High-performance XML parsing in Python with lxml

Stretch the limits of this full-featured XML parsing and serializing suite

Liza Daly

March 24, 2011
(First published October 28, 2008)

lxml is a fast yet flexible library for XML processing in Python. It comes bundled with support for XML Path Language (XPath) and Extensible Stylesheet Language Transformation (XSLT), and it implements the familiar ElementTree API. In this article, you focus both on the ease of use provided by lxml and on its high-performance profile when processing very large XML data.

Frequently used acronyms

- API: application programming interface
- DOM: Document Object Model
- HTML: Hypertext Markup Language
- SAX: Simple API for XML
- XML: Extensible Markup Language
- XPath: XML Path Language
- XSLT: Extensible Stylesheet Language Transformations

Introducing lxml

Python has never suffered from a scarcity of XML libraries. Since version 2.0, it has included the familiar `xml.dom.minidom` and related pulldom and Simple API for XML (SAX) models. Since 2.4, it has included the popular ElementTree API. In addition, there have always been third-party libraries that offer higher-level or more pythonic interfaces.

While any XML library is sufficient for simple Document Object Model (DOM) or SAX parsing of small files, developers are increasingly faced with larger datasets and a need for real-time parsing of XML in a Web services context. Meanwhile, experienced XML developers may prefer XML-native languages such as XPath or XSLT for their compactness and expressivity. It would be ideal to have access to the declarative syntax of XPath while retaining the general-purpose functionality available in Python.

Software versions and sample data used in this article

- lxml 2.1.2

- libxml2 2.7.1
- libxslt 1.1.24
- cElementTree 1.0.5
- United States copyright renewal data, supplied by Google
- Open Directory Resource Description Framework (RDF) content

See Related topics for more information about this software and data.

I performed the benchmarks on a Pentium M 1.86GHz ThinkPad T43 with 2GB of RAM running Ubuntu, using IPython's `timeit` command. Timings are supplied only for comparison among approaches and should not be taken as representative benchmarks for the software described.

lxml is the first Python XML library that demonstrates high-performance characteristics and includes native support for XPath 1.0, XSLT 1.0, custom element classes, and even a pythonic data-binding interface. It is built on top of two C libraries: `libxml2` and `libxslt`. They provide most of the horsepower behind the core tasks of parsing, serializing, and transforming.

Which parts of lxml you use in your code depends on your needs: Are you comfortable with XPath? Do you prefer to work with Python-like objects? How much memory do you have on the system to keep large trees available?

This article does not cover all of lxml but instead demonstrates techniques to efficiently process very large XML files, optimizing for high speed and low memory usage. Two freely available example documents are used: U.S. copyright renewal data converted into XML by Google and the Open Directory RDF content.

lxml is compared here only to cElementTree and not to the dozens of other Python libraries available. I chose cElementTree because it is a native part of Python 2.5 and, like lxml, built on C libraries.

What's so hard about very large data?

XML libraries are often designed for and tested on small sample files. Indeed, many real-world projects are begun without complete data available. Programmers work diligently for weeks or months using sample content and writing code such as that shown in Listing 1.

Listing 1. A simple parse operation

```
from lxml import etree
doc = etree.parse('content-sample.xml')
```

cElementTree

If parsing and simple analysis is all your program needs to do, consider the cElementTree module, which is bundled as part of Python 2.5. It is a C implementation of ElementTree that uses `expat` for parsing and is superior to all other libraries when you need to parse the entire document tree. However, its API is more restricted than even ElementTree, and it is slower than lxml in most other tasks, especially serialization.

The lxml `parse` method reads the entire document and builds an in-memory tree. Relative to cElementTree, an lxml tree is much more expensive because it retains more information about a

node's context, including references to its parent. Parsing a 2G document this way immediately puts a machine with 2G RAM into swap, with disastrous performance implications. If the whole application is written assuming this data will be available in memory, a major refactor is in order.

Iterative parsing

When building an in-memory tree is not desired or practical, use an iterative parsing technique that does not rely on reading the entire source file. lxml offers two approaches:

- Supplying a target parser class
- Using the `iterparse` method

Using the target parser method

The target parser method is familiar to developers who are comfortable with SAX event-driven code. A target parser is a class that implements the following methods:

1. **`start`** fires on element open. The data and children of the element are not yet available.
2. **`end`** fires on element close. All of the element's child nodes, including text nodes, are now available.
3. **`data`** fires on text children and has access to that text.
4. **`close`** fires when parsing is complete.

Listing 2 demonstrates creating a target parser class (here called `TitleTarget`) that implements the required methods. This parser collects the text children of the `Title` element in an internal list (`self.text`) and, upon reaching the `close()` method, returns that list.

