Parsing an XML Document

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

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 more convenient, 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:

```
<!-- podcasts.opml -->

<?xml version="1.0" encoding="UTF-8"?>
<opml version="1.0">
<head>
    <title>My Podcasts</title>
    <dateCreated>Sat, 06 Aug 2016 15:53:26 GMT</dateCreated>
    <dateModified>Sat, 06 Aug 2016 15:53:26 GMT</dateModified>
</head>
<body>
  <outline text="Non-tech">
    <outline
        text="99% Invisible" type="rss"
        xmlUrl="http://feeds.99percentinvisible.org/99percentinvisible"
        htmlUrl="http://99percentinvisible.org" />
  </outline>
  <outline text="Python">
    <outline
        text="Talk Python to Me" type="rss"
        xmlUrl="https://talkpython.fm/episodes/rss"
        htmlUrl="https://talkpython.fm" />
    <outline
        text="Podcast.__init__" type="rss"
        xmlUrl="http://podcastinit.podbean.com/feed/"
        htmlUrl="http://podcastinit.com" />
  </outline>
</body>
</opml>
```

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

```
# ElementTree_parse_opml.py

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.

```
$ python3 ElementTree_parse_opml.py

<xml.etree.ElementTree.ElementTree object at 0x1013e5630>
```

Traversing the Parsed Tree

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

```
# ElementTree_dump_opml.py

from xml.etree import ElementTree
import pprint

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

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

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

```
$ python3 ElementTree_dump_opml.py

opml
head
title
dateCreated
dateModified
body
outline
outline
outline
outline
outline
```

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 by looking up the values in the `attrib` dictionary.

```
# ElementTree_show_feed_urls.py

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

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

```
$ python3 ElementTree_show_feed_urls.py

Non-tech
  99% Invisible
    http://feeds.99percentinvisible.org/99percentinvisible
Python
  Talk Python to Me
    https://talkpython.fm/episodes/rss
  Podcast.__init__
    http://podcastinit.podbean.com/feed/
```

Finding Nodes in a Document

Walking the entire tree like this, searching for relevant nodes, can be error prone. The previous example 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, the logic could be simplified using `findall()` to look

for nodes with more descriptive search characteristics.

As a first pass at converting the first version, an XPath argument can be used to look for all outline nodes.

```
# ElementTree_find_feeds_by_tag.py

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.

```
$ python3 ElementTree_find_feeds_by_tag.py

http://feeds.99percentinvisible.org/99percentinvisible
https://talkpython.fm/episodes/rss
http://podcastinit.podbean.com/feed/
```

It is possible to take advantage of the fact that the outline nodes are only nested two levels deep. Changing the search path to `.//outline/outline` means the loop will process only the second level of outline nodes.

```
# ElementTree_find_feeds_by_structure.py

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

```
$ python3 ElementTree_find_feeds_by_structure.py

http://feeds.99percentinvisible.org/99percentinvisible
https://talkpython.fm/episodes/rss
http://podcastinit.podbean.com/feed/
```

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`.

```
<!-- data.xml -->

<?xml version="1.0" encoding="UTF-8"?>
<top>
  <child>Regular text.</child>
  <child_with_tail>Regular text.</child_with_tail>"Tail" text.
  <with_attributes name="value" foo="bar" />
  <entity_expansion attribute="This &#38; That">
    That &#38; This
  </entity_expansion>
```

```
9 </top>
```

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

```
# ElementTree_node_attributes.py

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`.

```
$ python3 ElementTree_node_attributes.py

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

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

```
# ElementTree_node_text.py

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 whitespace).

```
$ python3 ElementTree_node_text.py

child
  child node text: Regular text.
  and tail text  :

child_with_tail
  child node text: Regular text.
  and tail text  : "Tail" text.
```

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

```
# ElementTree_entity_references.py

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['attribute'])
print('  in text     :', node.text.strip())
```

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

```
$ python3 ElementTree_entity_references.py

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

Watching Events While Parsing

The other API 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.

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.

`iterparse()` returns an iterable that produces tuples containing the name of the event and the node triggering the event.

```
# ElementTree_show_all_events.py

from xml.etree.ElementTree import iterparse

depth = 0
prefix_width = 8
prefix_dots = '.' * prefix_width
line_template = ''.join([
    '{prefix:<0.{prefix_len}}',
    '{event:<8}',
    '{suffix:<{suffix_len}} ',
    '{node.tag:<12} ',
    '{node_id}',
])

EVENT_NAMES = ['start', 'end', 'start-ns', 'end-ns']

