20.13. `xml.etree.ElementTree` — The ElementTree XML API¶

New in version 2.5.

The Element type is a flexible container object, designed to store hierarchical data structures in memory. The type can be described as a cross between a list and a dictionary.

Each element has a number of properties associated with it:

- a tag which is a string identifying what kind of data this element represents (the element type, in other words).
- a number of attributes, stored in a Python dictionary.
- a text string.
- an optional tail string.
- a number of child elements, stored in a Python sequence

To create an element instance, use the Element or SubElement factory functions.

The `ElementTree` class can be used to wrap an element structure, and convert it from and to XML.

A C implementation of this API is available as `xml.etree.cElementTree`.

See http://effbot.org/zone/element-index.htm for tutorials and links to other docs. Fredrik Lundh's page is also the location of the development version of the xml.etree.ElementTree.

20.13.1. Functions¶

`xml.etree.ElementTree.Comment`([*text*])¶
Comment element factory. This factory function creates a special element that will be serialized as an XML comment. The comment string can be either an 8-bit ASCII string or a Unicode string. *text* is a string containing the comment string. Returns an element instance representing a comment.

`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.Element`(*tag*[, *attrib*][, ***extra*])¶

Element factory. This function returns an object implementing the standard Element interface. The exact class or type of that object is implementation dependent, but it will always be compatible with the _ElementInterface class in this module.

The element name, attribute names, and attribute values can be either 8-bit ASCII strings or Unicode strings. *tag* is the element name. *attrib* is an optional dictionary, containing element attributes. *extra* contains additional attributes, given as keyword arguments. Returns an element instance.

`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.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*])¶

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 list of events to report back. If omitted, only "end" events are reported. Returns an *iterator* providing `(event, elem)` pairs.

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.

`xml.etree.ElementTree.parse`(*source*[, *parser*])¶
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 XMLTreeBuilder parser is used. Returns an ElementTree instance.

`xml.etree.ElementTree.ProcessingInstruction`(*target*[, *text*])¶
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.

`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 8-bit ASCII strings 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*])¶
Generates a string representation of an XML element, including all subelements. *element* is an Element instance. *encoding* is the output encoding (default is US-ASCII). Returns an encoded string containing the XML data.

`xml.etree.ElementTree.XML`(*text*)¶
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. Returns an Element instance.

`xml.etree.ElementTree.XMLID`(*text*)¶
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. Returns a tuple containing an Element instance and a dictionary.

20.13.2. The Element Interface¶

Element objects returned by Element or SubElement have the following methods and attributes.

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

`Element.text`¶
The *text* attribute can be used to hold additional data associated with the element. As the name implies this attribute is usually a string but may be any application-specific object. If the element is created from an XML file the attribute will contain any text found between the element tags.

`Element.tail`¶
The *tail* attribute can be used to hold additional data associated with the element. This attribute is usually a string but may be any application-specific object. If the element is created from an XML file the attribute will contain any text found after the element’s end tag and before the next tag.

`Element.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.

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

`Element.get`(*key*[, *default*=*None*])¶

Gets the element attribute named *key*.

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

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

`Element.keys`()¶
Returns the elements attribute names as a list. The names are returned in an arbitrary order.

`Element.set`(*key*, *value*)¶
Set the attribute *key* on the element to *value*.

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

`Element.append`(*subelement*)¶
Adds the element *subelement* to the end of this elements internal list of subelements.

`Element.find`(*match*)¶
Finds the first subelement matching *match*. *match* may be a tag name or path. Returns an element instance or `None`.

`Element.findall`(*match*)¶
Finds all subelements matching *match*. *match* may be a tag name or path. Returns an iterable yielding all matching elements in document order.

`Element.findtext`(*condition*[, *default=None*])¶
Finds text for the first subelement matching *condition*. *condition* may be a tag name or 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.

`Element.getchildren`()¶
Returns all subelements. The elements are returned in document order.

`Element.getiterator`([*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.

`Element.insert`(*index*, *element*)¶
Inserts a subelement at the given position in this element.

`Element.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.

