

lxml 2.0.11



2008-12-12

# Contents

<b>Contents</b>	<b>2</b>
<b>I lxml</b>	<b>11</b>
<b>1 lxml</b>	<b>12</b>
1.1 Introduction	12
1.2 Documentation	12
1.3 Download	13
1.4 Mailing list	14
1.5 Bug tracker	14
1.6 License	14
<b>2 Why lxml?</b>	<b>15</b>
2.1 Motto	15
2.2 Aims	15
<b>3 Installing lxml</b>	<b>17</b>
3.1 Requirements	17
3.2 Installation	17
3.3 Building lxml from sources	17
3.4 MS Windows	18
3.5 MacOS-X	18
<b>4 What's new in lxml 2.0?</b>	<b>19</b>
4.1 Changes in etree and objectify	19
4.1.1 Incompatible changes	19
4.1.2 Enhancements	20
4.1.3 Deprecation	21
4.2 New modules	21
4.2.1 lxml.usedoctest	21
4.2.2 lxml.html	22
4.2.3 lxml.cssselect	22
<b>5 Benchmarks and Speed</b>	<b>23</b>
5.1 General notes	23
5.2 How to read the timings	23
5.3 Parsing and Serialising	24
5.4 The ElementTree API	25
5.4.1 Child access	26
5.4.2 Element creation	27
5.4.3 Merging different sources	27
5.4.4 deepcopy	28

5.4.5	Tree traversal	28
5.5	XPath	29
5.6	A longer example	30
5.7	lxml.objectify	32
5.7.1	ObjectPath	32
5.7.2	Caching Elements	32
5.7.3	Further optimisations	33
<b>6</b>	<b>ElementTree compatibility of lxml.etree</b>	<b>34</b>
<b>7</b>	<b>lxml FAQ - Frequently Asked Questions</b>	<b>37</b>
7.1	General Questions	37
7.1.1	Is there a tutorial?	37
7.1.2	Where can I find more documentation about lxml?	37
7.1.3	What standards does lxml implement?	37
7.1.4	Who uses lxml?	38
7.1.5	What is the difference between lxml.etree and lxml.objectify?	38
7.1.6	How can I make my application run faster?	39
7.1.7	What about that trailing text on serialised Elements?	39
7.2	Installation	40
7.2.1	Which version of libxml2 and libxslt should I use or require?	40
7.2.2	Where are the Windows binaries?	40
7.2.3	Why do I get errors about missing UCS4 symbols when installing lxml?	40
7.3	Contributing	41
7.3.1	Why is lxml not written in Python?	41
7.3.2	How can I contribute?	41
7.4	Bugs	42
7.4.1	My application crashes!	42
7.4.2	I think I have found a bug in lxml. What should I do?	42
7.5	Threading	43
7.5.1	Can I use threads to concurrently access the lxml API?	43
7.5.2	Does my program run faster if I use threads?	43
7.5.3	Would my single-threaded program run faster if I turned off threading?	43
7.5.4	Why can't I reuse XSLT stylesheets in other threads?	43
7.5.5	My program crashes when run with mod_python/Pyro/Zope/Plone/...	44
7.6	Parsing and Serialisation	45
7.6.1	Why doesn't the <code>pretty_print</code> option reformat my XML output?	45
7.6.2	Why can't lxml parse my XML from unicode strings?	45
7.6.3	What is the difference between <code>str(xslt(doc))</code> and <code>xslt(doc).write()</code> ?	45
7.6.4	Why can't I just delete parents or clear the root node in <code>iterparse()</code> ?	46
7.6.5	How do I output null characters in XML text?	46
7.7	XPath and Document Traversal	46
7.7.1	What are the <code>findall()</code> and <code>xpath()</code> methods on <code>Element(Tree)</code> ?	46
7.7.2	Why doesn't <code>findall()</code> support full XPath expressions?	46
7.7.3	How can I find out which namespace prefixes are used in a document?	46
7.7.4	How can I specify a default namespace for XPath expressions?	47
<b>II</b>	<b>Developing with lxml</b>	<b>48</b>
<b>8</b>	<b>The lxml.etree Tutorial</b>	<b>49</b>
8.1	The Element class	50
8.1.1	Elements are lists	50
8.1.2	Elements carry attributes	52
8.1.3	Elements contain text	52

8.1.4	Using XPath to find text	53
8.1.5	Tree iteration	54
8.1.6	Serialisation	55
8.2	The ElementTree class	56
8.3	Parsing from strings and files	57
8.3.1	The fromstring() function	57
8.3.2	The XML() function	58
8.3.3	The parse() function	58
8.3.4	Parser objects	59
8.3.5	Incremental parsing	59
8.3.6	Event-driven parsing	60
8.4	Namespaces	61
8.5	The E-factory	62
8.6	ElementPath	64
<b>9</b>	<b>APIs specific to lxml.etree</b>	<b>65</b>
9.1	lxml.etree	65
9.2	Other Element APIs	65
9.3	Trees and Documents	66
9.4	Iteration	67
9.5	Error handling on exceptions	67
9.6	Error logging	68
9.7	Serialisation	68
9.8	XInclude and ElementInclude	69
9.9	write_c14n on ElementTree	70
<b>10</b>	<b>Parsing XML and HTML with lxml</b>	<b>71</b>
10.1	Parsers	71
10.1.1	Parser options	72
10.1.2	Error log	72
10.1.3	Parsing HTML	73
10.1.4	Doctype information	73
10.2	The target parser interface	74
10.3	The feed parser interface	74
10.4	iterparse and iterwalk	75
10.4.1	Selective tag events	76
10.4.2	Modifying the tree	77
10.4.3	iterwalk	77
10.5	Python unicode strings	78
10.5.1	Serialising to Unicode strings	78
<b>11</b>	<b>Validation with lxml</b>	<b>80</b>
11.1	Validation at parse time	80
11.2	DTD	81
11.3	RelaxNG	81
11.4	XMLSchema	83
11.5	Schematron	84
<b>12</b>	<b>XPath and XSLT with lxml</b>	<b>86</b>
12.1	XPath	86
12.1.1	The xpath() method	86
12.1.2	XPath return values	88
12.1.3	Generating XPath expressions	88
12.1.4	The XPath class	88
12.1.5	The XPathEvaluator classes	89

12.1.6	ETXPath	90
12.1.7	Error handling	90
12.2	XSLT	91
12.2.1	XSLT result objects	91
12.2.2	Stylesheet parameters	92
12.2.3	The <code>xslt()</code> tree method	93
12.2.4	Profiling	93
<b>13</b>	<b>lxml.objectify</b>	<b>94</b>
13.1	The lxml.objectify API	94
13.1.1	Creating objectify trees	95
13.1.2	Element access through object attributes	95
13.1.3	Tree generation with the E-factory	97
13.1.4	Namespace handling	98
13.2	Asserting a Schema	99
13.3	ObjectPath	100
13.4	Python data types	103
13.4.1	Recursive tree dump	104
13.4.2	Recursive string representation of elements	106
13.5	How data types are matched	106
13.5.1	Type annotations	107
13.5.2	XML Schema datatype annotation	108
13.5.3	The DataElement factory	109
13.5.4	Defining additional data classes	111
13.5.5	Advanced element class lookup	112
13.6	What is different from lxml.etree?	113
<b>14</b>	<b>lxml.html</b>	<b>114</b>
14.1	Parsing HTML	114
14.1.1	Parsing HTML fragments	114
14.1.2	Really broken pages	114
14.2	HTML Element Methods	115
14.3	Running HTML doctests	115
14.4	Creating HTML with the E-factory	116
14.4.1	Viewing your HTML	117
14.5	Working with links	117
14.5.1	Functions	117
14.6	Forms	118
14.6.1	Form Filling Example	119
14.6.2	Form Submission	119
14.7	Cleaning up HTML	120
14.7.1	autolink	122
14.7.2	wordwrap	122
14.8	HTML Diff	122
14.9	Examples	123
14.9.1	Microformat Example	123
<b>15</b>	<b>lxml.cssselect</b>	<b>125</b>
15.1	The CSSSelector class	125
15.2	CSS Selectors	125
15.2.1	Namespaces	126
15.3	Limitations	126
<b>16</b>	<b>BeautifulSoup Parser</b>	<b>127</b>
16.1	Entity handling	128

16.2 Using soupparser as a fallback . . . . .	128
<b>III Extending lxml</b>	<b>130</b>
<b>17 Document loading and URL resolving</b>	<b>131</b>
17.1 Resolvers . . . . .	131
17.2 Document loading in context . . . . .	132
17.3 I/O access control in XSLT . . . . .	134
<b>18 Extension functions for XPath and XSLT</b>	<b>135</b>
18.1 The FunctionNamespace . . . . .	135
18.2 Global prefix assignment . . . . .	136
18.3 The XPath context . . . . .	136
18.4 Evaluators and XSLT . . . . .	137
18.5 Evaluator-local extensions . . . . .	138
18.6 What to return from a function . . . . .	139
<b>19 Using custom Element classes in lxml</b>	<b>142</b>
19.1 Element initialization . . . . .	142
19.2 Setting up a class lookup scheme . . . . .	143
19.2.1 Default class lookup . . . . .	144
19.2.2 Namespace class lookup . . . . .	145
19.2.3 Attribute based lookup . . . . .	145
19.2.4 Custom element class lookup . . . . .	145
19.2.5 Tree based element class lookup in Python . . . . .	146
19.3 Implementing namespaces . . . . .	146
<b>20 Sax support</b>	<b>149</b>
20.1 Building a tree from SAX events . . . . .	149
20.2 Producing SAX events from an ElementTree or Element . . . . .	149
20.3 Interfacing with pulldom/minidom . . . . .	150
<b>21 The public C-API of lxml.etree</b>	<b>151</b>
21.1 Writing external modules in Cython . . . . .	151
21.2 Writing external modules in C . . . . .	152
<b>IV Developing lxml</b>	<b>153</b>
<b>22 How to build lxml from source</b>	<b>154</b>
22.1 Cython . . . . .	154
22.2 Subversion . . . . .	154
22.3 Setuptools . . . . .	155
22.4 Running the tests and reporting errors . . . . .	155
22.5 Contributing an egg . . . . .	156
22.6 Providing newer library versions on Mac-OS X . . . . .	156
22.7 Static linking on Windows . . . . .	157
22.8 Building Debian packages from SVN sources . . . . .	158
<b>23 How to read the source of lxml</b>	<b>159</b>
23.1 What is Cython? . . . . .	159
23.2 Where to start? . . . . .	159
23.2.1 Concepts . . . . .	160
23.2.2 The documentation . . . . .	160
23.3 lxml.etree . . . . .	161

23.4 Python modules . . . . .	162
23.5 lxml.objectify . . . . .	162
23.6 lxml.pyclasslookup . . . . .	162
23.7 lxml.html . . . . .	163
<b>24 Credits</b>	<b>164</b>
24.1 Special thanks goes to: . . . . .	164
<b>A Changes</b>	<b>166</b>
A.1 Changes in version 2.0.11, released 2008-12-12 . . . . .	166
A.2 Changes in version 2.0.10, released 2008-11-17 . . . . .	166
A.3 Changes in version 2.0.9, released 2008-09-05 . . . . .	166
A.4 Changes in version 2.0.8, released 2008-07-24 . . . . .	166
A.5 Changes in version 2.0.7, released 2008-06-20 . . . . .	167
A.6 Changes in version 2.0.6, released 2008-05-31 . . . . .	167
A.7 Changes in version 2.0.5, released 2008-05-01 . . . . .	168
A.8 Changes in version 2.0.4, released 2008-04-13 . . . . .	168
A.9 Changes in version 2.0.3, released 2008-03-26 . . . . .	168
A.10 Changes in version 2.0.2, released 2008-02-22 . . . . .	169
A.11 Changes in version 2.0.1, released 2008-02-13 . . . . .	169
A.12 Changes in version 2.0, released 2008-02-01 . . . . .	170
A.13 Changes in version 2.0beta2, released 2008-01-26 . . . . .	170
A.14 Changes in version 2.0beta1, released 2008-01-11 . . . . .	171
A.15 Changes in version 2.0alpha6, released 2007-12-19 . . . . .	172
A.16 Changes in version 2.0alpha5, released 2007-11-24 . . . . .	172
A.17 Changes in version 1.3.6, released 2007-10-29 . . . . .	173
A.18 Changes in version 1.3.5, released 2007-10-22 . . . . .	173
A.19 Changes in version 2.0alpha4, released 2007-10-07 . . . . .	174
A.20 Changes in version 2.0alpha3, released 2007-09-26 . . . . .	174
A.21 Changes in version 2.0alpha2, released 2007-09-15 . . . . .	175
A.22 Changes in version 2.0alpha1, released 2007-09-02 . . . . .	175
A.23 Changes in version 1.3.4, released 2007-08-30 . . . . .	177
A.24 Changes in version 1.3.3, released 2007-07-26 . . . . .	177
A.25 Changes in version 1.3.2, released 2007-07-03 . . . . .	178
A.26 Changes in version 1.3.1, released 2007-07-02 . . . . .	178
A.27 Changes in version 1.3, released 2007-06-24 . . . . .	178
A.28 Changes in version 1.2.1, released 2007-02-27 . . . . .	180
A.29 Changes in version 1.2, released 2007-02-20 . . . . .	180
A.30 Changes in version 1.1.2, released 2006-10-30 . . . . .	181
A.31 Changes in version 1.1.1, released 2006-09-21 . . . . .	181
A.32 Changes in version 1.1, released 2006-09-13 . . . . .	182
A.33 Changes in version 1.0.4, released 2006-09-09 . . . . .	183
A.34 Changes in version 1.0.3, released 2006-08-08 . . . . .	184
A.35 Changes in version 1.0.2, released 2006-06-27 . . . . .	184
A.36 Changes in version 1.0.1, released 2006-06-09 . . . . .	184
A.37 Changes in version 1.0, released 2006-06-01 . . . . .	185
A.38 Changes in version 0.9.2, released 2006-05-10 . . . . .	187
A.39 Changes in version 0.9.1, released 2006-03-30 . . . . .	187
A.40 Changes in version 0.9, released 2006-03-20 . . . . .	188
A.41 Changes in version 0.8, released 2005-11-03 . . . . .	189
A.42 Changes in version 0.7, released 2005-06-15 . . . . .	189
A.43 Changes in version 0.6, released 2005-05-14 . . . . .	190
A.44 Changes in version 0.5.1, released 2005-04-09 . . . . .	191
A.45 Changes in version 0.5, released 2005-04-08 . . . . .	191

<b>B</b>	<b>Generated API documentation</b>	<b>192</b>
B.1	Package lxml	193
B.1.1	Modules	193
B.2	Module lxml.ElementInclude	194
B.2.1	Functions	194
B.2.2	Variables	194
B.2.3	Class FatalIncludeError	195
B.3	Module lxml.builder	197
B.3.1	Variables	197
B.3.2	Class ElementMaker	197
B.4	Module lxml.cssselect	200
B.4.1	Class SelectorSyntaxError	200
B.4.2	Class ExpressionError	201
B.4.3	Class CSSSelector	202
B.5	Module lxml.doctestcompare	204
B.5.1	Functions	204
B.5.2	Variables	205
B.5.3	Class LXMLOutputChecker	205
B.5.4	Class LHTMLOutputChecker	206
B.6	Module lxml.etree	208
B.6.1	Functions	208
B.6.2	Variables	214
B.6.3	Class AncestorsIterator	215
B.6.4	Class AttributeBasedElementClassLookup	216
B.6.5	Class C14NError	217
B.6.6	Class CommentBase	218
B.6.7	Class CustomElementClassLookup	219
B.6.8	Class DTD	220
B.6.9	Class DTDError	222
B.6.10	Class DTDParseError	223
B.6.11	Class DTDValidateError	224
B.6.12	Class DocInfo	225
B.6.13	Class DocumentInvalid	226
B.6.14	Class ETCompatXMLParser	227
B.6.15	Class ETXPath	228
B.6.16	Class ElementBase	229
B.6.17	Class ElementChildIterator	231
B.6.18	Class ElementClassLookup	232
B.6.19	Class ElementDefaultClassLookup	232
B.6.20	Class ElementDepthFirstIterator	233
B.6.21	Class ElementNamespaceClassLookup	235
B.6.22	Class ElementTextIterator	236
B.6.23	Class EntityBase	237
B.6.24	Class Error	238
B.6.25	Class ErrorDomains	239
B.6.26	Class ErrorLevels	240
B.6.27	Class ErrorTypes	240
B.6.28	Class FallbackElementClassLookup	272
B.6.29	Class HTMLParser	274
B.6.30	Class LxmlError	276
B.6.31	Class LxmlRegistryError	277
B.6.32	Class LxmlSyntaxError	278
B.6.33	Class NamespaceRegistryError	279
B.6.34	Class PIBase	280
B.6.35	Class ParseError	282

B.6.36	Class ParserBasedElementClassLookup	283
B.6.37	Class ParserError	284
B.6.38	Class PyErrorLog	285
B.6.39	Class QName	286
B.6.40	Class RelaxNG	288
B.6.41	Class RelaxNGError	289
B.6.42	Class RelaxNGErrorTypes	290
B.6.43	Class RelaxNGParseError	293
B.6.44	Class RelaxNGValidateError	294
B.6.45	Class Resolver	295
B.6.46	Class Schematron	296
B.6.47	Class SchematronError	297
B.6.48	Class SchematronParseError	298
B.6.49	Class SchematronValidateError	299
B.6.50	Class SiblingsIterator	300
B.6.51	Class TreeBuilder	301
B.6.52	Class XInclude	303
B.6.53	Class XIncludeError	304
B.6.54	Class XMLParser	305
B.6.55	Class XMLSchema	307
B.6.56	Class XMLSchemaError	308
B.6.57	Class XMLSchemaParseError	309
B.6.58	Class XMLSchemaValidateError	310
B.6.59	Class XMLSyntaxError	311
B.6.60	Class XPath	312
B.6.61	Class XPathDocumentEvaluator	314
B.6.62	Class XPathElementEvaluator	315
B.6.63	Class XPathError	317
B.6.64	Class XPathEvalError	318
B.6.65	Class XPathFunctionError	319
B.6.66	Class XPathResultError	320
B.6.67	Class XPathSyntaxError	322
B.6.68	Class XSLT	323
B.6.69	Class XSLTAccessControl	325
B.6.70	Class XSLTApplyError	326
B.6.71	Class XSLTError	327
B.6.72	Class XSLTExtensionError	328
B.6.73	Class XSLTParseError	329
B.6.74	Class XSLTSaveError	330
B.6.75	Class iterparse	331
B.6.76	Class iterwalk	333
B.7	Package lxml.html	335
B.7.1	Modules	335
B.7.2	Functions	335
B.7.3	Variables	338
B.8	Module lxml.html.ElementSoup	339
B.8.1	Functions	339
B.9	Module lxml.html.builder	340
B.9.1	Functions	340
B.9.2	Variables	340
B.10	Module lxml.html.clean	344
B.10.1	Functions	344
B.10.2	Variables	345
B.10.3	Class Cleaner	345
B.11	Module lxml.html.defs	348

---

B.11.1 Variables	348
B.12 Module lxml.html.diff	349
B.12.1 Functions	349
B.13 Module lxml.html.formfill	350
B.13.1 Functions	350
B.13.2 Class FormNotFound	350
B.13.3 Class DefaultErrorCreator	351
B.14 Module lxml.html.soupparser	353
B.14.1 Functions	353
B.15 Module lxml.html.usedoctest	354
B.16 Module lxml.objectify	355
B.16.1 Functions	355
B.16.2 Variables	359
B.16.3 Class BoolElement	360
B.16.4 Class ElementMaker	362
B.16.5 Class FloatElement	363
B.16.6 Class IntElement	365
B.16.7 Class LongElement	366
B.16.8 Class NoneElement	368
B.16.9 Class NumberElement	370
B.16.10 Class ObjectPath	374
B.16.11 Class ObjectifiedDataElement	376
B.16.12 Class ObjectifiedElement	378
B.16.13 Class ObjectifyElementClassLookup	381
B.16.14 Class PyType	382
B.16.15 Class StringElement	384
B.17 Module lxml.pyclasslookup	387
B.17.1 Class PythonElementClassLookup	387
B.18 Module lxml.sax	389
B.18.1 Functions	389
B.18.2 Class SaxError	389
B.18.3 Class ElementTreeContentHandler	390
B.18.4 Class ElementTreeProducer	394
B.19 Module lxml.usedoctest	396

**Part I**

**lxml**

# Chapter 1

## lxml

» lxml takes all the pain out of XML. «  
Stephan Richter

lxml is the most feature-rich  
and easy-to-use library  
for working with XML and HTML  
in the Python language.

### 1.1 Introduction

lxml is a Pythonic binding for the [libxml2](#) and [libxslt](#) libraries. It is unique in that it combines the speed and feature completeness of these libraries with the simplicity of a native Python API, mostly compatible but superior to the well-known [ElementTree](#) API. See the [introduction](#) for more information about background and goals. Some common questions are answered in the [FAQ](#).

### 1.2 Documentation

The complete lxml documentation is available for download as [PDF documentation](#). The HTML documentation from this web site is part of the normal [source download](#).

- [ElementTree](#):
  - [ElementTree API](#)
  - [compatibility](#) and differences of `lxml.etree`
  - [benchmark results](#)
- `lxml.etree`:
  - the [lxml.etree Tutorial](#)
  - [lxml.etree specific API](#) documentation

- the generated API documentation as a reference
  - [parsing](#) and [validating](#) XML
  - [XPath and XSLT](#) support
  - Python [extension functions](#) for XPath and XSLT
  - [custom element classes](#) for custom XML APIs
  - a [SAX compliant API](#) for interfacing with other XML tools
  - a [C-level API](#) for interfacing with external C/Pyrex modules
- `lxml.objectify`:
    - [lxml.objectify](#) API documentation
    - a brief comparison of [objectify](#) and [etree](#)

`lxml.etree` follows the [ElementTree](#) API as much as possible, building it on top of the native `libxml2` tree. If you are new to `ElementTree`, start with the [lxml.etree Tutorial](#). See also the [ElementTree compatibility](#) overview and the [benchmark results](#) comparing `lxml` to the original `ElementTree` and `cElementTree` implementations.

Right after the [lxml.etree Tutorial](#) and the `ElementTree` documentation, the most important place to look is the [lxml.etree specific API](#) documentation. It describes how `lxml` extends the `ElementTree` API to expose `libxml2` and `libxslt` specific functionality, such as [XPath](#), [Relax NG](#), [XML Schema](#), [XSLT](#), and [c14n](#). Python code can be called from XPath expressions and XSLT stylesheets through the use of [extension functions](#). `lxml` also offers a [SAX compliant API](#), that works with the SAX support in the standard library.

There is a separate module [lxml.objectify](#) that implements a data-binding API on top of `lxml.etree`. See the [objectify and etree](#) FAQ entry for a comparison.

In addition to the `ElementTree` API, `lxml` also features a sophisticated API for [custom element classes](#). This is a simple way to write arbitrary XML driven APIs on top of `lxml`. As of version 1.1, `lxml.etree` has a new [C-level API](#) that can be used to efficiently extend `lxml.etree` in external C modules, including custom element class support.

## 1.3 Download

The best way to download `lxml` is to visit [lxml at the Python Package Index \(PyPI\)](#). It has the source that compiles on various platforms. The source distribution is signed with [this key](#). Binary builds for MS Windows usually become available through PyPI a few days after a source release. If you can't wait, consider trying a less recent release version first.

The latest version is `lxml 2.0.11`, released 2008-12-12 ([changes for 2.0.11](#)). [Older versions](#) are listed below.

Please take a look at the [installation instructions](#)!

This complete web site (including the generated API documentation) is part of the source distribution, so if you want to download the documentation for offline use, take the source archive and copy the `doc/html` directory out of the source tree.

It's also possible to check out the latest development version of `lxml` from `svn` directly, using a command like this:

```
svn co http://codespeak.net/svn/lxml/trunk lxml
```

You can also browse the [Subversion repository](#) through the web, or take a look at the [Subversion history](#). Please read [how to build lxml from source](#) first. The latest [CHANGES](#) of the developer version are also accessible. You can check there if a bug you found has been fixed or a feature you want has been implemented in the latest trunk version.

## 1.4 Mailing list

Questions? Suggestions? Code to contribute? We have a [mailing list](#).

You can search the archive with [Gmane](#) or [Google](#).

## 1.5 Bug tracker

lxml uses the [launchpad bug tracker](#). If you are sure you found a bug in lxml, please file a bug report there. If you are not sure whether some unexpected behaviour of lxml is a bug or not, please ask on the [mailing list](#) first. Do not forget to search the archive (e.g. with [Gmane](#))!

## 1.6 License

The lxml library is shipped under a [BSD license](#). libxml2 and libxslt2 itself are shipped under the [MIT license](#). There should therefore be no obstacle to using lxml in your codebase.

## Chapter 2

# Why lxml?

### 2.1 Motto

“the thrills without the strangeness”

To explain the motto:

“Programming with libxml2 is like the thrilling embrace of an exotic stranger. It seems to have the potential to fulfill your wildest dreams, but there’s a nagging voice somewhere in your head warning you that you’re about to get screwed in the worst way.” (a quote by [Mark Pilgrim](#))

Mark Pilgrim was describing in particular the experience a Python programmer has when dealing with libxml2. The default Python bindings of libxml2 are fast, thrilling, powerful, and your code might fail in some horrible way that you really shouldn’t have to worry about when writing Python code. lxml combines the power of libxml2 with the ease of use of Python.

### 2.2 Aims

The C libraries [libxml2](#) and [libxslt](#) have huge benefits:

- Standards-compliant XML support.
- Support for (broken) HTML.
- Full-featured.
- Actively maintained by XML experts.
- fast. fast! FAST!

These libraries already ship with Python bindings, but these Python bindings mimic the C-level interface. This yields a number of problems:

- very low level and C-ish (not Pythonic).
- underdocumented and huge, you get lost in them.
- UTF-8 in API, instead of Python unicode strings.

- Can easily cause segfaults from Python.
- Require manual memory management!

lxml is a new Python binding for libxml2 and libxslt, completely independent from these existing Python bindings. Its aims:

- Pythonic API.
- Documented.
- Use Python unicode strings in API.
- Safe (no segfaults).
- No manual memory management!

lxml aims to provide a Pythonic API by following as much as possible the [ElementTree API](#). We're trying to avoid inventing too many new APIs, or you having to learn new things -- XML is complicated enough.

## Chapter 3

# Installing lxml

### 3.1 Requirements

You need Python 2.3 or later.

You need libxml2 and libxslt, in particular:

- libxml2 2.6.20 or later. It can be found here: <http://xmlsoft.org/downloads.html>

If you want to use XPath, do not use libxml2 2.6.27.

- libxslt 1.1.15 or later. It can be found here: <http://xmlsoft.org/XSLT/downloads.html>

Newer versions generally contain less bugs and are therefore recommended. The HTML parser benefits from libxml2 version 2.6.21 or later, which support parsing horribly broken HTML. XML Schema support is also still worked on in libxml2, so newer versions will give you better compliance with the W3C spec.

### 3.2 Installation

If you have `easy_install`, you can run the following as super-user (or administrator):

```
easy_install lxml
```

This has been reported to work on Linux, MacOS-X 10.4 and Windows, as long as libxml2 and libxslt are properly installed (including development packages, i.e. header files, etc.).

### 3.3 Building lxml from sources

If you want to build lxml from SVN you should read [how to build lxml from source](#) (or the file `build.txt` in the `doc` directory of the source tree). Building from Subversion sources or from modified distribution sources requires [Cython](#) to translate the lxml sources into C code. The source distribution ships with pre-generated C source files, so you do not need Cython installed to build from release sources.

If you have read these instructions and still cannot manage to install lxml, you can check the archives of the [mailing list](#) to see if your problem is known or otherwise send a mail to the list.

## 3.4 MS Windows

For MS Windows, the [binary egg distribution of lxml](#) is statically built against the libraries, i.e. it already includes them. There is no need to install the external libraries if you use an official lxml build from cheeseshop.

If you want to upgrade the libraries and/or compile lxml from sources, you should install a [binary distribution](#) of libxml2 and libxslt. You need both libxml2 and libxslt, as well as iconv and zlib.

## 3.5 MacOS-X

On MacOS-X 10.4, you can try to use the installed system libraries when you build lxml yourself. However, the library versions on this system are older than the required versions, so you may encounter certain differences in behaviour or even crashes. A number of users reported success with updated libraries (e.g. using [fink](#)), but needed to set the environment variable DYLD\_LIBRARY\_PATH to the directory where fink keeps the libraries.

A MacPort of lxml is available. Try `port install py25-lxml`.

## Chapter 4

# What's new in lxml 2.0?

During the development of the lxml 1.x series, a couple of quirks were discovered in the design that made the API less obvious and its future extensions harder than necessary. lxml 2.0 is a soft evolution of lxml 1.x towards a simpler, more consistent and more powerful API - with some major extensions. Wherever possible, lxml 1.3 comes close to the semantics of lxml 2.0, so that migrating should be easier for code that currently runs with 1.3.