Listing 2. A target parser that returns a list of all text children of the `Title` tag

```
class TitleTarget(object):
    def __init__(self):
        self.text = []
    def start(self, tag, attrib):
        self.is_title = True if tag == 'Title' else False
    def end(self, tag):
        pass
    def data(self, data):
        if self.is_title:
            self.text.append(data.encode('utf-8'))
    def close(self):
        return self.text

parser = etree.XMLParser(target = TitleTarget())

# This and most other samples read in the Google copyright data
infile = 'copyright.xml'

results = etree.parse(infile, parser)

# When iterated over, 'results' will contain the output from
# target parser's close() method

out = open('titles.txt', 'w')
out.write('\n'.join(results))
```

```
out.close()
```

This code was timed at 54 seconds when run against the copyright data. Target parsing can be reasonably fast and does not generate a memory-consuming parse tree, but all events fire for all elements in the data. For very large documents, this might not be desirable when only a few elements are of interest, such as in this example. Is it possible to limit processing to a selected tag and get better performance?

Using the `iterparse` method

lxml's `iterparse` method is an extension of the ElementTree API. `iterparse` returns a Python iterator for the selected element context. It accepts two useful arguments: a tuple of events to monitor and a tag name. In this case, I'm only interested in the text content of `<Title>` (which is available upon reaching the `end` event). The output from Listing 3 will be identical to that of the target parser method in Listing 2 but ought to be much faster because lxml can optimize the event handling internally. It's also many fewer lines of code.

Listing 3. Simple iteration over a named tag and event

```
context = etree.iterparse(infile, events=('end',), tag='Title')

for event, elem in context:
       out.write('%s\n' % elem.text.encode('utf-8'))
```

If you run this code and monitor the output, you see it begins by appending titles very quickly but soon slows to a crawl. A quick check of `top` shows that the computer has gone into swap:

```
PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
170 root      15  -5     0    0    0 D  3.9  0.0   0:01.32 kswapd0
```

What's going on? Although `iterparse` does not consume the entire file at first, it does not free the references to nodes from each iteration. When the whole document will be accessed repeatedly, this is a feature. However, in this case I would much rather reclaim that memory at the end of each loop. This includes both references to children or text nodes that were already been processed and preceding siblings of the current node, whose references from the root node are also implicitly preserved, as in Listing 4:

Listing 4. Revised iteration which clears unneeded node references

```
for event, elem in context:
    out.write('%s\n' % elem.text.encode('utf-8'))

    # It's safe to call clear() here because no descendants will be accessed
    elem.clear()

    # Also eliminate now-empty references from the root node to <Title>
    while elem.getprevious() is not None:
        del elem.getparent()[0]
```

For convenience, I refactor Listing 4 into a function that takes a callable `func` to do an operation against the current node, as in Listing 5. I'll use this method in the succeeding examples.

Listing 5. A function to loop through a context, calling `func` each time, and then clean up unneeded references

```
def fast_iter(context, func):
    for event, elem in context:
        func(elem)
        elem.clear()
        while elem.getprevious() is not None:
            del elem.getparent()[0]
    del context
```

Performance characteristics

This optimized `iterparse` approach in Listing 4 produces output that's identical to that produced by the target parser in Listing 2, but in half the time. It is even faster than cElementTree when the task is restricted to a particular event and tag name, as it is here. (In most cases, though, cElementTree will outperform lxml when parsing is the primary activity.)

Table 1 shows timings of various parser techniques as measured against the copyright data on the computer described in the benchmarks sidebar.

Table 1. Comparisons of iterative parsing methods: extract `text()` from `<Title>`

XML library	Method	Average time, in seconds
cElementTree	Iterparse	32
lxml	Target parser	54
lxml	Optimized iterparse	25

Does it scale?

Running the same `iterparse` method in Listing 4 on the Open Directory data takes 122 seconds per run, or slightly more than five times longer than parsing the copyright data. As the Open Directory data is also slightly more than five times as large (at 1.9 gigabytes), you should expect roughly linear time performance from this method, even on very large files.

Serialization

If all you need to do with an XML file is grab some text from within a single node, it might be possible to use a simple regular expression that will probably operate faster than any XML parser. In practice, though, this is nearly impossible to get right when the data is at all complex, and I do not recommend it. XML libraries are invaluable when true data manipulation is required.

Serializing XML to a string or file is where lxml excels because it relies on `libxml2` C code directly. If your task requires any serialization at all, lxml is a clear choice, but there are some tricks to get the best performance out of the library.

Use `deepcopy` when serializing subtrees

lxml retains references between child nodes and their parents. One effect of this is that a node in lxml can have one and only one parent. (cElementTree has no concept of parent nodes.)

Listing 6 takes each `<Record>` in the copyright file and writes a simplified record containing only the title and the copyright information.

