20.5. xml.etree.ElementTree — The ElementTree XML API

Source code: Lib/xml/etree/ElementTree.py

The xml.etree.ElementTree module implements a simple and efficient API for parsing and creating XML data.

Changed in version 3.3: This module will use a fast implementation whenever available. The xml.etree.cElementTree module is deprecated.

Warning: The xml.etree.ElementTree module is not secure against maliciously constructed data. If you need to parse untrusted or unauthenticated data see XML vulnerabilities.

20.5.1. Tutorial

This is a short tutorial for using xml.etree.ElementTree (ET in short). The goal is to demonstrate some of the building blocks and basic concepts of the module.

20.5.1.1. XML tree and elements

XML is an inherently hierarchical data format, and the most natural way to represent it is with a tree. ET has two classes for this purpose - ElementTree represents the whole XML document as a tree, and Element represents a single node in this tree. Interactions with the whole document (reading and writing to/from files) are usually done on the ElementTree level. Interactions with a single XML element and its sub-elements are done on the Element level.

20.5.1.2. Parsing XML

We'll be using the following XML document as the sample data for this section:

```
<?xml version="1.0"?>
<data>
    <country name="Liechtenstein">
        <rank>1</rank>
        <year>2008</year>
        <gdppc>141100</gdppc>
        <neighbor name="Austria" direction="E"/>
        <neighbor name="Switzerland" direction="W"/>
    </country>
    <country name="Singapore">
        <rank>4</rank>
        <year>2011</year>
        <gdppc>59900</gdppc>
```

```
        <neighbor name="Malaysia" direction="N"/>
    </country>
    <country name="Panama">
        <rank>68</rank>
        <year>2011</year>
        <gdppc>13600</gdppc>
        <neighbor name="Costa Rica" direction="W"/>
        <neighbor name="Colombia" direction="E"/>
    </country>
</data>
```

We can import this data by reading from a file:

```
import xml.etree.ElementTree as ET
tree = ET.parse('country_data.xml')
root = tree.getroot()
```

Or directly from a string:

```
root = ET.fromstring(country_data_as_string)
```

`fromstring()` parses XML from a string directly into an `Element`, which is the root element of the parsed tree. Other parsing functions may create an `ElementTree`. Check the documentation to be sure.

As an `Element`, `root` has a tag and a dictionary of attributes:

```
>>> root.tag
'data'
>>> root.attrib
{}
```

It also has children nodes over which we can iterate:

```
>>> for child in root:
...     print(child.tag, child.attrib)
...
country {'name': 'Liechtenstein'}
country {'name': 'Singapore'}
country {'name': 'Panama'}
```

Children are nested, and we can access specific child nodes by index:

```
>>> root[0][1].text
'2008'
```

Note: Not all elements of the XML input will end up as elements of the parsed tree. Currently, this module skips over any XML comments, processing instructions, and document type declarations in the input. Nevertheless, trees built using this module's API rather than parsing from XML text can have comments and processing instructions in them; they will be included when generating XML output. A document type declaration may be accessed by passing a custom `TreeBuilder` instance to the `XMLParser` constructor.

20.5.1.3. Pull API for non-blocking parsing

Most parsing functions provided by this module require the whole document to be read at once before returning any result. It is possible to use an `XMLParser` and feed data into it incrementally, but it is a push API that calls methods on a callback target, which is too low-level and inconvenient for most needs. Sometimes what the user really wants is to be able to parse XML incrementally, without blocking operations, while enjoying the convenience of fully constructed `Element` objects.

The most powerful tool for doing this is `XMLPullParser`. It does not require a blocking read to obtain the XML data, and is instead fed with data incrementally with `XMLPullParser.feed()` calls. To get the parsed XML elements, call `XMLPullParser.read_events()`. Here is an example:

```
>>> parser = ET.XMLPullParser(['start', 'end'])
>>> parser.feed('<mytag>sometext')
>>> list(parser.read_events())
[('start', <Element 'mytag' at 0x7fa66db2be58>)]
>>> parser.feed(' more text</mytag>')
>>> for event, elem in parser.read_events():
...     print(event)
...     print(elem.tag, 'text=', elem.text)
...
end
```

The obvious use case is applications that operate in a non-blocking fashion where the XML data is being received from a socket or read incrementally from some storage device. In such cases, blocking reads are unacceptable.