One of the important internal changes was the switch from the [Pyrex](#) compiler to [Cython](#), which provides better optimisation and improved support for newer Python language features. This allows the code of lxml to become more Python-like again, while the performance improves as Cython continues its own development. The code simplification, which will continue throughout the 2.x series, will hopefully make it even easier for users to contribute.

### 4.1 Changes in etree and objectify

A graduation towards a more consistent API cannot go without a certain amount of incompatible changes. The following is a list of those differences that applications need to take into account when migrating from lxml 1.x to lxml 2.0.

#### 4.1.1 Incompatible changes

- lxml 0.9 introduced a feature called [namespace implementation](#). The global `Namespace` factory was added to register custom element classes and have `lxml.etree` look them up automatically. However, the later development of further class lookup mechanisms made it appear less and less adequate to register this mapping at a global level, so lxml 1.1 first removed the namespace based lookup from the default setup and lxml 2.0 finally removes the global namespace registry completely. As all other lookup mechanisms, the namespace lookup is now local to a parser, including the registry itself. Applications that use a module-level parser can easily map its `get_namespace()` method to a global `Namespace` function to mimic the old behaviour.
- Some API functions now require passing options as keyword arguments, as opposed to positional arguments. This restriction was introduced to make the API usage independent of future extensions such as the addition of new positional arguments. Users should not rely on the position of an optional argument in function signatures and instead pass it explicitly named. This also improves code readability - it is common good practice to pass options in a consistent way independent of

their position, so many people may not even notice the change in their code. Another important reason is compatibility with `cElementTree`, which also enforces keyword-only arguments in a couple of places.

- XML tag names are validated when creating an `Element`. This does not apply to HTML tags, where only HTML special characters are forbidden. The distinction is made by the `SubElement()` factory, which tests if the tree it works on is an HTML tree, and by the `.makeelement()` methods of parsers, which behave differently for the `XMLParser()` and the `HTMLParser()`.
- XPath now raises exceptions specific to the part of the execution that failed: `XPathSyntaxError` for parser errors and `XPathEvalError` for errors that occurred during the evaluation. Note that the distinction only works for the `XPath()` class. The other two evaluators only have a single evaluation call that includes the parsing step, and will therefore only raise an `XPathEvalError`. Applications can catch both exceptions through the common base class `XPathError` (which also exists in earlier `lxml` versions).
- Network access in parsers is now disabled by default, i.e. the `no_network` option defaults to `True`. Due to a somewhat 'interesting' implementation in `libxml2`, this does not affect the first document (i.e. the URL that is parsed), but only subsequent documents, such as a DTD when parsing with validation. This means that you will have to check the URL you pass, instead of relying on `lxml` to prevent *any* access to external resources. As this can be helpful in some use cases, `lxml` does not work around it.
- The type annotations in `lxml.objectify` (the `pytype` attribute) now use `NoneType` for the `None` value as this is the correct Python type name. Previously, `lxml 1.x` used a lower case `none`.
- Another change in `objectify` regards the way it deals with ambiguous types. Previously, setting a value like the string `"3"` through normal attribute access would let it come back as an integer when reading the object attribute. `lxml 2.0` prevents this by always setting the `pytype` attribute to the type the user passed in, so `"3"` will come back as a string, while the number `3` will come back as a number. To remove the type annotation on serialisation, you can use the `deannotate()` function.
- The C-API function `findOrBuildNodeNs()` was replaced by the more generic `findOrBuildNodeNsPrefix()` that accepts an additional default prefix.

### 4.1.2 Enhancements

Most of the enhancements of `lxml 2.0` were made under the hood. Most people won't even notice them, but they make the maintenance of `lxml` easier and thus facilitate further enhancements and an improved integration between `lxml`'s features.

- `lxml.objectify` now has its own implementation of the [E factory](#). It uses the built-in type lookup mechanism of `lxml.objectify`, thus removing the need for an additional type registry mechanism (as previously available through the `typemap` parameter).
- XML entities are supported through the `Entity()` factory, an `Entity` element class and a parser option `resolve_entities` that allows to keep entities in the element tree when set to `False`. Also, the parser will now report undefined entities as errors if it needs to resolve them (which is still the default, as in `lxml 1.x`).
- A major part of the XPath code was rewritten and can now benefit from a bigger overlap with the XSLT code. The main benefits are improved thread safety in the XPath evaluators and Python RegExp support in standard XPath.
- The string results of an XPath evaluation have become 'smart' string subclasses. Formerly, there was no easy way to find out where a string originated from. In `lxml 2.0`, you can call its `getparent()`

method to [find the Element that carries it](#). This works for attributes (`//@attribute`) and for `text()` nodes, i.e. Element text and tails. Strings that were constructed in the path expression, e.g. by the `string()` function or extension functions, will return None as their parent.

- Setting a QName object as value of the `.text` property or as an attribute value will resolve its prefix in the respective context
- Following ElementTree 1.3, the `iterfind()` method supports efficient iteration based on XPath-like expressions.

The parsers also received some major enhancements:

- `iterparse()` can parse HTML when passing the boolean `html` keyword.
- Parse time XML Schema validation by passing an XMLSchema object to the `schema` keyword argument of a parser.
- Support for a `target` object that implements ElementTree's [TreeBuilder interface](#).
- The `encoding` keyword allows overriding the document encoding.

### 4.1.3 Deprecation

The following functions and methods are now deprecated. They are still available in lxml 2.0 and will be removed in lxml 2.1:

- The `tounicode()` function was replaced by the call `tostring(encoding=unicode)`.
- CamelCaseNamed module functions and methods were renamed to their underscore equivalents to follow [PEP 8](#) in naming.
  - `etree.setDefaultParser()` -> `etree.set_default_parser()`
  - `etree.getDefaultParser()` -> `etree.get_default_parser()`
  - `etree.useGlobalPythonLog()` -> `etree.use_global_python_log()`
  - `XMLParser.setElementClassLookup()` -> `.set_element_class_lookup()`
  - `HTMLParser.setElementClassLookup()` -> `.set_element_class_lookup()`
- The `.getiterator()` method on Elements and ElementTrees was renamed to `.iter()` to follow ElementTree 1.3.

## 4.2 New modules

The most visible changes in lxml 2.0 regard the new modules that were added.

### 4.2.1 lxml.usedoctest

A very useful module for doctests based on XML or HTML is `lxml.doctestcompare`. It provides a relaxed comparison mechanism for XML and HTML in doctests. Using it for XML comparisons is as simple as:

```
>>> import lxml.usedoctest
```

and for HTML comparisons:

```
>>> import lxml.html.usedoctest
```

### 4.2.2 `lxml.html`

The largest new package that was added to lxml 2.0 is [lxml.html](#). It contains various tools and modules for HTML handling. The major features include support for cleaning up HTML (removing unwanted content), a readable HTML diff and various tools for working with links.

### 4.2.3 `lxml.cssselect`

The Cascading Stylesheet Language ([CSS](#)) has a very short and generic path language for pointing at elements in XML/HTML trees ([CSS selectors](#)). The module [lxml.cssselect](#) provides an implementation based on XPath.

## Chapter 5

# Benchmarks and Speed

**Author:** Stefan Behnel

As an XML library, lxml.etree is very fast. It is also slow. As with all software, it depends on what you do with it. Rest assured that lxml is fast enough for most applications, so lxml is probably somewhere between 'fast enough' and 'the best choice' for yours. Read some [messages](#) from [happy users](#) to see what we mean.

This text describes where lxml.etree (abbreviated to 'lxe') excels, gives hints on some performance traps and compares the overall performance to the original [ElementTree](#) (ET) and [cElementTree](#) (cET) libraries by Fredrik Lundh. The cElementTree library is a fast C-implementation of the original ElementTree.

### 5.1 General notes

First thing to say: there *is* an overhead involved in having a DOM-like C library mimic the ElementTree API. As opposed to ElementTree, lxml has to generate Python representations of tree nodes on the fly when asked for them, and the internal tree structure of libxml2 results in a higher maintenance overhead than the simpler top-down structure of ElementTree. What this means is: the more of your code runs in Python, the less you can benefit from the speed of lxml and libxml2. Note, however, that this is true for most performance critical Python applications. No one would implement fourier transformations in pure Python when you can use NumPy.

The up side then is that lxml provides powerful tools like tree iterators, XPath and XSLT, that can handle complex operations at the speed of C. Their pythonic API in lxml makes them so flexible that most applications can easily benefit from them.

### 5.2 How to read the timings

The statements made here are backed by the (micro-)benchmark scripts [bench\\_etree.py](#), [bench\\_xpath.py](#) and [bench\\_objectify.py](#) that come with the lxml source distribution. They are distributed under the same BSD license as lxml itself, and the lxml project would like to promote them as a general benchmarking suite for all ElementTree implementations. New benchmarks are very easy to add as tiny test methods, so if you write a performance test for a specific part of the API yourself, please consider sending it to the lxml mailing list.

The timings cited below compare lxml 2.0 final (with libxml2 2.6.31) to the January 2008 SVN trunk versions of ElementTree (1.3alpha) and cElementTree (1.2.7). They were run single-threaded on a 1.8GHz Intel Core Duo machine under Ubuntu Linux 7.10 (Gutsy). The C libraries were compiled with the same platform specific optimisation flags. The Python interpreter (2.5.1) was used as provided by the distribution.

The scripts run a number of simple tests on the different libraries, using different XML tree configurations: different tree sizes (T1-4), with or without attributes (-/A), with or without ASCII string or unicode text (-/S/U), and either against a tree or its serialised XML form (T/X). In the result extracts cited below, T1 refers to a 3-level tree with many children at the third level, T2 is swapped around to have many children below the root element, T3 is a deep tree with few children at each level and T4 is a small tree, slightly broader than deep. If repetition is involved, this usually means running the benchmark in a loop over all children of the tree root, otherwise, the operation is run on the root node (C/R).

As an example, the character code (SATR T1) states that the benchmark was running for tree T1, with plain string text (S) and attributes (A). It was run against the root element (R) in the tree structure of the data (T).

Note that very small operations are repeated in integer loops to make them measurable. It is therefore not always possible to compare the absolute timings of, say, a single access benchmark (which usually loops) and a 'get all in one step' benchmark, which already takes enough time to be measurable and is therefore measured as is. An example is the index access to a single child, which cannot be compared to the timings for `getchildren()`. Take a look at the concrete benchmarks in the scripts to understand how the numbers compare.

### 5.3 Parsing and Serialising

Serialisation is an area where lxml excels. The reason is that it executes entirely at the C level, without any interaction with Python code. The results are rather impressive, especially for UTF-8, which is native to libxml2. While 20 to 40 times faster than (c)ElementTree 1.2, lxml is still more than 5 times as fast as the much improved ElementTree 1.3:

```

lxe: tostring_utf16 (SATR T1)  19.0921 msec/pass
cET: tostring_utf16 (SATR T1) 129.8430 msec/pass
ET : tostring_utf16 (SATR T1) 136.1301 msec/pass

lxe: tostring_utf16 (UATR T1)  20.4630 msec/pass
cET: tostring_utf16 (UATR T1) 130.1570 msec/pass
ET : tostring_utf16 (UATR T1) 136.3101 msec/pass

lxe: tostring_utf16 (S-TR T2)  18.8632 msec/pass
cET: tostring_utf16 (S-TR T2) 136.9388 msec/pass
ET : tostring_utf16 (S-TR T2) 143.9550 msec/pass

lxe: tostring_utf8  (S-TR T2)  14.4310 msec/pass
cET: tostring_utf8  (S-TR T2) 137.0859 msec/pass
ET : tostring_utf8  (S-TR T2) 144.3110 msec/pass

lxe: tostring_utf8  (U-TR T3)   2.6381 msec/pass
cET: tostring_utf8  (U-TR T3)  52.1040 msec/pass
ET : tostring_utf8  (U-TR T3)  53.1070 msec/pass

```

For parsing, on the other hand, the advantage is clearly with cElementTree. The (c)ET libraries use a very thin layer on top of the expat parser, which is known to be extremely fast:

```

lxe: parse_stringIO (SAXR T1) 144.1851 msec/pass
cET: parse_stringIO (SAXR T1) 14.4269 msec/pass
ET : parse_stringIO (SAXR T1) 245.9190 msec/pass

lxe: parse_stringIO (S-XR T3) 5.6100 msec/pass
cET: parse_stringIO (S-XR T3) 5.3229 msec/pass
ET : parse_stringIO (S-XR T3) 82.4831 msec/pass

lxe: parse_stringIO (UAXR T3) 23.4420 msec/pass
cET: parse_stringIO (UAXR T3) 30.2689 msec/pass
ET : parse_stringIO (UAXR T3) 165.7169 msec/pass

```

While about as fast for smaller documents, the expat parser allows cET to be up to 10 times faster than lxml on plain parser performance for large input documents. Similar timings can be observed for the `iterparse()` function:

```

lxe: iterparse_stringIO (SAXR T1) 160.3689 msec/pass
cET: iterparse_stringIO (SAXR T1) 19.1891 msec/pass
ET : iterparse_stringIO (SAXR T1) 274.8971 msec/pass

lxe: iterparse_stringIO (UAXR T3) 24.9629 msec/pass
cET: iterparse_stringIO (UAXR T3) 31.7740 msec/pass
ET : iterparse_stringIO (UAXR T3) 173.8000 msec/pass

```

However, if you benchmark the complete round-trip of a serialise-parse cycle, the numbers will look similar to these:

```

lxe: write_utf8_parse_stringIO (S-TR T1) 160.0718 msec/pass
cET: write_utf8_parse_stringIO (S-TR T1) 207.6778 msec/pass
ET : write_utf8_parse_stringIO (S-TR T1) 450.2120 msec/pass

lxe: write_utf8_parse_stringIO (UATR T2) 173.5830 msec/pass
cET: write_utf8_parse_stringIO (UATR T2) 253.0849 msec/pass
ET : write_utf8_parse_stringIO (UATR T2) 519.2261 msec/pass

lxe: write_utf8_parse_stringIO (S-TR T3) 8.4269 msec/pass
cET: write_utf8_parse_stringIO (S-TR T3) 75.7639 msec/pass
ET : write_utf8_parse_stringIO (S-TR T3) 156.1930 msec/pass

lxe: write_utf8_parse_stringIO (SATR T4) 1.2100 msec/pass
cET: write_utf8_parse_stringIO (SATR T4) 6.4859 msec/pass
ET : write_utf8_parse_stringIO (SATR T4) 9.9051 msec/pass

```

For applications that require a high parser throughput and do little serialization, cET is the best choice. Also for iterparse applications that extract small amounts of data from large XML data sets. If it comes to round-trip performance, however, lxml tends to be between 30% and multiple times faster in total. So, whenever the input documents are not considerably bigger than the output, lxml is the clear winner.

## 5.4 The ElementTree API

Since all three libraries implement the same API, their performance is easy to compare in this area. A major disadvantage for lxml's performance is the different tree model that underlies libxml2. It allows lxml to provide parent pointers for elements, but also increases the overhead of tree building and restructuring. This can be seen from the tree setup times of the benchmark (given in seconds):

```

lxe:      --      S-      U-      -A      SA      UA
T1: 0.0783 0.0777 0.0774 0.0787 0.0781 0.0783
T2: 0.0799 0.0796 0.0799 0.0879 0.0882 0.0886
T3: 0.0245 0.0216 0.0217 0.0577 0.0575 0.0572
T4: 0.0003 0.0003 0.0003 0.0011 0.0011 0.0011
cET:      --      S-      U-      -A      SA      UA
T1: 0.0272 0.0264 0.0267 0.0268 0.0261 0.0265
T2: 0.0280 0.0274 0.0273 0.0273 0.0276 0.0275
T3: 0.0065 0.0066 0.0065 0.0111 0.0088 0.0088
T4: 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001
ET :      --      S-      U-      -A      SA      UA
T1: 0.1302 0.1903 0.2208 0.1265 0.2542 0.1267
T2: 0.2994 0.1301 0.3402 0.3746 0.1326 0.4170
T3: 0.0301 0.0310 0.0302 0.0348 0.3654 0.0349
T4: 0.0006 0.0005 0.0008 0.0006 0.0007 0.0006

```

While lxml is still faster than ET in most cases (10-70%), cET can be up to three times faster than lxml here. One of the reasons is that lxml must additionally discard the created Python elements after their use, when they are no longer referenced. ET and cET represent the tree itself through these objects, which reduces the overhead in creating them.

### 5.4.1 Child access

The same reason makes operations like collecting children as in `list(element)` more costly in lxml. Where ET and cET can quickly create a shallow copy of their list of children, lxml has to create a Python object for each child and collect them in a list:

```

lxe: root_list_children      (--TR T1)      0.0160 msec/pass
cET: root_list_children      (--TR T1)      0.0081 msec/pass
ET : root_list_children      (--TR T1)      0.0541 msec/pass

lxe: root_list_children      (--TR T2)      0.2100 msec/pass
cET: root_list_children      (--TR T2)      0.0319 msec/pass
ET : root_list_children      (--TR T2)      0.4420 msec/pass

```

This handicap is also visible when accessing single children:

```

lxe: first_child      (--TR T2)      0.2429 msec/pass
cET: first_child      (--TR T2)      0.2170 msec/pass
ET : first_child      (--TR T2)      0.9968 msec/pass

lxe: last_child      (--TR T1)      0.2470 msec/pass
cET: last_child      (--TR T1)      0.2291 msec/pass
ET : last_child      (--TR T1)      0.9830 msec/pass

```

... unless you also add the time to find a child index in a bigger list. ET and cET use Python lists here, which are based on arrays. The data structure used by libxml2 is a linked tree, and thus, a linked list of children:

```

lxe: middle_child      (--TR T1)      0.2759 msec/pass
cET: middle_child      (--TR T1)      0.2229 msec/pass
ET : middle_child      (--TR T1)      1.0030 msec/pass

lxe: middle_child      (--TR T2)      1.7071 msec/pass
cET: middle_child      (--TR T2)      0.2229 msec/pass

```

```
ET : middle_child          (--TR T2)    0.9930 msec/pass
```

### 5.4.2 Element creation

As opposed to ET, libxml2 has a notion of documents that each element must be in. This results in a major performance difference for creating independent Elements that end up in independently created documents:

```
lxe: create_elements      (--TC T2)    2.8961 msec/pass
cET: create_elements      (--TC T2)    0.1929 msec/pass
ET : create_elements      (--TC T2)    1.3590 msec/pass
```

Therefore, it is always preferable to create Elements for the document they are supposed to end up in, either as SubElements of an Element or using the explicit `Element.makeelement()` call:

```
lxe: makeelement          (--TC T2)    1.9000 msec/pass
cET: makeelement          (--TC T2)    0.3211 msec/pass
ET : makeelement          (--TC T2)    1.6358 msec/pass

lxe: create_subelements   (--TC T2)    1.7891 msec/pass
cET: create_subelements   (--TC T2)    0.2351 msec/pass
ET : create_subelements   (--TC T2)    3.2270 msec/pass
```

So, if the main performance bottleneck of an application is creating large XML trees in memory through calls to Element and SubElement, cET is the best choice. Note, however, that the serialisation performance may even out this advantage, especially for smaller trees and trees with many attributes.

### 5.4.3 Merging different sources

A critical action for lxml is moving elements between document contexts. It requires lxml to do recursive adaptations throughout the moved tree structure.

The following benchmark appends all root children of the second tree to the root of the first tree:

```
lxe: append_from_document  (--TR T1,T2)  3.3841 msec/pass
cET: append_from_document  (--TR T1,T2)  0.2699 msec/pass
ET : append_from_document  (--TR T1,T2)  1.2650 msec/pass

lxe: append_from_document  (--TR T3,T4)  0.0441 msec/pass
cET: append_from_document  (--TR T3,T4)  0.0169 msec/pass
ET : append_from_document  (--TR T3,T4)  0.0820 msec/pass
```

Although these are fairly small numbers compared to parsing, this easily shows the different performance classes for lxml and (c)ET. Where the latter do not have to care about parent pointers and tree structures, lxml has to deep traverse the appended tree. The performance difference therefore increases with the size of the tree that is moved.

This difference is not always as visible, but applies to most parts of the API, like inserting newly created elements:

```
lxe: insert_from_document  (--TR T1,T2)  5.7020 msec/pass
cET: insert_from_document  (--TR T1,T2)  0.4041 msec/pass
ET : insert_from_document  (--TR T1,T2)  1.4789 msec/pass
```

or replacing the child slice by a newly created element:

```

lxe: replace_children_element  (--TC T1)    0.2210 msec/pass
cET: replace_children_element  (--TC T1)    0.0238 msec/pass
ET : replace_children_element  (--TC T1)    0.1600 msec/pass

```

as opposed to replacing the slice with an existing element from the same document:

```

lxe: replace_children          (--TC T1)    0.0179 msec/pass
cET: replace_children          (--TC T1)    0.0119 msec/pass
ET : replace_children          (--TC T1)    0.0739 msec/pass

```

You should keep this difference in mind when you merge very large trees.

#### 5.4.4 deepcopy

Deep copying a tree is fast in lxml:

```

lxe: deepcopy_all              (--TR T1)    9.7558 msec/pass
cET: deepcopy_all              (--TR T1)   120.6188 msec/pass
ET : deepcopy_all              (--TR T1)   902.6880 msec/pass

lxe: deepcopy_all              (-ATR T2)   12.3210 msec/pass
cET: deepcopy_all              (-ATR T2)   136.9810 msec/pass
ET : deepcopy_all              (-ATR T2)   944.2801 msec/pass

lxe: deepcopy_all              (S-TR T3)    8.3981 msec/pass
cET: deepcopy_all              (S-TR T3)   35.6541 msec/pass
ET : deepcopy_all              (S-TR T3)   221.6041 msec/pass

```

So, for example, if you have a database-like scenario where you parse in a large tree and then search and copy independent subtrees from it for further processing, lxml is by far the best choice here.

#### 5.4.5 Tree traversal

Another area where lxml is very fast is iteration for tree traversal. If your algorithms can benefit from step-by-step traversal of the XML tree and especially if few elements are of interest or the target element tag name is known, lxml is a good choice:

```

lxe: getiterator_all           (--TR T1)    5.7251 msec/pass
cET: getiterator_all           (--TR T1)   39.9489 msec/pass
ET : getiterator_all           (--TR T1)   23.0000 msec/pass

lxe: getiterator_islice        (--TR T2)    0.0830 msec/pass
cET: getiterator_islice        (--TR T2)    0.3440 msec/pass
ET : getiterator_islice        (--TR T2)    0.2429 msec/pass

lxe: getiterator_tag           (--TR T2)    0.3011 msec/pass
cET: getiterator_tag           (--TR T2)   14.1001 msec/pass
ET : getiterator_tag           (--TR T2)    7.4241 msec/pass

lxe: getiterator_tag_all       (--TR T2)    0.6340 msec/pass
cET: getiterator_tag_all       (--TR T2)   40.7901 msec/pass
ET : getiterator_tag_all       (--TR T2)   21.0390 msec/pass

```

This translates directly into similar timings for `Element.findall()`:

lxe: findall	(--TR T2)	7.8950 msec/pass
cET: findall	(--TR T2)	44.5340 msec/pass
ET : findall	(--TR T2)	27.1149 msec/pass
lxe: findall	(--TR T3)	1.7281 msec/pass
cET: findall	(--TR T3)	12.9611 msec/pass
ET : findall	(--TR T3)	8.6131 msec/pass
lxe: findall_tag	(--TR T2)	0.7720 msec/pass
cET: findall_tag	(--TR T2)	40.6358 msec/pass
ET : findall_tag	(--TR T2)	21.4581 msec/pass
lxe: findall_tag	(--TR T3)	0.2050 msec/pass
cET: findall_tag	(--TR T3)	9.6831 msec/pass
ET : findall_tag	(--TR T3)	5.2109 msec/pass

Note that all three libraries currently use the same Python implementation for `findall()`, except for their native tree iterator.

## 5.5 XPath

The following timings are based on the benchmark script [bench\\_xpath.py](#).

This part of lxml does not have an equivalent in ElementTree. However, lxml provides more than one way of accessing it and you should take care which part of the lxml API you use. The most straight forward way is to call the `xpath()` method on an Element or ElementTree:

lxe: xpath_method	(--TC T1)	1.7459 msec/pass
lxe: xpath_method	(--TC T2)	22.0850 msec/pass
lxe: xpath_method	(--TC T3)	0.1309 msec/pass
lxe: xpath_method	(--TC T4)	1.0772 msec/pass

This is well suited for testing and when the XPath expressions are as diverse as the trees they are called on. However, if you have a single XPath expression that you want to apply to a larger number of different elements, the XPath class is the most efficient way to do it:

lxe: xpath_class	(--TC T1)	0.6740 msec/pass
lxe: xpath_class	(--TC T2)	3.1760 msec/pass
lxe: xpath_class	(--TC T3)	0.0548 msec/pass
lxe: xpath_class	(--TC T4)	0.1700 msec/pass

Note that this still allows you to use variables in the expression, so you can parse it once and then adapt it through variables at call time. In other cases, where you have a fixed Element or ElementTree and want to run different expressions on it, you should consider the XPathEvaluator:

lxe: xpath_element	(--TR T1)	0.4151 msec/pass
lxe: xpath_element	(--TR T2)	11.6129 msec/pass
lxe: xpath_element	(--TR T3)	0.1299 msec/pass
lxe: xpath_element	(--TR T4)	0.3409 msec/pass

While it looks slightly slower, creating an XPath object for each of the expressions generates a much higher overhead here:

lxe: xpath_class_repeat	(--TC T1)	1.6699 msec/pass
lxe: xpath_class_repeat	(--TC T2)	20.4420 msec/pass

```
lxe: xpath_class_repeat  (--TC T3)    0.1230 msec/pass
lxe: xpath_class_repeat  (--TC T4)    0.9859 msec/pass
```

## 5.6 A longer example

... based on lxml 1.3.

A while ago, Uche Ogbuji posted a [benchmark proposal](#) that would read in a 3MB XML version of the [Old Testament](#) of the Bible and look for the word *begat* in all verses. Apparently, it is contained in 120 out of almost 24000 verses. This is easy to implement in ElementTree using `findall()`. However, the fastest and most memory friendly way to do this is obviously `iterparse()`, as most of the data is not of any interest.

Now, Uche's original proposal was more or less the following:

```
def bench_ET():
    tree = ElementTree.parse("ot.xml")
    result = []
    for v in tree.findall("//v"):
        text = v.text
        if 'begat' in text:
            result.append(text)
    return len(result)
```

which takes about one second on my machine today. The faster `iterparse()` variant looks like this:

```
def bench_ET_iterparse():
    result = []
    for event, v in ElementTree.iterparse("ot.xml"):
        if v.tag == 'v':
            text = v.text
            if 'begat' in text:
                result.append(text)
        v.clear()
    return len(result)
```

The improvement is about 10%. At the time I first tried (early 2006), lxml didn't have `iterparse()` support, but the `findall()` variant was already faster than ElementTree. This changes immediately when you switch to cElementTree. The latter only needs 0.17 seconds to do the trick today and only some impressive 0.10 seconds when running the `iterparse` version. And even back then, it was quite a bit faster than what lxml could achieve.

Since then, lxml has matured a lot and has gotten much faster. The `iterparse` variant now runs in 0.14 seconds, and if you remove the `v.clear()`, it is even a little faster (which isn't the case for cElementTree).

One of the many great tools in lxml is XPath, a swiss army knife for finding things in XML documents. It is possible to move the whole thing to a pure XPath implementation, which looks like this:

```
def bench_lxml_xpath_all():
    tree = etree.parse("ot.xml")
    result = tree.xpath("//v[contains(., 'begat')]/text()")
    return len(result)
```

This runs in about 0.13 seconds and is about the shortest possible implementation (in lines of Python code) that I could come up with. Now, this is already a rather complex XPath expression compared to

the simple “//v” ElementPath expression we started with. Since this is also valid XPath, let’s try this instead:

```
def bench_lxml_xpath():
    tree = etree.parse("ot.xml")
    result = []
    for v in tree.xpath("//v"):
        text = v.text
        if 'begat' in text:
            result.append(text)
    return len(result)
```

This gets us down to 0.12 seconds, thus showing that a generic XPath evaluation engine cannot always compete with a simpler, tailored solution. However, since this is not much different from the original findall variant, we can remove the complexity of the XPath call completely and just go with what we had in the beginning. Under lxml, this runs in the same 0.12 seconds.

But there is one thing left to try. We can replace the simple ElementPath expression with a native tree iterator:

```
def bench_lxml_getiterator():
    tree = etree.parse("ot.xml")
    result = []
    for v in tree.getiterator("v"):
        text = v.text
        if 'begat' in text:
            result.append(text)
    return len(result)
```

This implements the same thing, just without the overhead of parsing and evaluating a path expression. And this makes it another bit faster, down to 0.11 seconds. For comparison, cElementTree runs this version in 0.17 seconds.

So, what have we learned?

