Building Documents With Element Nodes

In addition to its parsing capabilities, `xml.etree.ElementTree` also supports creating well-formed XML documents from `Element` objects constructed in an application. The `Element` class used when a document is parsed also knows how to generate a serialized form of its contents, which can then be written to a file or other data stream.

There are three helper functions useful for creating a hierarchy of `Element` nodes. `Element()` creates a standard node, `SubElement()` attaches a new node to a parent, and `Comment()` creates a node that serializes using XML's comment syntax.

```
# ElementTree_create.py

from xml.etree.ElementTree import (
    Element, SubElement, Comment, tostring,
)

top = Element('top')

comment = Comment('Generated for PyMOTW')
top.append(comment)

child = SubElement(top, 'child')
child.text = 'This child contains text.'

child_with_tail = SubElement(top, 'child_with_tail')
child_with_tail.text = 'This child has text.'
child_with_tail.tail = 'And "tail" text.'

child_with_entity_ref = SubElement(top, 'child_with_entity_ref')
child_with_entity_ref.text = 'This & that'

print(tostring(top))
```

The output contains only the XML nodes in the tree, not the XML declaration with version and encoding.

```
$ python3 ElementTree_create.py

b'<top><!--Generated for PyMOTW--><child>This child contains text.</
child><child_with_tail>This child has text.</child_with_tail>And "ta
il" text.<child_with_entity_ref>This &amp; that</child_with_entity_r
ef></top>'
```

The & character in the text of `child_with_entity_ref` is converted to the entity reference `&` automatically.

Pretty-Printing XML

`ElementTree` makes no effort to format the output of `tostring()` to make it easy to read because adding extra whitespace changes the contents of the document. To make the output easier to follow, the rest of the examples will use `xml.dom.minidom` to re-parse the XML then use its `toprettyxml()` method.

```
# ElementTree_pretty.py

from xml.etree import ElementTree
from xml.dom import minidom


def prettify(elem):
    """Return a pretty-printed XML string for the Element.
    """
    rough_string = ElementTree.tostring(elem, 'utf-8')
    reparsed = minidom.parseString(rough_string)
    return reparsed.toprettyxml(indent="  ")
```

```
    return reparsed.toprettyxml(indent="  ")
```

The updated example now looks like

```
# ElementTree_create_pretty.py

from xml.etree.ElementTree import Element, SubElement, Comment
from ElementTree_pretty import prettify

top = Element('top')

comment = Comment('Generated for PyMOTW')
top.append(comment)

child = SubElement(top, 'child')
child.text = 'This child contains text.'

child_with_tail = SubElement(top, 'child_with_tail')
child_with_tail.text = 'This child has text.'
child_with_tail.tail = 'And "tail" text.'

child_with_entity_ref = SubElement(top, 'child_with_entity_ref')
child_with_entity_ref.text = 'This & that'

print(prettify(top))
```

and the output is easier to read.

```
$ python3 ElementTree_create_pretty.py

<?xml version="1.0" ?>
<top>
  <!--Generated for PyMOTW-->
  <child>This child contains text.</child>
  <child_with_tail>This child has text.</child_with_tail>
  And &quot;tail&quot; text.
  <child_with_entity_ref>This &amp; that</child_with_entity_ref>
</top>
```

In addition to the extra whitespace for formatting, the `xml.dom.minidom` pretty-printer also adds an XML declaration to the output.

