

Python XML processing with lxml



John W. Shipman

2009-01-22 20:51

Abstract

Describes the `lxml` package for reading and writing XML files with the Python programming language.

This publication is available in Web form¹ and also as a PDF document². Please forward any comments to tcc-doc@nmt.edu.

Table of Contents

1. Introduction: Python and XML	2
2. How <code>ElementTree</code> represents XML	3
3. Reading an XML document	4
4. Creating a new XML document	5
5. Modifying an existing XML document	6
6. Features of the <code>etree</code> module	6
6.1. The <code>Comment()</code> constructor	6
6.2. The <code>Element()</code> constructor	7
6.3. The <code>ElementTree()</code> constructor	7
6.4. The <code>fromstring()</code> function: Create an element from a string	9
6.5. The <code>parse()</code> function: build an <code>ElementTree</code> from a file	9
6.6. The <code>ProcessingInstruction()</code> constructor	9
6.7. The <code>QName()</code> constructor	10
6.8. The <code>SubElement()</code> constructor	10
6.9. The <code>tostring()</code> function: Serialize as XML	11
6.10. The <code>XMLID()</code> function: Convert text to XML with a dictionary of <code>id</code> values	11
7. class <code>ElementTree</code> : A complete XML document	12
7.1. <code>ElementTree.find()</code>	12
7.2. <code>ElementTree.findall()</code> : Find matching elements	13
7.3. <code>ElementTree.findtext()</code> : Retrieve the text content from an element	13
7.4. <code>ElementTree.getiterator()</code> : Make an iterator	13
7.5. <code>ElementTree.getroot()</code> : Find the root element	14
7.6. <code>ElementTree.xpath()</code> : Evaluate an <i>XPath</i> expression	14
7.7. <code>ElementTree.write()</code> : Translate back to XML	14
8. class <code>Element</code> : One element in the tree	14
8.1. Attributes of an <code>Element</code> instance	15
8.2. Accessing the list of child elements	15
8.3. <code>Element.append()</code> : Add a new element child	16
8.4. <code>Element.clear()</code> : Make an element empty	16

¹ <http://www.nmt.edu/tcc/help/pubs/pylxml/>

² <http://www.nmt.edu/tcc/help/pubs/pylxml/pylxml.pdf>

8.5. <code>Element.find()</code> : Find a matching sub-element	16
8.6. <code>Element.findall()</code> : Find all matching sub-elements	17
8.7. <code>Element.findtext()</code> : Extract text content	17
8.8. <code>Element.get()</code> : Retrieve an attribute value with defaulting	17
8.9. <code>Element.getchildren()</code> : Get element children	18
8.10. <code>Element.getiterator()</code> : Make an iterator to walk a subtree	18
8.11. <code>Element.getroottree()</code> : Find the <code>ElementTree</code> containing this element	19
8.12. <code>Element.insert()</code> : Insert a new child element	20
8.13. <code>Element.items()</code> : Produce attribute names and values	20
8.14. <code>Element.iterancestors()</code> : Find an element's ancestors	20
8.15. <code>Element.iterchildren()</code> : Find all children	21
8.16. <code>Element.iterdescendants()</code> : Find all descendants	21
8.17. <code>Element.itorsiblings()</code> : Find other children of the same parent	22
8.18. <code>Element.keys()</code> : Find all attribute names	22
8.19. <code>Element.remove()</code> : Remove a child element	23
8.20. <code>Element.set()</code> : Set an attribute value	23
8.21. <code>Element.xpath()</code> : Evaluate an <i>XPath</i> expression	23
9. <i>XPath</i> processing	23
9.1. An <i>XPath</i> example	24
10. Automated validation of input files	25
10.1. Validation with a Relax NG schema	25
10.2. Validation with an XSchema (XSD) schema	27
11. <code>etbuilder</code> : A simplified XML builder module	27
11.1. Using the <code>etbuilder</code> module	28
12. Implementation of <code>etbuilder</code>	29
12.1. Features differing from Lundh's original	29
12.2. Prologue	29
12.3. <code>class ElementMaker</code> : The factory class	30
12.4. <code>ElementMaker.__init__()</code> : Constructor	31
12.5. <code>ElementMaker.__call__()</code> : Handle calls to the factory instance	34
12.6. <code>ElementMaker.__handleArg()</code> : Process one positional argument	35
12.7. <code>ElementMaker.__getattr__()</code> : Handle arbitrary method calls	36
12.8. Epilogue	36
12.9. <code>testetbuilder</code> : A test driver for <code>etbuilder</code>	36

1. Introduction: Python and XML

With the continued growth of both Python and XML, there is a plethora of packages out there that help you read, generate, and modify XML files from Python scripts. Compared to most of them, the `lxml` package has two big advantages:

- Performance. Reading and writing even fairly large XML files takes an almost imperceptible amount of time.
- Ease of programming. The `lxml` package is based on `ElementTree`, which Fredrik Lundh invented to simplify and streamline XML processing.

`lxml` is similar in many ways to two other, earlier packages:

- Fredrik Lundh continues to maintain his original `ElementTree`³.

³ <http://effbot.org/zone/element-index.htm>

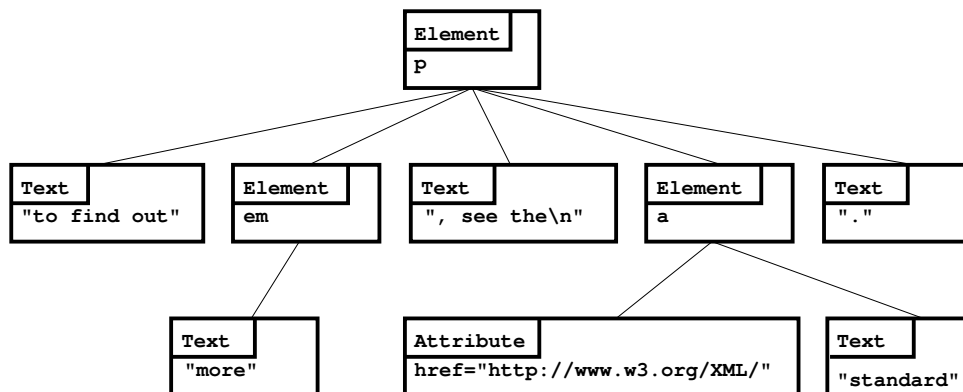
- `xml.etree.ElementTree`⁴ is now an official part of the Python library. There is a C-language version called `cElementTree` which may be even faster than `lxml` for some applications.

However, the author prefers `lxml` for providing a number of additional features that make life easier. In particular, support for *XPath* makes it considerably easier to manage more complex XML structures.

2. How ElementTree represents XML

If you have done XML work using the Document Object Model (DOM), you will find that the `lxml` package has a quite different way of representing documents as trees. In the DOM, trees are built out of nodes represented as `Node` instances. Some nodes are `Element` instances, representing whole elements. Each `Element` has an assortment of child nodes of various types: `Element` nodes for its element children; `Attribute` nodes for its attributes; and `Text` nodes for textual content.

Here is a small fragment of XHTML, and its representation as a DOM tree:



```
<p>To find out <em>more</em>, see the
<a href="http://www.w3.org/XML">standard</a>.</p>
```

The above diagram shows the conceptual structure of the XML. The `lxml` view of an XML document, by contrast, builds a tree of only one node type: the `Element`.

The main difference between the `ElementTree` view used in `lxml`, and the classical view, is the association of text with elements: it is very different in `lxml`.

An instance of `lxml`'s `Element` class contains these attributes:

.tag

The name of the element, such as "p" for a paragraph or "em" for emphasis.

.text

The text inside the element, if any, *up to the first child element*. This attribute is `None` if the element is empty or has no text before the first child element.

.tail

The text **following** the element. This is the most unusual departure. In the DOM model, any text following an element *E* is associated with the parent of *E*; in `lxml`, that text is considered the "tail" of *E*.