- Python code is not slow. The pure XPath solution was not even as fast as the first shot Python implementation. In general, a few more lines in Python make things more readable, which is much more important than the last 5% of performance.
- It’s important to know the available options - and it’s worth starting with the most simple one. In this case, a programmer would then probably have started with `getiterator("v")` or `iterparse()`. Either of them would already have been the most efficient, depending on which library is used.
- It’s important to know your tool. lxml and cElementTree are both very fast libraries, but they do not have the same performance characteristics. The fastest solution in one library can be comparatively slow in the other. If you optimise, optimise for the specific target platform.
- It’s not always worth optimising. After all that hassle we got from 0.12 seconds for the initial implementation to 0.11 seconds. Switching over to cElementTree and writing an `iterparse()` based version would have given us 0.10 seconds - not a big difference for 3MB of XML.
- Take care what operation is really dominating in your use case. If we split up the operations, we can see that lxml is slightly slower than cElementTree on `parse()` (both about 0.06 seconds), but more visibly slower on `iterparse()`: 0.07 versus 0.10 seconds. However, tree iteration in lxml is incredibly fast, so it can be better to parse the whole tree and then iterate over it rather than using `iterparse()` to do both in one step. Or, you can just wait for the lxml authors to optimise `iterparse` in one of the next releases...

## 5.7 lxml.objectify

The following timings are based on the benchmark script [bench\\_objectify.py](#).

Objectify is a data-binding API for XML based on lxml.etree, that was added in version 1.1. It uses standard Python attribute access to traverse the XML tree. It also features ObjectPath, a fast path language based on the same meme.

Just like lxml.etree, lxml.objectify creates Python representations of elements on the fly. To save memory, the normal Python garbage collection mechanisms will discard them when their last reference is gone. In cases where deeply nested elements are frequently accessed through the objectify API, the create-discard cycles can become a bottleneck, as elements have to be instantiated over and over again.

### 5.7.1 ObjectPath

ObjectPath can be used to speed up the access to elements that are deep in the tree. It avoids step-by-step Python element instantiations along the path, which can substantially improve the access time:

lxe: attribute	(--TR T1)	9.4581 msec/pass
lxe: attribute	(--TR T2)	52.5560 msec/pass
lxe: attribute	(--TR T4)	9.1729 msec/pass
lxe: objectpath	(--TR T1)	4.8690 msec/pass
lxe: objectpath	(--TR T2)	47.8780 msec/pass
lxe: objectpath	(--TR T4)	4.7870 msec/pass
lxe: attributes_deep	(--TR T1)	54.7471 msec/pass
lxe: attributes_deep	(--TR T2)	62.7451 msec/pass
lxe: attributes_deep	(--TR T4)	15.1050 msec/pass
lxe: objectpath_deep	(--TR T1)	48.2810 msec/pass
lxe: objectpath_deep	(--TR T2)	51.3949 msec/pass
lxe: objectpath_deep	(--TR T4)	6.1419 msec/pass

Note, however, that parsing ObjectPath expressions is not for free either, so this is most effective for frequently accessing the same element.

### 5.7.2 Caching Elements

A way to improve the normal attribute access time is static instantiation of the Python objects, thus trading memory for speed. Just create a cache dictionary and run:

```
cache[root] = list(root.iter())
```

after parsing and:

```
del cache[root]
```

when you are done with the tree. This will keep the Python element representations of all elements alive and thus avoid the overhead of repeated Python object creation. You can also consider using filters or generator expressions to be more selective. By choosing the right trees (or even subtrees and elements) to cache, you can trade memory usage against access speed:

lxe: attribute_cached	(--TR T1)	7.5061 msec/pass
-----------------------	-----------	------------------

lxe: attribute_cached	(--TR T2)	50.1881 msec/pass
lxe: attribute_cached	(--TR T4)	7.4170 msec/pass
lxe: attributes_deep_cached	(--TR T1)	48.7239 msec/pass
lxe: attributes_deep_cached	(--TR T2)	55.2199 msec/pass
lxe: attributes_deep_cached	(--TR T4)	9.9740 msec/pass
lxe: objectpath_deep_cached	(--TR T1)	43.4160 msec/pass
lxe: objectpath_deep_cached	(--TR T2)	47.6480 msec/pass
lxe: objectpath_deep_cached	(--TR T4)	3.4680 msec/pass

Things to note: you cannot currently use `weakref.WeakKeyDictionary` objects for this as `lxml`'s element objects do not support weak references (which are costly in terms of memory). Also note that new element objects that you add to these trees will not turn up in the cache automatically and will therefore still be garbage collected when all their Python references are gone, so this is most effective for largely immutable trees. You should consider using a set instead of a list in this case and add new elements by hand.

### 5.7.3 Further optimisations

Here are some more things to try if optimisation is required:

- A lot of time is usually spent in tree traversal to find the addressed elements in the tree. If you often work in subtrees, do what you would also do with deep Python objects: assign the parent of the subtree to a variable or pass it into functions instead of starting at the root. This allows accessing its descendents more directly.
- Try assigning data values directly to attributes instead of passing them through `DataElement`.
- If you use custom data types that are costly to parse, try running `objectify.annotate()` over read-only trees to speed up the attribute type inference on read access.

Note that none of these measures is guaranteed to speed up your application. As usual, you should prefer readable code over premature optimisations and profile your expected use cases before bothering to apply optimisations at random.

## Chapter 6

# ElementTree compatibility of lxml.etree

A lot of care has been taken to ensure compatibility between etree and ElementTree. Nonetheless, some differences and incompatibilities exist:

- Importing etree is obviously different; etree uses a lower-case package name, while ElementTree uses a combination of upper-case and lower case in imports:

```
# etree
from lxml.etree import Element

# ElementTree
from elementtree.ElementTree import Element

# ElementTree in the Python 2.5 standard library
from xml.etree.ElementTree import Element
```

When switching over code from ElementTree to lxml.etree, and you're using the package name prefix 'ElementTree', you can do the following:

```
# instead of
from elementtree import ElementTree
# use
from lxml import etree as ElementTree
```

- lxml.etree offers a lot more functionality, such as XPath, XSLT, Relax NG, and XML Schema support, which (c)ElementTree does not offer.
- etree has a different idea about Python unicode strings than ElementTree. In most parts of the API, ElementTree uses plain strings and unicode strings as what they are. This includes Element.text, Element.tail and many other places. However, the ElementTree parsers assume by default that any string (*str* or *unicode*) contains ASCII data. They raise an exception if strings do not match the expected encoding.

etree has the same idea about plain strings (*str*) as ElementTree. For unicode strings, however, etree assumes throughout the API that they are Python unicode encoded strings rather than byte data. This includes the parsers. It is therefore perfectly correct to pass XML unicode data into the etree parsers in form of Python unicode strings. It is an error, on the other hand, if unicode strings specify an encoding in their XML declaration, as this conflicts with the characteristic encoding of

Python unicode strings.

- ElementTree allows you to place an Element in two different trees at the same time. Thus, this:

```
a = Element('a')
b = SubElement(a, 'b')
c = Element('c')
c.append(b)
```

will result in the following tree a:

```
<a><b /></a>
```

and the following tree c:

```
<c><b /></c>
```

In lxml, this behavior is different, because lxml is built on top of a tree that maintains parent relationships for elements (like W3C DOM). This means an element can only exist in a single tree at the same time. Adding an element in some tree to another tree will cause this element to be moved.

So, for tree a we will get:

```
<a></a>
```

and for tree c we will get:

```
<c><b/></c>
```

Unfortunately this is a rather fundamental difference in behavior, which is hard to change. It won't affect some applications, but if you want to port code you must unfortunately make sure that it doesn't affect yours.

- etree allows navigation to the parent of a node by the `getparent()` method and to the siblings by calling `getnext()` and `getprevious()`. This is not possible in ElementTree as the underlying tree model does not have this information.
- When trying to set a subelement using `__setitem__` that is in fact not an Element but some other object, etree raises a `TypeError`, and ElementTree raises an `AssertionError`. This also applies to some other places of the API. In general, etree tries to avoid `AssertionErrors` in favour of being more specific about the reason for the exception.
- When parsing fails in `iterparse()`, ElementTree up to version 1.2.x raises a low-level `ExpatError` instead of a `SyntaxError` as the other parsers. Both lxml and ElementTree 1.3 raise a `ParseError` for parser errors.
- The `iterparse()` function in lxml is implemented based on the libxml2 parser and tree generator. This means that modifications of the document root or the ancestors of the current element during parsing can irritate the parser and even segfault. While this is not a problem in the Python object structure used by ElementTree, the C tree underlying lxml suffers from it. The golden rule for `iterparse()` on lxml therefore is: do not touch anything that will have to be touched again by the parser later on. See the lxml parser documentation on this.
- ElementTree ignores comments and processing instructions when parsing XML, while etree will read them in and treat them as `Comment` or `ProcessingInstruction` elements respectively. This is especially visible where comments are found inside text content, which is then split by the `Comment` element.

You can disable this behaviour by passing the boolean `remove_comments` and/or `remove_pis` key-

word arguments to the parser you use. For convenience and to support portable code, you can also use the `etree.ETCompatXMLParser` instead of the default `etree.XMLParser`. It tries to provide a default setup that is as close to the ElementTree parser as possible.

- The `TreeBuilder` class of `lxml.etree` uses a different signature for the `start()` method. It accepts an additional argument `nsmap` to propagate the namespace declarations of an element in addition to its own namespace. To assure compatibility with ElementTree (which does not support this argument), `lxml` checks if the method accepts 3 arguments before calling it, and otherwise drops the namespace mapping. This should work with most existing ElementTree code, although there may still be conflicting cases.
- ElementTree 1.2 has a bug when serializing an empty Comment (no text argument given) to XML, `etree` serializes this successfully.
- ElementTree adds whitespace around comments on serialization, `lxml` does not. This means that a comment text “text” that ElementTree serializes as “<!-- text -->” will become “<!--text-->” in `lxml`.
- When the string `*` is used as tag filter in the `Element.getiterator()` method, ElementTree returns all elements in the tree, including comments and processing instructions. `lxml.etree` only returns real Elements, i.e. tree nodes that have a string tag name. Without a filter, both libraries iterate over all nodes.

Note that currently only `lxml.etree` supports passing the `Element` factory function as filter to select only Elements. Both libraries support passing the `Comment` and `ProcessingInstruction` factories to select the respective tree nodes.

- ElementTree merges the target of a processing instruction into `PI.text`, while `lxml.etree` puts it into the `.target` property and leaves it out of the `.text` property. The `pi.text` in ElementTree therefore corresponds to `pi.target + " " + pi.text` in `lxml.etree`.
- Because `etree` is built on top of `libxml2`, which is namespace prefix aware, `etree` preserves namespace declarations and prefixes while ElementTree tends to come up with its own prefixes (`ns0`, `ns1`, etc). When no namespace prefix is given, however, `etree` creates ElementTree style prefixes as well.
- `etree` has a `'prefix'` attribute (read-only) on elements giving the Element’s prefix, if this is known, and `None` otherwise (in case of no namespace at all, or default namespace).
- `etree` further allows passing an `'nsmap'` dictionary to the `Element` and `SubElement` element factories to explicitly map namespace prefixes to namespace URIs. These will be translated into namespace declarations on that element. This means that in the probably rare case that you need to construct an attribute called `'nsmap'`, you need to be aware that unlike in ElementTree, you cannot pass it as a keyword argument to the `Element` and `SubElement` factories directly.
- ElementTree allows `QName` objects as attribute values and resolves their prefix on serialisation (e.g. an attribute value `QName("{myns}myname")` becomes “p:myname” if “p” is the namespace prefix of “myns”). `lxml.etree` also allows you to set attribute values from `QName` instances (and also `.text` values), but it resolves their prefix immediately and stores the plain text value. So, if prefixes are modified later on, e.g. by moving a subtree to a different tree (which reassigns the prefix mappings), the text values will not be updated and you might end up with an undefined prefix.
- `etree` elements can be copied using `copy.deepcopy()` and `copy.copy()`, just like ElementTree’s. However, `copy.copy()` does *not* create a shallow copy where elements are shared between trees, as this makes no sense in the context of `libxml2` trees. Note that `lxml` can deep-copy trees considerably faster than ElementTree, so a deep copy might still be fast enough to replace a shallow copy in your case.

## Chapter 7

# lxml FAQ - Frequently Asked Questions

Frequently asked questions on lxml. See also the notes on [compatibility](#) to ElementTree.

### 7.1 General Questions

#### 7.1.1 Is there a tutorial?

Read the [lxml.etree Tutorial](#). While this is still work in progress (just as any good documentation), it provides an overview of the most important concepts in `lxml.etree`. If you want to help out, improving the tutorial is a very good place to start.

There is also a [tutorial for ElementTree](#) which works for `lxml.etree`. The [API documentation](#) also contains many examples for `lxml.etree`. To learn using `lxml.objectify`, read the [objectify documentation](#).

John Shipman has written another tutorial called [Python XML processing with lxml](#) that contains lots of examples.

#### 7.1.2 Where can I find more documentation about lxml?

There is a lot of documentation as lxml implements the well-known [ElementTree API](#) and tries to follow its documentation as closely as possible. There are a couple of issues where lxml cannot keep up compatibility. They are described in the [compatibility](#) documentation. The lxml specific extensions to the API are described by individual files in the `doc` directory of the distribution and on [the web page](#).

#### 7.1.3 What standards does lxml implement?

The compliance to XML Standards depends on the support in libxml2 and libxslt. Here is a quote from <http://xmlsoft.org/>:

In most cases libxml2 tries to implement the specifications in a relatively strictly compliant way. As of release 2.4.16, libxml2 passed all 1800+ tests from the OASIS XML Tests Suite.

lxml currently supports libxml2 2.6.20 or later, which has even better support for various XML standards. Some of the more important ones are: HTML, XML namespaces, XPath, XInclude, XSLT, XML catalogs, canonical XML, RelaxNG, XML:ID. Support for XML Schema and especially Schematron is currently incomplete in libxml2, but is definitely usable and actively being worked on. libxml2 also supports loading documents through HTTP and FTP.

#### 7.1.4 Who uses lxml?

As an XML library, lxml is often used under the hood of in-house server applications, such as web servers or applications that facilitate some kind of document management. Therefore, it is hard to get an idea of who uses it, and the following list of 'users and projects we know of' is definitely not a complete list of lxml's users.

Also note that the compatibility to the ElementTree library does not require projects to set a hard dependency on lxml - as long as they do not take advantage of lxml's enhanced feature set.

- [cssutils](#), a CSS parser and toolkit, can be used with `lxml.cssselect`
- [Deliverance](#), a content theming tool
- [Enfold Proxy 4](#), a web server accelerator with on-the-fly XSLT processing
- [Inteproxy](#), a secure HTTP proxy
- [lwebstring](#), an XML template engine
- [OpenXMLlib](#), a library for handling OpenXML document meta data
- [Pycoon](#), a WSGI web development framework based on XML pipelines
- [rfadict](#), an RDFa parser with a simple dictionary-like interface.

Zope3 and some of its extensions have good support for lxml:

- [gocept.lxml](#), Zope3 interface bindings for lxml
- [z3c.rml](#), an implementation of ReportLab's RML format

And don't miss the quotes by our generally [happy users](#), and other [sites that link to lxml](#).

#### 7.1.5 What is the difference between lxml.etree and lxml.objectify?

The two modules provide different ways of handling XML. However, objectify builds on top of lxml.etree and therefore inherits most of its capabilities and a large portion of its API.

- lxml.etree is a generic API for XML and HTML handling. It aims for ElementTree [compatibility](#) and supports the entire XML infoset. It is well suited for both mixed content and data centric XML. Its generality makes it the best choice for most applications.
- lxml.objectify is a specialized API for XML data handling in a Python object syntax. It provides a very natural way to deal with data fields stored in a structurally well defined XML format. Data is automatically converted to Python data types and can be manipulated with normal Python operators. Look at the examples in the [objectify documentation](#) to see what it feels like to use it.

Objectify is not well suited for mixed contents or HTML documents. As it is built on top of lxml.etree, however, it inherits the normal support for XPath, XSLT or validation.

### 7.1.6 How can I make my application run faster?

lxml.etree is a very fast library for processing XML. There are, however, [a few caveats](#) involved in the mapping of the powerful libxml2 library to the simple and convenient ElementTree API. Not all operations are as fast as the simplicity of the API might suggest, while some use cases can heavily benefit from finding the right way of doing them. The [benchmark page](#) has a comparison to other ElementTree implementations and a number of tips for performance tweaking. As with any Python application, the rule of thumb is: the more of your processing runs in C, the faster your application gets. See also the section on [threading](#).

### 7.1.7 What about that trailing text on serialised Elements?

The ElementTree tree model defines an Element as a container with a tag name, contained text, child Elements and a tail text. This means that whenever you serialise an Element, you will get all parts of that Element:

```
>>> from lxml import etree
>>> root = etree.XML("<root><tag>text<child/></tag>tail</root>")
>>> print etree.tostring(root[0])
<tag>text<child/></tag>tail
```

Here is an example that shows why not serialising the tail would be even more surprising from an object point of view:

```
>>> root = etree.Element("test")

>>> root.text = "TEXT"
>>> etree.tostring(root)
<test>TEXT</test>

>>> etree.tail = "TAIL"
>>> etree.tostring(root)
<test>TEXT</test>TAIL

>>> etree.tail = None
>>> etree.tostring(root)
<test>TEXT</test>
```

Just imagine a Python list where you append an item and it doesn't show up when you look at the list.

The `.tail` property is a huge simplification for the tree model as it avoids text nodes to appear in the list of children and makes access to them quick and simple. So this is a benefit in most applications and simplifies many, many XML tree algorithms.

However, in document-like XML (and especially HTML), the above result can be unexpected to new users and can sometimes require a bit more overhead. A good way to deal with this is to use helper functions that copy the Element without its tail. The `lxml.html` package also deals with this in a couple of places, as most HTML algorithms benefit from a tail-free behaviour.

## 7.2 Installation

### 7.2.1 Which version of libxml2 and libxslt should I use or require?

It really depends on your application, but the rule of thumb is: more recent versions contain less bugs and provide more features.

- Do not use libxml2 2.6.27 if you want to use XPath (including XSLT). You will get crashes when XPath errors occur during the evaluation (e.g. for unknown functions). This happens inside the evaluation call to libxml2, so there is nothing that lxml can do about it.
- Try to use versions of both libraries that were released together. At least the libxml2 version should not be older than the libxslt version.
- If you use XML Schema or Schematron which are still under development, the most recent version of libxml2 is usually a good bet.
- The same applies to XPath, where a substantial number of bugs and memory leaks were fixed over time. If you encounter crashes or memory leaks in XPath applications, try a more recent version of libxml2.
- For parsing and fixing broken HTML, lxml requires at least libxml2 2.6.21.
- For the normal tree handling, however, any libxml2 version starting with 2.6.20 should do.

Read the [release notes of libxml2](#) and the [release notes of libxslt](#) to see when (or if) a specific bug has been fixed.

### 7.2.2 Where are the Windows binaries?

Short answer: If you want to contribute a binary build, we are happy to put it up on the Cheeseshop.

Long answer: Two of the bigger problems with the Windows system are the lack of a pre-installed standard compiler and the missing package management. Both make it non-trivial to build lxml on this platform. We are trying hard to make lxml as platform-independent as possible and it is regularly tested on Windows systems. However, we currently cannot provide Windows binary distributions ourselves.

From time to time, users of different environments kindly contribute binary builds of lxml, most frequently for Windows or Mac-OS X. We put these on the Cheeseshop to make it as easy as possible for others to use lxml on their platform.

If there is not currently a binary distribution of the most recent lxml release for your platform available from the Cheeseshop, please look through the older versions to see if they provide a binary build. This is done by appending the version number to the cheeseshop URL, e.g.:

<http://cheeseshop.python.org/pypi/lxml/1.1.2>

### 7.2.3 Why do I get errors about missing UCS4 symbols when installing lxml?

Most likely, you use a Python installation that was configured for internal use of UCS2 unicode, meaning 16-bit unicode. The lxml egg distributions are generally compiled on platforms that use UCS4, a 32-bit unicode encoding, as this is used on the majority of platforms. Sadly, both are not compatible, so the eggs can only support the one they were compiled with.

This means that you have to compile lxml from sources for your system. Note that you do not need

---

Cython for this, the lxml source distribution is directly compilable on both platform types. See the [build instructions](#) on how to do this.

## 7.3 Contributing

### 7.3.1 Why is lxml not written in Python?

It *almost* is.

lxml is not written in plain Python, because it interfaces with two C libraries: libxml2 and libxslt. Accessing them at the C-level is required for performance reasons.

However, to avoid writing plain C-code and caring too much about the details of built-in types and reference counting, lxml is written in [Cython](#), a Python-like language that is translated into C-code. Chances are that if you know Python, you can write [code that Cython accepts](#). Again, the C-ish style used in the lxml code is just for performance optimisations. If you want to contribute, don't bother with the details, a Python implementation of your contribution is better than none. And keep in mind that lxml's flexible API often favours an implementation of features in pure Python, without bothering with C-code at all. For example, the `lxml.html` package is entirely written in Python.

Please contact the [mailing list](#) if you need any help.

### 7.3.2 How can I contribute?

Besides enhancing the code, there are a lot of places where you can help the project and its user base. You can

- spread the word and write about lxml. Many users (especially new Python users) have not yet heard about lxml, although our user base is constantly growing. If you write your own blog and feel like saying something about lxml, go ahead and do so. If we think your contribution or criticism is valuable to other users, we may even put a link or a quote on the project page.
- provide code examples for the general usage of lxml or specific problems solved with lxml. Readable code is a very good way of showing how a library can be used and what great things you can do with it. Again, if we hear about it, we can set a link on the project page.
- work on the documentation. The web page is generated from a set of [ReST text files](#). It is meant both as a representative project page for lxml and as a site for documenting lxml's API and usage. If you have questions or an idea how to make it more readable and accessible while you are reading it, please send a comment to the [mailing list](#).
- help with the tutorial. A tutorial is the most important starting point for new users, so it is important for us to provide an easy to understand guide into lxml. As allo documentation, the tutorial is work in progress, so we appreciate every helping hand.
- improve the docstrings. lxml uses docstrings to support Python's integrated online `help()` function. However, sometimes these are not sufficient to grasp the details of the function in question. If you find such a place, you can try to write up a better description and send it to the [mailing list](#).

## 7.4 Bugs

### 7.4.1 My application crashes!

One of the goals of lxml is “no segfaults”, so if there is no clear warning in the documentation that you were doing something potentially harmful, you have found a bug and we would like to hear about it. Please report this bug to the [mailing list](#). See the next section on how to do that.

However, there are a few things to try first, to make sure the problem is really within lxml (or libxml2 or libxslt):

- a) If your application (or e.g. your web container) uses threads, please see the FAQ section on [threading](#) to check if you touch on one of the potential pitfalls.
- b) If you are on Mac-OS X, make sure lxml uses the correct libraries. If you have updated the old system libraries (e.g. through fink), this is best achieved by building lxml statically to prevent the different library versions from interfering. If you choose to use a dynamically linked version, make sure the `DYLD_LIBRARY_PATH` environment variable contains the directory where you installed the libraries. To make sure the correct libraries are used, print the module level version numbers that `lxml.etree` provides from *within* your application rather than relying on what your operating system tells you.

In any case, try to reproduce the problem with the latest versions of libxml2 and libxslt. From time to time, bugs and race conditions are found in these libraries, so a more recent version might already contain a fix for your problem.

### 7.4.2 I think I have found a bug in lxml. What should I do?

First, you should look at the [current developer changelog](#) to see if this is a known problem that has already been fixed in the SVN trunk since the release you are using.

Also, the ‘crash’ section above has a few good advices what to try to see if the problem is really in lxml - and not in your setup. Believe it or not, that happens more often than you might think, especially when old libraries or even multiple library versions are installed.

You should always try to reproduce the problem with the latest versions of libxml2 and libxslt - and make sure they are used (`lxml.etree` can tell you what it runs with, see below).

Otherwise, we would really like to hear about it. Please report it to the [mailing list](#) so that we can fix it. It is very helpful in this case if you can come up with a short code snippet that demonstrates your problem. If others can reproduce and see the problem, it is much easier for them to fix it - and maybe even easier for you to describe it and get people convinced that it really is a problem to fix. Please also report the version of lxml, libxml2 and libxslt that you are using by calling this:

```
from lxml import etree
print "lxml.etree:      ", etree.LXML_VERSION
print "libxml used:    ", etree.LIBXML_VERSION
print "libxml compiled: ", etree.LIBXML_COMPILED_VERSION
print "libxslt used:   ", etree.LIBXSLT_VERSION
print "libxslt compiled: ", etree.LIBXSLT_COMPILED_VERSION
```

Since as a user of lxml you are likely a programmer, you might find [this article on bug reports](#) an interesting read.

## 7.5 Threading

### 7.5.1 Can I use threads to concurrently access the lxml API?

Yes, although not carelessly.

lxml frees the GIL (Python's global interpreter lock) internally when parsing from disk and memory, as long as you use either the default parser (which is replicated for each thread) or create a parser for each thread yourself. lxml also allows concurrency during validation (RelaxNG and XMLSchema) and XSL transformation. You can share RelaxNG, XMLSchema and (with restrictions) XSLT objects between threads. While you can also share parsers between threads, this will serialize the access to each of them, so it is better to `copy()` parsers or to just use the default parser (which is automatically copied for each thread).

Due to the way libxslt handles threading, concurrent access to stylesheets is currently only possible if it was parsed in the main thread. Parsing and applying a stylesheet inside one thread also works.

Warning: You should generally avoid modifying trees in other threads than the one it was generated in. Although this should work in many cases, there are certain scenarios where the termination of a thread that parsed a tree can crash the application if subtrees of this tree were moved to other documents. You should be on the safe side when passing trees between threads if you either

- a) do not modify these trees and do not move their elements to other trees, or
- b) do not terminate threads while the trees they parsed are still in use (e.g. by using a fixed size thread-pool or long-running threads in processing chains)

### 7.5.2 Does my program run faster if I use threads?

Depends. The best way to answer this is timing and profiling.

The global interpreter lock (GIL) in Python serializes access to the interpreter, so if the majority of your processing is done in Python code (walking trees, modifying elements, etc.), your gain will be close to 0. The more of your XML processing moves into lxml, however, the higher your gain. If your application is bound by XML parsing and serialisation, or by complex XSLTs, your speedup on multi-processor machines can be substantial.

See the question above to learn which operations free the GIL to support multi-threading.

### 7.5.3 Would my single-threaded program run faster if I turned off threading?

Can be. You can see for yourself by compiling lxml entirely without threading support. Pass the `--without-threading` option to `setup.py` when building lxml from source.

### 7.5.4 Why can't I reuse XSLT stylesheets in other threads?

lxml currently has the restriction that an XSLT object can only be used in a thread if it was created either in the thread itself or in the main thread. This is due to some interfering optimisations in libxslt and lxml.etree. To work around this, you can do a couple of things:

- create all XSLT objects in the main program and reuse them wherever you want.

- create them in the thread where you use them and maybe cache them in thread local storage (see the threading module).

If your stylesheets are diverse and status specific, you can still prepare them in advance if you:

- use XSLT parameters that you pass at call time to configure the stylesheets
- create the stylesheets (partially) programmatically in the main program, e.g. by adding `xsl:output` tags, `xsl:include` tags or templates (be careful with the order here) to the XSL tree, and then create the XSLT objects and store them in a read-only dictionary. That way, you can access and use them in any thread.

Note that passing the same XSL tree into multiple `XSLT()` instances will create independent stylesheets, so later modifications of the tree will not be reflected in the already created stylesheets. Also, since `lxml` 2.0, you can deep copy XSLT objects using the `copy` module from the standard library.

### 7.5.5 My program crashes when run with mod\_python/Pyro/Zope/Plone/...

These environments can use threads in a way that may not make it obvious when threads are created and what happens in which thread. This makes it hard to ensure `lxml`'s threading support is used in a reliable way. Sadly, if problems arise, they are as diverse as the applications, so it is difficult to provide any generally applicable solution. Also, these environments are so complex that problems become hard to debug and even harder to reproduce in a predictable way. If you encounter crashes in one of these systems, but your code runs perfectly when started by hand, the following gives you a few hints for possible approaches to solve your specific problem:

- make sure you use recent versions of `libxml2`, `libxslt` and `lxml`. The `libxml2` developers keep fixing bugs in each release, and `lxml` also tries to become more robust against possible pitfalls. So newer versions might already fix your problem in a reliable way.
- make sure the library versions you installed are really used. Do not rely on what your operating system tells you! Print the version constants in `lxml.etree` from within your runtime environment to make sure it is the case. This is especially a problem under MacOS-X when newer library versions were installed in addition to the outdated system libraries.
- if you use `mod_python`, try setting this option:

```
PythonInterpreter main_interpreter
```

There was a discussion on the mailing list about this problem:

<http://comments.gmane.org/gmane.comp.python.lxml.devel/2942>

- compile `lxml` without threading support by running `setup.py` with the `--without-threading` option. While this might be slower in certain scenarios on multi-processor systems, it *might* also keep your application from crashing, which should be worth more to you than peek performance. Remember that `lxml` is fast anyway, so concurrency may not even be worth it.
- avoid doing fancy XSLT stuff like foreign document access or passing in subtrees through XSLT variables. This might or might not work, depending on your specific usage.
- try copying trees at suspicious places and working with those instead of a tree shared between threads. A good candidate might be the result of an XSLT or the stylesheet itself.
- try keeping thread-local copies of XSLT stylesheets, i.e. one per thread, instead of sharing one. Also see the question above.
- you can try to serialise suspicious parts of your code with explicit thread locks, thus disabling the

concurrency of the runtime system.

