The lxml.etree Tutorial

Author: Stefan Behnel

This is a tutorial on XML processing with `lxml.etree`. It briefly overviews the main concepts of the ElementTree API, and some simple enhancements that make your life as a programmer easier.

For a complete reference of the API, see the generated API documentation.

Contents

A common way to import `lxml.etree` is as follows:

```
>>> from lxml import etree
```

If your code only uses the ElementTree API and does not rely on any functionality that is specific to `lxml.etree`, you can also use (any part of) the following import chain as a fall-back to the original ElementTree:

```
try:
  from lxml import etree
  print("running with lxml.etree")
except ImportError:
  try:
    # Python 2.5
    import xml.etree.cElementTree as etree
    print("running with cElementTree on Python 2.5+")
  except ImportError:
    try:
      # Python 2.5
      import xml.etree.ElementTree as etree
      print("running with ElementTree on Python 2.5+")
    except ImportError:
      try:
        # normal cElementTree install
        import cElementTree as etree
        print("running with cElementTree")
      except ImportError:
        try:
          # normal ElementTree install
          import elementtree.ElementTree as etree
          print("running with ElementTree")
        except ImportError:
          print("Failed to import ElementTree from any known place")
```

To aid in writing portable code, this tutorial makes it clear in the examples which part of the presented API is an extension of `lxml.etree` over the original ElementTree API, as defined by Fredrik Lundh's ElementTree library.

The Element class

An `Element` is the main container object for the ElementTree API. Most of the XML tree functionality is accessed through this class. Elements are easily created through the `Element` factory:

```
>>> root = etree.Element("root")
```

The XML tag name of elements is accessed through the `tag` property:

```
>>> print(root.tag)
root
```

Elements are organised in an XML tree structure. To create child elements and add them to a parent element, you can use the `append()` method:

```
>>> root.append( etree.Element("child1") )
```

However, this is so common that there is a shorter and much more efficient way to do this: the `SubElement` factory. It accepts the same arguments as the `Element` factory, but additionally requires the parent as first argument:

```
>>> child2 = etree.SubElement(root, "child2")
>>> child3 = etree.SubElement(root, "child3")
```

To see that this is really XML, you can serialise the tree you have created:

```
>>> print(etree.tostring(root, pretty_print=True))
<root>
  <child1/>
  <child2/>
  <child3/>
</root>
```

Elements are lists

To make the access to these subelements easy and straight forward, elements mimic the behaviour of normal Python lists as closely as possible:

```
>>> child = root[0]
>>> print(child.tag)
child1

>>> print(len(root))
3

>>> root.index(root[1]) # lxml.etree only!
1

>>> children = list(root)

>>> for child in root:
...     print(child.tag)
child1
child2
child3

>>> root.insert(0, etree.Element("child0"))
>>> start = root[:1]
>>> end   = root[-1:]
>>> print(start[0].tag)
child0
>>> print(end[0].tag)
child3
```

Prior to ElementTree 1.3 and lxml 2.0, you could also check the truth value of an Element to see if it has children, i.e. if the list of children is empty:

```
if root:   # this no longer works!
    print("The root element has children")
```

This is no longer supported as people tend to expect that a "something" evaluates to True and expect Elements to be "something", may they have children or not. So, many users find it surprising that any Element would evaluate to False in an if-statement like the above. Instead, use `len(element)`, which is both more explicit and less error prone.

```
>>> print(etree.iselement(root))  # test if it's some kind of Element
True
>>> if len(root):                 # test if it has children
...     print("The root element has children")
The root element has children
```

There is another important case where the behaviour of Elements in lxml (in 2.0 and later) deviates from that of lists and from that of the original ElementTree (prior to version 1.3 or Python 2.7/3.2):

```
>>> for child in root:
...     print(child.tag)
child0
child1
child2
child3
>>> root[0] = root[-1]  # this moves the element in lxml.etree!
>>> for child in root:
...     print(child.tag)
child3
child1
child2
```

In this example, the last element is moved to a different position, instead of being copied, i.e. it is automatically removed from its previous position when it is put in a different place. In lists, objects can appear in multiple positions at the same time, and the above assignment would just copy the item reference into the first position, so that both contain the exact same item:

```
>>> l = [0, 1, 2, 3]
>>> l[0] = l[-1]
>>> l
[3, 1, 2, 3]
```

Note that in the original ElementTree, a single Element object can sit in any number of places in any number of trees, which allows for the same copy operation as with lists. The obvious drawback is that modifications to such an Element will apply to all places where it appears in a tree, which may or may not be intended.