Listing 6. Serialize an element's children

```
from lxml import etree
import deepcopy

def serialize(elem):
    # Output a new tree like:
    # <SimplerRecord>
    #   <Title>This title</Title>
    #   <Copyright><Date>date</Date><Id>id</Id></Copyright>
    # </SimplerRecord>

    # Create a new root node
    r = etree.Element('SimplerRecord')

    # Create a new child
    t = etree.SubElement(r, 'Title')

    # Set this child's text attribute to the original text contents of <Title>
    t.text = elem.iterchildren(tag='Title').next().text

    # Deep copy a descendant tree
    for c in elem.iterchildren(tag='Copyright'):
        r.append( deepcopy(c) )
    return r

out = open('titles.xml', 'w')
context = etree.iterparse('copyright.xml', events=('end',), tag='Record')

# Iterate through each of the <Record> nodes using our fast iteration method
fast_iter(context,
          # For each <Record>, serialize a simplified version and write it
          # to the output file
          lambda elem:
              out.write(
                 etree.tostring(serialize(elem), encoding='utf-8')))
```

Don't use `deepcopy` to simply replicate the text of a single node. It's faster to create a new node, populate its text attribute manually, and then serialize it. In my tests, calling `deepcopy` for both `<Title>` and `<Copyright>` was 15 percent slower than the code in Listing 6. You'll see the greatest performance boosts from `deepcopy` when serializing large descendant trees.

When benchmarked against cElementTree using the code in Listing 7, lxml's serializer was almost twice as fast (50 seconds versus 95 seconds):

Listing 7. Serializing with cElementTree

```
def serialize_cet(elem):
    r = cet.Element('Record')

    # Create a new element with the same text child
    t = cet.SubElement(r, 'Title')
    t.text = elem.find('Title').text

    # ElementTree does not store parent references -- an element can
    # exist in multiple trees. It's not necessary to use deepcopy here.
    for c in elem.findall('Copyright'):
       r.append(h)
```

```
    return r

context = cet.iterparse('copyright.xml', events=('end','start'))
context = iter(context)
event, root = context.next()

for event, elem in context:
    if elem.tag == 'Record' and event =='end':
        result = serialize_cet(elem)
        out.write(cet.tostring(result, encoding='utf-8'))
        root.clear()
```

For more information about this iteration pattern, see "Incremental Parsing" of the ElementTree documentation. (See Related topics for a link.)

Finding elements quickly

After parsing, the most common XML task is to locate specific data of interest inside the parsed tree. lxml offers several approaches, from a simplified search syntax to full XPath 1.0. As a user, you should be aware of the performance characteristics and optimization techniques for each approach.

Avoid use of `find` and `findall`

The `find` and `findall` methods, inherited from the ElementTree API, locate one or more descendant nodes using a simplified XPath-like expression language called ElementPath. Users migrating from ElementTree to lxml can naturally continue to use the find/ElementPath syntax.

lxml supplies two other options for discovering subnodes: the `iterchildren`/`iterdescendants` methods and true XPath. In cases where the expression should match a node name, it is far faster (in some cases twice as fast) to use the `iterchildren` or `iterdescendants` methods with their optional tag parameter when compared to their equivalent ElementPath expressions.

For more complex patterns, use the `XPath` class to precompile search patterns. Simple patterns that mimic the behavior of `iterchildren` with tag arguments (for example, `etree.XPath("child::Title")`) execute in effectively the same time as their `iterchildren` equivalents. It's important to precompile, though. Compiling the pattern in each execution of the loop or using the `xpath()` method on an element (described in the lxml documentation, see Resources) can be almost twice as slow as compiling once and then using that pattern repeatedly.

XPath evaluation in lxml is *fast*. If only a subset of nodes needs to be serialized, it is much better to limit with precise XPath expressions up front than to inspect all the nodes later. For example, limiting the sample serialization to include only titles containing the word `night`, as in Listing 8, takes 60 percent of the time to serialize the full set.

Listing 8. Conditional serialization with XPath classes

```
def write_if_node(out, node):
    if node is not None:
        out.write(etree.tostring(node, encoding='utf-8'))

def serialize_with_xpath(elem, xp1, xp2):
    '''Take our source <Record> element and apply two pre-compiled XPath classes.
```

```
    Return a node only if the first expression matches.
    '''
    r = etree.Element('Record')

    t = etree.SubElement(r, 'Title')
    x = xp1(elem)
    if x:
        t.text = x[0].text
        for c in xp2(elem):
            r.append(deepcopy(c))
        return r

xp1 = etree.XPath("child::Title[contains(text(), 'night')]")
xp2 = etree.XPath("child::Copyright")
out = open('out.xml', 'w')
context = etree.iterparse('copyright.xml', events=('end',), tag='Record')
fast_iter(context,
   lambda elem: write_if_node(out, serialize_with_xpath(elem, xp1, xp2)))
```

