Chapter 23. Structured Text: XML

XML, the *eXtensible Markup Language*, is a widely used data interchange format. On top of XML itself, the XML community (in good part within the World Wide Web Consortium, or W3C) has standardized many other technologies, such as schema languages, namespaces, XPath, XLink, XPointer, and XSLT.

Industry consortia have additionally defined industry-specific markup languages on top of XML for data exchange among applications in their respective fields. XML, XML-based markup languages, and other XML-related technologies are often used for inter-application, cross-language, cross-platform data interchange in specific fields.

Python's standard library, for historical reasons, has multiple modules supporting XML under the `xml` package, with overlapping functionality; this book does not cover them all, but interested readers can find details in the **online documentation**.

This book (and, specifically, this chapter) covers only the most Pythonic approach to XML processing: `ElementTree`, created by the **deeply missed Fredrik Lundh**, best known as "the effbot."[1] Its elegance, speed, generality, multiple implementations, and Pythonic architecture make this the package of choice for Python XML applications. For tutorials and complete details on the `xml.etree.ElementTree` module beyond what this chapter provides, see the **online docs**. This book takes for granted some elementary knowledge of XML itself; if you need to learn more about XML, we recommend ***XML in a Nutshell*** by Elliotte Rusty Harold and W. Scott Means (O'Reilly).

Parsing XML from untrusted sources puts your application at risk of many possible attacks. We do not cover this issue specifically, but the **on-**

line documentation recommends third-party modules to help safeguard your application if you do have to parse XML from sources you can't fully trust. In particular, if you need an `ElementTree` implementation with safeguards against parsing untrusted sources, consider `defusedxml.ElementTree`.

ElementTree

Python and third-party add-ons offer several alternative implementations of the `ElementTree` functionality; the one you can always rely on in the standard library is the module `xml.etree.ElementTree`. Just importing `xml.etree.ElementTree` gets you the fastest implementation available in your Python installation's standard library. The third-party package `defusedxml`, mentioned in this chapter's introduction, offers slightly slower but safer implementations if you ever need to parse XML from untrusted sources; another third-party package, `lxml`, gets you faster performance, and some extra functionality, via `lxml.etree`.

Traditionally, you get whatever available implementation of `ElementTree` you prefer using a **`from...import...as`** statement such as this:

```
from xml.etree import ElementTree as et
```

Or this, which tries to import `lxml` and, if unable, falls back to the version provided in the standard library:

```
try:
    from lxml import etree as et
except ImportError:
    from xml.etree import ElementTree as et
```

Once you succeed in importing an implementation, use it as `et` (some prefer the uppercase variant, `ET`) in the rest of your code.

`ElementTree` supplies one fundamental class representing a *node* within the *tree* that naturally maps an XML document: the class `Element`. `ElementTree` also supplies other important classes, chiefly the one representing the whole tree, with methods for input and output and many convenience classes equivalent to ones on its `Element` *root*—that's the class `ElementTree`. In addition, the `ElementTree` module supplies several utility functions, and auxiliary classes of lesser importance.

The Element Class

The `Element` class represents a node in the tree that maps an XML document, and it's the core of the whole `ElementTree` ecosystem. Each element is a bit like a mapping, with *attributes* that map string keys to string values, and also a bit like a sequence, with *children* that are other elements (sometimes referred to as the element's "subelements"). In addition, each element offers a few extra attributes and methods. Each `Element` instance *`e`* has four data attributes or properties, detailed in Table 23-1.

Table 23-1. Attributes of an `Element` instance `e`

`attrib`	A `dict` containing all of the XML node's attributes, with strings, the attributes' names, as its keys (and, usually, strings as corresponding values as well). For example, parsing the XML fragment `<a x="y">b</a>c`, you get an *`e`* whose *`e`*`.attrib` is `{'x': 'y'}`. **AVOID ACCESSING ATTRIB ON ELEMENT INSTANCES** It's normally best to avoid accessing *`e`*`.attrib` when possible, because the implementation might need to build it on the fly when you access it. *`e`* itself offers some typical mapping methods (listed in Table 23-2) that you might otherwise want to call on *`e`*`.attrib`; going through *`e`*'s own methods allows an implementation to optimize things for you, compared to the performance you'd get via the actual `dict` *`e`*`.attrib`.