The upside of this difference is that an Element in `lxml.etree` always has exactly one parent, which can be queried through the `getparent()` method. This is not supported in the original ElementTree.

```
>>> root is root[0].getparent()  # lxml.etree only!
True
```

If you want to copy an element to a different position in `lxml.etree`, consider creating an independent deep copy using the copy module from Python's standard library:

```
>>> from copy import deepcopy
>>> element = etree.Element("neu")
>>> element.append( deepcopy(root[1]) )

>>> print(element[0].tag)
child1
>>> print([ c.tag for c in root ])
['child3', 'child1', 'child2']
```

The siblings (or neighbours) of an element are accessed as next and previous elements:

```
>>> root[0] is root[1].getprevious() # lxml.etree only!
True
>>> root[1] is root[0].getnext() # lxml.etree only!
True
```

Elements carry attributes as a dict

XML elements support attributes. You can create them directly in the Element factory:

```
>>> root = etree.Element("root", interesting="totally")
>>> etree.tostring(root)
b'<root interesting="totally"/>'
```

Attributes are just unordered name-value pairs, so a very convenient way of dealing with them is through the dictionary-like interface of Elements:

```
>>> print(root.get("interesting"))
totally

>>> print(root.get("hello"))
None
>>> root.set("hello", "Huhu")
>>> print(root.get("hello"))
Huhu

>>> etree.tostring(root)
b'<root interesting="totally" hello="Huhu"/>'

>>> sorted(root.keys())
['hello', 'interesting']

>>> for name, value in sorted(root.items()):
...     print('%s = %r' % (name, value))
hello = 'Huhu'
interesting = 'totally'
```

For the cases where you want to do item lookup or have other reasons for getting a 'real' dictionary-like object, e.g. for passing it around, you can use the `attrib` property:

```
>>> attributes = root.attrib

>>> print(attributes["interesting"])
totally
>>> print(attributes.get("no-such-attribute"))
None

>>> attributes["hello"] = "Guten Tag"
>>> print(attributes["hello"])
Guten Tag
>>> print(root.get("hello"))
Guten Tag
```

Note that `attrib` is a dict-like object backed by the Element itself. This means that any changes to the Element are reflected in `attrib` and vice versa. It also means that the XML tree stays alive in memory as long as the `attrib` of one of its Elements is in use. To get an independent snapshot of the attributes that does not depend on the XML tree, copy it into a dict:

```
>>> d = dict(root.attrib)
>>> sorted(d.items())
[('hello', 'Guten Tag'), ('interesting', 'totally')]
```

Elements contain text

Elements can contain text:

```
>>> root = etree.Element("root")
>>> root.text = "TEXT"

>>> print(root.text)
TEXT

>>> etree.tostring(root)
b'<root>TEXT</root>'
```

In many XML documents (data-centric documents), this is the only place where text can be found. It is encapsulated by a leaf tag at the very bottom of the tree hierarchy.

However, if XML is used for tagged text documents such as (X)HTML, text can also appear between different elements, right in the middle of the tree:

```
<html><body>Hello<br/>World</body></html>
```

Here, the `<br/>` tag is surrounded by text. This is often referred to as document-style or mixed-content XML. Elements support this through their `tail` property. It contains the text that directly follows the element, up to the next element in the XML tree:

```
>>> html = etree.Element("html")
>>> body = etree.SubElement(html, "body")
>>> body.text = "TEXT"

>>> etree.tostring(html)
b'<html><body>TEXT</body></html>'

>>> br = etree.SubElement(body, "br")
>>> etree.tostring(html)
b'<html><body>TEXT<br/></body></html>'

>>> br.tail = "TAIL"
>>> etree.tostring(html)
b'<html><body>TEXT<br/>TAIL</body></html>'
```

The two properties `.text` and `.tail` are enough to represent any text content in an XML document. This way, the ElementTree API does not require any special text nodes in addition to the Element class, that tend to get in the way fairly often (as you might know from classic DOM APIs).

However, there are cases where the tail text also gets in the way. For example, when you serialise an Element from within the tree, you do not always want its tail text in the result (although you would still want the tail text of its children). For this purpose, the `tostring()` function accepts the keyword argument `with_tail`:

```
>>> etree.tostring(br)
b'<br/>TAIL'
>>> etree.tostring(br, with_tail=False) # lxml.etree only!
b'<br/>'
```

If you want to read only the text, i.e. without any intermediate tags, you have to recursively concatenate all `text` and `tail` attributes in the correct order. Again, the `tostring()` function comes to the rescue, this time using the `method` keyword:

```
>>> etree.tostring(html, method="text")
b'TEXTTAIL'
```

Using XPath to find text

Another way to extract the text content of a tree is XPath, which also allows you to extract the separate text chunks into a list:

```
>>> print(html.xpath("string()")) # lxml.etree only!
TEXTTAIL
>>> print(html.xpath("//text()")) # lxml.etree only!
['TEXT', 'TAIL']
```

