Parsing XML Documents

Parsed XML documents are represented in memory by **`ElementTree`** and **`Element`** objects connected into a tree structure based on the way the nodes in the XML document are nested.

Parsing an Entire Document

Parsing an entire document with **`parse()`** returns an **`ElementTree`** instance. The tree knows about all of the data in the input document, and the nodes of the tree can be searched or manipulated in place. While this flexibility can make working with the parsed document a little easier, it typically takes more memory than an event-based parsing approach since the entire document must be loaded at one time.

The memory footprint of small, simple documents such as this list of podcasts represented as an OPML outline is not significant:

```
<?xml version="1.0" encoding="UTF-8"?>
<opml version="1.0">
<head>
        <title>My Podcasts</title>
        <dateCreated>Sun, 07 Mar 2010 15:53:26 GMT</da
        <dateModified>Sun, 07 Mar 2010 15:53:26 GMT</d
</head>
<body>
  <outline text="Science and Tech">
    <outline text="APM: Future Tense" type="rss"
             xmlUrl="http://www.publicradio.org/colum
             htmlUrl="http://www.publicradio.org/colu
        <outline text="Engines Of Our Ingenuity Podca
             xmlUrl="http://www.npr.org/rss/podcast.p
             htmlUrl="http://www.uh.edu/engines/engine
        <outline text="Science &#38; the City" type="
             xmlUrl="http://www.nyas.org/Podcasts/Aton
             htmlUrl="http://www.nyas.org/WhatWeDo/Sc
  </outline>
  <outline text="Books and Fiction">
        <outline text="Podiobooker" type="rss"
             xmlUrl="http://feeds.feedburner.com/podi
             htmlUrl="http://www.podiobooks.com/blog"
        <outline text="The Drabblecast" type="rss"
             xmlUrl="http://web.me.com/normsherman/Si
             htmlUrl="http://web.me.com/normsherman/S
        <outline text="tor.com / category / tordotsto
             xmlUrl="http://www.tor.com/rss/category/
             htmlUrl="http://www.tor.com/" />
  </outline>
  <outline text="Computers and Programming">
        <outline text="MacBreak Weekly" type="rss"
             xmlUrl="http://leo.am/podcasts/mbw"
             htmlUrl="http://twit.tv/mbw" />
        <outline text="FLOSS Weekly" type="rss"
             xmlUrl="http://leo.am/podcasts/floss"
             htmlUrl="http://twit.tv" />
        <outline text="Core Intuition" type="rss"
             xmlUrl="http://www.coreint.org/podcast.x
             htmlUrl="http://www.coreint.org/" />
  </outline>
  <outline text="Python">
    <outline text="PyCon Podcast" type="rss"
             xmlUrl="http://advocacy.python.org/podca
             htmlUrl="http://advocacy.python.org/podc
        <outline text="A Little Bit of Python" type="
             xmlUrl="http://advocacy.python.org/podca
             htmlUrl="http://advocacy.python.org/podc
        <outline text="Django Dose Everything Feed" t
             xmlUrl="http://djangodose.com/everything
  </outline>
  <outline text="Miscelaneous">
```

Page Contents

Navigation

Table of Contents
Previous: xml.etree.ElementTree – XML Manipulation API
Next: Creating XML Documents

This Page

Show Source

Examples

The output from all the example programs from PyMOTW has been generated with Python 2.7.8, unless otherwise noted. Some of the features described here may not be available in earlier versions of Python.

If you are looking for examples that work under Python 3, please refer to the PyMOTW-3 section of the site.

```
        <outline text="dhellmann's CastSampler Feed" t
             xmlUrl="http://www.castsampler.com/cast/f
             htmlUrl="http://www.castsampler.com/users
    </outline>
  </body>
  </opml>
```

To parse the file, pass an open file handle to **`parse()`**.

```
from xml.etree import ElementTree

with open('podcasts.opml', 'rt') as f:
    tree = ElementTree.parse(f)

print tree
```

It will read the data, parse the XML, and return an **`ElementTree`** object.

```
$ python ElementTree_parse_opml.py

<xml.etree.ElementTree.ElementTree object at 0
```

Traversing the Parsed Tree

To visit all of the children in order, use **`iter()`** to create a generator that iterates over the **`ElementTree`** instance.