- report back on the mailing list to see if there are other ways to work around your specific problems. Do not forget to report the version numbers of `lxml`, `libxml2` and `libxslt` you are using (see the question on reporting a bug).

## 7.6 Parsing and Serialisation

### 7.6.1 Why doesn't the `pretty_print` option reformat my XML output?

Pretty printing (or formatting) an XML document means adding white space to the content. These modifications are harmless if they only impact elements in the document that do not carry (text) data. They corrupt your data if they impact elements that contain data. If `lxml` cannot distinguish between whitespace and data, it will not alter your data. Whitespace is therefore only added between nodes that do not contain data. This is always the case for trees constructed element-by-element, so no problems should be expected here. For parsed trees, a good way to assure that no conflicting whitespace is left in the tree is the `remove_blank_text` option:

```
>>> parser = etree.XMLParser(remove_blank_text=True)
>>> tree = etree.parse(file, parser)
```

This will allow the parser to drop blank text nodes when constructing the tree. If you now call a serialization function to pretty print this tree, `lxml` can add fresh whitespace to the XML tree to indent it.

### 7.6.2 Why can't `lxml` parse my XML from unicode strings?

`lxml` can read Python unicode strings and even tries to support them if `libxml2` does not. However, if the unicode string declares an XML encoding internally (`<?xml encoding="..."?>`), parsing is bound to fail, as this encoding is most likely not the real encoding used in Python unicode. The same is true for HTML unicode strings that contain charset meta tags, although the problems may be more subtle here. The `libxml2` HTML parser may not be able to parse the meta tags in broken HTML and may end up ignoring them, so even if parsing succeeds, later handling may still fail with character encoding errors.

Note that Python uses different encodings for unicode on different platforms, so even specifying the real internal unicode encoding is not portable between Python interpreters. Don't do it.

Python unicode strings with XML data or HTML data that carry encoding information are broken. `lxml` will not parse them. You must provide parsable data in a valid encoding.

### 7.6.3 What is the difference between `str(xslt(doc))` and `xslt(doc).write()` ?

The `str()` implementation of the `XSLTResultTree` class (a subclass of the `ElementTree` class) knows about the output method chosen in the stylesheet (`xsl:output`), `write()` doesn't. If you call `write()`, the result will be a normal XML tree serialization in the requested encoding. Calling this method may also fail for XSLT results that are not XML trees (e.g. string results).

If you call `str()`, it will return the serialized result as specified by the XSL transform. This correctly serializes string results to encoded Python strings and honours `xsl:output` options like `indent`. This almost certainly does what you want, so you should only use `write()` if you are sure that the XSLT result is an XML tree and you want to override the encoding and indentation options requested by the

stylesheet.

#### 7.6.4 Why can't I just delete parents or clear the root node in iterparse()?

The `iterparse()` implementation is based on the libxml2 parser. It requires the tree to be intact to finish parsing. If you delete or modify parents of the current node, chances are you modify the structure in a way that breaks the parser. Normally, this will result in a segfault. Please refer to the [iterparse section](#) of the lxml API documentation to find out what you can do and what you can't do.

#### 7.6.5 How do I output null characters in XML text?

Don't. What you would produce is not well-formed XML. XML parsers will refuse to parse a document that contains null characters. The right way to embed binary data in XML is using a text encoding such as uuencode or base64.

## 7.7 XPath and Document Traversal

### 7.7.1 What are the `findall()` and `xpath()` methods on `Element(Tree)`?

`findall()` is part of the original [ElementTree API](#). It supports a [simple subset of the XPath language](#), without predicates, conditions and other advanced features. It is very handy for finding specific tags in a tree. Another important difference is namespace handling, which uses the `{namespace}tagname` notation. This is not supported by XPath. The `findall`, `find` and `findtext` methods are compatible with other `ElementTree` implementations and allow writing portable code that runs on `ElementTree`, `cElementTree` and `lxml.etree`.

`xpath()`, on the other hand, supports the complete power of the XPath language, including predicates, XPath functions and Python extension functions. The syntax is defined by the [XPath specification](#). If you need the expressiveness and selectivity of XPath, the `xpath()` method, the `XPath` class and the `XPathEvaluator` are the best [choice](#).

### 7.7.2 Why doesn't `findall()` support full XPath expressions?

It was decided that it is more important to keep compatibility with `ElementTree` to simplify code migration between the libraries. The main difference compared to XPath is the `{namespace}tagname` notation used in `findall()`, which is not valid XPath.

`ElementTree` and `lxml.etree` use the same implementation, which assures 100% compatibility. Note that `findall()` is [so fast](#) in lxml that a native implementation would not bring any performance benefits.

### 7.7.3 How can I find out which namespace prefixes are used in a document?

You can traverse the document (`getiterator()`) and collect the prefix attributes from all Elements into a set. However, it is unlikely that you really want to do that. You do not need these prefixes, honestly. You only need the namespace URIs. All namespace comparisons use these, so feel free to make up your own prefixes when you use XPath expressions or extension functions.

The only place where you might consider specifying prefixes is the serialization of Elements that were created through the API. Here, you can specify a prefix mapping through the `nsmap` argument when creating the root Element. Its children will then inherit this prefix for serialization.

#### **7.7.4 How can I specify a default namespace for XPath expressions?**

You can't. In XPath, there is no such thing as a default namespace. Just use an arbitrary prefix and let the namespace dictionary of the XPath evaluators map it to your namespace. See also the question above.

## Part II

# Developing with lxml

## Chapter 8

# The lxml.etree Tutorial

**Author:** Stefan Behnel

This tutorial briefly overviews the main concepts of the [ElementTree API](#) as implemented by `lxml.etree`, and some simple enhancements that make your life as a programmer easier.

For a complete reference of the API, see the [generated API documentation](#).

A common way to import `lxml.etree` is as follows:

```
>>> from lxml import etree
```

If your code only uses the ElementTree API and does not rely on any functionality that is specific to `lxml.etree`, you can also use (any part of) the following import chain as a fall-back to the original ElementTree:

```
try:
    from lxml import etree
    print "running with lxml.etree"
except ImportError:
    try:
        # Python 2.5
        import xml.etree.cElementTree as etree
        print "running with cElementTree on Python 2.5+"
    except ImportError:
        try:
            # Python 2.5
            import xml.etree.ElementTree as etree
            print "running with ElementTree on Python 2.5+"
        except ImportError:
            try:
                # normal cElementTree install
                import cElementTree as etree
                print "running with cElementTree"
            except ImportError:
                try:
                    # normal ElementTree install
                    import elementtree.ElementTree as etree
                    print "running with ElementTree"
                except ImportError:
```

```
print "Failed to import ElementTree from any known place"
```

To aid in writing portable code, this tutorial makes it clear in the examples which part of the presented API is an extension of `lxml.etree` over the original `ElementTree` API, as defined by Fredrik Lundh's `ElementTree` library.

## 8.1 The Element class

An `Element` is the main container object for the `ElementTree` API. Most of the XML tree functionality is accessed through this class. Elements are easily created through the `Element` factory:

```
>>> root = etree.Element("root")
```

The XML tag name of elements is accessed through the `tag` property:

```
>>> print root.tag
root
```

Elements are organised in an XML tree structure. To create child elements and add them to a parent element, you can use the `append()` method:

```
>>> root.append( etree.Element("child1") )
```

However, this is so common that there is a shorter and much more efficient way to do this: the `SubElement` factory. It accepts the same arguments as the `Element` factory, but additionally requires the parent as first argument:

```
>>> child2 = etree.SubElement(root, "child2")
>>> child3 = etree.SubElement(root, "child3")
```

To see that this is really XML, you can serialise the tree you have created:

```
>>> print etree.tostring(root, pretty_print=True),
<root>
  <child1/>
  <child2/>
  <child3/>
</root>
```

### 8.1.1 Elements are lists

To make the access to these subelements as easy and straight forward as possible, elements behave like normal Python lists:

```
>>> child = root[0]
>>> print child.tag
child1

>>> print len(root)
3

>>> root.index(root[1]) # lxml.etree only!
1
```

```

>>> children = list(root)

>>> for child in root:
...     print child.tag
child1
child2
child3

>>> root.insert(0, etree.Element("child0"))
>>> start = root[:1]
>>> end   = root[-1:]

>>> print start[0].tag
child0
>>> print end[0].tag
child3

>>> root[0] = root[-1] # this moves the element!
>>> for child in root:
...     print child.tag
child3
child1
child2

```

Prior to ElementTree 1.3 and lxml 2.0, you could also check the truth value of an Element to see if it has children, i.e. if the list of children is empty. This is no longer supported as people tend to find it surprising that a non-None reference to an existing Element can evaluate to False. Instead, use `len(element)`, which is both more explicit and less error prone.

Note in the examples that the last element was *moved* to a different position in the last example. This is a difference from the original ElementTree (and from lists), where elements can sit in multiple positions of any number of trees. In lxml.etree, elements can only sit in one position of one tree at a time.

If you want to *copy* an element to a different position, consider creating an independent *deep copy* using the copy module from Python's standard library:

```

>>> from copy import deepcopy

>>> element = etree.Element("neu")
>>> element.append( deepcopy(root[1]) )

>>> print element[0].tag
child1
>>> print [ c.tag for c in root ]
['child3', 'child1', 'child2']

```

The way up in the tree is provided through the `getparent()` method:

```

>>> root is root[0].getparent() # lxml.etree only!
True

```

The siblings (or neighbours) of an element are accessed as next and previous elements:

```

>>> root[0] is root[1].getprevious() # lxml.etree only!
True
>>> root[1] is root[0].getnext() # lxml.etree only!
True

```

## 8.1.2 Elements carry attributes

XML elements support attributes. You can create them directly in the Element factory:

```
>>> root = etree.Element("root", interesting="totally")
>>> print etree.tostring(root)
<root interesting="totally"/>
```

Fast and direct access to these attributes is provided by the `set()` and `get()` methods of elements:

```
>>> print root.get("interesting")
totally

>>> root.set("interesting", "somewhat")
>>> print root.get("interesting")
somewhat
```

However, a very convenient way of dealing with them is through the dictionary interface of the `attrib` property:

```
>>> attributes = root.attrib

>>> print attributes["interesting"]
somewhat

>>> print attributes.get("hello")
None

>>> attributes["hello"] = "Guten Tag"
>>> print attributes.get("hello")
Guten Tag
>>> print root.get("hello")
Guten Tag
```

## 8.1.3 Elements contain text

Elements can contain text:

```
>>> root = etree.Element("root")
>>> root.text = "TEXT"

>>> print root.text
TEXT

>>> print etree.tostring(root)
<root>TEXT</root>
```

In many XML documents (*data-centric* documents), this is the only place where text can be found. It is encapsulated by a leaf tag at the very bottom of the tree hierarchy.

However, if XML is used for tagged text documents such as (X)HTML, text can also appear between different elements, right in the middle of the tree:

```
<html><body>Hello<br/>World</body></html>
```

Here, the `<br/>` tag is surrounded by text. This is often referred to as *document-style* or *mixed-content* XML. Elements support this through their `tail` property. It contains the text that directly follows the element, up to the next element in the XML tree:

```
>>> html = etree.Element("html")
>>> body = etree.SubElement(html, "body")
>>> body.text = "TEXT"

>>> print etree.tostring(html)
<html><body>TEXT</body></html>

>>> br = etree.SubElement(body, "br")
>>> print etree.tostring(html)
<html><body>TEXT<br/></body></html>

>>> br.tail = "TAIL"
>>> print etree.tostring(html)
<html><body>TEXT<br/>TAIL</body></html>
```

The two properties `.text` and `.tail` are enough to represent any text content in an XML document. This way, the ElementTree API does not require any [special text nodes](#) in addition to the Element class, that tend to get in the way fairly often (as you might know from classic [DOM APIs](#)).

However, there are cases where the tail text also gets in the way. For example, when you serialise an Element from within the tree, you do not always want its tail text in the result (although you would still want the tail text of its children). For this purpose, the `tostring()` function accepts the keyword argument `with_tail`:

```
>>> print etree.tostring(br)
<br/>TAIL
>>> print etree.tostring(br, with_tail=False) # lxml.etree only!
<br/>
```

If you want to read *only* the text, i.e. without any intermediate tags, you have to recursively concatenate all `text` and `tail` attributes in the correct order. Again, the `tostring()` function comes to the rescue, this time using the `method` keyword:

```
>>> print etree.tostring(html, method="text")
TEXTTAIL
```

#### 8.1.4 Using XPath to find text

Another way to extract the text content of a tree is [XPath](#), which also allows you to extract the separate text chunks into a list:

```
>>> print html.xpath("string()") # lxml.etree only!
TEXTTAIL
>>> print html.xpath("//text()") # lxml.etree only!
['TEXT', 'TAIL']
```

If you want to use this more often, you can wrap it in a function:

```
>>> build_text_list = etree.XPath("//text()") # lxml.etree only!
>>> print build_text_list(html)
['TEXT', 'TAIL']
```

Note that a string result returned by XPath is a special 'smart' object that knows about its origins. You can ask it where it came from through its `getparent()` method, just as you would with Elements:

```
>>> texts = build_text_list(html)
>>> print texts[0]
TEXT
>>> parent = texts[0].getparent()
>>> print parent.tag
body

>>> print texts[1]
TAIL
>>> print texts[1].getparent().tag
br
```

You can also find out if it's normal text content or tail text:

```
>>> print texts[0].is_text
True
>>> print texts[1].is_text
False
>>> print texts[1].is_tail
True
```

While this works for the results of the `text()` function, lxml will not tell you the origin of a string value that was constructed by the XPath functions `string()` or `concat()`:

```
>>> stringify = etree.XPath("string()")
>>> print stringify(html)
TEXTTAIL
>>> print stringify(html).getparent()
None
```

### 8.1.5 Tree iteration

For problems like the above, where you want to recursively traverse the tree and do something with its elements, tree iteration is a very convenient solution. Elements provide a tree iterator for this purpose. It yields elements in *document order*, i.e. in the order their tags would appear if you serialised the tree to XML:

```
>>> root = etree.Element("root")
>>> etree.SubElement(root, "child").text = "Child 1"
>>> etree.SubElement(root, "child").text = "Child 2"
>>> etree.SubElement(root, "another").text = "Child 3"

>>> print etree.tostring(root, pretty_print=True),
<root>
  <child>Child 1</child>
  <child>Child 2</child>
  <another>Child 3</another>
</root>

>>> for element in root.iter():
...     print element.tag, '-', element.text
root - None
```

```

child - Child 1
child - Child 2
another - Child 3

```

If you know you are only interested in a single tag, you can pass its name to `iter()` to have it filter for you:

```

>>> for element in root.iter("child"):
...     print element.tag, '-', element.text
child - Child 1
child - Child 2

```

By default, iteration yields all nodes in the tree, including `ProcessingInstructions`, `Comments` and `Entity` instances. If you want to make sure only `Element` objects are returned, you can pass the `Element` factory as tag parameter:

```

>>> root.append(etree.Entity("#234"))
>>> root.append(etree.Comment("some comment"))

>>> for element in root.iter():
...     if isinstance(element.tag, basestring):
...         print element.tag, '-', element.text
...     else:
...         print 'SPECIAL:', element, '-', element.text
root - None
child - Child 1
child - Child 2
another - Child 3
SPECIAL: &#234; - &#234;
SPECIAL: <!--some comment--> - some comment

>>> for element in root.iter(tag=etree.Element):
...     print element.tag, '-', element.text
root - None
child - Child 1
child - Child 2
another - Child 3

>>> for element in root.iter(tag=etree.Entity):
...     print element.text
&#234;

```

In `lxml.etree`, elements provide [further iterators](#) for all directions in the tree: children, parents (or rather ancestors) and siblings.

### 8.1.6 Serialisation

Serialisation commonly uses the `tostring()` function that returns a string, or the `ElementTree.write()` method that writes to a file or file-like object. Both accept the same keyword arguments like `pretty_print` for formatted output or `encoding` to select a specific output encoding other than plain ASCII:

```

>>> root = etree.XML('<root><a><b/></a></root>')

>>> print etree.tostring(root)
<root><a><b/></a></root>

```

```

>>> print etree.tostring(root, xml_declaration=True)
<?xml version='1.0' encoding='ASCII'?>
<root><a><b/></a></root>

>>> print etree.tostring(root, encoding='iso-8859-1')
<?xml version='1.0' encoding='iso-8859-1'?>
<root><a><b/></a></root>

>>> print etree.tostring(root, pretty_print=True),
<root>
  <a>
    <b/>
  </a>
</root>

```

Note the newline that is appended at the end when pretty printing the output.

Since lxml 2.0 (and ElementTree 1.3), the serialisation functions can do more than XML serialisation. You can serialise to HTML or extract the text content by passing the `method` keyword:

```

>>> root = etree.XML('<html><head/><body><p>Hello<br/>World</p></body></html>')

>>> print etree.tostring(root) # default: method = 'xml'
<html><head/><body><p>Hello<br/>World</p></body></html>

>>> print etree.tostring(root, method='xml') # same as above
<html><head/><body><p>Hello<br/>World</p></body></html>

>>> print etree.tostring(root, method='html')
<html><head></head><body><p>Hello<br>World</p></body></html>

>>> print etree.tostring(root, method='html', pretty_print=True),
<html>
<head></head>
<body><p>Hello<br>World</p></body>
</html>

>>> print etree.tostring(root, method='text')
HelloWorld

```

For the plain text output, serialising to a Python unicode string might become handy. Just pass the `unicode` type as encoding:

```

>>> etree.tostring(root, encoding=unicode, method='text')
u'HelloWorld'

```

## 8.2 The ElementTree class

An `ElementTree` is mainly a document wrapper around a tree with a root node. It provides a couple of methods for parsing, serialisation and general document handling. One of the bigger differences is that it serialises as a complete document, as opposed to a single `Element`. This includes top-level processing instructions and comments, as well as a DOCTYPE and other DTD content in the document:

```

>>> from StringIO import StringIO
>>> tree = etree.parse(StringIO(''\
... <?xml version="1.0"?>
... <!DOCTYPE root SYSTEM "test" [ <!ENTITY tasty "eggs"> ]>
... <root>
...   <a>&tasty;</a>
... </root>
... '''))

>>> print tree.docinfo.doctype
<!DOCTYPE root SYSTEM "test">

>>> # lxml 1.3.4 and later
>>> print etree.tostring(tree)
<!DOCTYPE root SYSTEM "test" [
<!ENTITY tasty "eggs">
]>
<root>
  <a>eggs</a>
</root>

>>> # lxml 1.3.4 and later
>>> print etree.tostring(etree.ElementTree(tree.getroot()))
<!DOCTYPE "root" SYSTEM "test" [
<!ENTITY tasty "eggs">
]>
<root>
  <a>eggs</a>
</root>

>>> # ElementTree and lxml <= 1.3.3
>>> print etree.tostring(tree.getroot())
<root>
  <a>eggs</a>
</root>

```

Note that this has changed in lxml 1.3.4 to match the behaviour of the upcoming lxml 2.0. Before, both would serialise without DTD content, which made lxml lose DTD information in an input-output cycle.

## 8.3 Parsing from strings and files

`lxml.etree` supports parsing XML in a number of ways and from all important sources, namely strings, files, URLs (`http/ftp`) and file-like objects. The main parse functions are `fromstring()` and `parse()`, both called with the source as first argument. By default, they use the standard parser, but you can always pass a different parser as second argument.

### 8.3.1 The `fromstring()` function

The `fromstring()` function is the easiest way to parse a string:

```
>>> some_xml_data = "<root>data</root>"
```

```
>>> root = etree.fromstring(some_xml_data)
>>> print root.tag
root
>>> print etree.tostring(root)
<root>data</root>
```

### 8.3.2 The XML() function

The XML() function behaves like the fromstring() function, but is commonly used to write XML literals right into the source:

```
>>> root = etree.XML("<root>data</root>")
>>> print root.tag
root
>>> print etree.tostring(root)
<root>data</root>
```

### 8.3.3 The parse() function

The parse() function is used to parse from files and file-like objects:

```
>>> some_file_like = StringIO("<root>data</root>")
>>> tree = etree.parse(some_file_like)
>>> print etree.tostring(tree)
<root>data</root>
```

Note that parse() returns an ElementTree object, not an Element object as the string parser functions:

```
>>> root = tree.getroot()
>>> print root.tag
root
>>> print etree.tostring(root)
<root>data</root>
```

The reasoning behind this difference is that parse() returns a complete document from a file, while the string parsing functions are commonly used to parse XML fragments.

The parse() function supports any of the following sources:

- an open file object
- a file-like object that has a .read(byte\_count) method returning a byte string on each call
- a filename string
- an HTTP or FTP URL string

Note that passing a filename or URL is usually faster than passing an open file.

### 8.3.4 Parser objects

By default, `lxml.etree` uses a standard parser with a default setup. If you want to configure the parser, you can create a you instance:

```
>>> parser = etree.XMLParser(remove_blank_text=True) # lxml.etree only!
```

This creates a parser that removes empty text between tags while parsing, which can reduce the size of the tree and avoid dangling tail text if you know that whitespace-only content is not meaningful for your data. An example:

```
>>> root = etree.XML("<root> <a/> <b> </b> </root>", parser)

>>> print etree.tostring(root)
<root><a/><b> </b></root>
```

Note that the whitespace content inside the `<b>` tag was not removed, as content at leaf elements tends to be data content (even if blank). You can easily remove it in an additional step by traversing the tree:

```
>>> for element in root.iter("*"):
...     if element.text is not None and not element.text.strip():
...         element.text = None

>>> print etree.tostring(root)
<root><a/><b/></root>
```

See `help(etree.XMLParser)` to find out about the available parser options.

### 8.3.5 Incremental parsing

`lxml.etree` provides two ways for incremental step-by-step parsing. One is through file-like objects, where it calls the `read()` method repeatedly. This is best used where the data arrives from a source like `urllib` or any other file-like object that can provide data on request. Note that the parser will block and wait until data becomes available in this case:

```
>>> class DataSource:
...     data = iter(["<roo", "t><", "a/", "><", "/root>"])
...     def read(self, requested_size):
...         try:
...             return self.data.next()
...         except StopIteration:
...             return ""

>>> tree = etree.parse(DataSource())

>>> print etree.tostring(tree)
<root><a/></root>
```

The second way is through a feed parser interface, given by the `feed(data)` and `close()` methods:

```
>>> parser = etree.XMLParser()

>>> parser.feed("<roo")
>>> parser.feed("t><")
>>> parser.feed("a/")
>>> parser.feed("><")
```

```
>>> parser.feed("/root>")

>>> root = parser.close()

>>> print etree.tostring(root)
<root><a/></root>
```

Here, you can interrupt the parsing process at any time and continue it later on with another call to the `feed()` method. This comes in handy if you want to avoid blocking calls to the parser, e.g. in frameworks like Twisted, or whenever data comes in slowly or in chunks and you want to do other things while waiting for the next chunk.

After calling the `close()` method (or when an exception was raised by the parser), you can reuse the parser by calling its `feed()` method again:

```
>>> parser.feed("<root/>")
>>> root = parser.close()
>>> print etree.tostring(root)
<root/>
```

### 8.3.6 Event-driven parsing

Sometimes, all you need from a document is a small fraction somewhere deep inside the tree, so parsing the whole tree into memory, traversing it and dropping it can be too much overhead. `lxml.etree` supports this use case with two event-driven parser interfaces, one that generates parser events while building the tree (`iterparse`), and one that does not build the tree at all, and instead calls feedback methods on a target object in a SAX-like fashion.

Here is a simple `iterparse()` example:

```
>>> some_file_like = StringIO("<root><a>data</a></root>")

>>> for event, element in etree.iterparse(some_file_like):
...     print "%s, %4s, %s" % (event, element.tag, element.text)
end,   a, data
end,  root, None
```

By default, `iterparse()` only generates events when it is done parsing an element, but you can control this through the `events` keyword argument:

```
>>> some_file_like = StringIO("<root><a>data</a></root>")

>>> for event, element in etree.iterparse(some_file_like,
...                                       events=("start", "end")):
...     print "%5s, %4s, %s" % (event, element.tag, element.text)
start, root, None
start,   a, data
end,    a, data
end,  root, None
```

Note that the text, tail and children of an `Element` are not necessarily there yet when receiving the `start` event. Only the `end` event guarantees that the `Element` has been parsed completely.

It also allows to `.clear()` or modify the content of an `Element` to save memory. So if you parse a large tree and you want to keep memory usage small, you should clean up parts of the tree that you no longer need:

```

>>> some_file_like = StringIO(
...     "<root><a><b>data</b></a><a><b/></a></root>")

>>> for event, element in etree.iterparse(some_file_like):
...     if element.tag == 'b':
...         print element.text
...     elif element.tag == 'a':
...         print "** cleaning up the subtree"
...         element.clear()
data
** cleaning up the subtree
None
** cleaning up the subtree

```

If memory is a real bottleneck, or if building the tree is not desired at all, the target parser interface of `lxml.etree` can be used. It creates SAX-like events by calling the methods of a target object. By implementing some or all of these methods, you can control which events are generated:

```

>>> class ParserTarget:
...     events = []
...     def start(self, tag, attrib):
...         self.events.append(("start", tag, attrib))
...     def close(self):
...         return self.events

>>> parser = etree.XMLParser(target=ParserTarget())
>>> events = etree.fromstring('<root test="true"/>', parser)

>>> for event in events:
...     print 'event: %s - tag: %s' % (event[0], event[1])
...     for attr, value in event[2].iteritems():
...         print ' * %s = %s' % (attr, value)
event: start - tag: root
 * test = true

```

## 8.4 Namespaces

The ElementTree API avoids [namespace prefixes](#) wherever possible and deploys the real namespaces instead:

```

>>> xhtml = etree.Element("{http://www.w3.org/1999/xhtml}html")
>>> body = etree.SubElement(xhtml, "{http://www.w3.org/1999/xhtml}body")
>>> body.text = "Hello World"

>>> print etree.tostring(xhtml, pretty_print=True),
<html:html xmlns:html="http://www.w3.org/1999/xhtml">
  <html:body>Hello World</html:body>
</html:html>

```

As you can see, prefixes only become important when you serialise the result. However, the above code becomes somewhat verbose due to the lengthy namespace names. And retyping or copying a string over and over again is error prone. It is therefore common practice to store a namespace URI in a global variable. To adapt the namespace prefixes for serialisation, you can also pass a mapping to the Element factory, e.g. to define the default namespace:

```

>>> XHTML_NAMESPACE = "http://www.w3.org/1999/xhtml"
>>> XHTML = "{%s}" % XHTML_NAMESPACE

>>> NSMAP = {None : XHTML_NAMESPACE} # the default namespace (no prefix)

>>> xhtml = etree.Element(XHTML + "html", nsmmap=NSMAP) # lxml only!
>>> body = etree.SubElement(xhtml, XHTML + "body")
>>> body.text = "Hello World"

>>> print etree.tostring(xhtml, pretty_print=True),
<html xmlns="http://www.w3.org/1999/xhtml">
  <body>Hello World</body>
</html>

```

Namespaces on attributes work alike:

```

>>> body.set(XHTML + "bgcolor", "#CCFFAA")

>>> print etree.tostring(xhtml, pretty_print=True),
<html xmlns="http://www.w3.org/1999/xhtml">
  <body bgcolor="#CCFFAA">Hello World</body>
</html>

>>> print body.get("bgcolor")
None
>>> body.get(XHTML + "bgcolor")
'#CCFFAA'

```

You can also use XPath in this way:

```

>>> find_xhtml_body = etree.XPath(      # lxml only !
...   "{%s}/body" % XHTML_NAMESPACE)
>>> results = find_xhtml_body(xhtml)

>>> print results[0].tag
{http://www.w3.org/1999/xhtml}body

```

## 8.5 The E-factory

The E-factory provides a simple and compact syntax for generating XML and HTML:

```

>>> from lxml.builder import E

>>> def CLASS(*args): # class is a reserved word in Python
...   return {"class":' '.join(args)}

>>> html = page = (
...   E.html(      # create an Element called "html"
...     E.head(
...       E.title("This is a sample document")
...     ),
...     E.body(
...       E.h1("Hello!", CLASS("title")),
...       E.p("This is a paragraph with ", E.b("bold"), " text in it!"),

```

```

...     E.p("This is another paragraph, with a", "\n      ",
...         E.a("link", href="http://www.python.org"), "."),
...     E.p("Here are some reserved characters: <spam&egg>."),
...     etree.XML("<p>And finally an embedded XHTML fragment.</p>"),
...     )
...   )
... )

>>> print etree.tostring(page, pretty_print=True),
<html>
  <head>
    <title>This is a sample document</title>
  </head>
  <body>
    <h1 class="title">Hello!</h1>
    <p>This is a paragraph with <b>bold</b> text in it!</p>
    <p>This is another paragraph, with a
      <a href="http://www.python.org">link</a>.</p>
    <p>Here are some reserved characters: &lt;spam&amp;egg&gt;.</p>
    <p>And finally an embedded XHTML fragment.</p>
  </body>
</html>

```

The Element creation based on attribute access makes it easy to build up a simple vocabulary for an XML language:

```

>>> from lxml.builder import ElementMaker # lxml only !

>>> E = ElementMaker(namespace="http://my.de/fault/namespace",
...                   nsmap={'p' : "http://my.de/fault/namespace"})

>>> DOC = E.doc
>>> TITLE = E.title
>>> SECTION = E.section
>>> PAR = E.par

>>> my_doc = DOC(
...   TITLE("The dog and the hog"),
...   SECTION(
...     TITLE("The dog"),
...     PAR("Once upon a time, ..."),
...     PAR("And then ...")
...   ),
...   SECTION(
...     TITLE("The hog"),
...     PAR("Sooner or later ...")
...   )
... )

>>> print etree.tostring(my_doc, pretty_print=True),
<p:doc xmlns:p="http://my.de/fault/namespace">
  <p:title>The dog and the hog</p:title>
  <p:section>
    <p:title>The dog</p:title>
    <p:par>Once upon a time, ...</p:par>

```

```

    <p:par>And then ...</p:par>
  </p:section>
  <p:section>
    <p:title>The hog</p:title>
    <p:par>Sooner or later ...</p:par>
  </p:section>
</p:doc>

```

One such example is the module `lxml.html.builder`, which provides a vocabulary for HTML.