⁴ <http://docs.python.org/lib/module-xml.etree.ElementTree.html>

.attrib

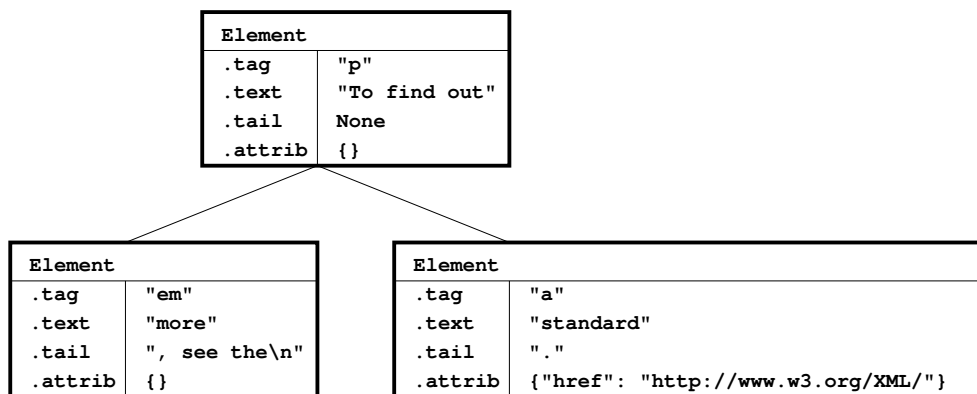
A Python dictionary containing the element's XML attribute names and their corresponding values. For example, for the element "`<h2 class="arch" id="N15">`", that element's `.attrib` would be the dictionary "`{"class": "arch", "id": "N15"}`".

(element children)

To access sub-elements, treat an element as a list. For example, if `node` is an `Element` instance, `node[0]` is the first sub-element of `node`. If `node` doesn't have any sub-elements, this operation will raise an `IndexError` exception.

You can find out the number of sub-elements using the `len()` function. For example, if `node` has five children, `len(node)` will return a value of 5.

One advantage of the `lxml` view is that a tree is now made of only one type of node: each node is an `Element` instance. Here is our XML fragment again, and a picture of its representation in `lxml`.



```
<p>To find out <em>more</em>, see the
<a href="http://www.w3.org/XML">standard</a>.</p>
```

Notice that in the `lxml` view, the text `", see the\n"` (which includes the newline) is contained in the `.tail` attribute of the `em` element, not associated with the `p` element as it would be in the DOM view. Also, the `."` at the end of the paragraph is in the `.tail` attribute of the `a` (link) element.

Now that you know how XML is represented in `lxml`, there are three general application areas.

- Section 3, "Reading an XML document" (p. 4).
- Section 4, "Creating a new XML document" (p. 5).
- Section 5, "Modifying an existing XML document" (p. 6).

3. Reading an XML document

Suppose you want to extract some information from an XML document. Here's the general procedure:

1. You'll need to import the `lxml` package. Here is one way to do it:

```
from lxml import etree
```

2. Typically your XML document will be in a file somewhere. Suppose your file is named `test.xml`; to read the document, you might say something like:

```
doc = etree.parse ( 'test.xml' )
```

The returned value `doc` is an instance of the `ElementTree` class that represents your XML document in tree form.

Once you have your document in this form, refer to Section 7, “class `ElementTree`: A complete XML document” (p. 12) to learn how to navigate around the tree and extract the various parts of its structure.

For other methods of creating an `ElementTree`, refer to Section 6, “Features of the `etree` module” (p. 6).

4. Creating a new XML document

If your program needs to write some output as an XML document, the `lxml` package makes this operation easy.

1. First import the `lxml` package. Here is one way:

```
from lxml import etree
```

2. Create the root element. For example, suppose you're creating a Web page; the root element is `html`. Use the `etree.Element()` constructor to build that element.

```
page = etree.Element ( 'html' )
```

3. Next, use the `etree.ElementTree()` constructor to make a new document tree, using our `html` element as its root:

```
doc = etree.ElementTree ( page )
```

4. The `etree.SubElement()` constructor is perfect for adding new child elements to our document. Here's the code to add a `head` element, and then a `body` as element, as new children of the `html` element:

```
headElt = etree.SubElement ( page, 'head' )  
bodyElt = etree.SubElement ( page, 'body' )
```

5. Your page will need a `title` element child under the `head` element. Add text to this element by storing a string in its `.text` attribute:

```
title = etree.SubElement ( headElt, 'title' )  
title.text = 'Your page title here'
```

6. To supply attribute values, use keyword arguments to the `SubElement()` constructor. For example, suppose you want a stylesheet link inside the `head` element that looks like this:

```
<link rel='stylesheet' href='mystyle.css' type='text/css'>
```

This code would do it:

```
linkElt = etree.SubElement ( headElt, 'link', rel='stylesheet',  
                             href='mystyle.css', type='text/css' )
```

7. Continue building your new document using the various functions described in Section 6, “Features of the `etree` module” (p. 6) and Section 8, “class `Element`: One element in the tree” (p. 14).
8. When the document is completely built, write it to a file using the `ElementTree` instance's `.write()` method, which takes a file argument.

```
outFile = open ( 'homemade.xml' )
doc.write ( outFile )
```

5. Modifying an existing XML document

If your program needs to read in an XML document, modify it, and write it back out, this operation is straightforward with `lxml`.

1. Start by reading the document using the techniques from Section 3, “Reading an XML document” (p. 4).
2. Modify the document tree by adding, deleting, or replacing elements, attributes, text, and other features.

For example, suppose your program has a variable `linkNode` that contains an `Element` instance representing an HTML “a” (hyperlink) element, and you want to change the value of its `href` attribute to point to a different URL, such as `http://www.nmt.edu/`. This code would do it:

```
linkNode.attrib['href'] = 'http://www.nmt.edu/'
```

3. Finally, write the document back out to a file as described in Section 4, “Creating a new XML document” (p. 5).

6. Features of the `etree` module

The `etree` contains numerous functions and class constructors.

6.1. The `Comment()` constructor

To create a comment node, use this constructor:

```
etree.Comment ( text=None )
```

text

The text to be placed within the comment. When serialized back into XML form, this text will be preceded by “`<!--`” and followed by “`-->`”. Note that one space will be added around each end of the text you supply.

The return value is an instance of the `Comment` class. Use the `.append()` method on the parent element to place the comment into your document.

For example, suppose `bodyElt` is an HTML `body` element. To add a comment under this element containing string `s`, you would use this code:

```
newComment = etree.Comment ( s )
bodyElt.append ( newComment )
```

6.2. The Element () constructor

This constructor creates and returns a new `Element` instance.

```
etree.Element ( tag, attrib={}, **extras )
```

tag

A string containing the name of the element to be created.

attrib

A dictionary containing attribute names and values to be added to the element. The default is to have no attributes.

extras

Any keyword arguments of the form *name=value* that you supply to the constructor are added to the element's attributes. For example, this code:

```
newReed = etree.Element ( 'reed', pitch='440', id='a4' )
```

will produce an element that looks like this:

```
<reed pitch='440' id='a4'/>
```

If you are working with multiple namespaces, you may also include an `nsmap` keyword argument defining your preferred namespace prefixes. For more on the `nsmap` argument, see Section 6.3, “The `ElementTree()` constructor” (p. 7).

There is one minor pathology of this constructor. If you pass in a pre-constructed dictionary as the `attrib` argument, and you also supply keyword arguments, the values of the keyword arguments will be added into that dictionary as if you had used the `.update()` method on the `attrib` dictionary. Here is a conversational example showing this side effect:

```
>>> from lxml import etree
>>> d = { 'name': 'Clem', 'clan': 'bozo' }
>>> clownElt = etree.Element ( 'clown', d, attitude='bad' )
>>> d
{'clan': 'bozo', 'attitude': 'bad', 'name': 'Clem'}
>>> etree.tostring ( clownElt )
'<clown clan="bozo" attitude="bad" name="Clem"/>'
>>>
```

6.3. The ElementTree () constructor

To create a new, empty document, use this constructor. It returns a new `ElementTree` instance.

```
etree.ElementTree ( element=None, file=None, nsmap=None )
```

element

An `Element` instance to be used as the root element.

file

To construct an `ElementTree` that represents an existing file, pass either a writeable `file` object, or a string containing the name of the file. Do not use the `element` argument; if you do, the `file` argument will be ignored.

For example, to transform a file named `balrog.xml` into an `ElementTree`, use this statement:

```
balrogTree = etree.ElementTree ( file='balrog.xml' )
```

nsmmap

If your document contains multiple XML namespaces, you can supply a dictionary that defines the namespace prefixes you would like to use when this document is converted to XML.

In this dictionary, the keys are namespace prefixes, and each corresponding value is the URI of that namespace. Use `None` as the key to define the blank namespace's URI.

Here is an example of creation of a document with such a dictionary:

```
HTML_NS = "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"
XSL_NS = "http://www.w3.org/1999/XSL/Transform"
EXSL_NS = "http://exslt.org/common"
NS_MAP = {
    None: HTML_NS,
    "xsl": XSL_NS,
    "exsl": EXSL_NS }
rootName = '{%s}%s' % (XSL_NS, 'stylesheet' )
root = etree.Element ( rootName )
sheet = etree.ElementTree ( root, nsmmap=NS_MAP )
```

When this root element is serialized into XML, it will look something like this:

```
<xsl:stylesheet
  xmlns="http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"
  xmlns:exsl="http://exslt.org/common"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
```

Exceptions that can be raised by this constructor include:

IOError

If the file is nonexistent or unreadable.

etree.XMLSyntaxError

If the file is readable, but its contents are not well-formed XML.