Because it's so flexible, `XMLPullParser` can be inconvenient to use for simpler use-cases. If you don't mind your application blocking on reading XML data but would still like to have incremental parsing capabilities, take a look at `iterparse()`. It can be useful when you're reading a large XML document and don't want to hold it wholly in memory.

20.5.1.4. Finding interesting elements

`Element` has some useful methods that help iterate recursively over all the sub-tree below it (its children, their children, and so on). For example, `Element.iter()`:

```
>>> for neighbor in root.iter('neighbor'):
...     print(neighbor.attrib)
...
{'name': 'Austria', 'direction': 'E'}
{'name': 'Switzerland', 'direction': 'W'}
{'name': 'Malaysia', 'direction': 'N'}
{'name': 'Costa Rica', 'direction': 'W'}
{'name': 'Colombia', 'direction': 'E'}
```

`Element.findall()` finds only elements with a tag which are direct children of the current element. `Element.find()` finds the *first* child with a particular tag, and `Element.text` accesses the element's text content. `Element.get()` accesses the element's attributes:

```
>>> for country in root.findall('country'):
...     rank = country.find('rank').text
...     name = country.get('name')
...     print(name, rank)
...
Liechtenstein 1
Singapore 4
Panama 68
```

More sophisticated specification of which elements to look for is possible by using XPath.

20.5.1.5. Modifying an XML File

`ElementTree` provides a simple way to build XML documents and write them to files. The `ElementTree.write()` method serves this purpose.

Once created, an `Element` object may be manipulated by directly changing its fields (such as `Element.text`), adding and modifying attributes (`Element.set()` method), as well as adding new children (for example with `Element.append()`).

Let's say we want to add one to each country's rank, and add an `updated` attribute to the rank element:

```
>>> for rank in root.iter('rank'):
...     new_rank = int(rank.text) + 1
...     rank.text = str(new_rank)
...     rank.set('updated', 'yes')
...
>>> tree.write('output.xml')
```

Our XML now looks like this:

```
<?xml version="1.0"?>
<data>
    <country name="Liechtenstein">
        <rank updated="yes">2</rank>
        <year>2008</year>
        <gdppc>141100</gdppc>
        <neighbor name="Austria" direction="E"/>
        <neighbor name="Switzerland" direction="W"/>
    </country>
    <country name="Singapore">
        <rank updated="yes">5</rank>
        <year>2011</year>
        <gdppc>59900</gdppc>
        <neighbor name="Malaysia" direction="N"/>
    </country>
    <country name="Panama">
        <rank updated="yes">69</rank>
```

```
        <year>2011</year>
        <gdppc>13600</gdppc>
        <neighbor name="Costa Rica" direction="W"/>
        <neighbor name="Colombia" direction="E"/>
    </country>
</data>
```

We can remove elements using `Element.remove()`. Let's say we want to remove all countries with a rank higher than 50:

```
>>> for country in root.findall('country'):
...     rank = int(country.find('rank').text)
...     if rank > 50:
...         root.remove(country)
...
>>> tree.write('output.xml')
```

Our XML now looks like this:

```
<?xml version="1.0"?>
<data>
    <country name="Liechtenstein">
        <rank updated="yes">2</rank>
        <year>2008</year>
        <gdppc>141100</gdppc>
        <neighbor name="Austria" direction="E"/>
        <neighbor name="Switzerland" direction="W"/>
    </country>
    <country name="Singapore">
        <rank updated="yes">5</rank>
        <year>2011</year>
        <gdppc>59900</gdppc>
        <neighbor name="Malaysia" direction="N"/>
    </country>
</data>
```

20.5.1.6. Building XML documents

The `SubElement()` function also provides a convenient way to create new sub-elements for a given element:

```
>>> a = ET.Element('a')
>>> b = ET.SubElement(a, 'b')
>>> c = ET.SubElement(a, 'c')
>>> d = ET.SubElement(c, 'd')
>>> ET.dump(a)
<a><b /><c><d /></c></a>
```

20.5.1.7. Parsing XML with Namespaces

If the XML input has namespaces, tags and attributes with prefixes in the form `prefix:sometag` get expanded to `{uri}sometag` where the *prefix* is replaced by the full

URI. Also, if there is a default namespace, that full URI gets prepended to all of the non-prefixed tags.