Setting Element Properties

The previous example created nodes with tags and text content, but did not set any attributes of the nodes. Many of the examples from Parsing an XML Document worked with an OPML file listing podcasts and their feeds. The `outline` nodes in the tree used attributes for the group names and podcast properties. `ElementTree` can be used to construct a similar XML file from a CSV input file, setting all of the element attributes as the tree is constructed.

```
# ElementTree_csv_to_xml.py

import csv
from xml.etree.ElementTree import (
    Element, SubElement, Comment, tostring,
)
import datetime
from ElementTree_pretty import prettify

generated_on = str(datetime.datetime.now())

# Configure one attribute with set()
root = Element('opml')
root.set('version', '1.0')

root.append(
    Comment('Generated by ElementTree_csv_to_xml.py for PyMOTW')
)

head = SubElement(root, 'head')
```

```
head = SubElement(root, 'head')
title = SubElement(head, 'title')
title.text = 'My Podcasts'
dc = SubElement(head, 'dateCreated')
dc.text = generated_on
dm = SubElement(head, 'dateModified')
dm.text = generated_on

body = SubElement(root, 'body')

with open('podcasts.csv', 'rt') as f:
    current_group = None
    reader = csv.reader(f)
    for row in reader:
        group_name, podcast_name, xml_url, html_url = row
        if (current_group is None or
                group_name != current_group.text):
            # Start a new group
            current_group = SubElement(
                body, 'outline',
                {'text': group_name},
            )
        # Add this podcast to the group,
        # setting all its attributes at
        # once.
        podcast = SubElement(
            current_group, 'outline',
            {'text': podcast_name,
             'xmlUrl': xml_url,
             'htmlUrl': html_url},
        )

print(prettify(root))
```

This example uses two techniques to set the attribute values of new nodes. The root node is configured using `set()` to change one attribute at a time. The podcast nodes are given all of their attributes at once by passing a dictionary to the node factory.

```
$ python3 ElementTree_csv_to_xml.py

<?xml version="1.0" ?>
<opml version="1.0">
  <!--Generated by ElementTree_csv_to_xml.py for PyMOTW-->
  <head>
    <title>My Podcasts</title>
    <dateCreated>2016-08-06 17:09:00.524979</dateCreated>
    <dateModified>2016-08-06 17:09:00.524979</dateModified>
  </head>
  <body>
    <outline text="Non-tech">
      <outline htmlUrl="http://99percentinvisible.org" text="99%\
 Invisible" xmlUrl="http://feeds.99percentinvisible.org/99percen\
tinvisible"/>
    </outline>
    <outline text="Python">
      <outline htmlUrl="https://talkpython.fm" text="Talk Python\
 to Me" xmlUrl="https://talkpython.fm/episodes/rss"/>
    </outline>
    <outline text="Python">
      <outline htmlUrl="http://podcastinit.com" text="Podcast.__\
init__" xmlUrl="http://podcastinit.podbean.com/feed/"/>
    </outline>
  </body>
</opml>
```

Building Trees from Lists of Nodes

Multiple children can be added to an `Element` instance together with the `extend()` method. The argument to `extend()` is any iterable, including a `list` or another `Element` instance.

```
# ElementTree_extend.py
```

```
# ElementTree_extend.py

from xml.etree.ElementTree import Element, tostring
from ElementTree_pretty import prettify

top = Element('top')

children = [
    Element('child', num=str(i))
    for i in range(3)
]

top.extend(children)

print(prettify(top))
```

When a `list` is given, the nodes in the list are added directly to the new parent.

```
$ python3 ElementTree_extend.py

<?xml version="1.0" ?>
<top>
  <child num="0"/>
  <child num="1"/>
  <child num="2"/>
</top>
```

When another `Element` instance is given, the children of that node are added to the new parent.

```
# ElementTree_extend_node.py

from xml.etree.ElementTree import (
    Element, SubElement, tostring, XML,
)
from ElementTree_pretty import prettify

top = Element('top')

parent = SubElement(top, 'parent')

children = XML(
    '<root><child num="0" /><child num="1" />'
    '<child num="2" /></root>'
)
parent.extend(children)

print(prettify(top))
```

In this case, the node with tag `root` created by parsing the XML string has three children, which are added to the `parent` node. The `root` node is not part of the output tree.

```
$ python3 ElementTree_extend_node.py

<?xml version="1.0" ?>
<top>
  <parent>
    <child num="0"/>
    <child num="1"/>
    <child num="2"/>
  </parent>
</top>
```

