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`. In most circumstances, in v2, you can use the faster C-coded implementation `xml.etree.cElementTree`; in v3, just importing `xml.etree.ElementTree` gets you the fastest implementation available. The third-party package `defusedxml`, mentioned in the previous section of this chapter, 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, by a `from...import...as` statement such as:

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

(or more than one such statement, with `try...except ImportError:` guards to discover what's the best implementation available), then use `et` (some prefer the uppercase variant, `ET`) as the module's name 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 ones 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 are a mapping from string keys to string values, and 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:

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, if feasible

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, as covered later in this section, offers some typical mapping methods that you might otherwise want to call on `e.attrib`; going through `e`'s own methods allows a smart 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`:

clear	`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 attribute named `key` to `value`.

The other methods of `e` (including indexing with the `e[i]` syntax, and length as in `len(e)`) deal with all `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 to Python 3.6, an `Element` instance `e` tests as false if `e` has no children, following the normal rule for Python containers' implicit `bool` conversion. However, it's documented that this behavior may change in some future version of v3. For future compatibility, if you want to check whether `e` has no children, explicitly check `if len(e) == 0:`—don't use the normal Python idiom `if not e:`.

The named methods of `e` dealing with children or descendants are the following (we do not cover XPath in this book: see the online docs):

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`. In v3 only, `namespaces` is an optional mapping with XML namespace prefixes as keys and corresponding XML namespace full names as values.
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`. In v3, only, `namespaces` is an optional mapping with XML namespace prefixes as keys and corresponding XML namespace full names as values.
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`. In v3, only, `namespaces` is an optional mapping with XML namespace prefixes as keys and corresponding XML namespace full names as values.
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`. In v3 only, `namespaces` is an optional mapping with XML namespace prefixes as keys and corresponding XML namespace full names as values.
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`. In v3 only, `namespaces` is an optional mapping with XML namespace prefixes as keys and corresponding XML namespace full names as values.
remove	`e.remove(se)`	Removes the descendant that `is` element `se` (as covered in Identity tests, in Table 3-2).

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, namely:

Table 23-1. Elementree 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's the content of `et`. `encoding` should be spelled according to the standard—for example, `'iso-8859-1'`, not `'latin-1'`, even though Python itself accepts both spellings for this encoding. In v3 only, you can pass `encoding` as `'unicode'` to output text (Unicode) strings, if *`file`*`.write` accepts such strings; otherwise, *`file`*`.write` must accept bytestrings, and that is the type of strings `et.write` outputs, using XML character references for characters not in the encoding—for example, with the default ASCII encoding, e with an acute accent, `é`, is output as `é`. You can pass `xml_declaration` as `False` to not have the declaration in the resulting text, 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 (v3 only) `'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. In v3 only, 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, as `<a></a>` in v3 if you pass `short_empty_elements` as `False`.

In addition, an instance `et` of `ElementTree` supplies the method `getroot`—*`et`*`.getroot()` returns 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 23-2.

Table 23-2.

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 get comment nodes.
ProcessingInstruction	`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 get processing instruction nodes.
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.
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 class `XMLParser`; see the online docs).
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).
dump	`dump(`*`e`*`)` Writes `e`, which can be an `Element` or an `ElementTree`, as XML to `sys.stdout`; it 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`.

iterparse	`iterparse(`*`source`*`,events=['end'],parser=None)` *`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 tuples `(`*`event, element`*`)`, where `event` is one of the strings listed in argument `events` (which must be `'start'`, `'end'`, `'start-ns'`, or `'end-ns'`), as the parsing progresses and `iterparse` incrementally builds the corresponding `ElementTree`. `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 (this book does not cover class `XMLParser`; see the online docs). 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 23-1, except that it returns the `ElementTree` instance it creates.
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.
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 23-1.
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 `Elemente`. Arguments have the same meaning as for the `write` method of `ElementTree`, covered in Table 23-1.

The `ElementTree` module also supplies the classes `QName`, `TreeBuilder`, and `XMLParser`, which we do not cover in this book. In v3 only, it also supplies 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 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`; in v2, the simplest way is:

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

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

and similarly, in v3, the simplest way uses the `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. The code for this task is the same in v2 and v3:

```
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('{} {}'.format(e.find('calories').text,
                       e.find('name').text))
```

When run, this prints:

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

Editing an ElementTree

Once an `ElementTree` is built (be that via parsing, or otherwise), it can be "edited"—inserting, deleting, and/or altering nodes (elements)—via the various methods of `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')
             'Buttered
tname.text = 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      600 French      650 Belgian        950 Homestyle
Toast             Toast           Waffles            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 document or some kind of database.

The code for such tasks is similar to the one 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 name of various foods, one food per row. Your task is to build an XML file, *menu.xml*, similar to the one we parsed in 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).

So, again, what we want to do is print on standard output the calories and names of foods, this time only the 10 lowest-calorie foods, in order of increasing calories, with ties broken alphabetically; and *menu.xml*, which for simplicity's sake we now suppose is a local file, lists millions of foods, so we'd rather not keep it all in memory at once, since obviously we don't need complete access to all of it at once.

Here's some code that one might think would let us ace this task:

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

# initialize the heap with dummy
entries
heap = [(999999, None)] * 10

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
    heapq.heappush(heap, (cals, name))

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

Simple but memory-intensive approach

This approach does indeed work, but it consumes just about as much memory as an approach based on a full `et.parse` would!

Why does the simple approach still eat memory? Because `iterparse`, as it runs, builds up a whole `ElementTree` in memory, incrementally, even though it only communicates back events such as (by default) just `'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 call to `heapq.heappush`, 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 end up with a huge list of empty children 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—that is, change the start of the loop to:

```
root = None
for event, elem in et.iterparse('menu.xml'):
    if event == 'start':
        if root is not None: root = elem
        continue
    if elem.tag != 'food': continue
# etc. as
before
```

and then, right after the call to `heapq.heappush`, add `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 (async) loop, as covered in Chapter 18. 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.

v2's `ElementTree` has no solution to offer to this conundrum. v3 does—specifically, it offers the class `XMLPullParser`. (In v2, you can get this functionality if you use the third-party package `lxml`, thanks to lxml.etree.)

In an async arrangement, as covered in Chapter 18, a typical task is to write a "filter" component, which is fed chunks of bytes as they happen to come from some upstream source, and yields events downstream as they get fully parsed. Here's how `XMLPullParser` lets you write such a "filter" component:

```
from xml.etree import ElementTree as et

def filter(events=None):
    pullparser = et.XMLPullParser(events)
    data = yield
    while data:
        pullparser.feed(data)
        for tup in pullparser.read_events
():
            data = yield tup
    pullparser.close()
    for tup in pullparser.read_events():
        data = yield tup
```

This assumes that `filter` is used via `.send(chunk)` calls to its result (passing new chunks of bytes as they are received), and `yield`s `(event, element)` tuples for the caller to loop on and process. So, essentially, `filter` turns an async stream of chunks of raw bytes into an async stream of `(event, element)` pairs, to be consumed by iteration—a typical design pattern in modern Python's async programming.