Here is an XML example that incorporates two namespaces, one with the prefix “fictional” and the other serving as the default namespace:

```
<?xml version="1.0"?>
<actors xmlns:fictional="http://characters.example.com"
        xmlns="http://people.example.com">
    <actor>
        <name>John Cleese</name>
        <fictional:character>Lancelot</fictional:character>
        <fictional:character>Archie Leach</fictional:character>
    </actor>
    <actor>
        <name>Eric Idle</name>
        <fictional:character>Sir Robin</fictional:character>
        <fictional:character>Gunther</fictional:character>
        <fictional:character>Commander Clement</fictional:character>
    </actor>
</actors>
```

One way to search and explore this XML example is to manually add the URI to every tag or attribute in the xpath of a `find()` or `findall()`:

```
root = fromstring(xml_text)
for actor in root.findall('{http://people.example.com}actor'):
    name = actor.find('{http://people.example.com}name')
    print(name.text)
    for char in actor.findall('{http://characters.example.com}character
        print(' |-->', char.text)
```

A better way to search the namespaced XML example is to create a dictionary with your own prefixes and use those in the search functions:

```
ns = {'real_person': 'http://people.example.com',
      'role': 'http://characters.example.com'}

for actor in root.findall('real_person:actor', ns):
    name = actor.find('real_person:name', ns)
    print(name.text)
    for char in actor.findall('role:character', ns):
        print(' |-->', char.text)
```

These two approaches both output:

```
John Cleese
 |--> Lancelot
 |--> Archie Leach
Eric Idle
 |--> Sir Robin
 |--> Gunther
 |--> Commander Clement
```

20.5.1.8. Additional resources

See http://effbot.org/zone/element-index.htm for tutorials and links to other docs.

20.5.2. XPath support

This module provides limited support for XPath expressions for locating elements in a tree. The goal is to support a small subset of the abbreviated syntax; a full XPath engine is outside the scope of the module.

20.5.2.1. Example

Here's an example that demonstrates some of the XPath capabilities of the module. We'll be using the `countrydata` XML document from the Parsing XML section:

```
import xml.etree.ElementTree as ET

root = ET.fromstring(countrydata)

# Top-level elements
root.findall(".")

# All 'neighbor' grand-children of 'country' children of the top-level
# elements
root.findall("./country/neighbor")

# Nodes with name='Singapore' that have a 'year' child
root.findall(".//year/..[@name='Singapore']")

# 'year' nodes that are children of nodes with name='Singapore'
root.findall(".//*[@name='Singapore']/year")

# All 'neighbor' nodes that are the second child of their parent
root.findall(".//neighbor[2]")
```

20.5.2.2. Supported XPath syntax

Syntax	Meaning
`tag`	Selects all child elements with the given tag. For example, `spam` selects all child elements named `spam`, and `spam/egg` selects all grandchildren named `egg` in all children named `spam`.
`*`	Selects all child elements. For example, `*/egg` selects all grandchildren named `egg`.
`.`	Selects the current node. This is mostly useful at the beginning of the path, to indicate that it's a relative path.
`//`	Selects all subelements, on all levels beneath the current element. For example, `.//egg` selects all `egg` elements in the entire tree.

Syntax	Meaning
`..`	Selects the parent element. Returns `None` if the path attempts to reach the ancestors of the start element (the element `find` was called on).
`[@attrib]`	Selects all elements that have the given attribute.
`[@attrib='value']`	Selects all elements for which the given attribute has the given value. The value cannot contain quotes.
`[tag]`	Selects all elements that have a child named `tag`. Only immediate children are supported.
`[tag='text']`	Selects all elements that have a child named `tag` whose complete text content, including descendants, equals the given `text`.
`[position]`	Selects all elements that are located at the given position. The position can be either an integer (1 is the first position), the expression `last()` (for the last position), or a position relative to the last position (e.g. `last()-1`).

Predicates (expressions within square brackets) must be preceded by a tag name, an asterisk, or another predicate. `position` predicates must be preceded by a tag name.