If you want to use this more often, you can wrap it in a function:

```
>>> build_text_list = etree.XPath("//text()") # lxml.etree only!
>>> print(build_text_list(html))
['TEXT', 'TAIL']
```

Note that a string result returned by XPath is a special 'smart' object that knows about its origins. You can ask it where it came from through its `getparent()` method, just as you would with Elements:

```
>>> texts = build_text_list(html)
>>> print(texts[0])
TEXT
>>> parent = texts[0].getparent()
>>> print(parent.tag)
body

>>> print(texts[1])
TAIL
>>> print(texts[1].getparent().tag)
br
```

You can also find out if it's normal text content or tail text:

```
>>> print(texts[0].is_text)
True
>>> print(texts[1].is_text)
False
>>> print(texts[1].is_tail)
True
```

While this works for the results of the `text()` function, lxml will not tell you the origin of a string value that was constructed by the XPath functions `string()` or `concat()`:

```
>>> stringify = etree.XPath("string()")
>>> print(stringify(html))
TEXTTAIL
>>> print(stringify(html).getparent())
None
```

Tree iteration

For problems like the above, where you want to recursively traverse the tree and do something with its elements, tree iteration is a very convenient solution. Elements provide a tree iterator for this purpose. It yields elements in document order, i.e. in the order their tags would appear if you serialised the tree to XML:

```
>>> root = etree.Element("root")
>>> etree.SubElement(root, "child").text = "Child 1"
>>> etree.SubElement(root, "child").text = "Child 2"
>>> etree.SubElement(root, "another").text = "Child 3"
>>> print(etree.tostring(root, pretty_print=True))
<root>
  <child>Child 1</child>
  <child>Child 2</child>
  <another>Child 3</another>
</root>

>>> for element in root.iter():
...     print("%s - %s" % (element.tag, element.text))
root - None
child - Child 1
child - Child 2
another - Child 3
```

If you know you are only interested in a single tag, you can pass its name to `iter()` to have it filter for you. Starting with lxml 3.0, you can also pass more than one tag to intercept on multiple tags during iteration.

```
>>> for element in root.iter("child"):
...     print("%s - %s" % (element.tag, element.text))
child - Child 1
child - Child 2

>>> for element in root.iter("another", "child"):
...     print("%s - %s" % (element.tag, element.text))
child - Child 1
child - Child 2
another - Child 3
```

By default, iteration yields all nodes in the tree, including ProcessingInstructions, Comments and Entity instances. If you want to make sure only Element objects are returned, you can pass the `Element` factory as tag parameter:

```
>>> root.append(etree.Entity("#234"))
>>> root.append(etree.Comment("some comment"))

>>> for element in root.iter():
...     if isinstance(element.tag, basestring):  # or 'str' in Python 3
...         print("%s - %s" % (element.tag, element.text))
...     else:
...         print("SPECIAL: %s - %s" % (element, element.text))
root - None
child - Child 1
child - Child 2
another - Child 3
SPECIAL: &#234; - &#234;
SPECIAL: <!--some comment--> - some comment

>>> for element in root.iter(tag=etree.Element):
...     print("%s - %s" % (element.tag, element.text))
root - None
child - Child 1
child - Child 2
another - Child 3

>>> for element in root.iter(tag=etree.Entity):
...     print(element.text)
&#234;
```

Note that passing a wildcard `"*"` tag name will also yield all `Element` nodes (and only elements).

In `lxml.etree`, elements provide further iterators for all directions in the tree: children, parents (or rather ancestors) and siblings.

Serialisation

Serialisation commonly uses the `tostring()` function that returns a string, or the `ElementTree.write()` method that writes to a file, a file-like object, or a URL (via FTP PUT or HTTP POST). Both calls accept the same keyword arguments like `pretty_print` for formatted output or `encoding` to select a specific output encoding other than plain ASCII:

```
>>> root = etree.XML('<root><a><b/></a></root>')

>>> etree.tostring(root)
b'<root><a><b/></a></root>'

>>> print(etree.tostring(root, xml_declaration=True))
<?xml version='1.0' encoding='ASCII'?>
<root><a><b/></a></root>

>>> print(etree.tostring(root, encoding='iso-8859-1'))
<?xml version='1.0' encoding='iso-8859-1'?>
<root><a><b/></a></root>

>>> print(etree.tostring(root, pretty_print=True))
<root>
  <a>
    <b/>
  </a>
</root>
```

Note that pretty printing appends a newline at the end.