`Element.remove`(*subelement*)¶
Removes *subelement* from the element. Unlike the findXYZ 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: Because Element objects do not define a `__nonzero__()` method, elements with no subelements will test as `False`.

```
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"
```

20.13.3. ElementTree Objects¶

class `xml.etree.ElementTree.ElementTree`([*element*][, *file*])¶

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`(*path*)¶
Finds the first toplevel element with given tag. Same as getroot().find(path). *path* is the element to look for. Returns the first matching element, or `None` if no element was found.

`findall`(*path*)¶
Finds all toplevel elements with the given tag. Same as getroot().findall(path). *path* is the element to look for. Returns a list or *iterator* containing all matching elements, in document order.

`findtext`(*path*[, *default*])¶
Finds the element text for the first toplevel element with given tag. Same as getroot().findtext(path). *path* is the toplevel element to look for. *default* is the value to return if the element was not found. Returns the text content of the first matching element, or the default value no element was found. Note that if the element has is found, but has no text content, this method returns an empty string.

`getiterator`([*tag*])¶
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)

`getroot`()¶
Returns the root element for this tree.

`parse`(*source*[, *parser*])¶
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 XMLTreeBuilder parser is used. Returns the section root element.

`write`(*file*[, *encoding*])¶
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).

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 b7d3f1ec>
>>> p = tree.find("body/p")     # Finds first occurrence of tag p in body
>>> p
<Element p at 8416e0c>
>>> links = p.getiterator("a")  # Returns list of all links
>>> links
[<Element a at b7d4f9ec>, <Element a at b7d4fb0c>]
>>> for i in links:             # Iterates through all found links
...     i.attrib["target"] = "blank"
>>> tree.write("output.xhtml")
```

20.13.4. QName Objects¶

class `xml.etree.ElementTree.QName`(*text_or_uri*[, *tag*])¶
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 an URI, and this argument is interpreted as a local name. `QName` instances are opaque.

20.13.5. TreeBuilder Objects¶

class `xml.etree.ElementTree.TreeBuilder`([*element_factory*])¶

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. The *element_factory* is called to create new Element instances when given.

`close`()¶
Flushes the parser 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 an 8-bit string containing ASCII text, 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.

20.13.6. XMLTreeBuilder Objects¶

class `xml.etree.ElementTree.XMLTreeBuilder`([*html*][, *target*])¶

Element structure builder for XML source data, based on the expat parser. *html* are predefined HTML entities. This flag is not supported by the current implementation. *target* is the target object. If omitted, the builder uses an instance of the standard TreeBuilder class.

`close`()¶
Finishes feeding data to the parser. Returns an element structure.

`doctype`(*name*, *pubid*, *system*)¶
Handles a doctype declaration. *name* is the doctype name. *pubid* is the public identifier. *system* is the system identifier.

`feed`(*data*)¶
Feeds data to the parser. *data* is encoded data.

`XMLTreeBuilder.feed()` calls *target*'s `start()` method for each opening tag, its `end()` method for each closing tag, and data is processed by method `data()`. `XMLTreeBuilder.close()` calls *target*'s method `close()`. `XMLTreeBuilder` 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 XMLTreeBuilder
>>> class MaxDepth:                     # The target object of the parser
...     maxDepth = 0
...     depth = 0
...     def start(self, tag, attrib):   # Called for each opening tag.
...         self.depth += 1
...         if self.depth > self.maxDepth:
...             self.maxDepth = self.depth
...     def end(self, tag):             # Called for each closing tag.
...         self.depth -= 1
...     def data(self, data):
...         pass            # We do not need to do anything with data.
...     def close(self):    # Called when all data has been parsed.
...         return self.maxDepth
...
>>> target = MaxDepth()
>>> parser = XMLTreeBuilder(target=target)
>>> exampleXml = """
... <a>
...   <b>
...   </b>
...   <b>
...     <c>
...       <d>
...       </d>
...     </c>
...   </b>
... </a>"""
>>> parser.feed(exampleXml)
>>> parser.close()
4
```

Footnotes

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

Previous topic

20.12. `xml.sax.xmlreader` — Interface for XML parsers

Next topic

21. Internet Protocols and Support

This Page

- Show Source

Navigation

- index
- modules |
- next |
- previous |
- Python v2.6.4 documentation »
- The Python Standard Library »