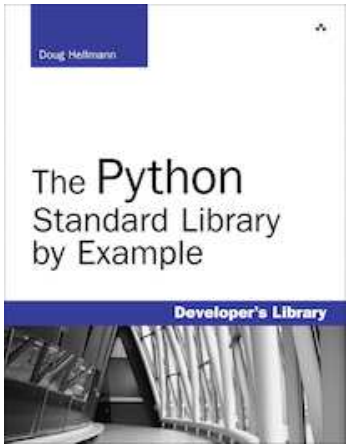
```
from xml.etree import ElementTree

with open('podcasts.opml', 'rt') as f:
    tree = ElementTree.parse(f)

for node in tree.iter():
    print node.tag, node.attrib
```

This example prints the entire tree, one tag at a time.

```
$ python ElementTree_dump_opml.py

opml {'version': '1.0'}
head {}
title {}
dateCreated {}
dateModified {}
body {}
outline {'text': 'Science and Tech'}
outline {'xmlUrl': 'http://www.publicradio.org
outline {'xmlUrl': 'http://www.npr.org/rss/pod
outline {'xmlUrl': 'http://www.nyas.org/Podcas
outline {'text': 'Books and Fiction'}
outline {'xmlUrl': 'http://feeds.feedburner.co
outline {'xmlUrl': 'http://web.me.com/normsher
outline {'xmlUrl': 'http://www.tor.com/rss/cat
outline {'text': 'Computers and Programming'}
outline {'xmlUrl': 'http://leo.am/podcasts/mbw
outline {'xmlUrl': 'http://leo.am/podcasts/flo
outline {'xmlUrl': 'http://www.coreint.org/pod
outline {'text': 'Python'}
outline {'xmlUrl': 'http://advocacy.python.org
outline {'xmlUrl': 'http://advocacy.python.org
outline {'xmlUrl': 'http://djangodose.com/ever
outline {'text': 'Miscelaneous'}
outline {'xmlUrl': 'http://www.castsampler.com
```

To print only the groups of names and feed URLs for the podcasts, leaving out of all of the data in the header section by iterating over only the `outline` nodes and print the *text* and *xmlUrl* attributes.

```
from xml.etree import ElementTree

with open('podcasts.opml', 'rt') as f:
    tree = ElementTree.parse(f)

for node in tree.iter('outline'):
    name = node.attrib.get('text')
    url = node.attrib.get('xmlUrl')
```



Buy the book!

```
    if name and url:
        print '  %s :: %s' % (name, url)
    else:
        print name
```

The `'outline'` argument to **`iter()`** means processing is limited to only nodes with the tag `'outline'`.

```
$ python ElementTree_show_feed_urls.py

Science and Tech
  APM: Future Tense :: http://www.publicradio.
  Engines Of Our Ingenuity Podcast :: http://w
  Science & the City :: http://www.nyas.org/Po
Books and Fiction
  Podiobooker :: http://feeds.feedburner.com/p
  The Drabblecast :: http://web.me.com/normshe
  tor.com / category / tordotstories :: http:/
Computers and Programming
  MacBreak Weekly :: http://leo.am/podcasts/mb
  FLOSS Weekly :: http://leo.am/podcasts/floss
  Core Intuition :: http://www.coreint.org/pod
Python
  PyCon Podcast :: http://advocacy.python.org/
  A Little Bit of Python :: http://advocacy.py
  Django Dose Everything Feed :: http://djangc
Miscelaneous
  dhellmann's CastSampler Feed :: http://www.c
```

Finding Nodes in a Document

Walking the entire tree like this searching for relevant nodes can be error prone. The example above had to look at each outline node to determine if it was a group (nodes with only a **`text`** attribute) or podcast (with both **`text`** and **`xmlUrl`**). To produce a simple list of the podcast feed URLs, without names or groups, for a podcast downloader application, the logic could be simplified using **`findall()`** to look for nodes with more descriptive search characteristics.

As a first pass at converting the above example, we can construct an XPath argument to look for all outline nodes.

```
from xml.etree import ElementTree

with open('podcasts.opml', 'rt') as f:
    tree = ElementTree.parse(f)

for node in tree.findall('.//outline'):
    url = node.attrib.get('xmlUrl')
    if url:
        print url
```

The logic in this version is not substantially different than the version using **`getiterator()`**. It still has to check for the presence of the URL, except that it does not print the group name when the URL is not found.

```
$ python ElementTree_find_feeds_by_tag.py

http://www.publicradio.org/columns/futuretense
http://www.npr.org/rss/podcast.php?id=510030
http://www.nyas.org/Podcasts/Atom.axd
http://feeds.feedburner.com/podiobooks
http://web.me.com/normsherman/Site/Podcast/rss
http://www.tor.com/rss/category/TorDotStories
http://leo.am/podcasts/mbw
http://leo.am/podcasts/floss
http://www.coreint.org/podcast.xml
http://advocacy.python.org/podcasts/pycon.rss
http://advocacy.python.org/podcasts/littlebit.
http://djangodose.com/everything/feed/
http://www.castsampler.com/cast/feed/rss/dhell
```