## 8.6 ElementPath

The ElementTree library comes with a simple XPath-like path language called `ElementPath`. The main difference is that you can use the `{namespace}tag` notation in `ElementPath` expressions. However, advanced features like value comparison and functions are not available.

In addition to a [full XPath implementation](#), `lxml.etree` supports the `ElementPath` language in the same way `ElementTree` does, even using (almost) the same implementation. The API provides four methods here that you can find on `Elements` and `ElementTrees`:

- `iterfind()` iterates over all `Elements` that match the path expression
- `findall()` returns a list of matching `Elements`
- `find()` efficiently returns only the first match
- `findtext()` returns the `.text` content of the first match

Here are some examples:

```
>>> root = etree.XML("<root><a x='123'>aText<b/><c/><b/></a></root>")
```

Find a child of an `Element`:

```
>>> print root.find("b")
None
>>> print root.find("a").tag
a
```

Find an `Element` anywhere in the tree:

```
>>> print root.find("./b").tag
b
>>> [ b.tag for b in root.iterfind("./b") ]
['b', 'b']
```

Find `Elements` with a certain attribute:

```
>>> print root.findall("./a[@x]")[0].tag
a
>>> print root.findall("./a[@y]")
[]
```

## Chapter 9

# APIs specific to lxml.etree

lxml.etree tries to follow established APIs wherever possible. Sometimes, however, the need to expose a feature in an easy way led to the invention of a new API. This page describes the major differences and a few additions to the main ElementTree API.

For a complete reference of the API, see the [generated API documentation](#).

Separate pages describe the support for [parsing XML](#), [executing XPath and XSLT](#), [validating XML](#) and [interfacing with other XML tools through the SAX-API](#).

lxml is extremely extensible through [XPath functions in Python](#), custom [Python element classes](#), custom [URL resolvers](#) and even [at the C-level](#).

### 9.1 lxml.etree

lxml.etree tries to follow the [ElementTree API](#) wherever it can. There are however some incompatibilities (see [compatibility](#)). The extensions are documented here.

If you need to know which version of lxml is installed, you can access the `lxml.etree.LXML_VERSION` attribute to retrieve a version tuple. Note, however, that it did not exist before version 1.0, so you will get an `AttributeError` in older versions. The versions of `libxml2` and `libxslt` are available through the attributes `LIBXML_VERSION` and `LIBXSLT_VERSION`.

The following examples usually assume this to be executed first:

```
>>> from lxml import etree
>>> from StringIO import StringIO
```

### 9.2 Other Element APIs

While lxml.etree itself uses the ElementTree API, it is possible to replace the Element implementation by [custom element subclasses](#). This has been used to implement well-known XML APIs on top of lxml. For example, lxml ships with a data-binding implementation called [objectify](#), which is similar to the [Amara](#) bindery tool.

lxml.etree comes with a number of [different lookup schemes](#) to customize the mapping between libxml2

nodes and the Element classes used by lxml.etree.

### 9.3 Trees and Documents

Compared to the original ElementTree API, lxml.etree has an extended tree model. It knows about parents and siblings of elements:

```
>>> root = etree.Element("root")
>>> a = etree.SubElement(root, "a")
>>> b = etree.SubElement(root, "b")
>>> c = etree.SubElement(root, "c")
>>> d = etree.SubElement(root, "d")
>>> e = etree.SubElement(d, "e")
>>> b.getparent() == root
True
>>> print b.getnext().tag
c
>>> print c.getprevious().tag
b
```

Elements always live within a document context in lxml. This implies that there is also a notion of an absolute document root. You can retrieve an ElementTree for the root node of a document from any of its elements:

```
>>> tree = d.getroottree()
>>> print tree.getroot().tag
root
```

Note that this is different from wrapping an Element in an ElementTree. You can use ElementTrees to create XML trees with an explicit root node:

```
>>> tree = etree.ElementTree(d)
>>> print tree.getroot().tag
d
>>> print etree.tostring(tree)
<d><e/></d>
```

ElementTree objects are serialised as complete documents, including preceding or trailing processing instructions and comments.

All operations that you run on such an ElementTree (like XPath, XSLT, etc.) will understand the explicitly chosen root as root node of a document. They will not see any elements outside the ElementTree. However, ElementTrees do not modify their Elements:

```
>>> element = tree.getroot()
>>> print element.tag
d
>>> print element.getparent().tag
root
>>> print element.getroottree().getroot().tag
root
```

The rule is that all operations that are applied to Elements use either the Element itself as reference point, or the absolute root of the document that contains this Element (e.g. for absolute XPath expressions). All operations on an ElementTree use its explicit root node as reference.

## 9.4 Iteration

The ElementTree API makes Elements iterable to supports iteration over their children. Using the tree defined above, we get:

```
>>> [ child.tag for child in root ]
['a', 'b', 'c', 'd']
```

To iterate in the opposite direction, use the `reversed()` function that exists in Python 2.4 and later.

Tree traversal should use the `element.iter()` method:

```
>>> [ el.tag for el in root.iter() ]
['root', 'a', 'b', 'c', 'd', 'e']
```

lxml.etree also supports this, but additionally features an extended API for iteration over the children, following/preceding siblings, ancestors and descendants of an element, as defined by the respective XPath axis:

```
>>> [ child.tag for child in root.iterchildren() ]
['a', 'b', 'c', 'd']
>>> [ child.tag for child in root.iterchildren(reversed=True) ]
['d', 'c', 'b', 'a']
>>> [ sibling.tag for sibling in b.itersiblings() ]
['c', 'd']
>>> [ sibling.tag for sibling in c.itersiblings(preceding=True) ]
['b', 'a']
>>> [ ancestor.tag for ancestor in e.iterancestors() ]
['d', 'root']
>>> [ el.tag for el in root.iterdescendants() ]
['a', 'b', 'c', 'd', 'e']
```

Note how `element.iterdescendants()` does not include the element itself, as opposed to `element.iter()`. The latter effectively implements the 'descendant-or-self' axis in XPath.

All of these iterators support an additional `tag` keyword argument that filters the generated elements by tag name:

```
>>> [ child.tag for child in root.iterchildren(tag='a') ]
['a']
>>> [ child.tag for child in d.iterchildren(tag='a') ]
[]
>>> [ el.tag for el in root.iterdescendants(tag='d') ]
['d']
>>> [ el.tag for el in root.iter(tag='d') ]
['d']
```

See also the section on the utility functions `iterparse()` and `iterwalk()` in the [parser documentation](#).

## 9.5 Error handling on exceptions

Libxml2 provides error messages for failures, be it during parsing, XPath evaluation or schema validation. The preferred way of accessing them is through the local `error_log` property of the respective evaluator or transformer object. See their documentation for details.

However, lxml also keeps a global error log of all errors that occurred at the application level. Whenever an exception is raised, you can retrieve the errors that occurred and “might have” lead to the problem from the error log copy attached to the exception:

```
>>> etree.clearErrorLog()
>>> broken_xml = '''
... <root>
...   <a>
... </root>
... '''
>>> try:
...     etree.parse(StringIO(broken_xml))
... except etree.XMLSyntaxError, e:
...     pass # just put the exception into e
```

Once you have caught this exception, you can access its `error_log` property to retrieve the log entries or filter them by a specific type, error domain or error level:

```
>>> log = e.error_log.filter_from_level(etree.ErrorLevels.FATAL)
>>> print log
<string>:4:8:FATAL:PARSER:ERR_TAG_NAME_MISMATCH: Opening and ending tag mis-
match: a line 3 and root
<string>:5:1:FATAL:PARSER:ERR_TAG_NOT_FINISHED: Premature end of data in tag root line 2
```

This might look a little cryptic at first, but it is the information that libxml2 gives you. At least the message at the end should give you a hint what went wrong and you can see that the fatal errors (FATAL) happened during parsing (PARSER) lines 4, column 8 and line 5, column 1 of a string (<string>, or the filename if available). Here, PARSER is the so-called error domain, see `lxml.etree.ErrorDomains` for that. You can get it from a log entry like this:

```
>>> entry = log[0]
>>> print entry.domain_name, entry.type_name, entry.filename
PARSER ERR_TAG_NAME_MISMATCH <string>
```

There is also a convenience attribute `last_error` that returns the last error or fatal error that occurred:

```
>>> entry = e.error_log.last_error
>>> print entry.domain_name, entry.type_name, entry.filename
PARSER ERR_TAG_NOT_FINISHED <string>
```

## 9.6 Error logging

`lxml.etree` supports logging libxml2 messages to the Python stdlib logging module. This is done through the `etree.PyErrorLog` class. It disables the error reporting from exceptions and forwards log messages to a Python logger. To use it, see the descriptions of the function `etree.useGlobalPythonLog` and the class `etree.PyErrorLog` for help. Note that this does not affect the local error logs of XSLT, XMLSchema, etc.

## 9.7 Serialisation

`lxml.etree` has direct support for pretty printing XML output. Functions like `ElementTree.write()` and `tostring()` support it through a keyword argument:

```

>>> root = etree.XML("<root><test/></root>")
>>> print etree.tostring(root)
<root><test/></root>

>>> print etree.tostring(root, pretty_print=True),
<root>
  <test/>
</root>

```

Note the newline that is appended at the end when pretty printing the output. It was added in lxml 2.0.

By default, lxml (just as ElementTree) outputs the XML declaration only if it is required by the standard:

```

>>> unicode_root = etree.Element(u"t\u3120st")
>>> unicode_root.text = u"t\u0A0Ast"
>>> etree.tostring(unicode_root, encoding="utf-8")
'<t\xe3\x84\xa0st>t\xe0\xa8\x8ast</t\xe3\x84\xa0st>'

>>> print etree.tostring(unicode_root, encoding="iso-8859-1")
<?xml version='1.0' encoding='iso-8859-1'?>
<t&#12576;st>t&#2570;st</t&#12576;st>

```

Also see the general remarks on [Unicode support](#).

You can enable or disable the declaration explicitly by passing another keyword argument for the serialisation:

```

>>> print etree.tostring(root, xml_declaration=True)
<?xml version='1.0' encoding='ASCII'?>
<root><test/></root>

>>> unicode_root.clear()
>>> etree.tostring(unicode_root, encoding="UTF-16LE",
...               xml_declaration=False)
'<\x00t\x00 1s\x00t\x00/\x00>\x00'

```

Note that a standard compliant XML parser will not consider the last line well-formed XML if the encoding is not explicitly provided somehow, e.g. in an underlying transport protocol:

```

>>> notxml = etree.tostring(unicode_root, encoding="UTF-16LE",
...                         xml_declaration=False)
>>> root = etree.XML(notxml)          #doctest: +ELLIPSIS
Traceback (most recent call last):
...
XMLSyntaxError: ...

```

## 9.8 XInclude and ElementInclude

You can let lxml process xinclude statements in a document by calling the `xinclude()` method on a tree:

```

>>> data = StringIO(''\'
... <doc xmlns:xi="http://www.w3.org/2001/XInclude">
... <foo/>
... <xi:include href="doc/test.xml" />
... </doc>'')

```

```
>>> tree = etree.parse(data)
>>> tree.xinclude()
>>> print etree.tostring(tree.getroot())
<doc xmlns:xi="http://www.w3.org/2001/XInclude">
<foo/>
<a xml:base="doc/test.xml"/>
</doc>
```

Note that the ElementTree compatible `ElementInclude` module is also supported as `lxml.ElementInclude`. It has the additional advantage of supporting custom [URL resolvers](#) at the Python level. The normal XInclude mechanism cannot deploy these. If you need ElementTree compatibility or custom resolvers, you have to stick to the external Python module.

## 9.9 write\_c14n on ElementTree

The `lxml.etree.ElementTree` class has a method `write_c14n`, which takes a file object as argument. This file object will receive an UTF-8 representation of the canonicalized form of the XML, following the W3C C14N recommendation. For example:

```
>>> f = StringIO('<a><b></b></a>')
>>> tree = etree.parse(f)
>>> f2 = StringIO()
>>> tree.write_c14n(f2)
>>> f2.getvalue()
'<a><b></b></a>'
```

## Chapter 10

# Parsing XML and HTML with lxml

`lxml` provides a very simple and powerful API for parsing XML and HTML. It supports one-step parsing as well as step-by-step parsing using an event-driven API (currently only for XML).

The usual setup procedure:

```
>>> from lxml import etree
>>> from StringIO import StringIO
```

### 10.1 Parsers

Parsers are represented by parser objects. There is support for parsing both XML and (broken) HTML. Note that XHTML is best parsed as XML, parsing it with the HTML parser can lead to unexpected results. Here is a simple example for parsing XML from an in-memory string:

```
>>> xml = '<a xmlns="test"><b xmlns="test"/></a>'

>>> root = etree.fromstring(xml)
>>> print etree.tostring(root)
<a xmlns="test"><b xmlns="test"/></a>
```

To read from a file or file-like object, you can use the `parse()` function, which returns an `ElementTree` object:

```
>>> tree = etree.parse(StringIO(xml))
>>> print etree.tostring(tree.getroot())
<a xmlns="test"><b xmlns="test"/></a>
```

Note how the `parse()` function reads from a file-like object here. If parsing is done from a real file, it is more common (and also somewhat more efficient) to pass a filename:

```
>>> tree = etree.parse("doc/test.xml")
```

`lxml` can parse from a local file, an HTTP URL or an FTP URL. It also auto-detects and reads gzip-compressed XML files (`.gz`).

If you want to parse from memory and still provide a base URL for the document (e.g. to support relative paths in an XInclude), you can pass the `base_url` keyword argument:

```
>>> root = etree.fromstring(xml, base_url="http://where.it/is/from.xml")
```

### 10.1.1 Parser options

The parsers accept a number of setup options as keyword arguments. The above example is easily extended to clean up namespaces during parsing:

```
>>> parser = etree.XMLParser(ns_clean=True)
>>> tree = etree.parse(StringIO(xml), parser)
>>> print etree.tostring(tree.getroot())
<a xmlns="test"><b/></a>
```

The keyword arguments in the constructor are mainly based on the libxml2 parser configuration. A DTD will also be loaded if validation or attribute default values are requested.

Available boolean keyword arguments:

- `attribute_defaults` - read the DTD (if referenced by the document) and add the default attributes from it
- `dtd_validation` - validate while parsing (if a DTD was referenced)
- `load_dtd` - load and parse the DTD while parsing (no validation is performed)
- `no_network` - prevent network access when looking up external documents
- `ns_clean` - try to clean up redundant namespace declarations
- `recover` - try hard to parse through broken XML
- `remove_blank_text` - discard blank text nodes between tags
- `remove_comments` - discard comments
- `compact` - use compact storage for short text content (on by default)

### 10.1.2 Error log

Parsers have an `error_log` property that lists the errors of the last parser run:

```
>>> parser = etree.XMLParser()
>>> print len(parser.error_log)
0

>>> tree = etree.XML("<root></b>", parser)
Traceback (most recent call last):
...
XMLSyntaxError: Opening and ending tag mismatch: root line 1 and b, line 1, column 11

>>> print len(parser.error_log)
1

>>> error = parser.error_log[0]
>>> print error.message
Opening and ending tag mismatch: root line 1 and b
```

```
>>> print error.line, error.column
1 11
```

### 10.1.3 Parsing HTML

HTML parsing is similarly simple. The parsers have a `recover` keyword argument that the `HTMLParser` sets by default. It lets `libxml2` try its best to return something usable without raising an exception. You should use `libxml2` version 2.6.21 or newer to take advantage of this feature:

```
>>> broken_html = "<html><head><title>test<body><h1>page title</h3>"

>>> parser = etree.HTMLParser()
>>> tree = etree.parse(StringIO(broken_html), parser)

>>> print etree.tostring(tree.getroot(), pretty_print=True),
<html>
  <head>
    <title>test</title>
  </head>
  <body>
    <h1>page title</h1>
  </body>
</html>
```

Lxml has an HTML function, similar to the XML shortcut known from `ElementTree`:

```
>>> html = etree.HTML(broken_html)
>>> print etree.tostring(html, pretty_print=True),
<html>
  <head>
    <title>test</title>
  </head>
  <body>
    <h1>page title</h1>
  </body>
</html>
```

The support for parsing broken HTML depends entirely on `libxml2`'s recovery algorithm. It is *not* the fault of `lxml` if you find documents that are so heavily broken that the parser cannot handle them. There is also no guarantee that the resulting tree will contain all data from the original document. The parser may have to drop seriously broken parts when struggling to keep parsing. Especially misplaced meta tags can suffer from this, which may lead to encoding problems.

### 10.1.4 Doctype information

The use of the `libxml2` parsers makes some additional information available at the API level. Currently, `ElementTree` objects can access the DOCTYPE information provided by a parsed document, as well as the XML version and the original encoding:

```
>>> pub_id = "-//W3C//DTD XHTML 1.0 Transitional//EN"
>>> sys_url = "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"
>>> doctype_string = '<!DOCTYPE html PUBLIC "%s" "%s">' % (pub_id, sys_url)
>>> xml_header = '<?xml version="1.0" encoding="ascii"?>'
```

```
>>> xhtml = xml_header + doctype_string + '<html><body></body></html>'

>>> tree = etree.parse(StringIO(xhtml))
>>> docinfo = tree.docinfo
>>> print docinfo.public_id
-//W3C//DTD XHTML 1.0 Transitional//EN
>>> print docinfo.system_url
http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd
>>> docinfo.doctype == doctype_string
True

>>> print docinfo.xml_version
1.0
>>> print docinfo.encoding
ascii
```

## 10.2 The target parser interface

As in `ElementTree`, and similar to a SAX event handler, you can pass a target object to the parser:

```
>>> class EchoTarget:
...     def start(self, tag, attrib):
...         print "start", tag, attrib
...     def end(self, tag):
...         print "end", tag
...     def data(self, data):
...         print "data", repr(data)
...     def close(self):
...         print "close"
...         return "closed!"

>>> parser = etree.XMLParser(target = EchoTarget())

>>> result = etree.XML("<element>some text</element>", parser)
start element {}
data u'some text'
end element
close

>>> print result
closed!
```

Note that the parser does *not* build a tree in this case. The result of the parser run is what the target object returns from its `close()` method. If you want to return an XML tree here, you have to create it programmatically in the target object.

## 10.3 The feed parser interface

Since `lxml` 2.0, the parsers have a feed parser interface that is compatible to the `ElementTree` parsers. You can use it to feed data into the parser in a controlled step-by-step way. Note that you can only use one interface at a time with each parser: the `parse()` or `XML()` functions, or the feed parser interface.

To start parsing with a feed parser, just call its `feed()` method:

```
>>> parser = etree.XMLParser()

>>> for data in ('<?xml versio', 'n="1.0"?', '><roo', 't><a', '/></root>'):
...     parser.feed(data)
```

When you are done parsing, you **must** call the `close()` method to retrieve the root Element of the parse result document, and to unlock the parser:

```
>>> root = parser.close()

>>> print root.tag
root
>>> print root[0].tag
a
```

If you do not call `close()`, the parser will stay locked and subsequent usages will block till the end of times. So make sure you also close it in the exception case.

Another way of achieving the same step-by-step parsing is by writing your own file-like object that returns a chunk of data on each `read()` call. Where the feed parser interface allows you to actively pass data chunks into the parser, a file-like object passively responds to `read()` requests of the parser itself. Depending on the data source, either way may be more natural.

Note that the feed parser has its own error log called `feed_error_log`. Errors in the feed parser do not show up in the normal `error_log` and vice versa.

You can also combine the feed parser interface with the target parser:

```
>>> parser = etree.XMLParser(target = EchoTarget())

>>> parser.feed("<eleme")
>>> parser.feed("<nt>some text</elem")
start element {}
data u'some text'
>>> parser.feed("<ent>")
end element

>>> result = parser.close()
close
>>> print result
closed!
```

Again, this prevents the automatic creating of an XML tree and leaves all the event handling to the target object. The `close()` method of the parser forwards the return value of the target's `close()` method.

## 10.4 iterparse and iterwalk

As known from ElementTree, the `iterparse()` utility function returns an iterator that generates parser events for an XML file (or file-like object), while building the tree. The values are tuples (`event-type, object`). The event types are 'start', 'end', 'start-ns' and 'end-ns'.

The 'start' and 'end' events represent opening and closing elements and are accompanied by the respective

element. By default, only 'end' events are generated:

```
>>> xml = '''\
... <root>
...   <element key='value'>text</element>
...   <element>text</element>tail
...   <empty-element xmlns="testns" />
... </root>
... '''

>>> context = etree.iterparse(StringIO(xml))
>>> for action, elem in context:
...     print action, elem.tag
end element
end element
end {testns}empty-element
end root
```

The resulting tree is available through the `root` property of the iterator:

```
>>> context.root.tag
'root'
```

The other event types can be activated with the `events` keyword argument:

```
>>> events = ("start", "end")
>>> context = etree.iterparse(StringIO(xml), events=events)
>>> for action, elem in context:
...     print action, elem.tag
start root
start element
end element
start element
end element
start {testns}empty-element
end {testns}empty-element
end root
```

### 10.4.1 Selective tag events

As an extension over `ElementTree`, `lxml.etree` accepts a `tag` keyword argument just like `element.iter(tag)`. This restricts events to a specific tag or namespace:

```
>>> context = etree.iterparse(StringIO(xml), tag="element")
>>> for action, elem in context:
...     print action, elem.tag
end element
end element

>>> events = ("start", "end")
>>> context = etree.iterparse(
...     StringIO(xml), events=events, tag="{testns}*" )
>>> for action, elem in context:
...     print action, elem.tag
start {testns}empty-element
```

```
end {testns}empty-element
```

## 10.4.2 Modifying the tree

You can modify the element and its descendants when handling the 'end' event. To save memory, for example, you can remove subtrees that are no longer needed:

```
>>> context = etree.iterparse(StringIO(xml))
>>> for action, elem in context:
...     print len(elem),
...     elem.clear()
0 0 0 3
>>> context.root.getchildren()
[]
```

**WARNING:** During the 'start' event, the descendants and following siblings are not yet available and should not be accessed. During the 'end' event, the element and its descendants can be freely modified, but its following siblings should not be accessed. During either of the two events, you **must not** modify or move the ancestors (parents) of the current element. You should also avoid moving or discarding the element itself. The golden rule is: do not touch anything that will have to be touched again by the parser later on.

If you have elements with a long list of children in your XML file and want to save more memory during parsing, you can clean up the preceding siblings of the current element:

```
>>> for event, element in etree.iterparse(StringIO(xml)):
...     # ... do something with the element
...     element.clear() # clean up children
...     while element.getprevious() is not None:
...         del element.getparent()[0] # clean up preceding siblings
```

The `while` loop deletes multiple siblings in a row. This is only necessary if you skipped over some of them using the `tag` keyword argument. Otherwise, a simple `if` should do. The more selective your tag is, however, the more thought you will have to put into finding the right way to clean up the elements that were skipped. Therefore, it is sometimes easier to traverse all elements and do the tag selection by hand in the event handler code.

The 'start-ns' and 'end-ns' events notify about namespace declarations and generate tuples (`prefix`, `URI`):

```
>>> events = ("start-ns", "end-ns")
>>> context = etree.iterparse(StringIO(xml), events=events)
>>> for action, obj in context:
...     print action, obj
start-ns ('', 'testns')
end-ns None
```

It is common practice to use a list as namespace stack and pop the last entry on the 'end-ns' event.

## 10.4.3 iterwalk

A second extension over `ElementTree` is the `iterwalk()` function. It behaves exactly like `iterparse()`, but works on `Elements` and `ElementTrees`:

```
>>> root = etree.XML(xml)
```

```

>>> context = etree.iterwalk(
...         root, events=("start", "end"), tag="element")
>>> for action, elem in context:
...     print action, elem.tag
start element
end element
start element
end element

>>> f = StringIO(xml)
>>> context = etree.iterparse(
...         f, events=("start", "end"), tag="element")

>>> for action, elem in context:
...     print action, elem.tag
start element
end element
start element
end element

```

## 10.5 Python unicode strings

lxml.etree has broader support for Python unicode strings than the ElementTree library. First of all, where ElementTree would raise an exception, the parsers in lxml.etree can handle unicode strings straight away. This is most helpful for XML snippets embedded in source code using the XML() function:

```

>>> uxml = u'<test> \uf8d1 + \uf8d2 </test>'
>>> uxml
u'<test> \uf8d1 + \uf8d2 </test>'
>>> root = etree.XML(uxml)

```

This requires, however, that unicode strings do not specify a conflicting encoding themselves and thus lie about their real encoding:

```

>>> etree.XML(u'<?xml version="1.0" encoding="ASCII"?>\n' + uxml)
Traceback (most recent call last):
...
ValueError: Unicode strings with encoding declaration are not supported.

```

Similarly, you will get errors when you try the same with HTML data in a unicode string that specifies a charset in a meta tag of the header. You should generally avoid converting XML/HTML data to unicode before passing it into the parsers. It is both slower and error prone.

### 10.5.1 Serialising to Unicode strings

To serialize the result, you would normally use the `tostring()` module function, which serializes to plain ASCII by default or a number of other byte encodings if asked for:

```

>>> etree.tostring(root)
'<test> &#63697; + &#63698; </test>'

>>> etree.tostring(root, encoding='UTF-8', xml_declaration=False)
'<test> \xef\xa3\x91 + \xef\xa3\x92 </test>'

```

As an extension, `lxml.etree` recognises the unicode type as encoding to build a Python unicode representation of a tree:

```
>>> etree.tostring(root, encoding=unicode)
u'<test> \uf8d1 + \uf8d2 </test>'

>>> el = etree.Element("test")
>>> etree.tostring(el, encoding=unicode)
u'<test/>'

>>> subel = etree.SubElement(el, "subtest")
>>> etree.tostring(el, encoding=unicode)
u'<test><subtest/></test>'

>>> tree = etree.ElementTree(el)
>>> etree.tostring(tree, encoding=unicode)
u'<test><subtest/></test>'
```

The result of `tostring(encoding=unicode)` can be treated like any other Python unicode string and then passed back into the parsers. However, if you want to save the result to a file or pass it over the network, you should use `write()` or `tostring()` with a byte encoding (typically UTF-8) to serialize the XML. The main reason is that unicode strings returned by `tostring(encoding=unicode)` are not byte streams and they never have an XML declaration to specify their encoding. These strings are most likely not parsable by other XML libraries.

For normal byte encodings, the `tostring()` function automatically adds a declaration as needed that reflects the encoding of the returned string. This makes it possible for other parsers to correctly parse the XML byte stream. Note that using `tostring()` with UTF-8 is also considerably faster in most cases.

# Chapter 11

## Validation with lxml

Apart from the built-in DTD support in parsers, lxml currently supports three schema languages: [DTD](#), [Relax NG](#) and [XML Schema](#). All three provide identical APIs in lxml, represented by validator classes with the obvious names.

There is also initial support for [Schematron](#). However, it does not currently support error reporting in the validation phase due to insufficiencies in the implementation as of libxml2 2.6.30.

The usual setup procedure:

```
>>> from lxml import etree
>>> from StringIO import StringIO
```

### 11.1 Validation at parse time

The parser in lxml can do on-the-fly validation of a document against a DTD or an XML schema. The DTD is retrieved automatically based on the DOCTYPE of the parsed document. All you have to do is use a parser that has DTD validation enabled:

```
>>> parser = etree.XMLParser(dtd_validation=True)
```

Obviously, a request for validation enables the DTD loading feature. There are two other options that enable loading the DTD, but that do not perform any validation. The first is the `load_dtd` keyword option, which simply loads the DTD into the parser and makes it available to the document as external subset. You can retrieve the DTD from the parsed document using the `docinfo` property of the result `ElementTree` object. The internal subset is available as `internalDTD`, the external subset is provided as `externalDTD`.

The third way way to activate DTD loading is with the `attribute_defaults` option, which loads the DTD and weaves attribute default values into the document. Again, no validation is performed unless explicitly requested.