It is important to understand that `extend()` does not modify any existing parent-child relationships with the nodes. If the values passed to `extend()` exist somewhere in the tree already, they will still be there, and will be repeated in the output.

```
# ElementTree_extend_node_copy.py

from xml.etree.ElementTree import (
    Element, SubElement, tostring, XML,
```

```
)
from ElementTree_pretty import prettify

top = Element('top')

parent_a = SubElement(top, 'parent', id='A')
parent_b = SubElement(top, 'parent', id='B')

# Create children
children = XML(
    '<root><child num="0" /><child num="1" />'
    '<child num="2" /></root>'
)

# Set the id to the Python object id of the node
# to make duplicates easier to spot.
for c in children:
    c.set('id', str(id(c)))

# Add to first parent
parent_a.extend(children)

print('A:')
print(prettify(top))
print()

# Copy nodes to second parent
parent_b.extend(children)

print('B:')
print(prettify(top))
print()
```

Setting the `id` attribute of these children to the Python unique object identifier highlights the fact that the same node objects appear in the output tree more than once.

```
$ python3 ElementTree_extend_node_copy.py

A:
<?xml version="1.0" ?>
<top>
  <parent id="A">
    <child id="4316789880" num="0"/>
    <child id="4316789960" num="1"/>
    <child id="4316790040" num="2"/>
  </parent>
  <parent id="B"/>
</top>


B:
<?xml version="1.0" ?>
<top>
  <parent id="A">
    <child id="4316789880" num="0"/>
    <child id="4316789960" num="1"/>
    <child id="4316790040" num="2"/>
  </parent>
  <parent id="B">
    <child id="4316789880" num="0"/>
    <child id="4316789960" num="1"/>
    <child id="4316790040" num="2"/>
  </parent>
</top>
```

Serializing XML to a Stream

`tostring()` is implemented by writing to an in-memory file-like object, then returning a string representing the entire element tree. When working with large amounts of data, it will take less memory and make more efficient use of the I/O libraries to write directly to a file handle using the `write()` method of `ElementTree`.

```
# ElementTree_write.py
```

```
import io
import sys
from xml.etree.ElementTree import (
    Element, SubElement, Comment, ElementTree,
)

top = Element('top')

comment = Comment('Generated for PyMOTW')
top.append(comment)

child = SubElement(top, 'child')
child.text = 'This child contains text.'

child_with_tail = SubElement(top, 'child_with_tail')
child_with_tail.text = 'This child has regular text.'
child_with_tail.tail = 'And "tail" text.'

child_with_entity_ref = SubElement(top, 'child_with_entity_ref')
child_with_entity_ref.text = 'This & that'

empty_child = SubElement(top, 'empty_child')

ElementTree(top).write(sys.stdout.buffer)
```

The example uses `sys.stdout.buffer` to write to the console instead of `sys.stdout` because `ElementTree` produces encoded bytes instead of a Unicode string. It could also write to a file opened in binary mode or socket.

```
$ python3 ElementTree_write.py

<top><!--Generated for PyMOTW--><child>This child contains text.</ch
ild><child_with_tail>This child has regular text.</child_with_tail>A
nd "tail" text.<child_with_entity_ref>This &amp; that</child_with_en
tity_ref><empty_child /></top>
```

The last node in the tree contains no text or sub-nodes, so it is written as an empty tag, `<empty_child />`. `write()` takes a `method` argument to control the handling for empty nodes.

```
# ElementTree_write_method.py
```

```
import io
import sys
from xml.etree.ElementTree import (
    Element, SubElement, ElementTree,
)

top = Element('top')

child = SubElement(top, 'child')
child.text = 'Contains text.'

empty_child = SubElement(top, 'empty_child')

for method in ['xml', 'html', 'text']:
    print(method)
    sys.stdout.flush()
    ElementTree(top).write(sys.stdout.buffer, method=method)
    print('\n')
```

Three methods are supported:

`xml`
: The default method, produces `<empty_child />`.

`html`
: Produce the tag pair, as is required in HTML documents (`<empty_child></empty_child>`).

`text`
: Prints only the text of nodes, and skips empty tags entirely.

Prints only the text of nodes, and skips empty tags entirely.

```
$ python3 ElementTree_write_method.py

xml
<top><child>Contains text.</child><empty_child /></top>

html
<top><child>Contains text.</child><empty_child></empty_child></t
op>

text
Contains text.
```

Parsing an XML Document

csv — Comma-separated Value Files

Quick Links

This page was last updated 2016-12-29.

Navigation

Parsing an XML Document
csv — Comma-separated Value Files

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?