In lxml 2.0 and later (as well as ElementTree 1.3), the serialisation functions can do more than XML serialisation. You can serialise to HTML or extract the text content by passing the `method` keyword:

```
>>> root = etree.XML(
...    '<html><head/><body><p>Hello<br/>World</p></body></html>')

>>> etree.tostring(root) # default: method = 'xml'
b'<html><head/><body><p>Hello<br/>World</p></body></html>'

>>> etree.tostring(root, method='xml') # same as above
b'<html><head/><body><p>Hello<br/>World</p></body></html>'

>>> etree.tostring(root, method='html')
b'<html><head></head><body><p>Hello<br>World</p></body></html>'

>>> print(etree.tostring(root, method='html', pretty_print=True))
<html>
<head></head>
<body><p>Hello<br>World</p></body>
</html>

>>> etree.tostring(root, method='text')
b'HelloWorld'
```

As for XML serialisation, the default encoding for plain text serialisation is ASCII:

```
>>> br = next(root.iter('br'))  # get first result of iteration
>>> br.tail = u'W\xf6rld'

>>> etree.tostring(root, method='text')  # doctest: +ELLIPSIS
Traceback (most recent call last):
  ...
UnicodeEncodeError: 'ascii' codec can't encode character u'\xf6' ...

>>> etree.tostring(root, method='text', encoding="UTF-8")
b'HelloW\xc3\xb6rld'
```

Here, serialising to a Python unicode string instead of a byte string might become handy. Just pass the name `'unicode'` as encoding:

```
>>> etree.tostring(root, encoding='unicode', method='text')
u'HelloW\xf6rld'
```

The W3C has a good article about the Unicode character set and character encodings.

The ElementTree class

An `ElementTree` is mainly a document wrapper around a tree with a root node. It provides a couple of methods for serialisation and general document handling.

```
>>> root = etree.XML('''\
... <?xml version="1.0"?>
... <!DOCTYPE root SYSTEM "test" [ <!ENTITY tasty "parsnips"> ]>
... <root>
...   <a>&tasty;</a>
... </root>
... ''')

>>> tree = etree.ElementTree(root)
>>> print(tree.docinfo.xml_version)
1.0
>>> print(tree.docinfo.doctype)
<!DOCTYPE root SYSTEM "test">

>>> tree.docinfo.public_id = '-//W3C//DTD XHTML 1.0 Transitional//EN'
>>> tree.docinfo.system_url = 'file://local.dtd'
>>> print(tree.docinfo.doctype)
<!DOCTYPE root PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "file://local.dtd">
```

An `ElementTree` is also what you get back when you call the `parse()` function to parse files or file-like objects (see the parsing section below).

One of the important differences is that the `ElementTree` class serialises as a complete document, as opposed to a single `Element`. This includes top-level processing instructions and comments, as well as a DOCTYPE and other DTD content in the document:

```
>>> print(etree.tostring(tree))  # lxml 1.3.4 and later
<!DOCTYPE root PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "file://local.dtd" [
<!ENTITY tasty "parsnips">
]>
<root>
  <a>parsnips</a>
</root>
```

In the original xml.etree.ElementTree implementation and in lxml up to 1.3.3, the output looks the same as when serialising only the root Element:

```
>>> print(etree.tostring(tree.getroot()))
<root>
  <a>parsnips</a>
</root>
```

This serialisation behaviour has changed in lxml 1.3.4. Before, the tree was serialised without DTD content, which made lxml lose DTD information in an input-output cycle.

Parsing from strings and files

`lxml.etree` supports parsing XML in a number of ways and from all important sources, namely strings, files, URLs (http/ftp) and file-like objects. The main parse functions are `fromstring()` and `parse()`, both called with the source as first argument. By default, they use the standard parser, but you can always pass a different parser as second argument.

The fromstring() function

The `fromstring()` function is the easiest way to parse a string:

```
>>> some_xml_data = "<root>data</root>"

>>> root = etree.fromstring(some_xml_data)
>>> print(root.tag)
root
>>> etree.tostring(root)
b'<root>data</root>'
```

The XML() function

The `XML()` function behaves like the `fromstring()` function, but is commonly used to write XML literals right into the source:

```
>>> root = etree.XML("<root>data</root>")
>>> print(root.tag)
root
>>> etree.tostring(root)
b'<root>data</root>'
```

There is also a corresponding function `HTML()` for HTML literals.

```
>>> root = etree.HTML("<p>data</p>")
>>> etree.tostring(root)
b'<html><body><p>data</p></body></html>'
```

The parse() function

The `parse()` function is used to parse from files and file-like objects.