`tag`	The XML tag of the node: a string, sometimes also known as the element's *type*. For example, parsing the XML fragment `<a x="y">b</a>c`, you get an *`e`* with *`e`*`.tag` set to `'a'`.
`tail`	Arbitrary data (a string) immediately "following" the element. For example, parsing the XML fragment `<a x="y">b</a>c`, you get an *`e`* with *`e`*`.tail` set to `'c'`.
`text`	Arbitrary data (a string) directly "within" the element. For example, parsing the XML fragment `<a x="y">b</a>c`, you get an *`e`* with *`e`*`.text` set to `'b'`.

`e` has some methods that are mapping-like and avoid the need to explicitly ask for the *`e`*`.attrib dict`. These are listed in Table 23-2.

Table 23-2. Mapping-like methods of an `Element` instance e

`clear`	*`e`*`.clear()` Leaves *`e`* "empty," except for its `tag`, removing all attributes and children, and setting `text` and `tail` to **`None`**.
`get`	*`e`*`.get(`*`key`*`, default=`**`None`**`)` Like *`e`*`.attrib.get(`*`key, default`*`)`, but potentially much faster. You cannot use *`e`*`[`*`key`*`]`, since indexing on *`e`* is used to access children, not attributes.
`items`	*`e`*`.items()` Returns the list of `(`*`name, value`*`)` tuples for all attributes, in arbitrary order.
`keys`	*`e`*`.keys()` Returns the list of all attribute names, in arbitrary order.

`set`	`e.set(key, value)` Sets the value of the attribute named *`key`* to *`value`*.

The other methods of *e* (including methods for indexing with the `e[i]` syntax and for getting the length, as in `len(e)`) deal with all of *e*'s children as a sequence, or in some cases—indicated in the rest of this section—with all descendants (elements in the subtree rooted at *e*, also known as subelements of *e*).

DON'T RELY ON IMPLICIT BOOL CONVERSION OF AN ELEMENT

In all versions up through Python 3.11, an `Element` instance *e* evaluates as false if *e* has no children, following the normal rule for Python containers' implicit `bool` conversion. However, it is documented that this behavior may change in some future version of Python. For future compatibility, if you want to check whether *e* has no children, explicitly check `if len(e) == 0:` instead of using the normal Python idiom `if not e:`.

The named methods of *e* dealing with children or descendants are listed in **Table 23-3** (we do not cover XPath in this book: see the **online docs** for information on that topic). Many of the following methods take an optional argument `namespaces`, defaulting to **`None`**. When present, `namespaces` is a mapping with XML namespace prefixes as keys and corresponding XML namespace full names as values.

Table 23-3. Methods of an `Element` instance e dealing with children or descendants

`append`	`e.append(se)` Adds subelement *`se`* (which must be an `Element`) at the end of *e*'s children.
`extend`	`e.extend(ses)` Adds each item of iterable *`ses`* (every item must be an `Element`) at the end of *e*'s children.

`find`	`e.find(`*`match`*`, namespaces=`**`None`**`)` Returns the first descendant matching *`match`*, which may be a tag name or an XPath expression within the subset supported by the current implementation of `ElementTree`. Returns **`None`** if no descendant matches *`match`*.
`findall`	`e.findall(`*`match`*`, namespaces=`**`None`**`)` Returns the list of all descendants matching *`match`*, which may be a tag name or an XPath expression within the subset supported by the current implementation of `ElementTree`. Returns `[]` if no descendants match *`match`*.
`findtext`	`e.findtext(`*`match`*`, default=`**`None`**`, namespaces=`**`None`**`)` Returns the `text` of the first descendant matching *`match`*, which may be a tag name or an XPath expression within the subset supported by the current implementation of `ElementTree`. The result may be an empty string, `''`, if the first descendant matching *`match`* has no `text`. Returns `default` if no descendant matches *`match`*.
`insert`	`e.insert(`*`index`*`, `*`se`*`)` Adds subelement *`se`* (which must be an `Element`) at index *`index`* within the sequence of *e*'s children.
`iter`	`e.iter(`*`tag`*`='*')` Returns an iterator walking in depth-first order over all of *e*'s descendants. When *`tag`* is not `'*'`, only yields subelements whose `tag` equals *`tag`*. Don't modify the subtree rooted at *`e`* while you're looping on `e.iter`.
`iterfind`	`e.iterfind(`*`match`*`, namespaces=`**`None`**`)` Returns an iterator over all descendants, in depth-first order, matching *`match`*, which may be a tag name or an XPath expression within the subset supported by the