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

    prefix_len = depth * 2

    print(line_template.format(
        prefix=prefix_dots,
        prefix_len=prefix_len,
        suffix='',
        suffix_len=(prefix_width - prefix_len),
        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.

```
$ python3 ElementTree_show_all_events.py

start            opml         4312612200
```

```
..start          head         4316174520
....start        title        4316254440
....end          title        4316254440
....start        dateCreated  4316254520
....end          dateCreated  4316254520
....start        dateModified 4316254680
....end          dateModified 4316254680
..end            head         4316174520
..start          body         4316254840
....start        outline      4316254920
......start      outline      4316255080
......end        outline      4316255080
....end          outline      4316254920
....start        outline      4316255160
......start      outline      4316255240
......end        outline      4316255240
......start      outline      4316255320
......end        outline      4316255320
....end          outline      4316255160
..end            body         4316254840
end              opml         4312612200
```

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.

```
# ElementTree_write_podcast_csv.py

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

writer = csv.writer(sys.stdout, quoting=csv.QUOTE_NONNUMERIC)

group_name = ''

parsing = iterparse('podcasts.opml', events=['start'])

for (event, node) in parsing:
    if node.tag != 'outline':
        # Ignore anything not part of the outline
        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.attrib['text'],
             node.attrib['xmlUrl'],
             node.attrib.get('htmlUrl', ''))
        )
```

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.

```
$ python3 ElementTree_write_podcast_csv.py

"Non-tech","99% Invisible","http://feeds.99percentinvisible.org/\
99percentinvisible","http://99percentinvisible.org"
"Python","Talk Python to Me","https://talkpython.fm/episodes/rss\
","https://talkpython.fm"
"Python","Podcast.__init__","http://podcastinit.podbean.com/feed\
/","http://podcastinit.com"
```

Note

The output from `ElementTree_write_podcast_csv.py` has been reformatted to fit on this page. The output lines ending with \ indicate an artificial line break.

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 `XMLParser` parser uses a `TreeBuilder` 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 re-implemented as a tree builder.

```
# ElementTree_podcast_csv_treebuilder.py

import csv
import io
from xml.etree.ElementTree import XMLParser
import sys


class PodcastListToCSV(object):

    def __init__(self, outputFile):
        self.writer = csv.writer(
            outputFile,
            quoting=csv.QUOTE_NONNUMERIC,
        )
        self.group_name = ''

    def start(self, tag, attrib):
        if tag != 'outline':
            # Ignore anything not part of the outline
            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_name,
                 attrib['text'],
                 attrib['xmlUrl'],
                 attrib.get('htmlUrl', ''))
            )

    def end(self, tag):
        "Ignore closing tags"

    def data(self, data):
        "Ignore data inside nodes"

    def close(self):
        "Nothing special to do here"


target = PodcastListToCSV(sys.stdout)
parser = XMLParser(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 `TreeBuilder`.

```
$ python3 ElementTree_podcast_csv_treebuilder.py

"Non-tech","99% Invisible","http://feeds.99percentinvisible.org/\
99percentinvisible","http://99percentinvisible.org"
```

```
"Python","Talk Python to Me","https://talkpython.fm/episodes/rss\
","https://talkpython.fm"
"Python","Podcast.__init__","http://podcastinit.podbean.com/feed\
/","http://podcastinit.com"
```

Note

The output from `ElementTree_podcast_csv_treebuidler.py` has been reformatted to fit on this page. The output lines ending with \ indicate an artificial line break.

Parsing Strings

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

```
# ElementTree_XML.py

from xml.etree.ElementTree import XML


def show_node(node):
    print(node.tag)
    if node.text is not None and node.text.strip():
        print('  text: "%s"' % node.text)
    if node.tail is not None and node.tail.strip():
        print('  tail: "%s"' % node.tail)
    for name, value in sorted(node.attrib.items()):
        print('  %-4s = "%s"' % (name, value))
    for child in node:
        show_node(child)


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:
    show_node(elem)
```

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()`.

```
$ python3 ElementTree_XML.py

parsed = <Element 'root' at 0x10079eef8>
group
child
  text: "This is child "a"."
  id   = "a"
child
  text: "This is child "b"."
  id   = "b"
group
child
  text: "This is child "c"."
  id   = "c"
```

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

```
# ElementTree_XMLID.py

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.

```
$ python3 ElementTree_XMLID.py

a = <Element 'child' at 0x10133aea8>
b = <Element 'child' at 0x10133aef8>
c = <Element 'child' at 0x10133af98>
```

xml.etree.ElementTree — XML Manipulation API

Building Documents With Element Nodes

Quick Links

Traversing the Parsed Tree
Finding Nodes in a Document
Parsed Node Attributes
Watching Events While Parsing
Creating a Custom Tree Builder
Parsing Strings

This page was last updated 2016-12-29.

Navigation

xml.etree.ElementTree — XML Manipulation API
Building Documents With Element Nodes

The output from all the example programs from PyMOTW-3 has been generated with Python 3.7.1, unless otherwise noted. Some of the features described here may not be available in earlier versions of Python.

Looking for examples for Python 2?