As an example of such a file-like object, the following code uses the `BytesIO` class for reading from a string instead of an external file. That class comes from the `io` module in Python 2.6 and later. In older Python versions, you will have to use the `StringIO` class from the `StringIO` module. However, in real life, you would obviously avoid doing this all together and use the string parsing functions above.

```
>>> from io import BytesIO
>>> some_file_or_file_like_object = BytesIO(b"<root>data</root>")

>>> tree = etree.parse(some_file_or_file_like_object)

>>> etree.tostring(tree)
b'<root>data</root>'
```

Note that `parse()` returns an ElementTree object, not an Element object as the string parser functions:

```
>>> root = tree.getroot()
>>> print(root.tag)
root
>>> etree.tostring(root)
b'<root>data</root>'
```

The reasoning behind this difference is that `parse()` returns a complete document from a file, while the string parsing functions are commonly used to parse XML fragments.

The `parse()` function supports any of the following sources:

- an open file object (make sure to open it in binary mode)
- a file-like object that has a `.read(byte_count)` method returning a byte string on each call
- a filename string
- an HTTP or FTP URL string

Note that passing a filename or URL is usually faster than passing an open file or file-like object. However, the HTTP/FTP client in libxml2 is rather simple, so things like HTTP authentication require a dedicated URL request library, e.g. `urllib2` or `request`. These libraries usually provide a file-like object for the result that you can parse from while the response is streaming in.

Parser objects

By default, `lxml.etree` uses a standard parser with a default setup. If you want to configure the parser, you can create a new instance:

```
>>> parser = etree.XMLParser(remove_blank_text=True) # lxml.etree only!
```

This creates a parser that removes empty text between tags while parsing, which can reduce the size of the tree and avoid dangling tail text if you know that whitespace-only content is not meaningful for your data. An example:

```
>>> root = etree.XML("<root>  <a/>   <b>  </b>     </root>", parser)

>>> etree.tostring(root)
b'<root><a/><b>  </b></root>'
```

Note that the whitespace content inside the <b> tag was not removed, as content at leaf elements tends to be data content (even if blank). You can easily remove it in an additional step by traversing the tree:

```
>>> for element in root.iter("*"):
...     if element.text is not None and not element.text.strip():
...         element.text = None

>>> etree.tostring(root)
b'<root><a/><b/></root>'
```

See `help(etree.XMLParser)` to find out about the available parser options.

Incremental parsing

`lxml.etree` provides two ways for incremental step-by-step parsing. One is through file-like objects, where it calls the `read()` method repeatedly. This is best used where the data arrives from a source like `urllib` or any other file-like object that can provide data on request. Note that the parser will block and wait until data becomes available in this case:

```
>>> class DataSource:
...     data = [ b"<roo", b"t><", b"a/", b"><", b"/root>" ]
...     def read(self, requested_size):
...         try:
...             return self.data.pop(0)
...         except IndexError:
...             return b''

>>> tree = etree.parse(DataSource())

>>> etree.tostring(tree)
b'<root><a/></root>'
```

The second way is through a feed parser interface, given by the `feed(data)` and `close()` methods:

```
>>> parser = etree.XMLParser()

>>> parser.feed("<roo")
>>> parser.feed("t><")
>>> parser.feed("a/")
>>> parser.feed("><")
>>> parser.feed("/root>")

>>> root = parser.close()

>>> etree.tostring(root)
b'<root><a/></root>'
```

Here, you can interrupt the parsing process at any time and continue it later on with another call to the `feed()` method. This comes in handy if you want to avoid blocking calls to the parser, e.g. in frameworks like Twisted, or whenever data comes in slowly or in chunks and you want to do other things while waiting for the next chunk.

After calling the `close()` method (or when an exception was raised by the parser), you can reuse the parser by calling its `feed()` method again:

```
>>> parser.feed("<root/>")
>>> root = parser.close()
>>> etree.tostring(root)
b'<root/>'
```

Event-driven parsing

Sometimes, all you need from a document is a small fraction somewhere deep inside the tree, so parsing the whole tree into memory, traversing it and dropping it can be too much overhead. `lxml.etree` supports this use case with two event-driven parser interfaces, one that generates parser events while building the tree (`iterparse`), and one that does not build the tree at all, and instead calls feedback methods on a target object in a SAX-like fashion.

Here is a simple `iterparse()` example:

```
>>> some_file_like = BytesIO(b"<root><a>data</a></root>")

>>> for event, element in etree.iterparse(some_file_like):
...     print("%s, %4s, %s" % (event, element.tag, element.text))
end,    a, data
end, root, None
```

By default, `iterparse()` only generates events when it is done parsing an element, but you can control this through the `events` keyword argument:

```
>>> some_file_like = BytesIO(b"<root><a>data</a></root>")

>>> for event, element in etree.iterparse(some_file_like,
...                                       events=("start", "end")):
...     print("%5s, %4s, %s" % (event, element.tag, element.text))
start, root, None
start,    a, data
  end,    a, data
  end, root, None
```

Note that the text, tail, and children of an Element are not necessarily present yet when receiving the `start` event. Only the `end` event guarantees that the Element has been parsed completely.