Another version can take advantage of the fact that the outline nodes are only nested two levels deep. Changing the search

path to `.//outline/outline` mean the loop will process only the second level of outline nodes.

```
from xml.etree import ElementTree

with open('podcasts.opml', 'rt') as f:
    tree = ElementTree.parse(f)

for node in tree.findall('.//outline/outline')
    url = node.attrib.get('xmlUrl')
    print url
```

All of those outline nodes nested two levels deep in the input are expected to have the *xmlURL* attribute refering to the podcast feed, so the loop can skip checking for for the attribute before using it.

```
$ python ElementTree_find_feeds_by_structure.p

http://www.publicradio.org/columns/futuretense
http://www.npr.org/rss/podcast.php?id=510030
http://www.nyas.org/Podcasts/Atom.axd
http://feeds.feedburner.com/podiobooks
http://web.me.com/normsherman/Site/Podcast/rss
http://www.tor.com/rss/category/TorDotStories
http://leo.am/podcasts/mbw
http://leo.am/podcasts/floss
http://www.coreint.org/podcast.xml
http://advocacy.python.org/podcasts/pycon.rss
http://advocacy.python.org/podcasts/littlebit.
http://djangodose.com/everything/feed/
http://www.castsampler.com/cast/feed/rss/dhell
```

This version is limited to the existing structure, though, so if the outline nodes are ever rearranged into a deeper tree it will stop working.

Parsed Node Attributes

The items returned by **`findall()`** and **`iter()`** are **`Element`** objects, each representing a node in the XML parse tree. Each **`Element`** has attributes for accessing data pulled out of the XML. This can be illustrated with a somewhat more contrived example input file, `data.xml`:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <top>
3    <child>This child contains text.</child>
4    <child_with_tail>This child has regular text.</child_with_tail>And "tail" text.
5    <with_attributes name="value" foo="bar" />
6    <entity_expansion attribute="This &#38; That">That &#38; This</entity_expansion>
7  </top>
```

The "attributes" of a node are available in the **`attrib`** property, which acts like a dictionary.

```
from xml.etree import ElementTree

with open('data.xml', 'rt') as f:
    tree = ElementTree.parse(f)

node = tree.find('./with_attributes')
print node.tag
for name, value in sorted(node.attrib.items())
    print '  %-4s = "%s"' % (name, value)
```

The node on line five of the input file has two attributes, **`name`** and **`foo`**.

```
$ python ElementTree_node_attributes.py

with_attributes
```

```
  foo  = "bar"
  name = "value"
```

The text content of the nodes is available, along with the "tail" text that comes after the end of a close tag.

```
from xml.etree import ElementTree

with open('data.xml', 'rt') as f:
    tree = ElementTree.parse(f)

for path in [ './child', './child_with_tail' ]
    node = tree.find(path)
    print node.tag
    print '  child node text:', node.text
    print '  and tail text  :', node.tail
```

The `child` node on line three contains embedded text, and the node on line four has text with a tail (including any whitespace).

```
$ python ElementTree_node_text.py

child
  child node text: This child contains text.
  and tail text  :

child_with_tail
  child node text: This child has regular text
  and tail text  : And "tail" text.
```

XML entity references embedded in the document are conveniently converted to the appropriate characters before values are returned.

```
from xml.etree import ElementTree

with open('data.xml', 'rt') as f:
    tree = ElementTree.parse(f)

node = tree.find('entity_expansion')
print node.tag
print '  in attribute:', node.attrib['attribut
print '  in text     :', node.text
```

The automatic conversion mean the implementation detail of representing certain characters in an XML document can be ignored.

```
$ python ElementTree_entity_references.py

entity_expansion
  in attribute: This & That
  in text     : That & This
```

Watching Events While Parsing

The other API useful for processing XML documents is event-based. The parser generates `start` events for opening tags and `end` events for closing tags. Data can be extracted from the document during the parsing phase by iterating over the event stream, which is convenient if it is not necessary to manipulate the entire document afterwards and there is no need to hold the entire parsed document in memory.

`iterparse()` returns an iterable that produces tuples containing the name of the event and the node triggering the event. Events can be one of:

`start`

A new tag has been encountered. The closing angle bracket of the tag was processed, but not the contents.

`end`

The closing angle bracket of a closing tag has been processed. All of the children were already processed.

`start-ns`

Start a namespace declaration.

`end-ns`

End a namespace declaration.