The returned exception value has an `.error_log` attribute that you can display to find out where in the file errors occurred. Here is an example:

```
>>> try:
...     bad = etree.fromstring ( "<a>\n<oops>\n</a>" )
... except etree.XMLSyntaxError, detail:
...     pass
...
>>> detail
<etree.XMLSyntaxError instance at 0xb7eba10c>
>>> detail.error_log
<string>:2:FATAL:PARSER:ERR_NAME_REQUIRED: StartTag: invalid element
name
<string>:3:FATAL:PARSER:ERR_TAG_NAME_MISMATCH: Opening and ending tag
mismatch: oops line 2 and a
<string>:3:FATAL:PARSER:ERR_TAG_NOT_FINISHED: Premature end of data in
tag a line 1
>>>
```

6.4. The `fromstring()` function: Create an element from a string

You can create an element or tree of elements from a string containing XML with this function; it returns a new `Element` instance representing all that XML.

```
etree.fromstring ( s )
```

where `s` is a string.

Here's an example:

```
>>> milne = '''<monster name='Heffalump'>
...   <trail>Woozle</trail>
...   <eeyore mood='boggy' />
... </monster>'''
>>> doc = etree.fromstring ( milne )
>>> print etree.tostring ( doc )
<monster name="Heffalump">
  <trail>Woozle</trail>
  <eeyore mood="boggy"/>
</monster>
>>>
```

6.5. The `parse()` function: build an `ElementTree` from a file

The quickest way to convert an XML file into an `ElementTree` is to use this function:

```
etree.parse ( source )
```

where `source` is the name of the file, or a `file` object containing the XML. If the file is well-formed, the function returns an `ElementTree` instance.

Exceptions raised include:

`IOError`

The file is nonexistent or not readable.

`etree.XMLSyntaxError`

The file is readable, but does not contain well-formed XML. The returned exception contains an `.error_log` attribute that you can print to see where the error occurred. For an example of the display of the `error_log`, see Section 6.3, “The `ElementTree()` constructor” (p. 7).

6.6. The `ProcessingInstruction()` constructor

To add an XML processing instruction to your document, use this constructor. It returns a new `ProcessingInstruction` instance; to place this into a tree, pass that instance to the parent element's `.append()` method.

```
etree.ProcessingInstruction ( target, text=None ):
```

`target`

A string containing the target portion of the processing instruction.

`text`

An optional string containing the rest of the processing instruction. The default value is empty.

Here's an example:

```
pi = etree.ProcessingInstruction ( 'decor', 'danish,modern,ducksOnWall'
```

When converted back to XML, this processing instruction would look like this:

```
<?decor danish,modern,ducksOnWall?>
```

6.7. The QName () constructor

When you are working with multiple namespaces, the `QName` object is useful for combining the “namespace URI” part with the “local name” part. A `QName` instance can be used for the name part of attributes that are in a different namespace than their containing element.

Although it is not legal in XML element names, there is a convention called “Clark notation” (after James Clark) that combines these two parts in a string of this form:

```
{nsURI}local
```

To construct a new `QName` instance, use a statement of this general form:

```
etree.QName ( text, tag=None )
```

- If the fully qualified element name is already in Clark notation, call the `QName` constructor with this argument alone.
- If you would like to pass the namespace URI and the local name separately, call `QName` with the namespace URI as the `text` argument, and the local name as the `tag` argument.

Here are two examples for creating a `QName` instance representing a qualified name in the XSL namespace with a local name of `template`:

- In Clark notation:

```
qn = etree.QName ( "{http://www.w3.org/1999/XSL/Transform}template" )
```

- With the namespace URI and local name supplied separately:

```
qn = etree.QName ( "http://www.w3.org/1999/XSL/Transform", "template" )
```

6.8. The SubElement () constructor

This is a handy constructor that accomplishes the two basic operations in adding an element to a tree:

- creating a new `Element` instance, and
- adding that new `Element` as the next child of its parent element.

Here is the general form:

```
SubElement ( parent, tag, attrib={}, **extras ):
```

The first argument, `parent`, is the `Element` instance under which the newly created `Element` instance is to be added as its next child.

The `tag`, `attrib`, and `**extras` arguments work exactly the same as they do in the call to `Element ()` described in Section 6.2, “The `Element ()` constructor” (p. 7).

If you are working with multiple namespaces, you may also include an `nsmap` keyword argument defining your preferred namespace prefixes. For more on the `nsmap` argument, see Section 6.3, “The `ElementTree()` constructor” (p. 7).

Here's an example. Suppose you want to build this XML:

```
<state name="New Mexico">
  <county name="Socorro">
    <ppl name="Luis Lopez"/>
  </county>
</state>
```

Here's the code to build it, and then display it, interactively:

```
>>> st=etree.Element('state', name='New Mexico')
>>> co=etree.SubElement(st, 'county', name='Socorro')
>>> ppl=etree.SubElement(co, 'ppl', name='Luis Lopez')
>>> print etree.tostring(st)
<state name="New Mexico"><county name="Socorro"><ppl name="Luis Lopez"/>
</county></state>
>>>
```

6.9. The `tostring()` function: Serialize as XML

To convert an `Element` and its content back to XML, use a function call of this form:

```
etree.tostring ( elt, pretty_print=False )
```

where `elt` is an `Element` instance. The function returns a string containing the XML. For an example, see Section 6.8, “The `SubElement()` constructor” (p. 10).

If you set the optional `pretty_print` argument to `True`, the method will attempt to insert line breaks to keep line lengths short where possible.

6.10. The `XMLID()` function: Convert text to XML with a dictionary of `id` values

To convert XML in the form of a string into an `Element` structure, use Section 6.4, “The `fromstring()` function: Create an element from a string” (p. 9). However, there is a similar function named `etree.XMLID()` that does this and also provides a dictionary that allows you to find elements in a tree by their unique `id` attribute values.

The XML standard stipulates that any element in any document can have an `id` attribute, but each value of this attribute must be unique within the document. The intent of this feature is that applications can refer to any element using its `id` value.

Here is the general form for this function:

```
etree.XMLID ( text )
```

The return value is a tuple (E, D) , where:

- E is the converted XML as an `Element` instance rooting the converted tree, just as if you had called `etree.fromstring(text)`.

- *D* is a dictionary whose keys are the values of `id` attributes in the converted tree, and each corresponding value is the `Element` instance that carried that `id` value.

Here's an example:

```
>>> SOURCE = '''<dog id="Fido">
...   Woof!
...   <cat id="Fluff">
...     Mao?
...   </cat>
...   <rhino id="ZR"/>
... </dog>'''
>>> tree, idMap = etree.XMLID(SOURCE)
>>> idList = idMap.keys()
>>> idList.sort()
>>> for id in idList:
...     elt = idMap[id]
...     if elt.text is None:
...         display = "None"
...     else:
...         display = elt.text.strip()
...     print "Tag %s, text is '%s'" % (elt.tag, display)
Tag dog, text is 'Woof!'
Tag cat, text is 'Mao?'
Tag rhino, text is 'None'
>>>
```

7. class ElementTree: A complete XML document

Once you have used the `etree.ElementTree` constructor to instantiate an XML document, you can use these attributes or methods on that instance.

7.1. ElementTree.find()

This method searches the tree for matching elements, and returns the first one that matches, or `None` if there are no matches. For an instance *ET* of an `ElementTree`:

```
ET.find ( path )
```

The `path` argument is a string describing the element for which you are searching. Possible values include:

"tag"

Find the first child element whose name is `"tag"`.

"tag₁/tag₂/.../tag_n"

Find the first child element whose name is `tag1`; then, under that child element, find its first child named `tag2`; and so forth.

For example, suppose you have an `ElementTree` instance named `page`, containing an XHTML page. Further suppose you want to find the `link` element inside the `head` element inside the root `html` element. This statement would set the variable `linkElt` to that `link` element:

```
linkElt = page.find ( "html/head/link" )
```

7.2. ElementTree.findall(): Find matching elements

This method searches the document in an ElementTree and returns a sequence of all the matching elements.

In general, for some ElementTree instance *ET*, this method will return a sequence of zero or more Elements:

```
ET.findall ( path )
```

The syntax of the `path` argument is the same as for the `path` argument in Section 7.1, “ElementTree.find()” (p. 12). The only difference is that this method returns a sequence containing *all* the elements that match `path`.