It also allows you to `.clear()` or modify the content of an Element to save memory. So if you parse a large tree and you want to keep memory usage small, you should clean up parts of the tree that you no longer need:

```
>>> some_file_like = BytesIO(
...     b"<root><a><b>data</b></a><a><b/></a></root>")

>>> for event, element in etree.iterparse(some_file_like):
...     if element.tag == 'b':
...         print(element.text)
...     elif element.tag == 'a':
...         print("** cleaning up the subtree")
...         element.clear()
data
** cleaning up the subtree
None
** cleaning up the subtree
```

A very important use case for `iterparse()` is parsing large generated XML files, e.g. database dumps. Most often, these XML formats only have one main data item element that hangs directly below the root node and that is repeated thousands of times. In this case, it is best practice to let `lxml.etree` do the tree building and only to intercept on exactly this one Element, using the normal tree API for data extraction.

```
>>> xml_file = BytesIO(b'''\
... <root>
...   <a><b>ABC</b><c>abc</c></a>
...   <a><b>MORE DATA</b><c>more data</c></a>
...   <a><b>XYZ</b><c>xyz</c></a>
... </root>''')

>>> for _, element in etree.iterparse(xml_file, tag='a'):
...     print('%s -- %s' % (element.findtext('b'), element[1].text))
...     element.clear()
ABC -- abc
MORE DATA -- more data
XYZ -- xyz
```

If, for some reason, building the tree is not desired at all, the target parser interface of `lxml.etree` can be used. It creates SAX-like events by calling the methods of a target object. By implementing some or all of these methods, you can control which events are generated:

```
>>> class ParserTarget:
...     events = []
...     close_count = 0
...     def start(self, tag, attrib):
...         self.events.append(("start", tag, attrib))
...     def close(self):
...         events, self.events = self.events, []
...         self.close_count += 1
...         return events

>>> parser_target = ParserTarget()

>>> parser = etree.XMLParser(target=parser_target)
>>> events = etree.fromstring('<root test="true"/>', parser)

>>> print(parser_target.close_count)
1

>>> for event in events:
...     print('event: %s - tag: %s' % (event[0], event[1]))
...     for attr, value in event[2].items():
...         print(' * %s = %s' % (attr, value))
event: start - tag: root
 * test = true
```

You can reuse the parser and its target as often as you like, so you should take care that the `.close()` method really resets the target to a usable state (also in the case of an error!).

```
>>> events = etree.fromstring('<root test="true"/>', parser)
>>> print(parser_target.close_count)
2
>>> events = etree.fromstring('<root test="true"/>', parser)
>>> print(parser_target.close_count)
3
>>> events = etree.fromstring('<root test="true"/>', parser)
>>> print(parser_target.close_count)
4

>>> for event in events:
...     print('event: %s - tag: %s' % (event[0], event[1]))
...     for attr, value in event[2].items():
...         print(' * %s = %s' % (attr, value))
event: start - tag: root
 * test = true
```

Namespaces

The ElementTree API avoids namespace prefixes wherever possible and deploys the real namespace (the URI) instead:

```
>>> xhtml = etree.Element("{http://www.w3.org/1999/xhtml}html")
>>> body = etree.SubElement(xhtml, "{http://www.w3.org/1999/xhtml}body")
>>> body.text = "Hello World"

>>> print(etree.tostring(xhtml, pretty_print=True))
<html:html xmlns:html="http://www.w3.org/1999/xhtml">
  <html:body>Hello World</html:body>
</html:html>
```

The notation that ElementTree uses was originally brought up by James Clark. It has the major advantage of providing a universally qualified name for a tag, regardless of any prefixes that may or may not have been used or defined in a document. By moving the indirection of prefixes out of the way, it makes namespace aware code much clearer and easier to get right.

As you can see from the example, prefixes only become important when you serialise the result. However, the above code looks somewhat verbose due to the lengthy namespace names. And retyping or copying a string over and over again is error prone. It is therefore common practice to store a namespace URI in a global variable. To adapt the namespace prefixes for serialisation, you can also pass a mapping to the Element factory function, e.g. to define the default namespace:

```
>>> XHTML_NAMESPACE = "http://www.w3.org/1999/xhtml"
>>> XHTML = "{%s}" % XHTML_NAMESPACE

>>> NSMAP = {None : XHTML_NAMESPACE} # the default namespace (no prefix)

>>> xhtml = etree.Element(XHTML + "html", nsmap=NSMAP) # lxml only!
>>> body = etree.SubElement(xhtml, XHTML + "body")
>>> body.text = "Hello World"

>>> print(etree.tostring(xhtml, pretty_print=True))
<html xmlns="http://www.w3.org/1999/xhtml">
  <body>Hello World</body>
</html>
```