20.5.3. Reference

20.5.3.1. Functions

`xml.etree.ElementTree.`**`Comment`**(*text=None*)

Comment element factory. This factory function creates a special element that will be serialized as an XML comment by the standard serializer. The comment string can be either a bytestring or a Unicode string. *text* is a string containing the comment string. Returns an element instance representing a comment.

Note that `XMLParser` skips over comments in the input instead of creating comment objects for them. An `ElementTree` will only contain comment nodes if they have been inserted into to the tree using one of the `Element` methods.

`xml.etree.ElementTree.`**`dump`**(*elem*)

Writes an element tree or element structure to sys.stdout. This function should be used for debugging only.

The exact output format is implementation dependent. In this version, it's written as an ordinary XML file.

elem is an element tree or an individual element.

`xml.etree.ElementTree.`**`fromstring`**(*text*)

Parses an XML section from a string constant. Same as `XML()`. *text* is a string containing XML data. Returns an `Element` instance.

`xml.etree.ElementTree.`**`fromstringlist`**(*sequence*, *parser=None*)

Parses an XML document from a sequence of string fragments. *sequence* is a list or other sequence containing XML data fragments. *parser* is an optional parser instance. If not given, the standard `XMLParser` parser is used. Returns an `Element` instance.

New in version 3.2.

`xml.etree.ElementTree.`**`iselement`**(*element*)

Checks if an object appears to be a valid element object. *element* is an element instance. Returns a true value if this is an element object.

`xml.etree.ElementTree.`**`iterparse`**(*source*, *events=None*, *parser=None*)

Parses an XML section into an element tree incrementally, and reports what's going on to the user. *source* is a filename or file object containing XML data. *events* is a sequence of events to report back. The supported events are the strings `"start"`, `"end"`, `"start-ns"` and `"end-ns"` (the "ns" events are used to get detailed namespace information). If *events* is omitted, only `"end"` events are reported. *parser* is an optional parser instance. If not given, the standard `XMLParser` parser is used. *parser* must be a subclass of `XMLParser` and can only use the default `TreeBuilder` as a target. Returns an iterator providing `(event, elem)` pairs.

Note that while `iterparse()` builds the tree incrementally, it issues blocking reads on *source* (or the file it names). As such, it's unsuitable for applications where blocking reads can't be made. For fully non-blocking parsing, see `XMLPullParser`.

> **Note:** `iterparse()` only guarantees that it has seen the ">" character of a starting tag when it emits a "start" event, so the attributes are defined, but the contents of the text and tail attributes are undefined at that point. The same applies to the element children; they may or may not be present.
> If you need a fully populated element, look for "end" events instead.

Deprecated since version 3.4: The *parser* argument.

`xml.etree.ElementTree.`**`parse`**(*source*, *parser=None*)

Parses an XML section into an element tree. *source* is a filename or file object containing XML data. *parser* is an optional parser instance. If not given, the standard `XMLParser` parser is used. Returns an `ElementTree` instance.

`xml.etree.ElementTree.`**`ProcessingInstruction`**(*target*, *text=None*)

PI element factory. This factory function creates a special element that will be serialized as an XML processing instruction. *target* is a string containing the PI target. *text* is a string containing the PI contents, if given. Returns an element instance, representing a processing instruction.

Note that `XMLParser` skips over processing instructions in the input instead of creating comment objects for them. An `ElementTree` will only contain processing instruction nodes if they have been inserted into to the tree using one of the `Element` methods.

`xml.etree.ElementTree.`**`register_namespace`**(*prefix*, *uri*)

Registers a namespace prefix. The registry is global, and any existing mapping for either the given prefix or the namespace URI will be removed. *prefix* is a namespace prefix. *uri* is a namespace uri. Tags and attributes in this namespace will be serialized with the given prefix, if at all possible.

New in version 3.2.

`xml.etree.ElementTree.`**`SubElement`**(*parent*, *tag*, *attrib={}*, ***extra*)

Subelement factory. This function creates an element instance, and appends it to an existing element.

The element name, attribute names, and attribute values can be either bytestrings or Unicode strings. *parent* is the parent element. *tag* is the subelement name. *attrib* is an optional dictionary, containing element attributes. *extra* contains additional attributes, given as keyword arguments. Returns an element instance.