XML schema is supported in a similar way, but requires an explicit schema to be provided:

```
>>> schema_root = etree.XML('''\
... <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
...   <xsd:element name="a" type="xsd:integer"/>
... </xsd:schema>
```

```
... '''
>>> schema = etree.XMLSchema(schema_root)

>>> parser = etree.XMLParser(schema = schema)
>>> root = etree.fromstring("<a>5</a>", parser)
```

If the validation fails (be it for a DTD or an XML schema), the parser will raise an exception:

```
>>> root = etree.fromstring("<a>no int</a>", parser)
Traceback (most recent call last):
XMLSyntaxError: Element 'a': 'no int' is not a valid value of the atomic type 'xs:integer'.
```

If you want the parser to succeed regardless of the outcome of the validation, you should use a non-validating parser and run the validation separately after parsing the document.

## 11.2 DTD

As described above, the parser support for DTDs depends on internal or external subsets of the XML file. This means that the XML file itself must either contain a DTD or must reference a DTD to make this work. If you want to validate an XML document against a DTD that is not referenced by the document itself, you can use the DTD class.

To use the DTD class, you must first pass a filename or file-like object into the constructor to parse a DTD:

```
>>> f = StringIO("<!ELEMENT b EMPTY>")
>>> dtd = etree.DTD(f)
```

Now you can use it to validate documents:

```
>>> root = etree.XML("<b/>")
>>> print dtd.validate(root)
True

>>> root = etree.XML("<b><a/></b>")
>>> print dtd.validate(root)
False
```

The reason for the validation failure can be found in the error log:

```
>>> print dtd.error_log.filter_from_errors()[0]
<string>:1:0:ERROR:VALID:DTD_NOT_EMPTY: Element b was declared EMPTY this one has content
```

As an alternative to parsing from a file, you can use the `external_id` keyword argument to parse from a catalog. The following example reads the DocBook DTD in version 4.2, if available in the system catalog:

```
dtd = etree.DTD(external_id = "-//OASIS//DTD DocBook XML V4.2//EN")
```

## 11.3 RelaxNG

The RelaxNG class takes an ElementTree object to construct a Relax NG validator:

```
>>> f = StringIO(''\
```

```

... <element name="a" xmlns="http://relaxng.org/ns/structure/1.0">
...   <zeroOrMore>
...     <element name="b">
...       <text />
...     </element>
...   </zeroOrMore>
... </element>
... '''
>>> relaxng_doc = etree.parse(f)
>>> relaxng = etree.RelaxNG(relaxng_doc)

```

Alternatively, pass a filename to the `file` keyword argument to parse from a file. This also enables correct handling of include files from within the RelaxNG parser.

You can then validate some ElementTree document against the schema. You'll get back `True` if the document is valid against the Relax NG schema, and `False` if not:

```

>>> valid = StringIO('<a><b></b></a>')
>>> doc = etree.parse(valid)
>>> relaxng.validate(doc)
True

>>> invalid = StringIO('<a><c></c></a>')
>>> doc2 = etree.parse(invalid)
>>> relaxng.validate(doc2)
False

```

Calling the schema object has the same effect as calling its `validate` method. This is sometimes used in conditional statements:

```

>>> invalid = StringIO('<a><c></c></a>')
>>> doc2 = etree.parse(invalid)
>>> if not relaxng(doc2):
...     print "invalid!"
invalid!

```

If you prefer getting an exception when validating, you can use the `assert_` or `assertValid` methods:

```

>>> relaxng.assertValid(doc2)
Traceback (most recent call last):
  [...]
DocumentInvalid: Did not expect element c there, line 1

>>> relaxng.assert_(doc2)
Traceback (most recent call last):
  [...]
AssertionError: Did not expect element c there, line 1

```

If you want to find out why the validation failed in the second case, you can look up the error log of the validation process and check it for relevant messages:

```

>>> log = relaxng.error_log
>>> print log.last_error
<string>:1:0:ERROR:RELAXNGV:RELAXNG_ERR_ELEMWRONG: Did not expect element c there

```

You can see that the error (ERROR) happened during RelaxNG validation (RELAXNGV). The message then tells you what went wrong. You can also look at the error domain and its type directly:

```

>>> error = log.last_error
>>> print error.domain_name
RELAXNGV
>>> print error.type_name
RELAXNG_ERR_ELEMWRONG

```

Note that this error log is local to the RelaxNG object. It will only contain log entries that appeared during the validation.

Similar to XSLT, there's also a less efficient but easier shortcut method to do one-shot RelaxNG validation:

```

>>> doc.relaxng(relaxng_doc)
True
>>> doc2.relaxng(relaxng_doc)
False

```

libxml2 does not currently support the [RelaxNG Compact Syntax](#). However, the `trang` translator can convert the compact syntax to the XML syntax, which can then be used with `lxml`.

## 11.4 XMLSchema

`lxml.etree` also has XML Schema (XSD) support, using the class `lxml.etree.XMLSchema`. The API is very similar to the Relax NG and DTD classes. Pass an `ElementTree` object to construct a `XMLSchema` validator:

```

>>> f = StringIO('''\
... <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
... <xsd:element name="a" type="AType"/>
... <xsd:complexType name="AType">
...   <xsd:sequence>
...     <xsd:element name="b" type="xsd:string" />
...   </xsd:sequence>
... </xsd:complexType>
... </xsd:schema>
... ''')
>>> xmlschema_doc = etree.parse(f)
>>> xmlschema = etree.XMLSchema(xmlschema_doc)

```

You can then validate some `ElementTree` document with this. Like with RelaxNG, you'll get back `true` if the document is valid against the XML schema, and `false` if not:

```

>>> valid = StringIO('<a><b></b></a>')
>>> doc = etree.parse(valid)
>>> xmlschema.validate(doc)
True

>>> invalid = StringIO('<a><c></c></a>')
>>> doc2 = etree.parse(invalid)
>>> xmlschema.validate(doc2)
False

```

Calling the schema object has the same effect as calling its `validate` method. This is sometimes used in conditional statements:

```
>>> invalid = StringIO('<a><c></c></a>')
>>> doc2 = etree.parse(invalid)
>>> if not xmlschema(doc2):
...     print "invalid!"
invalid!
```

If you prefer getting an exception when validating, you can use the `assert_` or `assertValid` methods:

```
>>> xmlschema.assertValid(doc2)
Traceback (most recent call last):
  [...]
DocumentInvalid: Element 'c': This element is not expected. Ex-
pected is ( b )., line 1

>>> xmlschema.assert_(doc2)
Traceback (most recent call last):
  [...]
AssertionError: Element 'c': This element is not expected. Expected is ( b )., line 1
```

Error reporting works as for the RelaxNG class:

```
>>> log = xmlschema.error_log
>>> error = log.last_error
>>> print error.domain_name
SCHEMASV
>>> print error.type_name
SCHEMAV_ELEMENT_CONTENT
```

If you were to print this log entry, you would get something like the following. Note that the error message depends on the libxml2 version in use:

```
<string>:1:ERROR::SCHEMAV_ELEMENT_CONTENT: Element 'c': This element is not ex-
pected. Expected is ( b ).
```

Similar to XSLT and RelaxNG, there's also a less efficient but easier shortcut method to do XML Schema validation:

```
>>> doc.xmlschema(xmlschema_doc)
True
>>> doc2.xmlschema(xmlschema_doc)
False
```

## 11.5 Schematron

Since version 2.0, `lxml.etree` features [Schematron](#) support, using the class `lxml.etree.Schematron`. It requires at least libxml2 2.6.21 to work. The API is the same as for the other validators. Pass an `ElementTree` object to construct a Schematron validator:

```
>>> f = StringIO(''\
... <schema xmlns="http://www.ascc.net/xml/schematron" >
...   <pattern name="Sum equals 100%.">
...     <rule context="Total">
...       <assert test="sum(//Percent)=100">Sum is not 100%.</assert>
...     </rule>
...   </pattern>
```

```
... </schema>
... '''
```

```
>>> sct_doc = etree.parse(f)
>>> schematron = etree.Schematron(sct_doc)
```

You can then validate some ElementTree document with this. Like with RelaxNG, you'll get back true if the document is valid against the schema, and false if not:

```
>>> valid = StringIO('''\
... <Total>
...   <Percent>20</Percent>
...   <Percent>30</Percent>
...   <Percent>50</Percent>
... </Total>
... ''')
```

```
>>> doc = etree.parse(valid)
>>> schematron.validate(doc)
True
```

```
>>> etree.SubElement(doc.getroot(), "Percent").text = "10"
```

```
>>> schematron.validate(doc)
False
```

Calling the schema object has the same effect as calling its validate method. This is sometimes used in conditional statements:

```
>>> is_valid = etree.Schematron(sct_doc)

>>> if not is_valid(doc):
...     print "invalid!"
invalid!
```

Note that libxml2 restricts error reporting to the parsing step (when creating the Schematron instance). There is not currently any support for error reporting during validation.

## Chapter 12

# XPath and XSLT with lxml

lxml supports both XPath and XSLT through libxml2 and libxslt in a standards compliant way.

The usual setup procedure:

```
>>> from lxml import etree
>>> from StringIO import StringIO
```

### 12.1 XPath

lxml.etree supports the simple path syntax of the [find](#), [findall](#) and [findtext](#) methods on `ElementTree` and `Element`, as known from the original `ElementTree` library ([ElementPath](#)). As an lxml specific extension, these classes also provide an `xpath()` method that supports expressions in the complete XPath syntax, as well as [custom extension functions](#).

There are also specialized XPath evaluator classes that are more efficient for frequent evaluation: `XPath` and `XPathEvaluator`. See the [performance comparison](#) to learn when to use which. Their semantics when used on `Elements` and `ElementTrees` are the same as for the `xpath()` method described here.

#### 12.1.1 The `xpath()` method

For `ElementTree`, the `xpath` method performs a global XPath query against the document (if absolute) or against the root node (if relative):

```
>>> f = StringIO('<foo><bar></bar></foo>')
>>> tree = etree.parse(f)

>>> r = tree.xpath('/foo/bar')
>>> len(r)
1
>>> r[0].tag
'bar'

>>> r = tree.xpath('bar')
>>> r[0].tag
'bar'
```

When `xpath()` is used on an Element, the XPath expression is evaluated against the element (if relative) or against the root tree (if absolute):

```
>>> root = tree.getroot()
>>> r = root.xpath('bar')
>>> r[0].tag
'bar'

>>> bar = root[0]
>>> r = bar.xpath('/foo/bar')
>>> r[0].tag
'bar'

>>> tree = bar.getroottree()
>>> r = tree.xpath('/foo/bar')
>>> r[0].tag
'bar'
```

The `xpath()` method has support for XPath variables:

```
>>> expr = "//*[local-name() = $name]"

>>> print root.xpath(expr, name = "foo")[0].tag
foo

>>> print root.xpath(expr, name = "bar")[0].tag
bar

>>> print root.xpath("$text", text = "Hello World!")
Hello World!
```

Optionally, you can provide a `namespaces` keyword argument, which should be a dictionary mapping the namespace prefixes used in the XPath expression to namespace URIs:

```
>>> f = StringIO('''\
... <a:foo xmlns:a="http://codespeak.net/ns/test1"
...     xmlns:b="http://codespeak.net/ns/test2">
...     <b:bar>Text</b:bar>
... </a:foo>
... ''')
>>> doc = etree.parse(f)

>>> r = doc.xpath('/t:foo/b:bar',
...               namespaces={'t': 'http://codespeak.net/ns/test1',
...                           'b': 'http://codespeak.net/ns/test2'})
>>> len(r)
1
>>> r[0].tag
'{http://codespeak.net/ns/test2}bar'
>>> r[0].text
'Text'
```

There is also an optional `extensions` argument which is used to define [custom extension functions](#) in Python that are local to this evaluation.

### 12.1.2 XPath return values

The return values of XPath evaluations vary, depending on the XPath expression used:

- True or False, when the XPath expression has a boolean result
- a float, when the XPath expression has a numeric result (integer or float)
- a 'smart' string (as described below), when the XPath expression has a string result.
- a list of items, when the XPath expression has a list as result. The items may include Elements (also comments and processing instructions), strings and tuples. Text nodes and attributes in the result are returned as 'smart' string values. Namespace declarations are returned as tuples of strings: (prefix, URI).

XPath string results are 'smart' in that they provide a `getparent()` method that knows their origin:

- for attribute values, `result.getparent()` returns the Element that carries them. An example is `//foo/@attribute`, where the parent would be a `foo` Element.
- for the `text()` function (as in `//text()`), it returns the Element that contains the text or tail that was returned.

You can distinguish between different text origins with the boolean properties `is_text`, `is_tail` and `is_attribute`.

Note that `getparent()` may not always return an Element. For example, the XPath functions `string()` and `concat()` will construct strings that do not have an origin. For them, `getparent()` will return `None`.

### 12.1.3 Generating XPath expressions

ElementTree objects have a method `getpath(element)`, which returns a structural, absolute XPath expression to find that element:

```
>>> a = etree.Element("a")
>>> b = etree.SubElement(a, "b")
>>> c = etree.SubElement(a, "c")
>>> d1 = etree.SubElement(c, "d")
>>> d2 = etree.SubElement(c, "d")

>>> tree = etree.ElementTree(c)
>>> print tree.getpath(d2)
/c/d[2]
>>> tree.xpath(tree.getpath(d2)) == [d2]
True
```

### 12.1.4 The XPath class

The XPath class compiles an XPath expression into a callable function:

```
>>> root = etree.XML("<root><a><b></a><b></root>")

>>> find = etree.XPath("//b")
>>> print find(root)[0].tag
```

b

The compilation takes as much time as in the `xpath()` method, but it is done only once per class instantiation. This makes it especially efficient for repeated evaluation of the same XPath expression.

Just like the `xpath()` method, the `XPath` class supports XPath variables:

```
>>> count_elements = etree.XPath("count(//*[local-name() = $name])")

>>> print count_elements(root, name = "a")
1.0
>>> print count_elements(root, name = "b")
2.0
```

This supports very efficient evaluation of modified versions of an XPath expression, as compilation is still only required once.

Prefix-to-namespace mappings can be passed as second parameter:

```
>>> root = etree.XML("<root xmlns='NS'><a><b/></a><b/></root>")

>>> find = etree.XPath("//n:b", namespaces={'n':'NS'})
>>> print find(root)[0].tag
{NS}b
```

By default, XPath supports regular expressions in the [EXSLT](#) namespace:

```
>>> regexpNS = "http://exslt.org/regular-expressions"
>>> find = etree.XPath("//*[re:test(., '^abc$', 'i')]",
...                       namespaces={'re':regexpNS})

>>> root = etree.XML("<root><a>aB</a><b>aBc</b></root>")
>>> print find(root)[0].text
aBc
```

You can disable this with the boolean keyword argument `regexp` which defaults to `True`.

### 12.1.5 The XPathEvaluator classes

`lxml.etree` provides two other efficient XPath evaluators that work on `ElementTrees` or `Elements` respectively: `XPathDocumentEvaluator` and `XPathElementEvaluator`. They are automatically selected if you use the `XPathEvaluator` helper for instantiation:

```
>>> root = etree.XML("<root><a><b/></a><b/></root>")
>>> xpatheval = etree.XPathEvaluator(root)

>>> print isinstance(xpatheval, etree.XPathElementEvaluator)
True

>>> print xpatheval("//b")[0].tag
b
```

This class provides efficient support for evaluating different XPath expressions on the same `Element` or `ElementTree`.

### 12.1.6 ETXPath

ElementTree supports a language named `ElementPath` in its `find*()` methods. One of the main differences between XPath and ElementPath is that the XPath language requires an indirection through prefixes for namespace support, whereas ElementTree uses the Clark notation (`{ns}name`) to avoid prefixes completely. The other major difference regards the capabilities of both path languages. Where XPath supports various sophisticated ways of restricting the result set through functions and boolean expressions, ElementPath only supports pure path traversal without nesting or further conditions. So, while the ElementPath syntax is self-contained and therefore easier to write and handle, XPath is much more powerful and expressive.

lxml.etree bridges this gap through the class `ETXPath`, which accepts XPath expressions with namespaces in Clark notation. It is identical to the `XPath` class, except for the namespace notation. Normally, you would write:

```
>>> root = etree.XML("<root xmlns='ns'><a><b/></a><b/></root>")

>>> find = etree.XPath("//p:b", namespaces={'p' : 'ns'})
>>> print find(root)[0].tag
{ns}b
```

ETXPath allows you to change this to:

```
>>> find = etree.ETXPath("/{ns}b")
>>> print find(root)[0].tag
{ns}b
```

### 12.1.7 Error handling

lxml.etree raises exceptions when errors occur while parsing or evaluating an XPath expression:

```
>>> find = etree.XPath("\\")
Traceback (most recent call last):
...
XPathSyntaxError: Invalid expression
```

lxml will also try to give you a hint what went wrong, so if you pass a more complex expression, you may get a somewhat more specific error:

```
>>> find = etree.XPath("//*[1.1.1]")
Traceback (most recent call last):
...
XPathSyntaxError: Invalid predicate
```

During evaluation, lxml will emit an `XPathEvalError` on errors:

```
>>> find = etree.XPath("//ns:a")
>>> find(root)
Traceback (most recent call last):
...
XPathEvalError: Undefined namespace prefix
```

This works for the `XPath` class, however, the other evaluators (including the `xpath()` method) are one-shot operations that do parsing and evaluation in one step. They therefore raise evaluation exceptions in all cases:

```

>>> root = etree.Element("test")
>>> find = root.xpath("//*[@1.1.1]")
Traceback (most recent call last):
...
XPathEvalError: Invalid predicate

>>> find = root.xpath("//ns:a")
Traceback (most recent call last):
...
XPathEvalError: Undefined namespace prefix

>>> find = root.xpath("\\\\")
Traceback (most recent call last):
...
XPathEvalError: Invalid expression

```

Note that lxml versions before 1.3 always raised an `XPathSyntaxError` for all errors, including evaluation errors. The best way to support older versions is to except on the superclass `XPathError`.

## 12.2 XSLT

`lxml.etree` introduces a new class, `lxml.etree.XSLT`. The class can be given an `ElementTree` object to construct an XSLT transformer:

```

>>> f = StringIO(''\
... <xsl:stylesheet version="1.0"
...   xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
...   <xsl:template match="/">
...     <foo><xsl:value-of select="/a/b/text()" /></foo>
...   </xsl:template>
... </xsl:stylesheet>'')
>>> xslt_doc = etree.parse(f)
>>> transform = etree.XSLT(xslt_doc)

```

You can then run the transformation on an `ElementTree` document by simply calling it, and this results in another `ElementTree` object:

```

>>> f = StringIO('<a><b>Text</b></a>')
>>> doc = etree.parse(f)
>>> result_tree = transform(doc)

```

By default, XSLT supports all extension functions from `libxslt` and `libexslt` as well as Python regular expressions through the [EXSLT regexp functions](#). Also see the documentation on [custom extension functions](#) and [document resolvers](#). There is a separate section on [controlling access](#) to external documents and resources.

### 12.2.1 XSLT result objects

The result of an XSL transformation can be accessed like a normal `ElementTree` document:

```

>>> f = StringIO('<a><b>Text</b></a>')
>>> doc = etree.parse(f)
>>> result = transform(doc)

```

```
>>> result.getroot().text
'Text'
```

but, as opposed to normal ElementTree objects, can also be turned into an (XML or text) string by applying the `str()` function:

```
>>> str(result)
'<?xml version="1.0"?>\n<foo>Text</foo>\n'
```

The result is always a plain string, encoded as requested by the `xsl:output` element in the stylesheet. If you want a Python unicode string instead, you should set this encoding to UTF-8 (unless the *ASCII* default is sufficient). This allows you to call the builtin `unicode()` function on the result:

```
>>> unicode(result)
u'<?xml version="1.0"?>\n<foo>Text</foo>\n'
```

You can use other encodings at the cost of multiple recoding. Encodings that are not supported by Python will result in an error:

```
>>> xslt_tree = etree.XML('''\
... <xsl:stylesheet version="1.0"
...   xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
...   <xsl:output encoding="UCS4"/>
...   <xsl:template match="/">
...     <foo><xsl:value-of select="/a/b/text()" /></foo>
...   </xsl:template>
... </xsl:stylesheet>''')
>>> transform = etree.XSLT(xslt_tree)

>>> result = transform(doc)
>>> unicode(result)
Traceback (most recent call last):
  [...]
LookupError: unknown encoding: UCS4
```

## 12.2.2 Stylesheet parameters

It is possible to pass parameters, in the form of XPath expressions, to the XSLT template:

```
>>> xslt_tree = etree.XML('''\
... <xsl:stylesheet version="1.0"
...   xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
...   <xsl:template match="/">
...     <foo><xsl:value-of select="$a" /></foo>
...   </xsl:template>
... </xsl:stylesheet>''')
>>> transform = etree.XSLT(xslt_tree)
>>> f = StringIO('<a><b>Text</b></a>')
>>> doc = etree.parse(f)
```

The parameters are passed as keyword parameters to the `transform` call. First let's try passing in a simple string expression:

```
>>> result = transform(doc, a="'A'")
>>> str(result)
```

```
'<?xml version="1.0"?>\n<foo>A</foo>\n'
```

Let's try a non-string XPath expression now:

```
>>> result = transform(doc, a="/a/b/text()")
>>> str(result)
'<?xml version="1.0"?>\n<foo>Text</foo>\n'
```

### 12.2.3 The `xslt()` tree method

There's also a convenience method on `ElementTree` objects for doing XSL transformations. This is less efficient if you want to apply the same XSL transformation to multiple documents, but is shorter to write for one-shot operations, as you do not have to instantiate a stylesheet yourself:

```
>>> result = doc.xslt(xslt_tree, a="'A'")
>>> str(result)
'<?xml version="1.0"?>\n<foo>A</foo>\n'
```

This is a shortcut for the following code:

```
>>> transform = etree.XSLT(xslt_tree)
>>> result = transform(doc, a="'A'")
>>> str(result)
'<?xml version="1.0"?>\n<foo>A</foo>\n'
```

### 12.2.4 Profiling

If you want to know how your stylesheet performed, pass the `profile_run` keyword to the transform:

```
>>> result = transform(doc, a="/a/b/text()", profile_run=True)
>>> profile = result.xslt_profile
```

The value of the `xslt_profile` property is an `ElementTree` with profiling data about each template, similar to the following:

```
<profile>
  <template rank="1" match="/" name="" mode="" calls="1" time="1" aver-
age="1"/>
</profile>
```

Note that this is a read-only document. You must not move any of its elements to other documents. Please deep-copy the document if you need to modify it. If you want to free it from memory, just do:

```
>>> del result.xslt_profile
```

## Chapter 13

# lxml.objectify

**Author:** Stefan Behnel

**Author:** Holger Joukl

lxml supports an alternative API similar to the [Amara](#) bindery or [gnosis.xml.objectify](#) through a custom Element implementation. The main idea is to hide the usage of XML behind normal Python objects, sometimes referred to as data-binding. It allows you to use XML as if you were dealing with a normal Python object hierarchy.

Accessing the children of an XML element deploys object attribute access. If there are multiple children with the same name, slicing and indexing can be used. Python data types are extracted from XML content automatically and made available to the normal Python operators.

To set up and use `objectify`, you need both the `lxml.etree` module and `lxml.objectify`:

```
>>> from lxml import etree
>>> from lxml import objectify
```

The `objectify` API is very different from the `ElementTree` API. If it is used, it should not be mixed with other element implementations (such as trees parsed with `lxml.etree`), to avoid non-obvious behaviour.

The [benchmark page](#) has some hints on performance optimisation of code using `lxml.objectify`.

To make the doctests in this document look a little nicer, we also use this:

```
>>> import lxml.usedoctest
```

Imported from within a doctest, this relieves us from caring about the exact formatting of XML output.

### 13.1 The `lxml.objectify` API

In `lxml.objectify`, element trees provide an API that models the behaviour of normal Python object trees as closely as possible.

### 13.1.1 Creating objectify trees

As with `lxml.etree`, you can either create an `objectify` tree by parsing an XML document or by building one from scratch. To parse a document, just use the `parse()` or `fromstring()` functions of the module:

```
>>> from StringIO import StringIO
>>> fileobject = StringIO('<test/>')

>>> tree = objectify.parse(fileobject)
>>> print isinstance(tree.getroot(), objectify.ObjectifiedElement)
True

>>> root = objectify.fromstring('<test/>')
>>> print isinstance(root, objectify.ObjectifiedElement)
True
```

To build a new tree in memory, `objectify` replicates the standard factory function `Element()` from `lxml.etree`:

```
>>> obj_el = objectify.Element("new")
>>> print isinstance(obj_el, objectify.ObjectifiedElement)
True
```

After creating such an `Element`, you can use the [usual API](#) of `lxml.etree` to add `SubElements` to the tree:

```
>>> child = etree.SubElement(obj_el, "newchild", attr="value")
```

New subelements will automatically inherit the `objectify` behaviour from their tree. However, all independent elements that you create through the `Element()` factory of `lxml.etree` (instead of `objectify`) will not support the `objectify` API by themselves:

```
>>> subel = etree.SubElement(obj_el, "sub")
>>> print isinstance(subel, objectify.ObjectifiedElement)
True

>>> independent_el = etree.Element("new")
>>> print isinstance(independent_el, objectify.ObjectifiedElement)
False
```

### 13.1.2 Element access through object attributes

The main idea behind the `objectify` API is to hide XML element access behind the usual object attribute access pattern. Asking an element for an attribute will return the sequence of children with corresponding tag names:

```
>>> root = objectify.Element("root")
>>> b = etree.SubElement(root, "b")
>>> print root.b[0].tag
b
>>> root.index(root.b[0])
0
>>> b = etree.SubElement(root, "b")
>>> print root.b[0].tag
b
```

```
>>> print root.b[1].tag
b
>>> root.index(root.b[1])
1
```

For convenience, you can omit the index '0' to access the first child:

```
>>> print root.b.tag
b
>>> root.index(root.b)
0
>>> del root.b
```

Iteration and slicing also obey the requested tag:

```
>>> x1 = etree.SubElement(root, "x")
>>> x2 = etree.SubElement(root, "x")
>>> x3 = etree.SubElement(root, "x")

>>> [ el.tag for el in root.x ]
['x', 'x', 'x']

>>> [ el.tag for el in root.x[1:3] ]
['x', 'x']

>>> [ el.tag for el in root.x[-1:] ]
['x']

>>> del root.x[1:2]
>>> [ el.tag for el in root.x ]
['x', 'x']
```

If you want to iterate over all children or need to provide a specific namespace for the tag, use the `iterchildren()` method. Like the other methods for iteration, it supports an optional tag keyword argument:

```
>>> [ el.tag for el in root.iterchildren() ]
['b', 'x', 'x']

>>> [ el.tag for el in root.iterchildren(tag='b') ]
['b']

>>> [ el.tag for el in root.b ]
['b']
```

XML attributes are accessed as in the normal ElementTree API:

```
>>> c = etree.SubElement(root, "c", myattr="someval")
>>> print root.c.get("myattr")
someval

>>> root.c.set("c", "oh-oh")
>>> print root.c.get("c")
oh-oh
```

In addition to the normal ElementTree API for appending elements to trees, subtrees can also be added by assigning them to object attributes. In this case, the subtree is automatically deep copied and the

tag name of its root is updated to match the attribute name:

```
>>> el = objectify.Element("yet_another_child")
>>> root.new_child = el
>>> print root.new_child.tag
new_child
>>> print el.tag
yet_another_child

>>> root.y = [ objectify.Element("y"), objectify.Element("y") ]
>>> [ el.tag for el in root.y ]
['y', 'y']
```

The latter is a short form for operations on the full slice:

```
>>> root.y[:] = [ objectify.Element("y") ]
>>> [ el.tag for el in root.y ]
['y']
```

You can also replace children that way:

```
>>> child1 = etree.SubElement(root, "child")
>>> child2 = etree.SubElement(root, "child")
>>> child3 = etree.SubElement(root, "child")

>>> el = objectify.Element("new_child")
>>> subel = etree.SubElement(el, "sub")

>>> root.child = el
>>> print root.child.sub.tag
sub

>>> root.child[2] = el
>>> print root.child[2].sub.tag
sub
```

Note that special care must be taken when changing the tag name of an element:

```
>>> print root.b.tag
b
>>> root.b.tag = "notB"
>>> root.b
Traceback (most recent call last):
...
AttributeError: no such child: b
>>> print root.notB.tag
notB
```

### 13.1.3 Tree generation with the E-factory

To simplify the generation of trees even further, you can use the E-factory:

```
>>> E = objectify.E
>>> root = E.root(
...     E.a(5L),
...     E.b(6.1),
```

```

...     E.c(True),
...     E.d("how", tell="me")
... )

>>> print etree.tostring(root, pretty_print=True)
<root xmlns:py="http://codespeak.net/lxml/objectify/pytype">
  <a py:pytype="long">5</a>
  <b py:pytype="float">6.1</b>
  <c py:pytype="bool">>true</c>
  <d py:pytype="str" tell="me">how</d>
</root>

```

This allows you to write up a specific language in tags:

```

>>> ROOT = objectify.E.root
>>> TITLE = objectify.E.title
>>> HOWMANY = getattr(objectify.E, "how-many")

>>> root = ROOT(
...     TITLE("The title"),
...     HOWMANY(5)
... )

>>> print etree.tostring(root, pretty_print=True)
<root xmlns:py="http://codespeak.net/lxml/objectify/pytype">
  <title py:pytype="str">The title</title>
  <how-many py:pytype="int">5</how-many>
</root>

```

`objectify.E` is an instance of `objectify.ElementMaker`. By default, it creates pytype annotated Elements without a namespace. You can switch off the pytype annotation by passing `False` to the `annotate` keyword argument of the constructor. You can also pass a default namespace and an `nsmap`:

```

>>> myE = objectify.ElementMaker(annotate=False,
...     namespace="http://my/ns", nsmap={None : "http://my/ns"})

>>> root = myE.root( myE.someint(2) )

>>> print etree.tostring(root, pretty_print=True)
<root xmlns="http://my/ns">
  <someint>2</someint>
</root>

```

### 13.1.4 Namespace handling

Namespaces are handled mostly behind the scenes. If you access a child of an Element without specifying a namespace, the lookup will use the namespace of the parent:

```

>>> root = objectify.Element("{ns}root")
>>> b = etree.SubElement(root, "{ns}b")
>>> c = etree.SubElement(root, "{other}c")

>>> print root.b.tag
{ns}b
>>> print root.c

```

```
Traceback (most recent call last):
...
AttributeError: no such child: {ns}c
```

You can access elements with different namespaces via `getattr()`:

```
>>> print getattr(root, "{other}c").tag
{other}c
```

For convenience, there is also a quick way through item access:

```
>>> print root["{other}c"].tag
{other}c
```

The same approach must be used to access children with tag names that are not valid Python identifiers:

```
>>> el = etree.SubElement(root, "{ns}tag-name")
>>> print root["tag-name"].tag
{ns}tag-name

>>> new_el = objectify.Element("{ns}new-element")
>>> el = etree.SubElement(new_el, "{ns}child")
>>> el = etree.SubElement(new_el, "{ns}child")
>>> el = etree.SubElement(new_el, "{ns}child")

>>> root["tag-name"] = [ new_el, new_el ]
>>> print len(root["tag-name"])
2
>>> print root["tag-name"].tag
{ns}tag-name

>>> print len(root["tag-name"].child)
3
>>> print root["tag-name"].child.tag
{ns}child
>>> print root["tag-name"][1].child.tag
{ns}child
```

or for names that have a special meaning in `lxml.objectify`:

```
>>> root = objectify.XML("<root><text>TEXT</text></root>")

>>> print root.text.text
Traceback (most recent call last):
...
AttributeError: 'NoneType' object has no attribute 'text'

>>> print root["text"].text
TEXT
```

## 13.2 Asserting a Schema

When dealing with XML documents from different sources, you will often require them to follow a common schema. In `lxml.objectify`, this directly translates to enforcing a specific object tree, i.e. expected object attributes are ensured to be there and to have the expected type. This can easily be achieved

through XML Schema validation at parse time. Also see the [documentation on validation](#) on this topic.