	current implementation of `ElementTree`. The resulting iterator is empty when no descendants match *`match`*.
`itertext`	*`e`*`.itertext(`*`match`*`, namespaces=`**`None`**`)` Returns an iterator over the `text` (not the `tail`) attribute of all descendants, in depth-first order, matching *`match`*, which may be a tag name or an XPath expression within the subset supported by the current implementation of `ElementTree`. The resulting iterator is empty when no descendants match *`match`*.
`remove`	*`e`*`.remove(`*`se`*`)` Removes the descendant that **`is`** element *`se`* (as covered in **Table 3-4**).

The ElementTree Class

The `ElementTree` class represents a tree that maps an XML document. The core added value of an instance *`et`* of `ElementTree` is to have methods for wholesale parsing (input) and writing (output) of a whole tree. These methods are described in **Table 23-4**.

Table 23-4. `ElementTree` instance parsing and writing methods

`parse`	*`et`*`.parse(`*`source`*`, parser=`**`None`**`)` *`source`* can be a file open for reading, or the name of a file to open and read (to parse a string, wrap it in `io.StringIO`, covered in **"In-Memory Files: io.StringIO and io.BytesIO"**), containing XML text. *`et`*`.parse` parses that text, builds its tree of `Element`s as the new content of *`et`* (discarding the previous content of *`et`*, if any), and returns the root element of the tree. `parser` is an optional parser instance; by default, *`et`*`.parse` uses an instance of class `XMLParser` supplied by the `ElementTree` module (this book does not cover `XMLParser`; see the **online docs**).

`write`	`*et*.write(*file*, encoding='us-ascii', xml_declaration=**None**, default_namespace=**None**, method='xml', short_empty_elements=True)` *`file`* can be a file open for writing, or the name of a file to open and write (to write into a string, pass as *`file`* an instance of `io.StringIO`, covered in **“In-Memory Files: io.StringIO and io.BytesIO”**). *`et`*`.write` writes into that file the text representing the XML document for the tree that is the content of *`et`*.
`write` *(cont.)*	*`encoding`* should be spelled according to the **standard**, not by using common “nicknames”—for example, `'iso-8859-1'`, not `'latin-1'`, even though Python itself accepts both spellings for this encoding, and similarly `'utf-8'`, with the dash, not `'utf8'`, without it. The best choice often is to pass `encoding` as `'unicode'`. This outputs text (Unicode) strings, when *`file`*`.write` accepts such strings; otherwise, *`file`*`.write` must accept bytestrings, and that will be the type of strings that *`et`*`.write` outputs, using XML character references for characters not in the encoding—for example, with the default US-ASCII encoding, “e with an acute accent,” `é`, is output as `&#233;`. You can pass `xml_declaration` as **`False`** to not have the declaration in the resulting text, or as **`True`** to have it; the default is to have the declaration in the result only when `encoding` is not one of `'us-ascii'`, `'utf-8'`, or `'unicode'`. You can optionally pass `default_namespace` to set the default namespace for `xmlns` constructs. You can pass `method` as `'text'` to output only the `text` and `tail` of each node (no tags). You can pass `method` as `'html'` to output the document in HTML format (which, for example, omits end tags not needed in HTML, such as `</br>`). The default is `'xml'`, to output in XML format.

You can optionally (only by name, not positionally) pass `short_empty_elements` as **`False`** to always use explicit start and end tags, even for elements that have no text or subelements; the default is to use the XML short form for such empty elements. For example, an empty element with tag `a` is output as `<a/>` by default, or as `<a></a>` if you pass `short_empty_elements` as **`False`**.

In addition, an instance *`et`* of `ElementTree` supplies the method `getroot` (to return the root of the tree) and the convenience methods `find`, `findall`, `findtext`, `iter`, and `iterfind`, each exactly equivalent to calling the same method on the root of the tree—that is, on the result of *`et`*`.getroot`.