`xml.etree.ElementTree.`**`tostring`**(*element*, *encoding="us-ascii"*, *method="xml"*, ***, *short_empty_elements=True*)

Generates a string representation of an XML element, including all subelements. *element* is an `Element` instance. *encoding* [1] is the output encoding (default is US-ASCII). Use `encoding="unicode"` to generate a Unicode string (otherwise, a bytestring is generated). *method* is either `"xml"`, `"html"` or `"text"` (default is `"xml"`). *short_empty_elements* has the same meaning as in `ElementTree.write()`. Returns an (optionally) encoded string containing the XML data.

New in version 3.4: The *short_empty_elements* parameter.

`xml.etree.ElementTree.`**`tostringlist`**(*element*, *encoding="us-ascii"*, *method="xml"*, ***, *short_empty_elements=True*)

Generates a string representation of an XML element, including all subelements. *element* is an `Element` instance. *encoding* [1] is the output encoding (default is US-ASCII). Use `encoding="unicode"` to generate a Unicode string (otherwise, a bytestring is generated). *method* is either `"xml"`, `"html"` or `"text"` (default is `"xml"`). *short_empty_elements* has the same meaning as in `ElementTree.write()`. Returns a list of (optionally) encoded strings containing the XML data. It does not guarantee any specific sequence, except that `b"".join(tostringlist(element)) == tostring(element)`.

New in version 3.2.

New in version 3.4: The *short_empty_elements* parameter.

`xml.etree.ElementTree.`**`XML`**(*text*, *parser=None*)

Parses an XML section from a string constant. This function can be used to embed “XML literals” in Python code. *text* is a string containing XML data. *parser* is an optional parser instance. If not given, the standard `XMLParser` parser is used. Returns an `Element` instance.

`xml.etree.ElementTree.`**`XMLID`**(*text*, *parser=None*)

Parses an XML section from a string constant, and also returns a dictionary which maps from element id:s to elements. *text* is a string containing XML data. *parser* is an optional parser instance. If not given, the standard `XMLParser` parser is used. Returns a tuple containing an `Element` instance and a dictionary.

20.5.3.2. Element Objects

class `xml.etree.ElementTree.`**`Element`**(*tag*, *attrib={}*, ***extra*)

Element class. This class defines the Element interface, and provides a reference implementation of this interface.

The element name, attribute names, and attribute values can be either bytestrings or Unicode strings. *tag* is the element name. *attrib* is an optional dictionary, containing element attributes. *extra* contains additional attributes, given as keyword arguments.

`tag`

A string identifying what kind of data this element represents (the element type, in other words).

`text`
`tail`

These attributes can be used to hold additional data associated with the element. Their values are usually strings but may be any application-specific object. If the element is created from an XML file, the *text* attribute holds either the text between the element's start tag and its first child or end tag, or `None`, and the *tail* attribute holds either the text between the element's end tag and the next tag, or `None`. For the XML data

```
<a><b>1<c>2<d/>3</c></b>4</a>
```

the *a* element has `None` for both *text* and *tail* attributes, the *b* element has *text* `"1"` and *tail* `"4"`, the *c* element has *text* `"2"` and *tail* `None`, and the *d* element has *text* `None` and *tail* `"3"`.

To collect the inner text of an element, see `itertext()`, for example `"".join(element.itertext())`.

Applications may store arbitrary objects in these attributes.

`attrib`

A dictionary containing the element's attributes. Note that while the *attrib* value is always a real mutable Python dictionary, an ElementTree implementation may choose to use another internal representation, and create the dictionary only if someone asks for it. To take advantage of such implementations, use the dictionary methods below whenever possible.

The following dictionary-like methods work on the element attributes.

`clear`()

Resets an element. This function removes all subelements, clears all attributes, and sets the text and tail attributes to `None`.

get(*key*, *default=None*)

Gets the element attribute named *key*.

Returns the attribute value, or *default* if the attribute was not found.

items()

Returns the element attributes as a sequence of (name, value) pairs. The attributes are returned in an arbitrary order.

keys()

Returns the elements attribute names as a list. The names are returned in an arbitrary order.

set(*key*, *value*)

Set the attribute *key* on the element to *value*.