First of all, we need a parser that knows our schema, so let's say we parse the schema from a file-like object (or file or filename):

```
>>> from StringIO import StringIO
>>> f = StringIO(''\
... <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
...   <xsd:element name="a" type="AType"/>
...   <xsd:complexType name="AType">
...     <xsd:sequence>
...       <xsd:element name="b" type="xsd:string" />
...     </xsd:sequence>
...   </xsd:complexType>
... </xsd:schema>
... ''')
>>> schema = etree.XMLSchema(file=f)
```

When creating the validating parser, we must make sure it [returns objectify trees](#). This is best done with the `makeparser()` function:

```
>>> parser = objectify.makeparser(schema = schema)
```

Now we can use it to parse a valid document:

```
>>> xml = "<a><b>test</b></a>"
>>> a = objectify.fromstring(xml, parser)

>>> print a.b
test
```

Or an invalid document:

```
>>> xml = "<a><b>test</b><c/></a>"
>>> a = objectify.fromstring(xml, parser)
Traceback (most recent call last):
XMLSyntaxError: Element 'c': This element is not expected.
```

Note that the same works for parse-time DTD validation, except that DTDs do not support any data types by design.

## 13.3 ObjectPath

For both convenience and speed, objectify supports its own path language, represented by the `ObjectPath` class:

```
>>> root = objectify.Element("{ns}root")
>>> b1 = etree.SubElement(root, "{ns}b")
>>> c = etree.SubElement(b1, "{ns}c")
>>> b2 = etree.SubElement(root, "{ns}b")
>>> d = etree.SubElement(root, "{other}d")

>>> path = objectify.ObjectPath("root.b.c")
>>> print path
root.b.c
```

```

>>> path.hasattr(root)
True
>>> print path.find(root).tag
{ns}c

>>> find = objectify.ObjectPath("root.b.c")
>>> print find(root).tag
{ns}c

>>> find = objectify.ObjectPath("root.{other}d")
>>> print find(root).tag
{other}d

>>> find = objectify.ObjectPath("root.{not}there")
>>> print find(root).tag
Traceback (most recent call last):
...
AttributeError: no such child: {not}there

>>> find = objectify.ObjectPath("{not}there")
>>> print find(root).tag
Traceback (most recent call last):
...
ValueError: root element does not match: need {not}there, got {ns}root

>>> find = objectify.ObjectPath("root.b[1]")
>>> print find(root).tag
{ns}b

>>> find = objectify.ObjectPath("root.{ns}b[1]")
>>> print find(root).tag
{ns}b

```

Apart from strings, ObjectPath also accepts lists of path segments:

```

>>> find = objectify.ObjectPath(['root', 'b', 'c'])
>>> print find(root).tag
{ns}c

>>> find = objectify.ObjectPath(['root', '{ns}b[1]'])
>>> print find(root).tag
{ns}b

```

You can also use relative paths starting with a `'.'` to ignore the actual root element and only inherit its namespace:

```

>>> find = objectify.ObjectPath(".b[1]")
>>> print find(root).tag
{ns}b

>>> find = objectify.ObjectPath(['.', 'b[1]'])
>>> print find(root).tag
{ns}b

>>> find = objectify.ObjectPath(".unknown[1]")
>>> print find(root).tag

```

```

Traceback (most recent call last):
...
AttributeError: no such child: {ns}unknown

>>> find = objectify.ObjectPath(".{other}unknown[1]")
>>> print find(root).tag
Traceback (most recent call last):
...
AttributeError: no such child: {other}unknown

```

For convenience, a single dot represents the empty ObjectPath (identity):

```

>>> find = objectify.ObjectPath(".")
>>> print find(root).tag
{ns}root

```

ObjectPath objects can be used to manipulate trees:

```

>>> root = objectify.Element("{ns}root")

>>> path = objectify.ObjectPath(".some.child.{other}unknown")
>>> path.hasattr(root)
False
>>> path.find(root)
Traceback (most recent call last):
...
AttributeError: no such child: {ns}some

>>> path.setattr(root, "my value") # creates children as necessary
>>> path.hasattr(root)
True
>>> print path.find(root).text
my value
>>> print root.some.child["{other}unknown"].text
my value

>>> print len( path.find(root) )
1
>>> path.addattr(root, "my new value")
>>> print len( path.find(root) )
2
>>> [ el.text for el in path.find(root) ]
['my value', 'my new value']

```

As with attribute assignment, `setattr()` accepts lists:

```

>>> path.setattr(root, ["v1", "v2", "v3"])
>>> [ el.text for el in path.find(root) ]
['v1', 'v2', 'v3']

```

Note, however, that indexing is only supported in this context if the children exist. Indexing of non existing children will not extend or create a list of such children but raise an exception:

```

>>> path = objectify.ObjectPath(".{non}existing[1]")
>>> path.setattr(root, "my value")
Traceback (most recent call last):
...

```

`TypeError: creating indexed path attributes is not supported`

It is worth noting that `ObjectPath` does not depend on the `objectify` module or the `ObjectifiedElement` implementation. It can also be used in combination with `Elements` from the normal `lxml.etree` API.

## 13.4 Python data types

The `objectify` module knows about Python data types and tries its best to let element content behave like them. For example, they support the normal math operators:

```
>>> root = objectify.fromstring(
...     "<root><a>5</a><b>11</b><c>true</c><d>hoi</d></root>")
>>> root.a + root.b
16
>>> root.a += root.b
>>> print root.a
16

>>> root.a = 2
>>> print root.a + 2
4
>>> print 1 + root.a
3

>>> print root.c
True
>>> root.c = False
>>> if not root.c:
...     print "false!"
false!

>>> print root.d + " test !"
hoi test !
>>> root.d = "%s - %s"
>>> print root.d % (1234, 12345)
1234 - 12345
```

However, data elements continue to provide the `objectify` API. This means that sequence operations such as `len()`, slicing and indexing (e.g. of strings) cannot behave as the Python types. Like all other tree elements, they show the normal slicing behaviour of `objectify` elements:

```
>>> root = objectify.fromstring("<root><a>test</a><b>toast</b></root>")
>>> print root.a + ' me' # behaves like a string, right?
test me
>>> len(root.a) # but there's only one 'a' element!
1
>>> [ a.tag for a in root.a ]
['a']
>>> print root.a[0].tag
a

>>> print root.a
test
>>> [ str(a) for a in root.a[:1] ]
```

```
['test']
```

If you need to run sequence operations on data types, you must ask the API for the *real* Python value. The string value is always available through the normal `ElementTree .text` attribute. Additionally, all data classes provide a `.pyval` attribute that returns the value as plain Python type:

```
>>> root = objectify.fromstring("<root><a>test</a><b>5</b></root>")
>>> root.a.text
'test'
>>> root.a.pyval
'test'

>>> root.b.text
'5'
>>> root.b.pyval
5
```

Note, however, that both attributes are read-only in `objectify`. If you want to change values, just assign them directly to the attribute:

```
>>> root.a.text = "25"
Traceback (most recent call last):
...
TypeError: attribute 'text' of 'StringElement' objects is not writable

>>> root.a.pyval = 25
Traceback (most recent call last):
...
TypeError: attribute 'pyval' of 'StringElement' objects is not writable

>>> root.a = 25
>>> print root.a
25
>>> print root.a.pyval
25
```

In other words, `objectify` data elements behave like immutable Python types. You can replace them, but not modify them.

### 13.4.1 Recursive tree dump

To see the data types that are currently used, you can call the module level `dump()` function that returns a recursive string representation for elements:

```
>>> root = objectify.fromstring("""
... <root xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
...   <a attr1="foo" attr2="bar">1</a>
...   <a>1.2</a>
...   <b>1</b>
...   <b>true</b>
...   <c>what?</c>
...   <d xsi:nil="true"/>
... </root>
... """)
```

```
>>> print objectify.dump(root)
root = None [ObjectifiedElement]
  a = 1 [IntElement]
    * attr1 = 'foo'
    * attr2 = 'bar'
  a = 1.2 [FloatElement]
  b = 1 [IntElement]
  b = True [BoolElement]
  c = 'what?' [StringElement]
  d = None [NoneElement]
    * xsi:nil = 'true'
```

You can freely switch between different types for the same child:

```
>>> root = objectify.fromstring("<root><a>5</a></root>")
>>> print objectify.dump(root)
root = None [ObjectifiedElement]
  a = 5 [IntElement]

>>> root.a = 'nice string!'
>>> print objectify.dump(root)
root = None [ObjectifiedElement]
  a = 'nice string!' [StringElement]
    * py:pytype = 'str'

>>> root.a = True
>>> print objectify.dump(root)
root = None [ObjectifiedElement]
  a = True [BoolElement]
    * py:pytype = 'bool'

>>> root.a = [1, 2, 3]
>>> print objectify.dump(root)
root = None [ObjectifiedElement]
  a = 1 [IntElement]
    * py:pytype = 'int'
  a = 2 [IntElement]
    * py:pytype = 'int'
  a = 3 [IntElement]
    * py:pytype = 'int'

>>> root.a = (1, 2, 3)
>>> print objectify.dump(root)
root = None [ObjectifiedElement]
  a = 1 [IntElement]
    * py:pytype = 'int'
  a = 2 [IntElement]
    * py:pytype = 'int'
  a = 3 [IntElement]
    * py:pytype = 'int'
```

### 13.4.2 Recursive string representation of elements

Normally, elements use the standard string representation for `str()` that is provided by `lxml.etree`. You can enable a pretty-print representation for objectify elements like this:

```
>>> objectify.enableRecursiveStr()

>>> root = objectify.fromstring("""
... <root xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
...   <a attr1="foo" attr2="bar">1</a>
...   <a>1.2</a>
...   <b>1</b>
...   <b>true</b>
...   <c>what?</c>
...   <d xsi:nil="true"/>
... </root>
... """)

>>> print str(root)
root = None [ObjectifiedElement]
  a = 1 [IntElement]
    * attr1 = 'foo'
    * attr2 = 'bar'
  a = 1.2 [FloatElement]
  b = 1 [IntElement]
  b = True [BoolElement]
  c = 'what?' [StringElement]
  d = None [NoneElement]
    * xsi:nil = 'true'
```

This behaviour can be switched off in the same way:

```
>>> objectify.enableRecursiveStr(False)
```

## 13.5 How data types are matched

Objectify uses two different types of Elements. Structural Elements (or tree Elements) represent the object tree structure. Data Elements represent the data containers at the leaves. You can explicitly create tree Elements with the `objectify.Element()` factory and data Elements with the `objectify.DataElement()` factory.

When Element objects are created, `lxml.objectify` must determine which implementation class to use for them. This is relatively easy for tree Elements and less so for data Elements. The algorithm is as follows:

1. If an element has children, use the default tree class.
2. If an element is defined as `xsi:nil`, use the `NoneElement` class.
3. If a “Python type hint” attribute is given, use this to determine the element class, see below.
4. If an XML Schema `xsi:type` hint is given, use this to determine the element class, see below.
5. Try to determine the element class from the text content type by trial and error.

6. If the element is a root node then use the default tree class.
7. Otherwise, use the default class for empty data classes.

You can change the default classes for tree Elements and empty data Elements at setup time. The `ObjectifyElementClassLookup()` call accepts two keyword arguments, `tree_class` and `empty_data_class`, that determine the Element classes used in these cases. By default, `tree_class` is a class called `ObjectifiedElement` and `empty_data_class` is a `StringElement`.

### 13.5.1 Type annotations

The “type hint” mechanism deploys an XML attribute defined as `lxml.objectify.PYTYPE_ATTRIBUTE`. It may contain any of the following string values: `int`, `long`, `float`, `str`, `unicode`, `NoneType`:

```
>>> print objectify.PYTYPE_ATTRIBUTE
{http://codespeak.net/lxml/objectify/pytype}pytype
>>> ns, name = objectify.PYTYPE_ATTRIBUTE[1:].split('}')

>>> root = objectify.fromstring("""\
... <root xmlns:py='%s'>
...   <a py:pytype='str'>5</a>
...   <b py:pytype='int'>5</b>
...   <c py:pytype='NoneType' />
... </root>
... """)

>>> print root.a + 10
510
>>> print root.b + 10
15
>>> print root.c
None
```

Note that you can change the name and namespace used for this attribute through the `set_pytype_attribute_tag(tag)` module function, in case your application ever needs to. There is also a utility function `annotate()` that recursively generates this attribute for the elements of a tree:

```
>>> root = objectify.fromstring("<root><a>test</a><b>5</b></root>")
>>> print objectify.dump(root)
root = None [ObjectifiedElement]
  a = 'test' [StringElement]
  b = 5 [IntElement]

>>> objectify.annotate(root)

>>> print objectify.dump(root)
root = None [ObjectifiedElement]
  a = 'test' [StringElement]
    * py:pytype = 'str'
  b = 5 [IntElement]
    * py:pytype = 'int'
```

### 13.5.2 XML Schema datatype annotation

A second way of specifying data type information uses XML Schema types as element annotations. Objectify knows those that can be mapped to normal Python types:

```
>>> root = objectify.fromstring('''\
...   <root xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
...       xmlns:xsd="http://www.w3.org/2001/XMLSchema">
...     <d xsi:type="xsd:double">5</d>
...     <l xsi:type="xsd:long" >5</l>
...     <s xsi:type="xsd:string">5</s>
...   </root>
...   ''')
>>> print objectify.dump(root)
root = None [ObjectifiedElement]
  d = 5.0 [FloatElement]
    * xsi:type = 'xsd:double'
  l = 5L [LongElement]
    * xsi:type = 'xsd:long'
  s = '5' [StringElement]
    * xsi:type = 'xsd:string'
```

Again, there is a utility function `xsiannotate()` that recursively generates the “xsi:type” attribute for the elements of a tree:

```
>>> root = objectify.fromstring('''\
...   <root><a>test</a><b>5</b><c>true</c></root>
...   ''')
>>> print objectify.dump(root)
root = None [ObjectifiedElement]
  a = 'test' [StringElement]
  b = 5 [IntElement]
  c = True [BoolElement]

>>> objectify.xsiannotate(root)

>>> print objectify.dump(root)
root = None [ObjectifiedElement]
  a = 'test' [StringElement]
    * xsi:type = 'xsd:string'
  b = 5 [IntElement]
    * xsi:type = 'xsd:int'
  c = True [BoolElement]
    * xsi:type = 'xsd:boolean'
```

Note, however, that `xsiannotate()` will always use the first XML Schema datatype that is defined for any given Python type, see also [Defining additional data classes](#).

The utility function `deannotate()` can be used to get rid of 'py:pytype' and/or 'xsi:type' information:

```
>>> root = objectify.fromstring('''\
...   <root xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
...       xmlns:xsd="http://www.w3.org/2001/XMLSchema">
...     <d xsi:type="xsd:double">5</d>
...     <l xsi:type="xsd:long" >5</l>
...     <s xsi:type="xsd:string">5</s>
...   </root>
...   ''')
```

```

... </root>'''
>>> objectify.annotate(root)
>>> print objectify.dump(root)
root = None [ObjectifiedElement]
  d = 5.0 [FloatElement]
    * xsi:type = 'xsd:double'
    * py:pytype = 'float'
  l = 5L [LongElement]
    * xsi:type = 'xsd:long'
    * py:pytype = 'long'
  s = '5' [StringElement]
    * xsi:type = 'xsd:string'
    * py:pytype = 'str'
>>> objectify.deannotate(root)
>>> print objectify.dump(root)
root = None [ObjectifiedElement]
  d = 5 [IntElement]
  l = 5 [IntElement]
  s = 5 [IntElement]

```

### 13.5.3 The DataElement factory

For convenience, the `DataElement()` factory creates an `Element` with a Python value in one step. You can pass the required Python type name or the XSI type name:

```

>>> root = objectify.Element("root")
>>> root.x = objectify.DataElement(5, _pytype="long")
>>> print objectify.dump(root)
root = None [ObjectifiedElement]
  x = 5L [LongElement]
    * py:pytype = 'long'

>>> root.x = objectify.DataElement(5, _pytype="str", myattr="someval")
>>> print objectify.dump(root)
root = None [ObjectifiedElement]
  x = '5' [StringElement]
    * py:pytype = 'str'
    * myattr = 'someval'

>>> root.x = objectify.DataElement(5, _xsi="integer")
>>> print objectify.dump(root)
root = None [ObjectifiedElement]
  x = 5L [LongElement]
    * py:pytype = 'long'
    * xsi:type = 'xsd:integer'

```

XML Schema types reside in the XML schema namespace thus `DataElement()` tries to correctly prefix the `xsi:type` attribute value for you:

```

>>> root = objectify.Element("root")
>>> root.s = objectify.DataElement(5, _xsi="string")

>>> objectify.deannotate(root, xsi=False)
>>> print etree.tostring(root, pretty_print=True)

```

```

<root xmlns:py="http://codespeak.net/lxml/objectify/pytype" xmlns:xsd="http://www.w3.org/2001/XML
instance">
  <s xsi:type="xsd:string">5</s>
</root>

```

DataElement() uses a default nsmmap to set these prefixes:

```

>>> el = objectify.DataElement('5', _xsi='string')
>>> for prefix, namespace in el.nsmmap.items():
...     print prefix, '-', namespace
py - http://codespeak.net/lxml/objectify/pytype
xsd - http://www.w3.org/2001/XMLSchema
xsi - http://www.w3.org/2001/XMLSchema-instance

>>> print el.get("{http://www.w3.org/2001/XMLSchema-instance}type")
xsd:string

```

While you can set custom namespace prefixes, it is necessary to provide valid namespace information if you choose to do so:

```

>>> el = objectify.DataElement('5', _xsi='foo:string',
...     nsmmap={'foo': 'http://www.w3.org/2001/XMLSchema'})
>>> for prefix, namespace in el.nsmmap.items():
...     print prefix, '-', namespace
py - http://codespeak.net/lxml/objectify/pytype
foo - http://www.w3.org/2001/XMLSchema
xsi - http://www.w3.org/2001/XMLSchema-instance

>>> print el.get("{http://www.w3.org/2001/XMLSchema-instance}type")
foo:string

```

Note how lxml chose a default prefix for the XML Schema Instance namespace. We can override it as in the following example:

```

>>> el = objectify.DataElement('5', _xsi='foo:string',
...     nsmmap={'foo': 'http://www.w3.org/2001/XMLSchema',
...             'myxsi': 'http://www.w3.org/2001/XMLSchema-instance'})
>>> for prefix, namespace in el.nsmmap.items():
...     print prefix, '-', namespace
py - http://codespeak.net/lxml/objectify/pytype
foo - http://www.w3.org/2001/XMLSchema
myxsi - http://www.w3.org/2001/XMLSchema-instance

>>> print el.get("{http://www.w3.org/2001/XMLSchema-instance}type")
foo:string

```

Care must be taken if different namespace prefixes have been used for the same namespace. Namespace information gets merged to avoid duplicate definitions when adding a new sub-element to a tree, but this mechanism does not adapt the prefixes of attribute values:

```

>>> root = objec-
tify.fromstring("<<root xmlns:schema="http://www.w3.org/2001/XMLSchema"/>""")
>>> print etree.tostring(root, pretty_print=True)
<root xmlns:schema="http://www.w3.org/2001/XMLSchema"/>

>>> s = objectify.DataElement("17", _xsi="string")
>>> print etree.tostring(s, pretty_print=True)

```

```

<value xmlns:py="http://codespeak.net/lxml/objectify/pytype" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
instance" py:pytype="str" xsi:type="xsd:string">17</value>

>>> root.s = s
>>> print etree.tostring(root, pretty_print=True)
<root xmlns:schema="http://www.w3.org/2001/XMLSchema">
  <s xmlns:py="http://codespeak.net/lxml/objectify/pytype" xmlns:xsi="http://www.w3.org/2001/XMLSchema"
instance" py:pytype="str" xsi:type="xsd:string">17</s>
</root>

```

It is your responsibility to fix the prefixes of attribute values if you choose to deviate from the standard prefixes. A convenient way to do this for `xsi:type` attributes is to use the `xsiannotate()` utility:

```

>>> objectify.xsiannotate(root)
>>> print etree.tostring(root, pretty_print=True)
<root xmlns:schema="http://www.w3.org/2001/XMLSchema">
  <s xmlns:py="http://codespeak.net/lxml/objectify/pytype" xmlns:xsi="http://www.w3.org/2001/XMLSchema"
instance" py:pytype="str" xsi:type="schema:string">17</s>
</root>

```

Of course, it is discouraged to use different prefixes for one and the same namespace when building up an objectify tree.

### 13.5.4 Defining additional data classes

You can plug additional data classes into objectify that will be used in exactly the same way as the predefined types. Data classes can either inherit from `ObjectifiedDataElement` directly or from one of the specialised classes like `NumberElement` or `BoolElement`. The numeric types require an initial call to the `NumberElement` method `self._setValueParser(function)` to set their type conversion function (string -> numeric Python type). This call should be placed into the element `_init()` method.

The registration of data classes uses the `PyType` class:

```

>>> class ChristmasDate(objectify.ObjectifiedDataElement):
...     def call_santa(self):
...         print "Ho ho ho!"

>>> def checkChristmasDate(date_string):
...     if not date_string.startswith('24.12.'):
...         raise ValueError # or TypeError

>>> xmas_type = objectify.PyType('date', checkChristmasDate, ChristmasDate)

```

The `PyType` constructor takes a string type name, an (optional) callable type check and the custom data class. If a type check is provided it must accept a string as argument and raise `ValueError` or `TypeError` if it cannot handle the string value.

`PyTypes` are used if an element carries a `py:pytype` attribute denoting its data type or, in absence of such an attribute, if the given type check callable does not raise a `ValueError/TypeError` exception when applied to the element text.

If you want, you can also register this type under an XML Schema type name:

```

>>> xmas_type.xmlSchemaTypes = ("date",)

```

XML Schema types will be considered if the element has an `xsi:type` attribute that specifies its data

type. The line above binds the XSD type `date` to the newly defined Python type. Note that this must be done before the next step, which is to register the type. Then you can use it:

```
>>> xmas_type.register()

>>> root = objectify.fromstring(
...     "<root><a>24.12.2000</a><b>12.24.2000</b></root>")
>>> root.a.call_santa()
Ho ho ho!
>>> root.b.call_santa()
Traceback (most recent call last):
...
AttributeError: no such child: call_santa
```

If you need to specify dependencies between the type check functions, you can pass a sequence of type names through the `before` and `after` keyword arguments of the `register()` method. The PyType will then try to register itself before or after the respective types, as long as they are currently registered. Note that this only impacts the currently registered types at the time of registration. Types that are registered later on will not care about the dependencies of already registered types.

If you provide XML Schema type information, this will override the type check function defined above:

```
>>> root = objectify.fromstring('''\
...     <root xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
...         <a xsi:type="date">12.24.2000</a>
...     </root>
...     ''')
>>> print root.a
12.24.2000
>>> root.a.call_santa()
Ho ho ho!
```

To unregister a type, call its `unregister()` method:

```
>>> root.a.call_santa()
Ho ho ho!
>>> xmas_type.unregister()
>>> root.a.call_santa()
Traceback (most recent call last):
...
AttributeError: no such child: call_santa
```

Be aware, though, that this does not immediately apply to elements to which there already is a Python reference. Their Python class will only be changed after all references are gone and the Python object is garbage collected.

### 13.5.5 Advanced element class lookup

In some cases, the normal data class setup is not enough. Being based on `lxml.etree`, however, `lxml.objectify` supports very fine-grained control over the Element classes used in a tree. All you have to do is configure a different [class lookup](#) mechanism (or write one yourself).

The first step for the setup is to create a new parser that builds objectify documents. The objectify API is meant for data-centric XML (as opposed to document XML with mixed content). Therefore, we configure the parser to let it remove whitespace-only text from the parsed document if it is not enclosed by an XML element. Note that this alters the document infoset, so if you consider the removed spaces as

data in your specific use case, you should go with a normal parser and just set the element class lookup. Most applications, however, will work fine with the following setup:

```
>>> parser = objectify.makeparser(remove_blank_text=True)
```

What this does internally, is:

```
>>> parser = etree.XMLParser(remove_blank_text=True)

>>> lookup = objectify.ObjectifyElementClassLookup()
>>> parser.set_element_class_lookup(lookup)
```

If you want to change the lookup scheme, say, to get additional support for [namespace specific classes](#), you can register the objectify lookup as a fallback of the namespace lookup. In this case, however, you have to take care that the namespace classes inherit from `objectify.ObjectifiedElement`, not only from the normal `lxml.etree.ElementBase`, so that they support the objectify API. The above setup code then becomes:

```
>>> lookup = etree.ElementNamespaceClassLookup(
...     objectify.ObjectifyElementClassLookup() )
>>> parser.set_element_class_lookup(lookup)
```

See the documentation on [class lookup](#) schemes for more information.

## 13.6 What is different from lxml.etree?

Such a different Element API obviously implies some side effects to the normal behaviour of the rest of the API.

- `len(<element>)` returns the sibling count, not the number of children of `<element>`. You can retrieve the number of children with the `countchildren()` method.
- Iteration over elements does not yield the children, but the siblings. You can access all children with the `iterchildren()` method on elements or retrieve a list by calling the `getchildren()` method.
- The `find`, `findall` and `findtext` methods require a different implementation based on `ETXPath`. In `lxml.etree`, they use a Python implementation based on the original iteration scheme. This has the disadvantage that they may not be 100% backwards compatible, and the additional advantage that they now support any XPath expression.

# Chapter 14

## lxml.html

Since version 2.0, lxml comes with a dedicated package for dealing with HTML: `lxml.html`. It provides a special Element API for HTML elements, as well as a number of utilities for common tasks.

The main API is based on the [lxml.etree](#) API, and thus, on the [ElementTree](#) API.

### 14.1 Parsing HTML

#### 14.1.1 Parsing HTML fragments

There are several functions available to parse HTML:

`parse(filename_url_or_file)`: Parses the named file or url, or if the object has a `.read()` method, parses from that.

If you give a URL, or if the object has a `.geturl()` method (as file-like objects from `urllib.urlopen()` have), then that URL is used as the base URL. You can also provide an explicit `base_url` keyword argument.

`document_fromstring(string)`: Parses a document from the given string. This always creates a correct HTML document, which means the parent node is `<html>`, and there is a body and possibly a head.

`fragment_fromstring(string, create_parent=False)`: Returns an HTML fragment from a string. The fragment must contain just a single element, unless `create_parent` is given; e.g., `fragment_fromstring(string, create_parent='div')` will wrap the element in a `<div>`.