Functions in the ElementTree Module

The `ElementTree` module also supplies several functions, described in Table 23-5.

Table 23-5. `ElementTree` functions

`Comment`	`Comment(text=`**`None`**`)` Returns an `Element` that, once inserted as a node in an `ElementTree`, will be output as an XML comment with the given `text` string enclosed between `'<!--'` and `'-->'`. `XMLParser` skips XML comments in any document it parses, so this function is the only way to insert comment nodes.
`dump`	`dump(`*`e`*`)` Writes `e`, which can be an `Element` or an `ElementTree`, as XML to `sys.stdout`. This function is meant only for debugging purposes.
`fromstring`	`fromstring(`*`text`*`, parser=`**`None`**`)` Parses XML from the *`text`* string and returns an

	`Element`, just like the XML function just covered.
`fromstringlist`	`fromstringlist(`*`sequence`*`, parser=`**`None`**`)` Just like `fromstring(''.join(`*`sequence`*`))`, but can be a bit faster by avoiding the `join`.
`iselement`	`iselement(`*`e`*`)` Returns **`True`** if *`e`* is an `Element`; otherwise, returns **`False`**.
`iterparse`	`iterparse(`*`source`*`, events=['end'], parser=`**`None`**`)` Parses an XML document and incrementally builds the corresponding `ElementTree`. *`source`* can be a file open for reading, or the name of a file to open and read, containing an XML document as text. `iterparse` returns an iterator yielding two-item tuples `(`*`event`*`, `*`element`*`)`, where *`event`* is one of the strings listed in the argument `events` (each string must be `'start'`, `'end'`, `'start-ns'`, or `'end-ns'`), as the parsing progresses. *`element`* is an `Element` for events `'start'` and `'end'`, **`None`** for event `'end-ns'`, and a tuple of two strings `(`*`namespace_prefix`*`, `*`namespace_uri`*`)` for event `'start-ns'`. `parser` is an optional parser instance; by default, `iterparse` uses an instance of the class `XMLParser` supplied by the `ElementTree` module (see the **online docs** for details on the `XMLParser` class). The purpose of `iterparse` is to let you iteratively parse a large XML document, without holding all of the resulting `ElementTree` in memory at once, whenever feasible. We cover `iterparse` in more detail in **"Parsing XML Iteratively"**.

`parse`	`parse(`*`source`*`, parser=`**`None`**`)` Just like the `parse` method of `ElementTree`, covered in Table 23-4, except that it returns the `ElementTree` instance it creates.
`Processing Instruction`	`ProcessingInstruction(`*`target`*`, text=`**`None`**`)` Returns an `Element` that, once inserted as a node in an `ElementTree`, will be output as an XML processing instruction with the given *`target`* and `text` strings enclosed between `'<?'` and `'?>'`. `XMLParser` skips XML processing instructions in any document it parses, so this function is the only way to insert processing instruction nodes.
`register_ namespace`	`register_namespace(`*`prefix`*`, `*`uri`*`)` Registers the string *`prefix`* as the namespace prefix for the string *`uri`*; elements in the namespace get serialized with this prefix.
`SubElement`	`SubElement(`*`parent`*`, `*`tag`*`, attrib={}, **`*`extra`*`)` Creates an `Element` with the given *`tag`*, attributes from `dict` `attrib`, and others passed as named arguments in *`extra`*, and appends it as the rightmost child of `Element` *`parent`*; returns the `Element` it has created.
`tostring`	`tostring(`*`e`*`, encoding='us-ascii, method='xml', short_empty_elements=`**`True`**`)` Returns a string with the XML representation of the subtree rooted at `Element` *`e`*. Arguments have the same meaning as for the `write` method of `ElementTree`, covered in Table 23-4.
`tostringlist`	`tostringlist(`*`e`*`, encoding='us-ascii', method='xml', short_empty_elements=`**`True`**`)`