The following methods work on the element's children (subelements).

append(*subelement*)

Adds the element *subelement* to the end of this element's internal list of subelements. Raises `TypeError` if *subelement* is not an `Element`.

extend(*subelements*)

Appends *subelements* from a sequence object with zero or more elements. Raises `TypeError` if a subelement is not an `Element`.

New in version 3.2.

find(*match*, *namespaces=None*)

Finds the first subelement matching *match*. *match* may be a tag name or a path. Returns an element instance or `None`. *namespaces* is an optional mapping from namespace prefix to full name.

findall(*match*, *namespaces=None*)

Finds all matching subelements, by tag name or path. Returns a list containing all matching elements in document order. *namespaces* is an optional mapping from namespace prefix to full name.

findtext(*match*, *default=None*, *namespaces=None*)

Finds text for the first subelement matching *match*. *match* may be a tag name or a path. Returns the text content of the first matching element, or *default* if no element was found. Note that if the matching element has no text content an empty string is returned. *namespaces* is an optional mapping from namespace prefix to full name.

getchildren()

Deprecated since version 3.2: Use `list(elem)` or iteration.

getiterator(*tag=None*)

Deprecated since version 3.2: Use method `Element.iter()` instead.

insert(*index*, *subelement*)

Inserts *subelement* at the given position in this element. Raises `TypeError` if *subelement* is not an `Element`.

iter(*tag=None*)

Creates a tree iterator with the current element as the root. The iterator iterates over this element and all elements below it, in document (depth first) order. If *tag* is not `None` or `'*'`, only elements whose tag equals *tag* are returned from the iterator. If the tree structure is modified during iteration, the result is undefined.

New in version 3.2.

iterfind(*match*, *namespaces=None*)

Finds all matching subelements, by tag name or path. Returns an iterable yielding all matching elements in document order. *namespaces* is an optional mapping from namespace prefix to full name.

New in version 3.2.

itertext()

Creates a text iterator. The iterator loops over this element and all subelements, in document order, and returns all inner text.

New in version 3.2.

makeelement(*tag*, *attrib*)

Creates a new element object of the same type as this element. Do not call this method, use the `SubElement()` factory function instead.

remove(*subelement*)

Removes *subelement* from the element. Unlike the find* methods this method compares elements based on the instance identity, not on tag value or contents.

`Element` objects also support the following sequence type methods for working with subelements: `__delitem__()`, `__getitem__()`, `__setitem__()`, `__len__()`.

Caution: Elements with no subelements will test as `False`. This behavior will change in future versions. Use specific `len(elem)` or `elem is None` test instead.

```
element = root.find('foo')

if not element:  # careful!
    print("element not found, or element has no subelements")

if element is None:
    print("element not found")
```

20.5.3.3. ElementTree Objects

class `xml.etree.ElementTree.`**`ElementTree`**(*element=None*, *file=None*)

ElementTree wrapper class. This class represents an entire element hierarchy, and adds some extra support for serialization to and from standard XML.

element is the root element. The tree is initialized with the contents of the XML *file* if given.

`_setroot`(*element*)

Replaces the root element for this tree. This discards the current contents of the tree, and replaces it with the given element. Use with care. *element* is an element instance.

`find`(*match*, *namespaces=None*)

Same as `Element.find()`, starting at the root of the tree.

`findall`(*match*, *namespaces=None*)

Same as `Element.findall()`, starting at the root of the tree.

`findtext`(*match*, *default=None*, *namespaces=None*)

Same as `Element.findtext()`, starting at the root of the tree.

`getiterator`(*tag=None*)

Deprecated since version 3.2: Use method `ElementTree.iter()` instead.

`getroot`()

Returns the root element for this tree.

`iter`(*tag=None*)

Creates and returns a tree iterator for the root element. The iterator loops over all elements in this tree, in section order. *tag* is the tag to look for (default is to return all elements).

`iterfind`(*match*, *namespaces=None*)

Same as `Element.iterfind()`, starting at the root of the tree.

New in version 3.2.

`parse`(*source*, *parser=None*)

Loads an external XML section into this element tree. *source* is a file name or file object. *parser* is an optional parser instance. If not given, the standard `XMLParser` parser is used. Returns the section root element.