7.3. ElementTree.findtext(): Retrieve the text content from an element

To retrieve the text inside some element, use this method on some ElementTree instance *ET*:

```
ET.findtext ( path, default=None ):
```

path

The syntax for describing the location of the desired element is the same as in Section 7.1, “ElementTree.find()” (p. 12).

default

The value to be returned if there is no element at that path.

If there is a child at the indicated path, the method returns its text content as a string. If there is a matching child, but it has no content, the return value is "" (the empty string). If there is no such child, the method returns `None`.

7.4. ElementTree.getiterator(): Make an iterator

In many applications, you will want to visit every element in a document, or perhaps to retrieve information from all the tags of a certain kind. This method, on some ElementTree instance *ET*, will return an iterator that visits all matching tags.

```
ET.getiterator ( tag=None )
```

If you omit the argument, you will get an iterator that generates every element in the tree, in document order.

If you want to visit only tags with a certain name, pass that name as the argument.

Here are some examples. In these examples, assume that `page` is an ElementTree instance that contains an XHTML page. The first example would print every tag name in the page, in document order.

```
for elt in page.getiterator():  
    print elt.tag
```

The second example would look at every `div` element in the page, and for those that have a `class` attribute, it prints those attributes.

```
for elt in page.getiterator ( 'div' ):
    if elt.attrib.has_key ( 'class' ):
        print elt.get('class')
```

7.5. `ElementTree.getroot()`: Find the root element

To obtain the root element of a document contained in an `ElementTree` instance `ET`, use this method call:

```
ET.getroot()
```

The return value will normally be the `Element` instance at the root of the tree. However, if you have created your `ElementTree` instance without specifying either a root element or an input file, this method will return `None`.

7.6. `ElementTree.xpath()`: Evaluate an *XPath* expression

For an `ElementTree` instance `ET`, use this method call to evaluate an *XPath* expression `s`, using the tree's root element as the context node.

```
ET.xpath ( s )
```

This methods returns the result of the *XPath* expression. For a general discussion of *XPath*, see Section 9, “*XPath* processing” (p. 23).

7.7. `ElementTree.write()`: Translate back to XML

To serialize (convert to XML) the content of a document contained in some `ElementTree` instance `ET`, use this method call:

```
ET.write ( file, pretty_print=False )
```

You must supply a writeable file object, or the name of a file to be written. If you set argument `pretty_print=True`, the method will attempt to fold long lines and indent the XML for legibility.

For example, if you have an `ElementTree` instance in a variable `page` containing an XHTML page, and you want to write it to the standard output stream, this statement would do it:

```
import sys
page.write ( sys.stdout )
```

8. `class Element`: One element in the tree

Each XML element is represented by an instance of the `Element` class.

- See Section 8.1, “Attributes of an `Element` instance” (p. 15) for attributes of an `Element` instance in the *Python* sense, as opposed to XML attributes.
- See Section 8.2, “Accessing the list of child elements” (p. 15) for the various ways to access the element children of an element.

- The various methods on `Element` instances follow in alphabetical order, starting with Section 8.3, “`Element.append()`: Add a new element child” (p. 16).

8.1. Attributes of an `Element` instance

Each instance of the `Element` class has these attributes.

.tag

The element's name.

.attrib

A dictionary containing the element's attributes. The keys are the attribute names, and each corresponding value is the attribute's value.

.tail

The text following this element's closing tag, up to the start tag of the next sibling element. If there was no text there, this attribute will have the value `None`.

This way of associating text with elements is not really typical of the way most XML processing models work; see Section 2, “How `ElementTree` represents XML” (p. 3).

.text

The text inside the element, up to the start tag of the first child element. If there was no text there, this attribute will have the value `None`.

8.2. Accessing the list of child elements

In many ways, an `Element` instance acts like a Python list, with its XML child elements acting as the members of that list.

You can use the Python `len()` function to determine how many children an element has. For example, if `node` is an `Element` instance with five element children, `len(node)` will return the value 5.

You can add, replace, or delete children of an element using regular Python list operations. For example, if an `Element` instance `node` has three child elements, `node[0]` is the first child, and `node[2]` is the third child.

In the examples that follow, assume that `E` is an `Element` instance.

- `E[i]` returns the child element of `E` at position `i`, if there is one. If there is no child element at that position, this operation raises an `IndexError` exception.
- `E[i:j]` returns a list of the child elements between positions `i` and `j`.

For example, `node[2:4]` returns a list containing the third and fourth children of `node`.

- You can replace one child of an element `E` with a new element `c` using a statement of this form:

```
E[i] = c
```

If `i` is not the position of an existing child, this operation will raise an `IndexError`.

- You can replace a sequence of adjacent children of an element `E` using slice assignment:

```
E[i:j] = seq
```

where `seq` is a sequence of `Element` instances.

If the slice `[i:j]` does not specify an existing set of children, this operation will raise an `IndexError` exception.

- You can delete one child of an element like this:

```
del E[i]
```

where *i* is the index of that child.

- You can delete a slice out of the list of element children like this:

```
del E[i:j]
```

8.3. `Element.append()`: Add a new element child

To add a new child *c* to an element *E*, use this method:

```
E.append ( c )
```

You can use this method to add `Comment` and `ProcessingInstruction` instances as children of an element, as well as `Element` instances.

Here is a conversational example:

```
>>> st = etree.Element ( "state", name="New Mexico" )
>>> etree.tostring(st)
'<state name="New Mexico"/>'
>>> co = etree.Element ( "county", name="Socorro" )
>>> st.append(co)
>>> etree.tostring(st)
'<state name="New Mexico"><county name="Socorro"/></state>'
>>> rem = etree.Comment ( "Just another day in paradise." )
>>> st.append ( rem )
>>> etree.tostring(st)
'<state name="New Mexico"><county name="Socorro"/><!-- Just another day in
paradise. --></state>'
>>>
```

8.4. `Element.clear()`: Make an element empty

Calling the `.clear()` method on an `Element` instance removes all its content:

- All values are removed from the `.attrib` dictionary.
- The `.text` and `.tail` attributes are both set to `None`.
- Any child elements are deleted.

8.5. `Element.find()`: Find a matching sub-element

You can search for sub-elements of an `Element` instance *E* using this method call:

```
E.find ( path )
```

The possible values of the *path* argument use the same syntax as the *path* argument in Section 7.1, “ElementTree.find()” (p. 12). However, the search starts at element *E*, rather than at the top of the document.

If there are any matching elements, this method returns the matching element that is first in document order. If there are no matching elements, the method returns `None`.

For example, if *node* is an `Element` instance that has an element child with a tag “county”, and that child in turn has an element child with tag “ppl”, this expression will return the `Element` corresponding to the “ppl” child:

```
node.find ( "county/ppl" )
```

8.6. Element.findall(): Find all matching sub-elements

The `.findall()` method on an `Element` instance is very similar to Section 7.2, “ElementTree.findall(): Find matching elements” (p. 13): it uses the same *path* description, and returns a sequence containing all the matching elements. However, the search starts at the given `Element` instead of starting at the top of the document. Syntax:

```
E.findall ( path )
```

8.7. Element.findtext(): Extract text content

Every `Element` instance *E* has a `.findtext()` method that works very similarly to Section 7.3, “ElementTree.findtext(): Retrieve the text content from an element” (p. 13). Here is the general form:

```
E.findtext ( path, default=None )
```

If any child elements of *E* exist that match the given *path*, this method returns the text content of the first matching element. If the element exists but has no text content, the method returns “”, the empty string. If there are no matching child elements, the method returns `None`.

8.8. Element.get(): Retrieve an attribute value with defaulting

There are two ways you can try to get an attribute value from an `Element` instance. See also the `.attrib` dictionary in Section 8.1, “Attributes of an `Element` instance” (p. 15).

The `.get()` method on an `Element` instance also attempts to retrieve an attribute value. The advantage of this method is that you can provide a default value that is returned if the element in question does not actually have an attribute by the given name.

Here is the general form, for some `Element` instance *E*.

```
E.get ( key, default=None )
```

The *key* argument is the name of the attribute whose value you want to retrieve.

- If *E* has an attribute by that name, the method returns that attribute's value as a string.
- If *E* has no such attribute, the method returns the `default` argument, which itself has a default value of `None`.

Here's an example:

```

>>> from lxml import etree
>>> node = etree.fromstring('<mount species="Jackalope"/>')
>>> print node.get('species')
Jackalope
>>> print node.get('source')
None
>>> print node.get('source', 'Unknown')
Unknown
>>>

```

8.9. Element.getChildren(): Get element children

For an `Element` instance `E`, this method returns a list of all `E`'s element children:

```
E.getChildren()
```

Here's an example:

```

>>> xml = '''<corral><horse n="2"/><cow n="17"/>
... <cowboy n="2"/></corral>'''
>>> pen = etree.fromstring ( xml )
>>> penContents = pen.getChildren()
>>> for content in penContents:
...     print "%-10s %3s" % (content.tag, content.get("n", "0"))
...
horse          2
cow            17
cowboy         2
>>>

```

8.10. Element.getiterator(): Make an iterator to walk a subtree

Sometimes you want to walk through all or part of a document, looking at all the elements in document order. Similarly, you may want to walk through all or part of a document and look for all the occurrences of a specific kind of element.