```
from xml.etree.ElementTree import iterparse

depth = 0
prefix_width = 8
prefix_dots = '.' * prefix_width
line_template = '{prefix:<0.{prefix_len}}{even

for (event, node) in iterparse('podcasts.opml'
    if event == 'end':
        depth -= 1

    prefix_len = depth * 2

    print line_template.format(prefix=prefix_d
                               prefix_len=pref
                               suffix='',
                               suffix_len=(pre
                               node=node,
                               node_id=id(node
                               event=event,
                               )

    if event == 'start':
        depth += 1
```

By default, only `end` events are generated. To see other events, pass the list of desired event names to **`iterparse()`**, as in this example:

```
$ python ElementTree_show_all_events.py

start            opml         4299786128
..start          head         4299786192
....start        title        4299786256
....end          title        4299786256
....start        dateCreated  4299786448
....end          dateCreated  4299786448
....start        dateModified 4299786640
....end          dateModified 4299786640
..end            head         4299786192
..start          body         4299787024
....start        outline      4299787088
......start      outline      4299787152
......end        outline      4299787152
......start      outline      4299787216
......end        outline      4299787216
......start      outline      4299787280
......end        outline      4299787280
....end          outline      4299787088
....start        outline      4299787344
......start      outline      4299787472
......end        outline      4299787472
......start      outline      4299787408
......end        outline      4299787408
......start      outline      4299787536
......end        outline      4299787536
....end          outline      4299787344
....start        outline      4299787600
......start      outline      4299787728
......end        outline      4299787728
......start      outline      4299787920
......end        outline      4299787920
......start      outline      4299787856
......end        outline      4299787856
....end          outline      4299787600
....start        outline      4299788048
......start      outline      4299788112
......end        outline      4299788112
......start      outline      4299788176
......end        outline      4299788176
......start      outline      4299792464
```

```
......end        outline      4299792464
....end          outline      4299788048
....start        outline      4299792592
......start      outline      4299792720
......end        outline      4299792720
....end          outline      4299792592
..end            body         4299787024
end              opml         4299786128
```

The event-style of processing is more natural for some operations, such as converting XML input to some other format. This technique can be used to convert list of podcasts from the earlier examples from an XML file to a CSV file, so they can be loaded into a spreadsheet or database application.

```
import csv
from xml.etree.ElementTree import iterparse
import sys

writer = csv.writer(sys.stdout, quoting=csv.QU

group_name = ''

for (event, node) in iterparse('podcasts.opml'
    if node.tag != 'outline':
        # Ignore anything not part of the outl
        continue
    if not node.attrib.get('xmlUrl'):
        # Remember the current group
        group_name = node.attrib['text']
    else:
        # Output a podcast entry
        writer.writerow( (group_name, node.att
                          node.attrib['xmlUrl'
                          node.attrib.get('htm
                          )
                         )
```

This conversion program does not need to hold the entire parsed input file in memory, and processing each node as it is encountered in the input is more efficient.

```
$ python ElementTree_write_podcast_csv.py

"Science and Tech","APM: Future Tense","http:/
"Science and Tech","Engines Of Our Ingenuity P
"Science and Tech","Science & the City","http:
"Books and Fiction","Podiobooker","http://feed
"Books and Fiction","The Drabblecast","http://
"Books and Fiction","tor.com / category / tord
"Computers and Programming","MacBreak Weekly",
"Computers and Programming","FLOSS Weekly","ht
"Computers and Programming","Core Intuition","
"Python","PyCon Podcast","http://advocacy.pyth
"Python","A Little Bit of Python","http://advo
"Python","Django Dose Everything Feed","http:/
"Miscelaneous","dhellmann's CastSampler Feed",
```

Creating a Custom Tree Builder

A potentially more efficient means of handling parse events is to replace the standard tree builder behavior with a custom version. The **`ElementTree`** parser uses an **`XMLTreeBuilder`** to process the XML and call methods on a target class to save the results. The usual output is an **`ElementTree`** instance created by the default **`TreeBuilder`** class. Replacing **`TreeBuilder`** with another class allows it to receive the events before the **`Element`** nodes are instantiated, saving that portion of the overhead.

The XML-to-CSV converter from the previous section can be translated to a tree builder.