You can also use the `QName` helper class to build or split qualified tag names:

```
>>> tag = etree.QName('http://www.w3.org/1999/xhtml', 'html')
>>> print(tag.localname)
html
>>> print(tag.namespace)
http://www.w3.org/1999/xhtml
>>> print(tag.text)
{http://www.w3.org/1999/xhtml}html

>>> tag = etree.QName('{http://www.w3.org/1999/xhtml}html')
>>> print(tag.localname)
html
>>> print(tag.namespace)
http://www.w3.org/1999/xhtml

>>> root = etree.Element('{http://www.w3.org/1999/xhtml}html')
>>> tag = etree.QName(root)
>>> print(tag.localname)
html

>>> tag = etree.QName(root, 'script')
>>> print(tag.text)
{http://www.w3.org/1999/xhtml}script
>>> tag = etree.QName('{http://www.w3.org/1999/xhtml}html', 'script')
>>> print(tag.text)
{http://www.w3.org/1999/xhtml}script
```

lxml.etree allows you to look up the current namespaces defined for a node through the `.nsmap` property:

```
>>> xhtml.nsmap
{None: 'http://www.w3.org/1999/xhtml'}
```

Note, however, that this includes all prefixes known in the context of an Element, not only those that it defines itself.

```
>>> root = etree.Element('root', nsmap={'a': 'http://a.b/c'})
>>> child = etree.SubElement(root, 'child',
...                          nsmap={'b': 'http://b.c/d'})
>>> len(root.nsmap)
1
>>> len(child.nsmap)
2
>>> child.nsmap['a']
'http://a.b/c'
>>> child.nsmap['b']
'http://b.c/d'
```

Therefore, modifying the returned dict cannot have any meaningful impact on the Element. Any changes to it are ignored.

Namespaces on attributes work alike, but as of version 2.3, `lxml.etree` will ensure that the attribute uses a prefixed namespace declaration. This is because unprefixed attribute names are not considered being in a namespace by the XML namespace specification (section 6.2), so they may end up losing their namespace on a serialise-parse roundtrip, even if they appear in a namespaced element.

```
>>> body.set(XHTML + "bgcolor", "#CCFFAA")

>>> print(etree.tostring(xhtml, pretty_print=True))
<html xmlns="http://www.w3.org/1999/xhtml">
  <body xmlns:html="http://www.w3.org/1999/xhtml" html:bgcolor="#CCFFAA">Hello World</body>
</html>

>>> print(body.get("bgcolor"))
None
>>> body.get(XHTML + "bgcolor")
'#CCFFAA'
```

You can also use XPath with fully qualified names:

```
>>> find_xhtml_body = etree.ETXPath(      # lxml only !
...     "//{%s}body" % XHTML_NAMESPACE)
>>> results = find_xhtml_body(xhtml)

>>> print(results[0].tag)
{http://www.w3.org/1999/xhtml}body
```

For convenience, you can use `"*"` wildcards in all iterators of `lxml.etree`, both for tag names and namespaces:

```
>>> for el in xhtml.iter('*'): print(el.tag)   # any element
{http://www.w3.org/1999/xhtml}html
{http://www.w3.org/1999/xhtml}body
>>> for el in xhtml.iter('{http://www.w3.org/1999/xhtml}*'): print(el.tag)
{http://www.w3.org/1999/xhtml}html
{http://www.w3.org/1999/xhtml}body
>>> for el in xhtml.iter('{*}body'): print(el.tag)
{http://www.w3.org/1999/xhtml}body
```

To look for elements that do not have a namespace, either use the plain tag name or provide the empty namespace explicitly:

```
>>> [ el.tag for el in xhtml.iter('{http://www.w3.org/1999/xhtml}body') ]
['{http://www.w3.org/1999/xhtml}body']
>>> [ el.tag for el in xhtml.iter('body') ]
[]
>>> [ el.tag for el in xhtml.iter('{}body') ]
[]
>>> [ el.tag for el in xhtml.iter('{}*') ]
[]
```