	Returns a list of strings with the XML representation of the subtree rooted at `Element` *`e`*. Arguments have the same meaning as for the `write` method of `ElementTree`, covered in **Table 23-4**.
`XML`	`XML(`*`text`*`, parser=`**`None`**`)` Parses XML from the *`text`* string and returns an `Element`. `parser` is an optional parser instance; by default, `XML` uses an instance of the class `XMLParser` supplied by the `ElementTree` module (this book does not cover the `XMLParser` class; see the **online docs** for details).
`XMLID`	`XMLID(`*`text`*`, parser=`**`None`**`)` Parses XML from the *`text`* string and returns a tuple with two items: an `Element` and a `dict` mapping `id` attributes to the only `Element` having each (XML forbids duplicate `id`s). `parser` is an optional parser instance; by default, `XMLID` uses an instance of the class `XMLParser` supplied by the `ElementTree` module (this book does not cover the `XMLParser` class; see the **online docs** for details).

The `ElementTree` module also supplies the classes `QName`, `TreeBuilder`, and `XMLParser`, which we do not cover in this book, and the class `XMLPullParser`, covered in **“Parsing XML Iteratively”**.

Parsing XML with ElementTree.parse

In everyday use, the most common way to make an `ElementTree` instance is by parsing it from a file or file-like object, usually with the module function `parse` or with the method `parse` of instances of the class `ElementTree`.

For the examples in the remainder of this chapter, we use the simple XML file found at ***http://www.w3schools.com/xml/simple.xml***; its root tag is `'breakfast_menu'`, and the root's children are elements with the tag `'food'`. Each `'food'` element has a child with the tag `'name'`, whose text is the food's name, and a child with the tag `'calories'`, whose text is the string representation of the integer number of calories in a portion of that food. In other words, a simplified representation of that XML file's content of interest to the examples is:

```
<breakfast_menu>
  <food>
    <name>Belgian Waffles</name>
    <calories>650</calories>
  </food>
  <food>
    <name>Strawberry Belgian Waffles</name>
    <calories>900</calories>
  </food>
  <food>
    <name>Berry-Berry Belgian Waffles</name>
    <calories>900</calories>
  </food>
  <food>
    <name>French Toast</name>
    <calories>600</calories>
  </food>
  <food>
    <name>Homestyle Breakfast</name>
    <calories>950</calories>
  </food>
</breakfast_menu>
```

Since the XML document lives at a WWW URL, you start by obtaining a file-like object with that content, and passing it to `parse`; the simplest way uses the `urllib.request` module:

```
from urllib import request
from xml.etree import ElementTree as et
```

```
content = request.urlopen('http://www.w3schools.com/xml/simple.xml')
tree = et.parse(content)
```

Selecting Elements from an ElementTree

Let’s say that we want to print on standard output the calories and names of the various foods, in order of increasing calories, with ties broken alphabetically. Here’s the code for this task:

```
def bycal_and_name(e):
    return int(e.find('calories').text), e.find('name').text

for e in sorted(tree.findall('food'), key=bycal_and_name):
    print(f"{e.find('calories').text} {e.find('name').text}")
```

When run, this prints:

```
600 French Toast
650 Belgian Waffles
900 Berry-Berry Belgian Waffles
900 Strawberry Belgian Waffles
950 Homestyle Breakfast
```

Editing an ElementTree

Once an `ElementTree` is built (be that via parsing, or otherwise), you can “edit” it—inserting, deleting, and/or altering nodes (elements)—via various methods of the `ElementTree` and `Element` classes, and module functions. For example, suppose our program is reliably informed that a new food has been added to the menu—buttered toast, two slices of white bread toasted and buttered, 180 calories—while any food whose name contains “berry,” case insensitive, has been removed. The “editing the tree” part for these specs can be coded as follows:

```
# add Buttered Toast to the menu
menu = tree.getroot()
toast = et.SubElement(menu, 'food')
tcals = et.SubElement(toast, 'calories')
tcals.text = '180'
tname = et.SubElement(toast, 'name')
tname.text = 'Buttered Toast'
# remove anything related to 'berry' from the menu
for e in menu.findall('food'):
    name = e.find('name').text
    if 'berry' in name.lower():
        menu.remove(e)
```

Once we insert these "editing" steps between the code parsing the tree and the code selectively printing from it, the latter prints:

```
180 Buttered Toast
600 French Toast
650 Belgian Waffles
950 Homestyle Breakfast
```

The ease of editing an `ElementTree` can sometimes be a crucial consideration, making it worth your while to keep it all in memory.