`write`(*file*, *encoding="us-ascii"*, *xml_declaration=None*, *default_namespace=None*, *method="xml"*, ***, *short_empty_elements=True*)

Writes the element tree to a file, as XML. *file* is a file name, or a file object opened for writing. *encoding* [1] is the output encoding (default is US-ASCII). *xml_declaration*

controls if an XML declaration should be added to the file. Use `False` for never, `True` for always, `None` for only if not US-ASCII or UTF-8 or Unicode (default is `None`). *default_namespace* sets the default XML namespace (for "xmlns"). *method* is either `"xml"`, `"html"` or `"text"` (default is `"xml"`). The keyword-only *short_empty_elements* parameter controls the formatting of elements that contain no content. If `True` (the default), they are emitted as a single self-closed tag, otherwise they are emitted as a pair of start/end tags.

The output is either a string (`str`) or binary (`bytes`). This is controlled by the *encoding* argument. If *encoding* is `"unicode"`, the output is a string; otherwise, it's binary. Note that this may conflict with the type of *file* if it's an open file object; make sure you do not try to write a string to a binary stream and vice versa.

New in version 3.4: The *short_empty_elements* parameter.

This is the XML file that is going to be manipulated:

```
<html>
    <head>
        <title>Example page</title>
    </head>
    <body>
        <p>Moved to <a href="http://example.org/">example.org</a>
        or <a href="http://example.com/">example.com</a>.</p>
    </body>
</html>
```

Example of changing the attribute "target" of every link in first paragraph:

```
>>> from xml.etree.ElementTree import ElementTree
>>> tree = ElementTree()
>>> tree.parse("index.xhtml")
<Element 'html' at 0xb77e6fac>
>>> p = tree.find("body/p")     # Finds first occurrence of tag p in bo
>>> p
<Element 'p' at 0xb77ec26c>
>>> links = list(p.iter("a"))   # Returns list of all links
>>> links
[<Element 'a' at 0xb77ec2ac>, <Element 'a' at 0xb77ec1cc>]
>>> for i in links:             # Iterates through all found links
...     i.attrib["target"] = "blank"
>>> tree.write("output.xhtml")
```

20.5.3.4. QName Objects

class `xml.etree.ElementTree.`**QName**(*text_or_uri*, *tag=None*)

QName wrapper. This can be used to wrap a QName attribute value, in order to get proper namespace handling on output. *text_or_uri* is a string containing the QName value, in the form {uri}local, or, if the tag argument is given, the URI part of a QName. If *tag* is given, the first argument is interpreted as a URI, and this argument is interpreted as a local name. `QName` instances are opaque.

20.5.3.5. TreeBuilder Objects

class `xml.etree.ElementTree.`**`TreeBuilder`**(*element_factory=None*)

Generic element structure builder. This builder converts a sequence of start, data, and end method calls to a well-formed element structure. You can use this class to build an element structure using a custom XML parser, or a parser for some other XML-like format. *element_factory*, when given, must be a callable accepting two positional arguments: a tag and a dict of attributes. It is expected to return a new element instance.

`close`()

Flushes the builder buffers, and returns the toplevel document element. Returns an `Element` instance.

`data`(*data*)

Adds text to the current element. *data* is a string. This should be either a bytestring, or a Unicode string.

`end`(*tag*)

Closes the current element. *tag* is the element name. Returns the closed element.

`start`(*tag*, *attrs*)

Opens a new element. *tag* is the element name. *attrs* is a dictionary containing element attributes. Returns the opened element.

In addition, a custom `TreeBuilder` object can provide the following method:

`doctype`(*name*, *pubid*, *system*)

Handles a doctype declaration. *name* is the doctype name. *pubid* is the public identifier. *system* is the system identifier. This method does not exist on the default `TreeBuilder` class.

New in version 3.2.

20.5.3.6. XMLParser Objects

class `xml.etree.ElementTree.`**`XMLParser`**(*html=0*, *target=None*, *encoding=None*)

This class is the low-level building block of the module. It uses `xml.parsers.expat` for efficient, event-based parsing of XML. It can be fed XML data incrementally with the `feed()` method, and parsing events are translated to a push API - by invoking callbacks on the *target* object. If *target* is omitted, the standard `TreeBuilder` is used. The *html* argument was historically used for backwards compatibility and is now deprecated. If *encoding* [1] is given, the value overrides the encoding specified in the XML file.