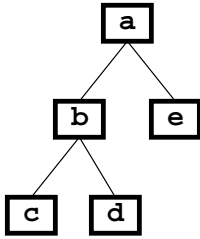
The `.getiterator()` method on an `Element` instance produces a Python iterator that tells Python how to visit elements in these ways. Here is the general form, for an `Element` instance `E`:

```
E.getiterator ( tag=None )
```

- If you omit the argument, you will get an iterator that visits `E` first, then all its element children and their children, in a *preorder traversal* of that subtree.
- If you want to visit only elements with a certain tag name, pass the desired tag name as the argument.

Preorder traversal of a tree means that we visit the root first, then the subtrees from left to right (that is, in document order). This is also called a **depth-first** traversal: we visit the root, then its first child, then its first child's first child, and so on until we run out of descendants. Then we move back up to the last element with more children, and repeat.

Here is an example showing the traversal of an entire tree. First, a diagram showing the tree structure:



A preorder traversal of this tree goes in this order: a, b, c, d, e.

```

>>> xml = '''<a><b><c/><d/></b><e/></a>'''
>>> tree = etree.fromstring ( xml )
>>> walkAll = tree.getiterator()
>>> for elt in walkAll:
...     print elt.tag,
...
a b c d e
>>>
  
```

In this example, we visit only the bird nodes.

```

>>> xml = '''<bio>
... <bird type="Bushtit"/>
... <butterfly type="Mourning Cloak"/>
... <bird type="Mew Gull"/>
... <group site="Water Canyon">
...   <snake type="Sidewinder"/>
...   <bird type="Verdin"/>
... </group>
... <bird type="Pygmy Nuthatch"/>
... </bio>'''
>>> root = etree.fromstring(xml)
>>> for elt in root.getiterator('bird'):
...     print elt.get('type', 'Unknown')
...
Bushtit
Mew Gull
Verdin
Pygmy Nuthatch
>>>
  
```

Note in the above example that the iterator visits the Verdin element even though it is not a direct child of the root element.

8.11. Element.getroottree(): Find the ElementTree containing this element

For any Element instance *E*, this method call returns the ElementTree instance that contains *E*:

```
E.getroottree()
```

8.12. Element.insert(): Insert a new child element

Use the `.insert()` method on an `Element` instance `E` to add a new element child `elt` in an arbitrary position. (To append a new element child at the last position, see Section 8.3, “`Element.append()`: Add a new element child” (p. 16).)

```
E.insert ( index, elt )
```

The `index` argument specifies the position into which element `elt` is inserted. For example, if you specify `index 0`, the new child will be inserted before any other children of `E`.

The `lxml` module is quite permissive about the values of the `index` argument: if it is negative, or greater than the position of the last existing child, the new child is added after all existing children.

Here is an example showing insertions at positions 0 and 2.

```
>>> node = etree.fromstring('<a><c0/><c1/><c2/></a>')
>>> newKid = etree.Element('c-1', laugh="Hi!")
>>> node.insert(0, newKid)
>>> etree.tostring(node)
'<a><c-1 laugh="Hi!"/><c0/><c1/><c2/></a>'
>>> newerKid = etree.Element('cn')
>>> node.insert(2, newerKid)
>>> etree.tostring(node)
'<a><c-1 laugh="Hi!"/><c0/><cn/><c1/><c2/></a>'
>>>
```

8.13. Element.items(): Produce attribute names and values

For any `Element` instance `E`, the `.items()` method returns the attributes as if they were a dictionary and you had called the `.items()` method on that dictionary: the result is a list of two-element tuples (`name`, `value`), one for each XML attribute of `E`.

Attribute values are returned in no particular order.

Here's an example.

```
>>> node = etree.fromstring ( "<event time='1830' cost='3.50'
rating='nc-03'/>" )
>>> node.items()
[('cost', '3.50'), ('time', '1830'), ('rating', 'nc-03')]
>>>
```

8.14. Element.iterancestors(): Find an element's ancestors

The *ancestors* of an element are its parent, its parent's parent, and so on up to the root element of the tree. For any `Element` instance `E`, this method returns an iterator that visits `E`'s ancestors in reverse document order:

```
E.iterancestors(tag=None)
```

If you omit the argument, the iterator will visit all ancestors. If you wish to visit only ancestors with a specific tag name, pass that tag name as an argument.

Examples:

```

>>> xml = '''<class sci='Aves' eng='Birds'>
...   <order sci='Strigiformes' eng='Owls'>
...     <family sci='Tytonidae' eng='Barn-Owls'>
...       <genus sci='Tyto'>
...         <species sci='Tyto alba' eng='Barn Owl' />
...       </genus>
...     </family>
...   </order>
... </class>'''
>>> root = etree.fromstring ( xml )
>>> barney = root.xpath ( '//species' ) [0]
>>> print "%s: %s" % (barney.get('sci'), barney.get('eng'))
Tyto alba: Barn Owl
>>> for ancestor in barney.iterancestors():
...     print ancestor.tag,
genus family order class
>>> for fam in barney.iterancestors('family'):
...     print "%s: %s" % (fam.get('sci'), fam.get('eng'))
Tytonidae: Barn-Owls

```

8.15. Element.iterchildren(): Find all children

For an Element instance *E*, this method returns an iterator that iterates over all of *E*'s children.

```
E.iterchildren(reversed=False, tag=None)
```

Normally, the resulting iterator will visit the children in document order. However, if you pass `reversed=True`, it will visit them in the opposite order.

If you want the iterator to visit only children with a specific name *N*, pass an argument `tag=N`.

Example:

```

>>> root=et.fromstring("<mom><aaron/><betty/><clarence/><dana/></mom>")
>>> for kid in root.getchildren():
...     print kid.tag
aaron
betty
clarence
dana
>>> for kid in root.iterchildren(reversed=True):
...     print kid.tag
...
dana
clarence
betty
aaron
>>>

```

8.16. Element.iterdescendants(): Find all descendants

The term *descendants* refers to an element's children, their children, and so on all the way to the leaves of the document tree.

For an `Element` instance E , this method returns an iterator that visits all of E 's descendants in document order.

```
E.iterdescendants(tag=None)
```

If you want the iterator to visit only elements with a specific tag name N , pass an argument `tag=N`.

Example:

```
>>> xml = '''<root>
...   <grandpa>
...     <dad>
...       <yuo/>
...     </dad>
...   </grandpa>
... </root>'''
>>> root = etree.fromstring(xml)
>>> you = root.xpath('./yuo')[0]
>>> for anc in you.iterancestors():
...     print anc.tag,
dad grandpa root
>>>
```

8.17. `Element.itersiblings()`: Find other children of the same parent

For any `Element` instance E , this method returns an iterator that visits all of E 's siblings, that is, the element children of its parent, in document order, but omitting E .

```
E.itersiblings(preceding=False)
```

If the `preceding` argument is false, the iterator will visit the siblings following E in document order. If you pass `preceding=True`, the iterator will visit the siblings that precede E in document order.

Example:

```
>>> root=etree.fromstring(
...   "<mom><aaron/><betty/><clarence/><dana/></mom>")
>>> betty=root.find('betty')
>>> for sib in betty.itersiblings(preceding=True):
...     print sib.tag
...
aaron
>>> for sib in betty.itersiblings():
...     print sib.tag
...
clarence
dana
>>>
```

8.18. `Element.keys()`: Find all attribute names

For any `Element` instance E , this method returns a list of the element's XML attribute names, in no particular order.

```
E.keys()
```

Here's an example:

```
>>> node = etree.fromstring ( "<event time='1830' cost='3.50'
rating='nc-03'/>" )
>>> node.keys()
['time', 'rating', 'cost']
>>>
```

8.19. Element.remove(): Remove a child element

To remove an element child *C* from an `Element` instance *E*, use this method call.

```
E.remove(C)
```

If *C* is not a child of *E*, this method will raise a `ValueError` exception.

8.20. Element.set(): Set an attribute value

To create or change an attribute named *A* to value *V* in an `Element` instance *E*, use this method:

```
E.set ( A, V )
```

Here's an example.

```
>>> node = etree.Element ( 'div', id='u401' )
>>> etree.tostring(node)
'<div id="u401"/>'
>>> node.set ( 'class', 'flyer' )
>>> etree.tostring(node)
'<div id="u401" class="flyer"/>'
>>> node.set ( 'class', 'broadside' )
>>> etree.tostring ( node )
'<div id="u401" class="broadside"/>'
>>>
```

This method is one of two ways to create or change an attribute value. The other method is to store values into the `.attrib` dictionary of the `Element` instance.