Building an ElementTree from Scratch

Sometimes, your task doesn't start from an existing XML document: rather, you need to make an XML document from data your code gets from a different source, such as a CSV file or some kind of database.

The code for such tasks is similar to the code we showed for editing an existing `ElementTree`—just add a little snippet to build an initially empty tree.

For example, suppose you have a CSV file, *menu.csv*, whose two comma-separated columns are the calories and names of various foods, one food per row. Your task is to build an XML file, *menu.xml*, similar to the one we parsed in the previous examples. Here's one way you could do that:

```
import csv
from xml.etree import ElementTree as et

menu = et.Element('menu')
tree = et.ElementTree(menu)
with open('menu.csv') as f:
    r = csv.reader(f)
    for calories, namestr in r:
        food = et.SubElement(menu, 'food')
        cals = et.SubElement(food, 'calories')
        cals.text = calories
        name = et.SubElement(food, 'name')
        name.text = namestr

tree.write('menu.xml')
```

Parsing XML Iteratively

For tasks focused on selecting elements from an existing XML document, sometimes you don't need to build the whole `ElementTree` in memory—a consideration that's particularly important if the XML document is very large (not the case for the tiny example document we've been dealing with, but stretch your imagination and visualize a similar menu-focused document that lists millions of different foods).

Suppose we have such a large document, and we want to print on standard output the calories and names of the 10 lowest-calorie foods, in order of increasing calories, with ties broken alphabetically. Our *menu.xml* file, which for simplicity's sake we'll assume is now a local file, lists millions of foods, so we'd rather not keep it all in memory (obviously, we don't need complete access to all of it at once).

The following code represents a naive attempt to parse without building the whole structure in memory:

```python
import heapq
from xml.etree import ElementTree as et

def cals_and_name():
    # generator for (calories, name) pairs
    for _, elem in et.iterparse('menu.xml'):
        if elem.tag != 'food':
            continue
        # just finished parsing a food, get calories and name
        cals = int(elem.find('calories').text)
        name = elem.find('name').text
        yield (cals, name)

lowest10 = heapq.nsmallest(10, cals_and_name())

for cals, name in lowest10:
    print(cals, name)
```

This approach does indeed work, but unfortunately it consumes just about as much memory as an approach based on a full `et.parse` would! This is because `iterparse` builds up a whole `ElementTree` in memory, even though it only communicates back events such as (and by default only) `'end'`, meaning “I just finished parsing this element.”

To actually save memory, we can at least toss all the contents of each element as soon as we’re done processing it—that is, right after the **`yield`**, we can add `elem.clear()` to make the just-processed element empty.

This approach would indeed save some memory—but not all of it, because the tree’s root would still end up with a huge list of empty child nodes. To be really frugal in memory consumption, we need to get `'start'` events as well, so we can get hold of the root of the `ElementTree` being built and remove each element from it as it’s used, rather than just clearing the element. That is, we want to change the generator into:

```
def cals_and_name():
    # memory-thrifty generator for (calories, name) pairs
    root = None
    for event, elem in et.iterparse('menu.xml', ['start', 'end']):
        if event == 'start':
            if root is None:
                root = elem
            continue
        if elem.tag != 'food':
            continue
        # just finished parsing a food, get calories and name
        cals = int(elem.find('calories').text)
        name = elem.find('name').text
        yield (cals, name)
        root.remove(elem)
```

This approach saves as much memory as feasible, and still gets the task done!

PARSING XML WITHIN AN ASYNCHRONOUS LOOP

While `iterparse`, used correctly, can save memory, it's still not good enough to use within an asynchronous loop. That's because `iterparse` makes blocking `read` calls to the file object passed as its first argument: such blocking calls are a no-no in async processing.

`ElementTree` offers the class `XMLPullParser` to help with this issue; see the **online docs** for the class's usage pattern.

---

[1] Alex is far too modest to mention it, but from around 1995 to 2005 both he and Fredrik were, along with Tim Peters, *the* Python bots. Known as such for their encyclopedic and detailed knowledge of the language, the effbot, the martellibot, and the timbot have created software and documentation that are of immense value to millions of people.