Finding nodes in other parts of the document

Note that, even when using `iterparse`, it is possible to use XPath predicates based on looking *ahead* of the current node. To find all `<Record>` nodes that are immediately followed by a record whose title contains the word `night`, do this:

```
etree.XPath("Title[contains(../Record/following::Record[1]/Title/text(), 'night')]")
```

However, when using the memory-efficient iteration strategy described in Listing 4, this command returns nothing because preceding nodes are cleared as parsing proceeds through the document:

```
etree.XPath("Title[contains(../Record/preceding::Record[1]/Title/text(), 'night')]")
```

While it is possible to write an efficient algorithm to solve this particular problem, tasks involving analyses across nodes—especially those that might be randomly distributed in the document—are usually more suited for an XML database that uses XQuery, such as eXist.

Other ways to increase performance

In addition to the use of specific methods *within* lxml, you can use approaches outside of the library to influence execution speed. Some of these are simple code changes; others require new thinking about how to handle large data problems.

Psyco

The Psyco module is an often-missed way to increase the speed of Python applications with minimal work. Typical gains for a pure Python program are between two and four times, but lxml does most of its work in C, so the difference is unusually small. When I ran Listing 4 with Psyco enabled, I reduced runtime by only three seconds (43.9 seconds versus 47.3 seconds). Psyco has a large memory overhead which might even negate any gains if the machine has to go to swap.

If your lxml-driven application has core pure Python code that's executed frequently (perhaps extensive string manipulation on text nodes), you might benefit if you enable Psyco for only those methods. For more information about Psyco, see Related topics.

Threading

If, instead, your application relies mostly on internal, C-driven lxml features, it might be to your advantage to run it as a threaded application in a multiprocessor environment. There are restrictions on how to start the threads—especially with XSLT. Consult the FAQ section on threads in the lxml documentation for more information.

Divide and conquer

If it is possible to divide extremely large documents into individually analyzable subtrees, then it becomes feasible to split the document at the subtree level (using lxml's fast serialization) and distribute the work on those files among multiple computers. Employing on-demand virtual servers is an increasingly popular solution for executing central processing unit (CPU) bound offline tasks.

General strategies for any high-volume XML task

The specific code samples presented here might not apply to your project, but consider a few principles—borne out by testing and the lxml documentation—when faced with XML data measured in gigabytes or more:

- Use an iterative parsing strategy to incrementally process large documents.
- If searching the entire document in random order is required, move to an indexed XML database.
- Be extremely conservative in the data that you select. If you are only interested in particular nodes, use methods that select by those names. If you require predicate syntax, try one of the XPath classes and methods available.
- Consider the task at hand and the comfort level of the developer. Object models such as lxml's objectify or Amara might be more natural for Python developers when speed is not a consideration. cElementTree is faster when only parsing is required.
- Take the time to do even simple benchmarking. When processing millions of records, small differences add up, and it is not always obvious which methods are the most efficient.

Conclusion

Many software products come with the *pick-two* caveat, meaning that you must choose only two: speed, flexibility, or readability. When used carefully, lxml can provide all three. XML developers who have struggled with DOM performance or with the event-driven model of SAX now have the chance to work with higher-level pythonic libraries. Programmers coming from a Python background who are new to XML have an easy way to explore the expressivity of XPath and XSLT. Both coding styles can co-exist happily in an lxml-based application.

lxml has more to offer than what was explored here. Be sure to look into the `lxml.objectify` module, especially for smaller datasets or applications that are not primarily XML-based. For content in HTML that might not be well formed, lxml provides two useful packages: the `lxml.html` module and the BeautifulSoup parser. It's also possible to extend lxml itself if you write Python modules that are callable from XSLT or create custom Python or C extensions. Find information about all of these in the lxml documentation mentioned in Related topics.

Related topics

- Help getting lxml to work reliably on MacOS-X: Read this thread for invaluable help on installing lxml on MacOS X.
- ElementTree Overview: Find information about the ElementTree API and cElementTree.
- Incremental Parsing: In this section of the ElementTree documentation, get more information about the iteration pattern used in Listing 6.
- developerWorks podcasts: Listen to interesting interviews and discussions for software developers.
- Google U.S. copyright renewal data: Download and experiment with this U.S. copyright renewal data converted into XML by Google (371MB, zipped, 426,907 individual records).
- Open Directory RDF content: Download RDF dumps of the Open Directory database (1.9GB, zipped, 5,354,663 individual records).
- eXist: Check out this open source database management system that uses XQuery.
- Psyco: Learn more about this Python extension module that can massively speed up the execution of Python code.
- IBM trial software for product evaluation: Build your next project with trial software available for download directly from developerWorks, including application development tools and middleware products from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.
- developerWorks podcasts: Listen to interesting interviews and discussions for software developers.