8.21. Element.xpath(): Evaluate an XPath expression

To evaluate an *XPath* expression *s* using some `Element` instance *E* as the context node:

```
E.xpath ( s )
```

For a general discussion of the use of *XPath*, see Section 9, “*XPath* processing” (p. 23).

9. XPath processing

One of the most significant advantages of the `Lxml` package over the other `ElementTree`-style packages is its support for the full *XPath* language. *XPath* expressions give you a much more powerful mechanism

for selecting and retrieving parts of a document, compared to the relatively simple “path” syntax used in Section 7.1, “`ElementTree.find()`” (p. 12).

If you are not familiar with *XPath*, see these sources:

- *XSLT reference*⁵, specifically the section entitled “XPath reference”⁶.
- The standard, *XML Path Language (XPath), Version 1.0*⁷.

Keep in mind that every *XPath* expression is evaluated using three items of context:

- The *context node* is the starting point for any operations whose meaning is relative to some point in the tree.
- The *context size* is the number of elements that are children of the context node's parent, that is, the context node and all its siblings.
- The *context position* is the context node's position relative to its siblings, counting the first sibling as position 1.

You can evaluate an *XPath* expression *s* by using the `.xpath(s)` method on either an `Element` instance or an `ElementTree` instance. See Section 8.21, “`Element.xpath()`: Evaluate an *XPath* expression” (p. 23) and Section 7.6, “`ElementTree.xpath()`: Evaluate an *XPath* expression” (p. 14).

Depending on the *XPath* expression you use, these `.xpath()` methods may return one of several kinds of values:

- For expressions that return a Boolean value, the `.xpath()` method will return `True` or `False`.
- Expressions with a numeric result will return a Python `float` (*never* an `int`).
- Expressions with a string result will return a Python `str` (string) or `unicode` value.
- Expressions that produce a list of values, such as node-sets, will return a Python `list`. Elements of this list may in turn be any of several types:
 - Elements, comments, and processing instructions will be represented as `lxml Element`, `Comment`, and `ProcessingInstruction` instances.
 - Text content and attribute values are returned as Python `str` (string) instances.
 - Namespace declarations are returned as a two-tuple (*prefix*, *namespaceURI*).

For further information on `lxml`'s *XPath* features, see *XPath and XSLT with lxml*⁸.

9.1. An *XPath* example

Here is an example of a situation where an *XPath* expression can save you a lot of work. Suppose you have a document with an element called `para` that represents a paragraph of text. Further suppose that your `para` has a mixed-content model, so its content is a free mixture of text and several kinds of inline markup. Your application, however, needs to extract just the text in the paragraph, discarding any and all tags.

Using just the classic `ElementTree` interface, this would require you to write some kind of function that recursively walks the `para` element and its subtree, extracting the `.text` and `.tail` attributes at each level and eventually gluing them all together.

However, there is a relatively simple *XPath* expression that does all this for you:

```
descendant-or-self::text()
```

⁵ <http://www.nmt.edu/tcc/help/pubs/xslt/>

⁶ <http://www.nmt.edu/tcc/help/pubs/xslt/xpath-sect.html>

⁷ <http://www.w3.org/TR/xpath>

⁸ <http://codespeak.net/lxml/xpathxslt.html>

The “`descendant-or-self::`” is an axis selector that limits the search to the context node, its children, their children, and so on out to the leaves of the tree. The “`text()`” function selects only text nodes, discarding any elements, comments, and other non-textual content. The return value is a list of strings.

Here's an example of this expression in practice.

```
>>> node=etree.fromstring(''<a>
...   a-text <b>b-text</b> b-tail <c>c-text</c> c-tail
... </a>'')
>>> alltext = node.xpath ( 'descendant-or-self::text()' )
>>> alltext
['\n a-text ', 'b-text', ' b-tail ', 'c-text', ' c-tail\n']
>>> clump = "".join(alltext)
>>> clump
'\n a-text b-text b-tail c-text c-tail\n'
>>>
```

10. Automated validation of input files

What happens to your application if you read a file that does not conform to the schema? There are two ways to deal with error handling.

- If you are a careful and defensive programmer, you will always check for the presence and validity of every part of the XML document you are reading, and issue an appropriate error message. If you aren't careful or defensive enough, your application may crash.
- It can make your application a lot simpler if you mechanically validate the input file against the schema that defines its document type.

With the `lxml` module, the latter approach is inexpensive both in programming effort and in runtime. You can validate a document using either of these major schema languages:

- Section 10.1, “Validation with a Relax NG schema” (p. 25).
- Section 10.2, “Validation with an XSchema (XSD) schema” (p. 27).

10.1. Validation with a Relax NG schema

The `lxml` module can validate a document, in the form of an `ElementTree`, against a schema expressed in the Relax NG notation. For more information about Relax NG, see *Relax NG Compact Syntax (RNC)*⁹.

A Relax NG schema can use two forms: the compact syntax (RNC), or an XML document type (RNG). If your schema uses RNC, you must translate it to RNG format. The *trang* utility does this conversion for you. Use a command of this form:

```
trang file.rnc file.rng
```

Once you have the schema available as an `.rng` file, use these steps to valid an element tree *ET*.

1. Parse the `.rng` file into its own `ElementTree`, as described in Section 6.3, “The `ElementTree()` constructor” (p. 7).
2. Use the constructor `etree.RelaxNG(S)` to convert that tree into a “schema instance,” where *S* is the `ElementTree` instance, containing the schema, from the previous step.

⁹ <http://www.nmt.edu/tcc/help/pubs/rnc/>

If the tree is not a valid Relax NG schema, the constructor will raise an `etree.RelaxNGParseError` exception.

3. Use the `.validate(ET)` method of the schema instance to validate `ET`.

This method returns `1` if `ET` validates against the schema, or `0` if it does not.

If the method returns `0`, the schema instance has an attribute named `.error_log` containing all the errors detected by the schema instance. You can print `.error_log.last_error` to see the most recent error detected.

Here is a small program that takes as command line arguments an RNG schema file name followed by one or more XML file names. It validates each XML file in turn against that schema.

```
#!/usr/bin/env python
#=====
# valrelax: Validate files against Relax NG
#-----
# Command line arguments:
#   valrelax SCHEMA.rng file1.xml file2.xml ...
#-----

import sys
from lxml import etree

def main():
    # [ schemaFile := first command line argument
    #   fileList := remaining command line arguments ]
    schemaFile = sys.argv[1]
    fileList = sys.argv[2:]

    # [ schema := an etree.RelaxNG instance that represents
    #       schemaFile ]
    schemaDoc = etree.parse ( schemaFile )
    try:
        schema = etree.RelaxNG ( schemaDoc )
    except etree.RelaxNGParseError, details:
        print >>sys.stderr, ( "*** %s is not a Relax NG schema: %s" %
                              (schemaFile, details) )
        raise SystemExit

    # [ sys.stdout += messages about files in fileList
    # [ sys.stderr += messages about files in fileList that
    #           don't validate against schema ]
    for fileName in fileList:
        print "=== Validating", fileName
        validate ( schema, fileName )

def validate ( schema, fileName ):
    """Validate one file against the schema.

    [ (schema is an etree.RelaxNG instance) and
      (fileName is a string) ->
      if fileName names a readable, well-formed XML file that
      validates against schema ->
```

```

        sys.stdout += success report
    else ->
        sys.stdout += failure report ]
"""

# [ if fileName names a readable, well-formed XML file ->
#   doc := an et.ElementTree representing that file
#   else ->
#     sys.stdout += failure report
#   return ]
try:
    doc = etree.ElementTree ( file=fileName )
except etree.XMLSyntaxError, detail:
    print "*** Not well-formed: %s" % detail
    return
except IOError, detail:
    print "*** I/O error reading '%s': %s" % (fileName, detail)
    return

# [ if doc is valid by schema ->
#   I
#   else ->
#     sys.stdout += failure report ]
result = schema.validate ( doc )
if not result:
    print "*** %s not valid: %s" % (fileName, schema.error_log)

#=====
if __name__ == '__main__':
    main()

```

10.2. Validation with an XSchema (XSD) schema

To validate a document against a schema written in the XSchema language, follow the steps shown in Section 10.1, “Validation with a Relax NG schema” (p. 25), with one variation.

Instead of using `etree.RelaxNG()` to parse your schema tree *S*, use `etree.XMLSchema(S)`.