The E-factory

The `E-factory` provides a simple and compact syntax for generating XML and HTML:

```
>>> from lxml.builder import E

>>> def CLASS(*args): # class is a reserved word in Python
...     return {"class":' '.join(args)}

>>> html = page = (
...   E.html(       # create an Element called "html"
...     E.head(
...       E.title("This is a sample document")
...     ),
...     E.body(
...       E.h1("Hello!", CLASS("title")),
...       E.p("This is a paragraph with ", E.b("bold"), " text in it!"),
...       E.p("This is another paragraph, with a", "\n      ",
...         E.a("link", href="http://www.python.org"), "."),
...       E.p("Here are some reservered characters: <spam&egg>."),
...       etree.XML("<p>And finally an embedded XHTML fragment.</p>"),
...     )
...   )
... )

>>> print(etree.tostring(page, pretty_print=True))
<html>
  <head>
    <title>This is a sample document</title>
  </head>
  <body>
    <h1 class="title">Hello!</h1>
    <p>This is a paragraph with <b>bold</b> text in it!</p>
    <p>This is another paragraph, with a
      <a href="http://www.python.org">link</a>.</p>
    <p>Here are some reservered characters: &lt;spam&amp;egg&gt;.</p>
    <p>And finally an embedded XHTML fragment.</p>
  </body>
</html>
```

Element creation based on attribute access makes it easy to build up a simple vocabulary for an XML language:

```
>>> from lxml.builder import ElementMaker # lxml only !

>>> E = ElementMaker(namespace="http://my.de/fault/namespace",
...                  nsmap={'p' : "http://my.de/fault/namespace"})

>>> DOC = E.doc
>>> TITLE = E.title
>>> SECTION = E.section
>>> PAR = E.par

>>> my_doc = DOC(
...   TITLE("The dog and the hog"),
...   SECTION(
...     TITLE("The dog"),
...     PAR("Once upon a time, ..."),
...     PAR("And then ...")
...   ),
...   SECTION(
...     TITLE("The hog"),
...     PAR("Sooner or later ...")
...   )
... )

>>> print(etree.tostring(my_doc, pretty_print=True))
<p:doc xmlns:p="http://my.de/fault/namespace">
  <p:title>The dog and the hog</p:title>
  <p:section>
    <p:title>The dog</p:title>
    <p:par>Once upon a time, ...</p:par>
    <p:par>And then ...</p:par>
  </p:section>
  <p:section>
    <p:title>The hog</p:title>
    <p:par>Sooner or later ...</p:par>
  </p:section>
</p:doc>
```

One such example is the module `lxml.html.builder`, which provides a vocabulary for HTML.

When dealing with multiple namespaces, it is good practice to define one ElementMaker for each namespace URI. Again, note how the above example predefines the tag builders in named constants. That makes it easy to put all tag declarations of a namespace into one Python module and to import/use the tag name constants from there. This avoids pitfalls like typos or accidentally missing namespaces.

ElementPath

The ElementTree library comes with a simple XPath-like path language called ElementPath. The main difference is that you can use the `{namespace}tag` notation in ElementPath expressions. However, advanced features like value comparison and functions are not available.

In addition to a full XPath implementation, `lxml.etree` supports the ElementPath language in the same way ElementTree does, even using (almost) the same implementation. The API provides four methods here that you can find on Elements and ElementTrees:

- `iterfind()` iterates over all Elements that match the path expression
- `findall()` returns a list of matching Elements
- `find()` efficiently returns only the first match
- `findtext()` returns the `.text` content of the first match

Here are some examples:

```
>>> root = etree.XML("<root><a x='123'>aText<b/><c/><b/></a></root>")
```

Find a child of an Element:

```
>>> print(root.find("b"))
None
>>> print(root.find("a").tag)
a
```

Find an Element anywhere in the tree:

```
>>> print(root.find(".//b").tag)
b
>>> [ b.tag for b in root.iterfind(".//b") ]
['b', 'b']
```

Find Elements with a certain attribute:

```
>>> print(root.findall(".//a[@x]")[0].tag)
a
>>> print(root.findall(".//a[@y]"))
[]
```

In lxml 3.4, there is a new helper to generate a structural ElementPath expression for an Element:

```
>>> tree = etree.ElementTree(root)
>>> a = root[0]
>>> print(tree.getelementpath(a[0]))
a/b[1]
>>> print(tree.getelementpath(a[1]))
a/c
```

```
>>> print(tree.getelementpath(a[2]))
a/b[2]
>>> tree.find(tree.getelementpath(a[2])) == a[2]
True
```

As long as the tree is not modified, this path expression represents an identifier for a given element that can be used to `find()` it in the same tree later. Compared to XPath, ElementPath expressions have the advantage of being self-contained even for documents that use namespaces.

The `.iter()` method is a special case that only finds specific tags in the tree by their name, not based on a path. That means that the following commands are equivalent in the success case:

```
>>> print(root.find(".//b").tag)
b
>>> print(next(root.iterfind(".//b")).tag)
b
>>> print(next(root.iter("b")).tag)
b
```

Note that the `.find()` method simply returns None if no match is found, whereas the other two examples would raise a `StopIteration` exception.

Generated on: 2017-01-08.