Generating XML

Python's support for XML is not limited to parsing existing documents. You can also create XML documents from scratch.

```
import xml.etree.ElementTree as etree
new_feed = etree.Element('{http://www.w3.org/2005/Atom}feed',     #①
     attrib={'{http://www.w3.org/XML/1998/namespace}lang': 'en'}) #②
print(etree.tostring(new_feed))                                   #③
#<ns0:feed xmlns:ns0='http://www.w3.org/2005/Atom' xml:lang='en'/>
```

① To create a new element, instantiate the `Element` class. You pass the element name (namespace + local name) as the first argument. This statement creates a `feed` element in the Atom namespace. This will be our new document's root element.

② To add attributes to the newly created element, pass a dictionary of attribute names and values in the `attrib` argument. Note that the attribute name should be in the standard ElementTree format, `{namespace}localname`.

③ At any time, you can serialize any element (and its children) with the ElementTree `tostring()` function.

Was that serialization surprising to you? The way ElementTree serializes namespaced XML elements is technically accurate but not optimal. The sample xml document at the beginning of this chapter defined a *default namespace* `(xmlns='http://www.w3.org/2005/Atom')`. Defining a default namespace is useful for documents — like Atom feeds — where every element is in the same namespace, because you can declare the namespace once and declare each element with just its local name (`<feed>`, `<link>`, `<entry>`). There is no need to use any prefixes unless you want to declare elements from another namespace.

An XML parser won't "see" any difference between an XML document with a default namespace and an XML document with a prefixed namespace. The resulting dom of this serialization:

```
<ns0:feed xmlns:ns0='http://www.w3.org/2005/Atom' xml:lang='en'/>
```

is identical to the DOM of this serialization:

```
<feed xmlns='http://www.w3.org/2005/Atom' xml:lang='en'/>
```

The only practical difference is that the second serialization is several characters shorter. If we were to recast our entire sample feed with a `ns0:` prefix in every start and end tag, it would add 4 characters per start tag × 79 tags + 4 characters for the namespace declaration itself, for a total of 320 characters. Assuming UTF-8 encoding, that's 320 extra bytes. (After gzipping, the difference drops to 21 bytes, but still, 21 bytes is 21 bytes.) Maybe that doesn't matter to you, but for something like an Atom feed, which may be downloaded several thousand times whenever it changes, saving a few bytes per request can quickly add up.

The built-in ElementTree library does not offer this fine-grained control over serializing namespaced elements, but `lxml` does.

```
import lxml.etree

NSMAP = {None: 'http://www.w3.org/2005/Atom'}                    #①
new_feed = lxml.etree.Element('feed', nsmap=NSMAP)               #②
print(lxml.etree.tounicode(new_feed))                            #③
#<feed xmlns='http://www.w3.org/2005/Atom'/>

new_feed.set('{http://www.w3.org/XML/1998/namespace}lang', 'en')  #④
print(lxml.etree.tounicode(new_feed))
#<feed xmlns='http://www.w3.org/2005/Atom' xml:lang='en'/>
```

① To start, define a namespace mapping as a dictionary. Dictionary values are namespaces; dictionary keys are the desired prefix. Using `None` as a prefix effectively declares a default namespace.

effectively declares a default namespace.

② Now you can pass the `lxml`-specific `nsmap` argument when you create an element, and `lxml` will respect the namespace prefixes you've defined.

③ As expected, this serialization defines the Atom namespace as the default namespace and declares the `feed` element without a namespace prefix.

④ Oops, we forgot to add the `xml:lang` attribute. You can always add attributes to any element with the `set()` method. It takes two arguments: the attribute name in standard ElementTree format, then the attribute value. (This method is not `lxml`-specific. The only `lxml`-specific part of this example was the `nsmap` argument to control the namespace prefixes in the serialized output.)

Are XML documents limited to one element per document? No, of course not. You can easily create child elements, too.

```
import lxml.etree
NSMAP = {None: 'http://www.w3.org/2005/Atom'}
new_feed = lxml.etree.Element('feed', nsmap=NSMAP)

title = lxml.etree.SubElement(new_feed, 'title',                                #
     attrib={'type':'html'})                                                    #
print(lxml.etree.tounicode(new_feed))                                           #
#<feed xmlns="http://www.w3.org/2005/Atom"><title type="html"/></feed>          #

print(lxml.etree.tounicode(new_feed))                                           #
#<feed xmlns="http://www.w3.org/2005/Atom"><title type="html"/></feed>

print(lxml.etree.tounicode(new_feed, pretty_print=True))                        #
#<feed xmlns="http://www.w3.org/2005/Atom">
#  <title type="html"/>
#</feed>
```

① To create a child element of an existing element, instantiate the `SubElement` class. The only required arguments are the parent element (`new_feed` in this case) and the new element's name. Since this child element will inherit the namespace mapping of its parent, there is no need to redeclare the namespace or prefix here.

② You can also pass in an attribute dictionary. Keys are attribute names; values are attribute values.

values are attribute values.

③ As expected, the new `title` element was created in the Atom namespace, and it was inserted as a child of the `feed` element. Since the `title` element has no text content and no children of its own, `lxml` serializes it as an empty element (with the /> shortcut).

④ To set the text content of an element, simply set its `.text` property.

⑤ Now the `title` element is serialized with its text content. Any text content that contains less-than signs or ampersands needs to be escaped when serialized. `lxml` handles this escaping automatically.

⑥ You can also apply "pretty printing" to the serialization, which inserts line breaks after end tags, and after start tags of elements that contain child elements but no text content. In technical terms, `lxml` adds "insignificant whitespace" to make the output more readable.

> *You might also want to check out xmlwitch, another third-party library for generating XML. It makes extensive use of the with statement to make XML generation code more readable.*