11. etbuilder: A simplified XML builder module

If you are building a lot of XML, it can be somewhat cumbersome to take several lines of code to build a single element. For elements with text content, you'll write a lot of two-line sequences like this:

```

mainTitle = et.Element('h1')
mainTitle.text = "Welcome to Your Title Here!"

```

The brilliant and productive Fredrik Lundh has written a very nice module called `builder.py` that makes building XML a lot easier.

- See Lundh's original page, *An ElementTree Builder*¹⁰, for an older version of his module, with documentation and examples.

¹⁰ <http://effbot.org/zone/element-builder.htm>

- You may wish to use the current version of `builder.py` from Lundh's SVN repository page¹¹.
- The author has written a modified version based heavily on Lundh's version. The source for this `et-builder` module is available online¹².

For the instructions for use of the author's version, see Section 11.1, “Using the `etbuilder` module” (p. 28).

For the actual implementation in lightweight literate programming form¹³, see Section 12, “Implementation of `etbuilder`” (p. 29).

11.1. Using the `etbuilder` module

Instead of importing the `ElementTree` package as `et`, use this importation:

```
from etbuilder import et, E
```

The name `E` is a factory object that creates `et.Element` instances.

Here is the calling sequence for `E`:

```
E(tag, *p, **kw)
```

The first argument, `tag`, is the element's name as a string. The return value is a new `et.Element` instance.

You can supply any number of positional arguments `p`, followed by any number of keyword arguments. The interpretation of each argument depends on its type. The displays with “>>>” prompts are interactive examples.

- Any keyword argument of the form “`name=value`” becomes an XML attribute “`name='value'`” of the new element.

```
>>> colElt=E('col', valign='top', align='left')
>>> et.tostring(colElt)
'<col align="left" valign="top" />'
```

- String arguments are added to the content of the tag.

```
>>> p14 = E("p", "Welcome to ", "Your Paragraph Here.")
>>> et.tostring(p14)
'<p>Welcome to Your Paragraph Here.</p>'
```

- An argument of type `int` is converted to a string and added to the tag's content.
- If you pass a dictionary to the factory, its members also become XML attributes. For instance, you might create an XHTML table cell element like this:

```
>>> cell = E('td', {'valign': 'top', 'align': 'right'}, 14)
>>> et.tostring(cell)
'<td align="right" valign="top">14</td>'
```

¹¹ <http://svn.effbot.org/public/stuff/sandbox/elementlib/>

¹² <http://www.nmt.edu/tcc/help/pubs/pylxml/etbuilder.py>

¹³ <http://www.nmt.edu/~shipman/soft/litprog/>

- You can pass in an `et.Element` instance, and it becomes a child element of the element being built. This allows you to nest calls within calls, like this:

```
>>> head = E('head',
...     E('title', 'Your Page Title Here'),
...     E('link', rel='stylesheet', href='/tcc/style.css'))
>>> print et.tostring(head, pretty_print=True)
<head>
  <title>Your Page Title Here</title>
  <link href="/tcc/style.css" rel="stylesheet" />
</head>
```

This module has one more nice wrinkle. If the name of the tag you are creating is also a valid Python name, you can use that name as the name of a method call on the `E` instance. That is,

```
E.name(...)
```

is functionally equivalent to

```
E("name", ...)
```

Here is an example:

```
>>> head = E.head(
...     E.title('Your title'),
...     E.link(rel='stylesheet', href='/tcc/style.css'))
>>> print et.tostring(head, pretty_print=True)
<head>
  <title>Your title</title>
  <link href="/tcc/style.css" rel="stylesheet" />
</head>
```

12. Implementation of `etbuilder`

Here is the author's `etbuilder.py` module, with narrative.

12.1. Features differing from Lundh's original

The author's version differs from Lundh's version in these respects:

- It requires the `lxml` package. Lundh's version did not use `lxml`; it uses `cElementTree`, or `element-tree` if that is not available.
- It requires Python 2.5 or later. Lundh's version will work with earlier versions, probably back to at least 2.2.
- The author's version also permits `int` values in the call to the `E` instance.

12.2. Prologue

The module begins with a comment pointing back to this documentation, and acknowledging Fredrik Lundh's work.

```

"""etbuilder.py: An element builder for lxml.etree
=====
$Revision: 1.28 $ $Date: 2009/01/23 03:50:52 $
=====
For documentation, see:
  http://www.nmt.edu/tcc/help/pubs/pylxml/
Borrows heavily from the work of Fredrik Lundh; see:
  http://effbot.org/
"""

```

The `et` module is `lxml.etree`.

```

#=====
# Imports
#-----

from lxml import etree as et

```

The `functools.partial()` function¹⁴ is used to curry a function call in Section 12.7, “Element-Maker.__getattr__(): Handle arbitrary method calls” (p. 36).

However, the `functools` module is new in Python 2.5. In order to make this module work in a Python 2.4 install, we will anticipate a possible failure to import `functools`, providing that functionality with a substitute `partial()` function. This function is stolen directly from the *Python Library Reference*¹⁵.

```

try:
    from functools import partial
except ImportError:
    def partial(func, *args, **keywords):
        def newfunc(*fargs, **fkeywords):
            newkeywords = keywords.copy()
            newkeywords.update(fkeywords)
            return func(*(args + fargs), **newkeywords)
        newfunc.func = func
        newfunc.args = args
        newfunc.keywords = keywords
        return newfunc

```

12.3. class ElementMaker: The factory class

The name `E` that the user imports is not a class. It is a factory object, that is, an instance of the `ElementMaker` factory class.

```

# - - - - - c l a s s   E l e m e n t M a k e r

class ElementMaker(object):
    '''ElementTree element factory class

    Exports:

```

¹⁴ <http://www.python.org/doc/2.5.2/lib/module-functools.html>

¹⁵ <http://docs.python.org/library/functools.html>

```

ElementMaker ( typeMap=None ):
    [ (typeMap is an optional dictionary whose keys are
      type objects T, and each corresponding value is a
      function with calling sequence
        f(elt, item)
      and generic intended function
        [ (elt is an et.Element) and
          (item has type T) ->
            elt := elt with item added ] ) ->
      return a new ElementMaker instance that has
      calling sequence
        E(*p, **kw)
      and intended function
        [ p[0] exists and is a str ->
          return a new et.Element instance whose name
          is p[0], and remaining elements of p become
          string content of that element (for types
          str, unicode, and int) or attributes (for
          type dict, and members of kw) or children
          (for type et.Element), plus additional
          handling from typeMap if it is provided ]
      and allows arbitrary method calls of the form
        E.tag(*p, **kw)
      with intended function
        [ return a new et.Element instance whose name
          is (tag), and elements of p and kw have
          the same effects as E(*(p[1:]), **kw) ]
    ...

```

For a discussion of intended functions and the Cleanroom software development methodology, see the author's Cleanroom page¹⁶.

You can use the optional `typeMap` argument to provide logic to handle types other than the ones defined in Section 11.1, "Using the `etbuilder` module" (p. 28). Refer to the constructor for a discussion of the internal state item `._typeMap` and how it works in element construction.

12.4. ElementMaker. `__init__`(): Constructor

The factory instance returned by the `ElementMaker` constructor must look at the type of each of its positional arguments in order to know what to do with it. Python's dictionary type makes this easy to do: we use a dictionary whose keys are Python type objects. Each of the corresponding values in this dictionary is a function that can be called to process arguments of that type.

The dictionary is a private attribute `._typeMap`, and all the constructor does is set this dictionary up.

The functions that process arguments all have this generic calling sequence:

```
f(elt, item)
```

where `elt` is the `et.Element` being built, and `item` is the argument to be processed.

The first step is to initialize the `._typeMap` dictionary. In most cases, the user will be satisfied with the type set described in Section 11.1, "Using the `etbuilder` module" (p. 28). However, as a conveni-

¹⁶ <http://www.nmt.edu/~shipman/soft/clean/>

ence, Lundh's original `builder.py` design allows the caller to supply a dictionary of additional type-function pairs as an optional argument; in that case, we will copy the supplied dictionary as the initial value of `self.__typeMap`.

`etbuilder.py`

```
# - - -   E l e m e n t M a k e r .   _ _   i n i t   _ _

def __init__ ( self, typeMap=None ):
    '''Constructor for the ElementMaker factory class.
    ...
    #-- 1 --
    # [ if typeMap is None ->
    #     self.__typeMap := a new, empty dictionary
    # else ->
    #     self.__typeMap := a copy of typeMap ]
    if typeMap is None:
        self.__typeMap = {}
    else:
        self.__typeMap = typeMap.copy()
```

The first types we'll need to handle are the `str` and `unicode` types. These types will use a function we define locally named `addText()`. Adding text to an element in the `ElementTree` world has two cases. If the element has no children, the text is added to the element's `.text` attribute. If the element has any children, the new text is added to the last child's `.tail` attribute. See Section 2, "How `ElementTree` represents XML" (p. 3) for a review of text handling.