Deprecated since version 3.4: The *html* argument. The remaining arguments should be passed via keyword to prepare for the removal of the *html* argument.

`close`()

Finishes feeding data to the parser. Returns the result of calling the `close()` method of the *target* passed during construction; by default, this is the toplevel document element.

doctype(*name*, *pubid*, *system*)

Deprecated since version 3.2: Define the `TreeBuilder.doctype()` method on a custom TreeBuilder target.

feed(*data*)

Feeds data to the parser. *data* is encoded data.

`XMLParser.feed()` calls *target*'s `start(tag, attrs_dict)` method for each opening tag, its `end(tag)` method for each closing tag, and data is processed by method `data(data)`. `XMLParser.close()` calls *target*'s method `close()`. `XMLParser` can be used not only for building a tree structure. This is an example of counting the maximum depth of an XML file:

```
>>> from xml.etree.ElementTree import XMLParser
>>> class MaxDepth:                     # The target object of the p
...     maxDepth = 0
...     depth = 0
...     def start(self, tag, attrib):   # Called for each opening ta
...         self.depth += 1
...         if self.depth > self.maxDepth:
...             self.maxDepth = self.depth
...     def end(self, tag):             # Called for each closing ta
...         self.depth -= 1
...     def data(self, data):
...         pass            # We do not need to do anything with dat
...     def close(self):    # Called when all data has been parsed.
...         return self.maxDepth
...
>>> target = MaxDepth()
>>> parser = XMLParser(target=target)
>>> exampleXml = """
... <a>
...   <b>
...   </b>
...   <b>
...     <c>
...       <d>
...       </d>
...     </c>
...   </b>
... </a>"""
>>> parser.feed(exampleXml)
>>> parser.close()
4
```

20.5.3.7. XMLPullParser Objects

class `xml.etree.ElementTree.`**XMLPullParser**(*events=None*)

A pull parser suitable for non-blocking applications. Its input-side API is similar to that of `XMLParser`, but instead of pushing calls to a callback target, `XMLPullParser` collects an internal list of parsing events and lets the user read from it. *events* is a sequence of events to report back. The supported events are the strings `"start"`, `"end"`, `"start-ns"` and `"end-ns"` (the "ns" events are used to get detailed namespace information). If *events* is omitted, only `"end"` events are reported.

`feed`(*data*)

Feed the given bytes data to the parser.

`close`()

Signal the parser that the data stream is terminated. Unlike `XMLParser.close()`, this method always returns `None`. Any events not yet retrieved when the parser is closed can still be read with `read_events()`.

`read_events`()

Return an iterator over the events which have been encountered in the data fed to the parser. The iterator yields `(event, elem)` pairs, where *event* is a string representing the type of event (e.g. `"end"`) and *elem* is the encountered `Element` object.

Events provided in a previous call to `read_events()` will not be yielded again. Events are consumed from the internal queue only when they are retrieved from the iterator, so multiple readers iterating in parallel over iterators obtained from `read_events()` will have unpredictable results.

Note: `XMLPullParser` only guarantees that it has seen the ">" character of a starting tag when it emits a "start" event, so the attributes are defined, but the contents of the text and tail attributes are undefined at that point. The same applies to the element children; they may or may not be present.
If you need a fully populated element, look for "end" events instead.

New in version 3.4.

20.5.3.8. Exceptions

class `xml.etree.ElementTree.`**`ParseError`**

XML parse error, raised by the various parsing methods in this module when parsing fails. The string representation of an instance of this exception will contain a user-friendly error message. In addition, it will have the following attributes available:

`code`

A numeric error code from the expat parser. See the documentation of `xml.parsers.expat` for the list of error codes and their meanings.

`position`

A tuple of *line*, *column* numbers, specifying where the error occurred.

Footnotes

[1] *(1, 2, 3, 4)* The encoding string included in XML output should conform to the appropriate standards. For example, “UTF-8” is valid, but “UTF8” is not. See https://www.w3.org/TR/2006/REC-xml11-20060816/#NT-EncodingDecl and https://www.iana.org/assignments/character-sets/character-sets.xhtml.