```
import csv
from xml.etree.ElementTree import XMLTreeBuild
import sys
```

```
class PodcastListToCSV(object):

    def __init__(self, outputFile):
        self.writer = csv.writer(outputFile, q
        self.group_name = ''
        return

    def start(self, tag, attrib):
        if tag != 'outline':
            # Ignore anything not part of the
            return
        if not attrib.get('xmlUrl'):
            # Remember the current group
            self.group_name = attrib['text']
        else:
            # Output a podcast entry
            self.writer.writerow( (self.group_
                                   attrib['xml
                                   attrib.get(
                                   )
                                  )

    def end(self, tag):
        # Ignore closing tags
        pass
    def data(self, data):
        # Ignore data inside nodes
        pass
    def close(self):
        # Nothing special to do here
        return


target = PodcastListToCSV(sys.stdout)
parser = XMLTreeBuilder(target=target)
with open('podcasts.opml', 'rt') as f:
    for line in f:
        parser.feed(line)
parser.close()
```

`PodcastListToCSV` implements the **`TreeBuilder`** protocol. Each time a new XML tag is encountered, **`start()`** is called with the tag name and attributes. When a closing tag is seen **`end()`** is called with the name. In between, **`data()`** is called when a node has content (the tree builder is expected to keep up with the "current" node). When all of the input is processed, **`close()`** is called. It can return a value, which will be returned to the user of the **`XMLTreeBuilder`**.

```
$ python ElementTree_podcast_csv_treebuilder.p

"Science and Tech","APM: Future Tense","http:/
"Science and Tech","Engines Of Our Ingenuity P
"Science and Tech","Science & the City","http:
"Books and Fiction","Podiobooker","http://feed
"Books and Fiction","The Drabblecast","http://
"Books and Fiction","tor.com / category / tord
"Computers and Programming","MacBreak Weekly",
"Computers and Programming","FLOSS Weekly","ht
"Computers and Programming","Core Intuition","
"Python","PyCon Podcast","http://advocacy.pyth
"Python","A Little Bit of Python","http://advo
"Python","Django Dose Everything Feed","http:/
"Miscelaneous","dhellmann's CastSampler Feed",
```

Parsing Strings

To work with smaller bits of XML text, especially string literals as might be embedded in the source of a program, use **`XML()`** and the string containing the XML to be parsed as the only argument.

```
from xml.etree.ElementTree import XML

parsed = XML('''
<root>
  <group>
    <child id="a">This is child "a".</child>
```

```
    <child id="b">This is child "b".</child>
  </group>
  <group>
    <child id="c">This is child "c".</child>
  </group>
</root>
''')

print 'parsed =', parsed

for elem in parsed:
    print elem.tag
    if elem.text is not None and elem.text.str
        print '  text: "%s"' % elem.text
    if elem.tail is not None and elem.tail.str
        print '  tail: "%s"' % elem.tail
    for name, value in sorted(elem.attrib.item
        print '  %-4s = "%s"' % (name, value)
    print
```

Notice that unlike with **`parse()`**, the return value is an **`Element`** instance instead of an **`ElementTree`**. An **`Element`** supports the iterator protocol directly, so there is no need to call **`getiterator()`**.

```
$ python ElementTree_XML.py

parsed = <Element 'root' at 0x100497710>
group

group
```

For structured XML that uses the **`id`** attribute to identify unique nodes of interest, **`XMLID()`** is a convenient way to access the parse results.

```
from xml.etree.ElementTree import XMLID

tree, id_map = XMLID('''
<root>
  <group>
    <child id="a">This is child "a".</child>
    <child id="b">This is child "b".</child>
  </group>
  <group>
    <child id="c">This is child "c".</child>
  </group>
</root>
''')

for key, value in sorted(id_map.items()):
    print '%s = %s' % (key, value)
```

`XMLID()` returns the parsed tree as an **`Element`** object, along with a dictionary mapping the **`id`** attribute strings to the individual nodes in the tree.

```
$ python ElementTree_XMLID.py

a = <Element 'child' at 0x100497850>
b = <Element 'child' at 0x100497910>
c = <Element 'child' at 0x100497b50>
```

See also:

Outline Processor Markup Language, OPML

Dave Winer's OPML specification and documentation.

XML Path Language, XPath

A syntax for identifying parts of an XML document.

XPath Support in ElementTree

Part of Fredrick Lundh's original documentation for ElementTree.

csv

Read and write comma-separated-value files