`etbuilder.py`

```
#-- 2 --
# [ self.__typeMap[str], self.__typeMap[unicode] :=
#     a function with calling sequence
#     addText(elt, item)
#     and intended function
#     [ (elt is an et.Element) and
#       (item is a str or unicode instance) ->
#         if elt has no children and elt.text is None ->
#           elt.text := item
#         else if elt has no children ->
#           elt.text += item
#         else if elt's last child has .text==None ->
#           that child's .text := item
#         else ->
#           that child's .text += item ]
def addText ( elt, item ):
    if len(elt):
        elt[-1].tail = (elt[-1].tail or "") + item
    else:
        elt.text = (elt.text or "") + item
    self.__typeMap[str] = self.__typeMap[unicode] = addText
```

Lundh's original module did not handle arguments of type `int`, but this ability is handy for many common tags, such as "`<table border='8'>`", which becomes "`E.table(border=8)`".

A little deviousness is required here. The `addInt()` function can't call the `addText()` function above directly, because the name `addText` is bound to that function only inside the constructor. The instance does not know that name. However, we can assume that `self.__typeMap[str]` is bound to that function, so we call it from there.

```

#-- 3 --
# [ self.__typeMap[str], self.__typeMap[unicode] :=
#   a function with calling sequence
#   addInt(elt, item)
#   and intended function
#   [ (elt is an et.Element) and
#     (item is an int instance) ->
#       if elt has no children and elt.text is None ->
#         elt.text := str(item)
#       else if elt has no children ->
#         elt.text += str(item)
#       else if elt's last child has .text==None ->
#         that child's .text := str(item)
#       else ->
#         that child's .text += str(item) ]
def addInt ( elt, item ):
    self.__typeMap[str](elt, str(item))
self.__typeMap[int] = addInt

```

The next type we need to handle is dict. Each key-value pair from the dictionary becomes an XML attribute. For user convenience, if the value is not a string, we'll use the `str()` function on it, allowing constructs like “E({border: 1})”.

```

#-- 4 --
# [ self.__typeMap[dict] := a function with calling
#   sequence
#   addDict(elt, item)
#   and intended function
#   [ (elt is an et.Element) and
#     (item is a dictionary) ->
#       elt := elt with an attribute made from
#       each key-value pair from item ]
def addDict ( elt, item ):
    for key, value in item.items():
        if isinstance(value, basestring):
            elt.attrib[key] = value
        else:
            elt.attrib[key] = str(value)
self.__typeMap[dict] = addDict

```

Note

In Lundh's original, the last line of the previous block was the equivalent of this:

```

elt.attrib[key] = \
    self.__typeMap[type(value)](None, value)

```

I'm not entirely sure what he had in mind here. If you have any good theories, please forward them to <tcc-doc@nmt.edu>.

Next up is the handler for arguments that are instances of `et.Element`. We'll actually create an `et.Element` to be sure that `self.__typeMap` uses the correct key.

```

#-- 5 --
# [ self.__typeMap[type(et.Element instances)] := a
#     function with calling sequence
#     addElt(elt, item)
#     and intended function
#     [ (elt and item are et.Element instances) ->
#       elt := elt with item added as its next
#           child element ]
def addDict ( elt, item ):
    elt.append ( item )
sample = et.Element ( 'sample' )
self.__typeMap[type(sample)] = addDict

```

12.5. ElementMaker.__call__(): Handle calls to the factory instance

This method is called when the user calls the factory instance E.

```

# - - -   E l e m e n t M a k e r . _ _ c a l l _ _

def __call__ ( self, tag, *argList, **attr):
    '''Handle calls to a factory instance.
    ...

```

First we create a new, empty element with the given tag name.

```

#-- 1 --
# [ elt := a new et.Element with name (tag) ]
elt = et.Element ( tag )

```

If the `attr` dictionary has anything in it, we can use the function stored in `self.__typeMap[dict]` to process those attributes.

```

#-- 2 --
# [ elt := elt with attributes made from the key-value
#     pairs in attr ]
# else -> I ]
if attr:
    self.__typeMap[dict](elt, attr)

```

Next, process the positional arguments in a loop, using each argument's type to extract from `self.__typeMap` the proper handler for that type. For this logic, see Section 12.6, “Element-Maker.__handleArg(): Process one positional argument” (p. 35).

```

#-- 3 --
# [ if the types of all the members of pos are also
#   keys in self.__typeMap ->
#     elt := elt modified as per the corresponding
#         functions from self.__typeMap
#   else -> raise TypeError ]
for arg in argList:
    #-- 3 body --

```

```

# [ if type(arg) is a key in self.__typeMap ->
#     elt := elt modified as per self.__typeMap[type(arg)]
#     else -> raise TypeError ]
self.__handleArg ( elt, arg )

```

Finally, return the shiny new element to the caller.

etbuilder.py

```

#-- 4 --
return elt

```

12.6. ElementMaker.__handleArg(): Process one positional argument

This method processes one of the positional arguments when the factory instance is called.

etbuilder.py

```

# - - -   E l e m e n t M a k e r . _ _ h a n d l e A r g

def __handleArg ( self, elt, arg ):
    '''Process one positional argument to the factory instance.

    [ (elt is an et.Element) ->
      if type(arg) is a key in self.__typeMap ->
        elt := elt modified as per self.__typeMap[type(arg)]
      else -> raise TypeError ]
    ...

```

As a convenience, if the caller passes some callable object, we'll call that object and use its result. Otherwise we'll use the object itself. (This is another Lundh feature, the utility of which I don't fully understand.)

etbuilder.py

```

#-- 1 --
# [ if arg is callable ->
#     value := arg()
#     else ->
#     value := arg ]
if callable(arg):
    value = arg()
else:
    value = arg

```

Next we look up the value's type in self.__typeMap, and call the corresponding function.

etbuilder.py

```

#-- 2 --
# [ if type(value) is a key in self.__typeMap ->
#     elt := elt modified as per self.__typeMap[type(value)]
#     else -> raise TypeError ]
try:
    handler = self.__typeMap[type(value)]
    handler(elt, value)
except KeyError:
    raise TypeError ( "Invalid argument type: %r" % value )

```

12.7. ElementMaker.__getattr__(): Handle arbitrary method calls

This method is called whenever the caller invokes an undefined method of a factory instance. It implements the feature that you can use an element name as a method name so that “E.tag(...)” is the equivalent of “E(tag, ...)”.

The method is a one-liner, but it's a rather abstruse one-liner for anyone that has never studied functional programming. See the `functools.partial` documentation¹⁷. The method returns a callable object that acts the same as a call to the factory instance, except with `tag` inserted before its other positional arguments.

The Wikipedia article on currying¹⁸ explains this technique in depth.

etbuilder.py

```
# - - -   E l e m e n t M a k e r . _ _ g e t a t t r _ _  
  
def __getattr__( self, tag ):  
    '''Handle arbitrary method calls.  
  
    [ tag is a string ->  
      return a new et.Element instance whose name  
      is (tag), and elements of p and kw have  
      the same effects as E(*(p[1:]), **kw) ]  
    ...  
    return partial ( self, tag )
```

12.8. Epilogue

The last step is to create the factory instance E.

etbuilder.py

```
# - - - - -   m a i n  
  
E = ElementMaker()
```

12.9. testetbuilder: A test driver for etbuilder

Here is a small script that exercises the `etbuilder` module.

This script generates a small XHTML page that looks like this:

```
<html>  
  <head>  
    <title>Sample page<title>  
    <link href="/tcc/style.css" rel="stylesheet"/>  
  </head>  
  <body>  
    <h1 class='big-title'>Sample page title</h1>  
    <p>A paragraph containing a <a href='http://www.nmt.edu/'>  
    >link to the NMT homepage</a>.</p>  
  </body>  
</html>
```

¹⁷ <http://www.python.org/doc/2.5.2/lib/module-functools.html>

¹⁸ <http://en.wikipedia.org/wiki/Currying>

The script follows.

testetbuilder

```
#!/usr/bin/env python
from etbuilder import E, et

page = E.html(
    E.head(
        E.title("Sample page"),
        E.link(href='/tcc/style.css', rel='stylesheet')),
    E.body(
        E.h1({'class': 'big-title'}, "Sample page title"),
        E.p("A paragraph containing ", 1, " ",
            E.a("link to the NMT homepage",
                href='http://www.nmt.edu/'),
            "."))))
print et.tostring(page, pretty_print=True)
```