`fragments_fromstring(string)`: Returns a list of the elements found in the fragment.

`fromstring(string)`: Returns `document_fromstring` or `fragment_fromstring`, based on whether the string looks like a full document, or just a fragment.

#### 14.1.2 Really broken pages

The normal HTML parser is capable of handling broken HTML, but for pages that are far enough from HTML to call them 'tag soup', it may still fail to parse the page. A way to deal with this is [ElementSoup](#), which deploys the well-known [BeautifulSoup](#) parser to build an lxml HTML tree.

## 14.2 HTML Element Methods

HTML elements have all the methods that come with `ElementTree`, but also include some extra methods:

- `.drop_tree()`: Drops the element and all its children. Unlike `el.getparent().remove(el)` this does *not* remove the tail text; with `drop_tree` the tail text is merged with the previous element.
- `.drop_tag()`: Drops the tag, but keeps its children and text.
- `.find_class(class_name)`: Returns a list of all the elements with the given CSS class name. Note that class names are space separated in HTML, so `doc.find_class_name('highlight')` will find an element like `<div class="sidebar highlight">`. Class names *are* case sensitive.
- `.find_rel_links(rel)`: Returns a list of all the `<a rel="{rel}">` elements. E.g., `doc.find_rel_links('tag')` returns all the links [marked as tags](#).
- `.get_element_by_id(id, default=None)`: Return the element with the given `id`, or the `default` if none is found. If there are multiple elements with the same `id` (which there shouldn't be, but there often is), this returns only the first.
- `.text_content()`: Returns the text content of the element, including the text content of its children, with no markup.
- `.cssselect(expr)`: Select elements from this element and its children, using a CSS selector expression. (Note that `.xpath(expr)` is also available as on all `lxml` elements.)
- `.label`: Returns the corresponding `<label>` element for this element, if any exists (None if there is none). Label elements have a `label.for_element` attribute that points back to the element.
- `.base_url`: The base URL for this element, if one was saved from the parsing. This attribute is not settable. Is None when no base URL was saved.

## 14.3 Running HTML doctests

One of the interesting modules in the `lxml.html` package deals with doctests. It can be hard to compare two HTML pages for equality, as whitespace differences aren't meaningful and the structural formatting can differ. This is even more a problem in doctests, where output is tested for equality and small differences in whitespace or the order of attributes can let a test fail. And given the verbosity of tag-based languages, it may take more than a quick look to find the actual differences in the doctest output.

Luckily, `lxml` provides the `lxml.doctestcompare` module that supports relaxed comparison of XML and HTML pages and provides a readable diff in the output when a test fails. The HTML comparison is most easily used by importing the `usedoctest` module in a doctest:

```
>>> import lxml.html.usedoctest
```

Now, if you have a HTML document and want to compare it to an expected result document in a doctest, you can do the following:

```
>>> import lxml.html
>>> html = lxml.html.fromstring('''\
...     <html><body onload="" color="white">
...         <p>Hi !</p>
...     </body></html>
... ''')
```

```

>>> print lxml.html.tostring(html)
<html><body onload="" color="white"><p>Hi !</p></body></html>

>>> print lxml.html.tostring(html)
<html> <body color="white" onload=""> <p>Hi    !</p> </body> </html>

>>> print lxml.html.tostring(html)
<html>
  <body color="white" onload="">
    <p>Hi !</p>
  </body>
</html>

```

In documentation, you would likely prefer the pretty printed HTML output, as it is the most readable. However, the three documents are equivalent from the point of view of an HTML tool, so the doctest will silently accept any of the above. This allows you to concentrate on readability in your doctests, even if the real output is a straight ugly HTML one-liner.

Note that there is also an `lxml.usedoctest` module which you can import for XML comparisons. The HTML parser notably ignores namespaces and some other XMLisms.

## 14.4 Creating HTML with the E-factory

`lxml.html` comes with a predefined HTML vocabulary for the [E-factory](#), originally written by Fredrik Lundh. This allows you to quickly generate HTML pages and fragments:

```

>>> from lxml.html import builder as E
>>> from lxml.html import usedoctest
>>> html = E.HTML(
...   E.HEAD(
...     E.LINK(rel="stylesheet", href="great.css", type="text/css"),
...     E.TITLE("Best Page Ever")
...   ),
...   E.BODY(
...     E.H1(E.CLASS("heading"), "Top News"),
...     E.P("World News only on this page", style="font-size: 200%"),
...     "Ah, and here's some more text, by the way.",
...     lxml.html.fromstring("<p>... and this is a parsed fragment ...</p>")
...   )
... )

>>> print lxml.html.tostring(html)
<html>
  <head>
    <link href="great.css" rel="stylesheet" type="text/css">
    <title>Best Page Ever</title>
  </head>
  <body>
    <h1 class="heading">Top News</h1>
    <p style="font-size: 200%">World News only on this page</p>
    Ah, and here's some more text, by the way.
    <p>... and this is a parsed fragment ...</p>
  </body>

```

```
</html>
```

Note that you should use `lxml.html.tostring` and **not** `lxml.tostring`. `lxml.tostring(doc)` will return the XML representation of the document, which is not valid HTML. In particular, things like `<script src="..."></script>` will be serialized as `<script src="..." />`, which completely confuses browsers.

### 14.4.1 Viewing your HTML

A handy method for viewing your HTML: `lxml.html.open_in_browser(lxml_doc)` will write the document to disk and open it in a browser (with the [webbrowser module](#)).

## 14.5 Working with links

There are several methods on elements that allow you to see and modify the links in a document.

`.iterlinks()`: This yields (`element`, `attribute`, `link`, `pos`) for every link in the document. `attribute` may be `None` if the link is in the text (as will be the case with a `<style>` tag with `@import`).

This finds any link in an `action`, `archive`, `background`, `cite`, `classid`, `codebase`, `data`, `href`, `longdesc`, `profile`, `src`, `usemap`, `dynsrc`, or `lowsrc` attribute. It also searches `style` attributes for `url(link)`, and `<style>` tags for `@import` and `url()`.

This function does *not* pay attention to `<base href>`.

`.resolve_base_href()`: This function will modify the document in-place to take account of `<base href>` if the document contains that tag. In the process it will also remove that tag from the document.

`.make_links_absolute(base_href, resolve_base_href=True)`: This makes all links in the document absolute, assuming that `base_href` is the URL of the document. So if you pass `base_href="http://localhost/foo"` and there is a link to `baz.html` that will be rewritten as `http://localhost/foo/baz.html`.

If `resolve_base_href` is true, then any `<base href>` tag will be taken into account (just calling `self.resolve_base_href()`).

`.rewrite_links(link_repl_func, resolve_base_href=True, base_href=None)`: This rewrites all the links in the document using your given link replacement function. If you give a `base_href` value, all links will be passed in after they are joined with this URL.

For each link `link_repl_func(link)` is called. That function then returns the new link, or `None` to remove the attribute or tag that contains the link. Note that all links will be passed in, including links like `"#anchor"` (which is purely internal), and things like `"mailto:bob@example.com"` (or `javascript:...`).

If you want access to the context of the link, you should use `.iterlinks()` instead.

### 14.5.1 Functions

In addition to these methods, there are corresponding functions:

- `iterlinks(html)`

- `make_links_absolute(html, base_href, ...)`
- `rewrite_links(html, link_repl_func, ...)`
- `resolve_base_href(html)`

These functions will parse `html` if it is a string, then return the new HTML as a string. If you pass in a document, the document will be copied (except for `iterlinks()`), the method performed, and the new document returned.

## 14.6 Forms

Any `<form>` elements in a document are available through the list `doc.forms` (e.g., `doc.forms[0]`). Form, input, select, and textarea elements each have special methods.

Input elements (including `<select>` and `<textarea>`) have these attributes:

**.name:** The name of the element.

**.value:** The value of an input, the content of a textarea, the selected option(s) of a select. This attribute can be set.

In the case of a select that takes multiple options (`<select multiple>`) this will be a set of the selected options; you can add or remove items to select and unselect the options.

Select attributes:

**.value\_options:** For select elements, this is all the *possible* values (the values of all the options).

**.multiple:** For select elements, true if this is a `<select multiple>` element.

Input attributes:

**.type:** The type attribute in `<input>` elements.

**.checkable:** True if this can be checked (i.e., true for `type=radio` and `type=checkbox`).

**.checked:** If this element is checkable, the checked state. Raises `AttributeError` on non-checkable inputs.

The form itself has these attributes:

**.inputs:** A dictionary-like object that can be used to access input elements by name. When there are multiple input elements with the same name, this returns list-like structures that can also be used to access the options and their values as a group.

**.fields:** A dictionary-like object used to access *values* by their name. `form.inputs` returns elements, this only returns values. Setting values in this dictionary will effect the form inputs. Basically `form.fields[x]` is equivalent to `form.inputs[x].value` and `form.fields[x] = y` is equivalent to `form.inputs[x].value = y`. (Note that sometimes `form.inputs[x]` returns a compound object, but these objects also have `.value` attributes.)

If you set this attribute, it is equivalent to `form.fields.clear(); form.fields.update(new_value)`

**.form\_values():** Returns a list of `[(name, value), ...]`, suitable to be passed to `urllib.urlencode()` for form submission.

**.action:** The `action` attribute. This is resolved to an absolute URL if possible.

**.method:** The `method` attribute, which defaults to `GET`.

### 14.6.1 Form Filling Example

Note that you can change any of these attributes (values, method, action, etc) and then serialize the form to see the updated values. You can, for instance, do:

```
>>> from lxml.html import fromstring, tostring
>>> form_page = fromstring('''<html><body><form>
...   Your name: <input type="text" name="name"> <br>
...   Your phone: <input type="text" name="phone"> <br>
...   Your favorite pets: <br>
...   Dogs: <input type="checkbox" name="interest" value="dogs"> <br>
...   Cats: <input type="checkbox" name="interest" value="cats"> <br>
...   Llamas: <input type="checkbox" name="interest" value="llamas"> <br>
...   <input type="submit"></form></body></html>''')
>>> form = form_page.forms[0]
>>> form.fields = dict(
...     name='John Smith',
...     phone='555-555-3949',
...     interest=set(['cats', 'llamas']))
>>> print tostring(form)
<html>
  <body>
    <form>
      Your name:
        <input name="name" type="text" value="John Smith">
      <br>Your phone:
        <input name="phone" type="text" value="555-555-3949">
      <br>Your favorite pets:
      <br>Dogs:
        <input name="interest" type="checkbox" value="dogs">
      <br>Cats:
        <input checked name="interest" type="checkbox" value="cats">
      <br>Llamas:
        <input checked name="interest" type="checkbox" value="llamas">
      <br>
        <input type="submit">
    </form>
  </body>
</html>
```

### 14.6.2 Form Submission

You can submit a form with `lxml.html.submit_form(form_element)`. This will return a file-like object (the result of `urllib.urlopen()`).

If you have extra input values you want to pass you can use the keyword argument `extra_values`, like `extra_values={'submit': 'Yes!'}`. This is the only way to get submit values into the form, as there is no state of “submitted” for these elements.

You can pass in an alternate opener with the `open_http` keyword argument, which is a function with the signature `open_http(method, url, values)`.

Example:

```
>>> from lxml.html import parse, submit_form
>>> page = parse('http://tinyurl.com').getroot()
>>> page.forms[1].fields['url'] = 'http://codespeak.net/lxml/'
>>> result = parse(submit_form(page.forms[1])).getroot()
>>> [a.attrib['href'] for a in result.xpath("//a[@target='_blank']")]
['http://tinyurl.com/2xae8s', 'http://preview.tinyurl.com/2xae8s']
```

## 14.7 Cleaning up HTML

The module `lxml.html.clean` provides a `Cleaner` class for cleaning up HTML pages. It supports removing embedded or script content, special tags, CSS style annotations and much more.

Say, you have an evil web page from an untrusted source that contains lots of content that upsets browsers and tries to run evil code on the client side:

```
>>> html = '''\
... <html>
... <head>
... <script type="text/javascript" src="evil-site"></script>
... <link rel="alternate" type="text/rss" src="evil-rss">
... <style>
...   body {background-image: url(javascript:do_evil)};
...   div {color: expression(evil)};
... </style>
... </head>
... <body onload="evil_function()">
... <!-- I am interpreted for EVIL! -->
... <a href="javascript:evil_function()">a link</a>
... <a href="#" onclick="evil_function()">another link</a>
... <p onclick="evil_function()">a paragraph</p>
... <div style="display: none">secret EVIL!</div>
... <object> of EVIL! </object>
... <iframe src="evil-site"></iframe>
... <form action="evil-site">
...   Password: <input type="password" name="password">
... </form>
... <blink>annoying EVIL!</blink>
... <a href="evil-site">spam spam SPAM!</a>
... <image src="evil!">
... </body>
... </html>'''
```

To remove the all suspicious content from this unparsed document, use the `clean_html` function:

```
>>> from lxml.html.clean import clean_html

>>> print clean_html(html)
<html>
  <body>
    <div>
      <style>/* deleted */</style>
      <a href="">a link</a>
      <a href="#">another link</a>
      <p>a paragraph</p>
```

```

    <div>secret EVIL!</div>
    of EVIL!
    Password:
    annoying EVIL!
    <a href="evil-site">spam spam SPAM!</a>
    
  </div>
</body>
</html>

```

The Cleaner class supports several keyword arguments to control exactly which content is removed:

```

>>> from lxml.html.clean import Cleaner

>>> cleaner = Cleaner(page_structure=False, links=False)
>>> print cleaner.clean_html(html)
<html>
  <head>
    <link rel="alternate" src="evil-rss" type="text/rss">
    <style>/* deleted */</style>
  </head>
  <body>
    <a href="">a link</a>
    <a href="#">another link</a>
    <p>a paragraph</p>
    <div>secret EVIL!</div>
    of EVIL!
    Password:
    annoying EVIL!
    <a href="evil-site">spam spam SPAM!</a>
    
  </body>
</html>

>>> cleaner = Cleaner(style=True, links=True, add_nofollow=True,
...                    page_structure=False, safe_attrs_only=False)

>>> print cleaner.clean_html(html)
<html>
  <head>
  </head>
  <body>
    <a href="">a link</a>
    <a href="#">another link</a>
    <p>a paragraph</p>
    <div>secret EVIL!</div>
    of EVIL!
    Password:
    annoying EVIL!
    <a href="evil-site" rel="nofollow">spam spam SPAM!</a>
    
  </body>
</html>

```

You can also whitelist some otherwise dangerous content with `Cleaner(host_whitelist=['www.youtube.com'])`,

which would allow embedded media from YouTube, while still filtering out embedded media from other sites.

See the docstring of `Cleaner` for the details of what can be cleaned.

### 14.7.1 autolink

In addition to cleaning up malicious HTML, `lxml.html.clean` contains functions to do other things to your HTML. This includes autolinking:

```
autolink(doc, ...)

autolink_html(html, ...)
```

This finds anything that looks like a link (e.g., `http://example.com`) in the *text* of an HTML document, and turns it into an anchor. It avoids making bad links.

Links in the elements `<textarea>`, `<pre>`, `<code>`, anything in the head of the document. You can pass in a list of elements to avoid in `avoid_elements=['textarea', ...]`.

Links to some hosts can be avoided. By default links to `localhost*`, `example.*` and `127.0.0.1` are not autolinked. Pass in `avoid_hosts=[list_of_regexes]` to control this.

Elements with the `nolink` CSS class are not autolinked. Pass in `avoid_classes=['code', ...]` to control this.

The `autolink_html()` version of the function parses the HTML string first, and returns a string.

### 14.7.2 wordwrap

You can also wrap long words in your html:

```
word_break(doc, max_width=40, ...)

word_break_html(html, ...)
```

This finds any long words in the text of the document and inserts `&#8203;` in the document (which is the Unicode zero-width space).

This avoids the elements `<pre>`, `<textarea>`, and `<code>`. You can control this with `avoid_elements=['textarea', ...]`.

It also avoids elements with the CSS class `nobreak`. You can control this with `avoid_classes=['code', ...]`.

Lastly you can control the character that is inserted with `break_character=u'\u200b'`. However, you cannot insert markup, only text.

`word_break_html(html)` parses the HTML document and returns a string.

## 14.8 HTML Diff

The module `lxml.html.diff` offers some ways to visualize differences in HTML documents. These differences are *content* oriented. That is, changes in markup are largely ignored; only changes in the

content itself are highlighted.

There are two ways to view differences: `htmldiff` and `html_annotate`. One shows differences with `<ins>` and `<del>`, while the other annotates a set of changes similar to `svn blame`. Both these functions operate on text, and work best with content fragments (only what goes in `<body>`), not complete documents.

Example of `htmldiff`:

```
>>> from lxml.html.diff import htmldiff, html_annotate
>>> doc1 = '''<p>Here is some text.</p>'''
>>> doc2 = '''<p>Here is <b>a lot</b> of <i>text</i>.</p>'''
>>> doc3 = '''<p>Here is <b>a little</b> <i>text</i>.</p>'''
>>> print htmldiff(doc1, doc2)
<p>Here is <ins><b>a lot</b> of <i>text</i>.</ins> <del>some text.</del> </p>
>>> print html_annotate([(doc1, 'author1'), (doc2, 'author2'),
...                      (doc3, 'author3')])
<p><span title="author1">Here is</span>
  <b><span title="author2">a</span>
  <span title="author3">little</span></b>
  <i><span title="author2">text</span></i>
  <span title="author2">.</span></p>
```

As you can see, it is imperfect as such things tend to be. On larger tracts of text with larger edits it will generally do better.

The `html_annotate` function can also take an optional second argument, `markup`. This is a function like `markup(text, version)` that returns the given text marked up with the given version. The default version, the output of which you see in the example, looks like:

```
def default_markup(text, version):
    return '<span title="%s">%s</span>' % (
        cgi.escape(unicode(version), 1), text)
```

## 14.9 Examples

### 14.9.1 Microformat Example

This example parses the [hCard](#) microformat.

First we get the page:

```
>>> import urllib
>>> from lxml.html import fromstring
>>> url = 'http://microformats.org/'
>>> content = urllib.urlopen(url).read()
>>> doc = fromstring(content)
>>> doc.make_links_absolute(url)
```

Then we create some objects to put the information in:

```
>>> class Card(object):
...     def __init__(self, **kw):
...         for name, value in kw:
...             setattr(self, name, value)
>>> class Phone(object):
```

```
...     def __init__(self, phone, types=()):
...         self.phone, self.types = phone, types
```

And some generally handy functions for microformats:

```
>>> def get_text(el, class_name):
...     els = el.find_class(class_name)
...     if els:
...         return els[0].text_content()
...     else:
...         return ''
>>> def get_value(el):
...     return get_text(el, 'value') or el.text_content()
>>> def get_all_texts(el, class_name):
...     return [e.text_content() for e in els.find_class(class_name)]
>>> def parse_addresses(el):
...     # Ideally this would parse street, etc.
...     return el.find_class('adr')
```

Then the parsing:

```
>>> for el in doc.find_class('hcard'):
...     card = Card()
...     card.el = el
...     card.fn = get_text(el, 'fn')
...     card.tels = []
...     for tel_el in card.find_class('tel'):
...         card.tels.append(Phone(get_value(tel_el),
...                                 get_all_texts(tel_el, 'type')))
...     card.addresses = parse_addresses(el)
```

## Chapter 15

# lxml.cssselect

lxml supports a number of interesting languages for tree traversal and element selection. The most important is obviously [XPath](#), but there is also [ObjectPath](#) in the `lxml.objectify` module. The newest child of this family is [CSS selection](#), which is implemented in the new `lxml.cssselect` module.

### 15.1 The CSSSelector class

The most important class in the `cssselect` module is `CSSSelector`. It provides the same interface as the [XPath](#) class, but accepts a CSS selector expression as input:

```
>>> from lxml.cssselect import CSSSelector
>>> sel = CSSSelector('div.content')
>>> sel #doctest: +ELLIPSIS
<CSSSelector ... for 'div.content'>
>>> sel.css
'div.content'
```

The selector actually compiles to XPath, and you can see the expression by inspecting the object:

```
>>> sel.path
"descendant-or-self::div[contains(concat(' ', normalize-space(@class), ' '), ' content ')]"
```

To use the selector, simply call it with a document or element object:

```
>>> from lxml.etree import fromstring
>>> h = fromstring('''<div id="outer">
...   <div id="inner" class="content body">
...     text
...   </div></div>''')
>>> [e.get('id') for e in sel(h)]
['inner']
```

### 15.2 CSS Selectors

This libraries attempts to implement CSS selectors as described in the [w3c specification](#). Many of the pseudo-classes do not apply in this context, including all [dynamic pseudo-classes](#). In particular these will

not be available:

- link state: `:link`, `:visited`, `:target`
- actions: `:hover`, `:active`, `:focus`
- UI states: `:enabled`, `:disabled`, `:indeterminate` (`:checked` and `:unchecked` *are* available)

Also, none of the psuedo-elements apply, because the selector only returns elements and psuedo-elements select portions of text, like `::first-line`.

### 15.2.1 Namespaces

In CSS you can use `namespace-prefix|element`, similar to `namespace-prefix:element` in an XPath expression. In fact, it maps one-to-one, and the same rules are used to map namespace prefixes to namespace URIs.

## 15.3 Limitations

These applicable pseudoclasses are not yet implemented:

- `:lang(language)`
- `:root`
- `*:first-of-type`, `*:last-of-type`, `*:nth-of-type`, `*:nth-last-of-type`, `*:only-of-type`. All of these work when you specify an element type, but not with `*`

Unlike XPath you cannot provide parameters in your expressions -- all expressions are completely static.

XPath has underspecified string quoting rules (there seems to be no string quoting at all), so if you use expressions that contain characters that requiring quoting you might have problems with the translation from CSS to XPath.

## Chapter 16

# BeautifulSoup Parser

`BeautifulSoup` is a Python package that parses broken HTML. While `libxml2` (and thus `lxml`) can also parse broken HTML, `BeautifulSoup` is a bit more forgiving and has superior support for encoding detection.

`lxml` can benefit from the parsing capabilities of `BeautifulSoup` through the `lxml.html.soupparser` module. It provides three main functions: `fromstring()` and `parse()` to parse a string or file using `BeautifulSoup`, and `convert_tree()` to convert an existing `BeautifulSoup` tree into a list of top-level Elements.

The functions `fromstring()` and `parse()` behave as known from `ElementTree`. The first returns a root Element, the latter returns an `ElementTree`.

There is also a legacy module called `lxml.html.ElementSoup`, which mimics the interface provided by `ElementTree`'s own `ElementSoup` module. Note that the `soupparser` module was added in `lxml 2.0.3`. Previous versions of `lxml 2.0.x` only have the `ElementSoup` module.

Here is a document full of tag soup, similar to, but not quite like, HTML:

```
>>> tag_soup = '<meta><head><title>Hello</head><body onload=crash()>Hi all<p>'
```

all you need to do is pass it to the `fromstring()` function:

```
>>> from lxml.html.soupparser import fromstring
>>> root = fromstring(tag_soup)
```

To see what we have here, you can serialise it:

```
>>> from lxml.etree import tostring
>>> print tostring(root, pretty_print=True),
<html>
  <meta/>
  <head>
    <title>Hello</title>
  </head>
  <body onload="crash()">Hi all<p/></body>
</html>
```

Not quite what you'd expect from an HTML page, but, well, it was broken already, right? `BeautifulSoup` did its best, and so now it's a tree.

To control which Element implementation is used, you can pass a `makeelement` factory function to

`parse()` and `fromstring()`. By default, this is based on the HTML parser defined in `lxml.html`.

## 16.1 Entity handling

By default, the BeautifulSoup parser also replaces the entities it finds by their character equivalent:

```
>>> tag_soup = '<body>&copy;&euro;&#45;&#245;&#445;<p>'
>>> body = fromstring(tag_soup).find('..//body')
>>> body.text
u'\xa9\u20ac-\xf5\u01bd'
```

If you want them back on the way out, you can serialise with the `'html'` method, which will always use escaping for safety reasons:

```
>>> tostring(body, method="html")
'<body>&#xA9;&#x20AC;-&#xF5;&#x1BD;<p></p></body>'

>>> tostring(body, method="html", encoding="utf-8")
'<body>&#xA9;&#x20AC;-&#xF5;&#x1BD;<p></p></body>'

>>> tostring(body, method="html", encoding=unicode)
u'<body>&#xA9;&#x20AC;-&#xF5;&#x1BD;<p></p></body>'
```

Otherwise, when serialising to XML, only the plain ASCII encoding will escape non-ASCII characters:

```
>>> tostring(body)
'<body>&#169;&#8364;-&#245;&#445;<p/></body>'

>>> tostring(body, encoding="utf-8")
'<body>\xc2\xa9\xe2\x82\xac-\xc3\xb5\xc6\xbd<p/></body>'

>>> tostring(body, encoding=unicode)
u'<body>\xa9\u20ac-\xf5\u01bd<p/></body>'
```

There is also a legacy module called `lxml.html.ElementSoup`, which mimics the interface provided by `ElementTree`'s own `ElementSoup` module.

## 16.2 Using soupparser as a fallback

The downside of using this parser is that it is **much slower** than the HTML parser of `lxml`. So if performance matters, you might want to consider using `soupparser` only as a fallback for certain cases.

One common problem of `lxml`'s parser is that it might not get the encoding right in cases where the document contains a `<meta>` tag at the wrong place. In this case, you can exploit the fact that `lxml` serialises much faster than most other HTML libraries for Python. Just serialise the document to unicode and if that gives you an exception, re-parse it with BeautifulSoup to see if that works better:

```
>>> tag_soup = '''\
... <meta http-equiv="Content-Type"
...     content="text/html; charset=utf-8" />
... <html>
...   <head>
...     <title>Hello W\xc3\xb6rld!</title>
```

```
... </head>
... <body>Hi all</body>
... </html>'''

>>> import lxml.html
>>> import lxml.html.soupparser

>>> root = lxml.html.fromstring(tag_soup)
>>> try:
...     ignore = tostring(root, encoding=unicode)
... except UnicodeDecodeError:
...     root = lxml.html.soupparser.fromstring(tag_soup)
```

## Part III

# Extending lxml

## Chapter 17

# Document loading and URL resolving

Lxml has support for custom document loaders in both the parsers and XSL transformations. These so-called resolvers are subclasses of the `etree.Resolver` class.

### 17.1 Resolvers

Here is an example of a custom resolver:

```
>>> from lxml import etree

>>> class DTDResolver(etree.Resolver):
...     def resolve(self, url, id, context):
...         print "Resolving URL '%s'" % url
...         return self.resolve_string(
...             '<!ENTITY myentity "[resolved text: %s]">' % url, context)
```

This defines a resolver that always returns a dynamically generated DTD fragment defining an entity. The `url` argument passes the system URL of the requested document, the `id` argument is the public ID. Note that any of these may be `None`. The context object is not normally used by client code.

Resolving is based on three methods of the `Resolver` object that build internal representations of the result document. The following methods exist:

- `resolve_string` takes a parsable string as result document
- `resolve_filename` takes a filename
- `resolve_file` takes an open file-like object that has at least a `read()` method
- `resolve_empty` resolves into an empty document

The `resolve()` method may choose to return `None`, in which case the next registered resolver (or the default resolver) is consulted. Resolving always terminates if `resolve()` returns the result of any of the above `resolve_*` methods.

Resolvers are registered local to a parser:

```
>>> parser = etree.XMLParser(load_dtd=True)
>>> parser.resolvers.add( DTDResolver() )
```

Note that we instantiate a parser that loads the DTD. This is not done by the default parser, which does no validation. When we use this parser to parse a document that requires resolving a URL, it will call our custom resolver:

```
>>> xml = u'<!DOCTYPE doc SYSTEM "MissingDTD.dtd"><doc>&myentity;</doc>'
>>> from StringIO import StringIO
>>> tree = etree.parse(StringIO(xml), parser)
Resolving URL 'MissingDTD.dtd'
>>> root = tree.getroot()
>>> print root.text
[resolved text: MissingDTD.dtd]
```

The entity in the document was correctly resolved by the generated DTD fragment.

## 17.2 Document loading in context

XML documents memorise their initial parser (and its resolvers) during their life-time. This means that a lookup process related to a document will use the resolvers of the document's parser. We can demonstrate this with a resolver that only responds to a specific prefix:

```
>>> class PrefixResolver(etree.Resolver):
...     def __init__(self, prefix):
...         self.prefix = prefix
...         self.result_xml = '''\
...             <xsl:stylesheet
...                 xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
...                 <test xmlns="testNS">%s-TEST</test>
...                 </xsl:stylesheet>
...             ''' % prefix
...     def resolve(self, url, pubid, context):
...         if url.startswith(self.prefix):
...             print "Resolved url %s as prefix %s" % (url, self.prefix)
...             return self.resolve_string(self.result_xml, context)
```

We demonstrate this in XSLT and use the following stylesheet as an example:

```
>>> xml_text = """\
... <xsl:stylesheet version="1.0"
...     xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
...     <xsl:include href="honk:test"/>
...     <xsl:template match="/">
...         <test>
...             <xsl:value-of select="document('hoi:test')/*/text()"/>
...         </test>
...     </xsl:template>
... </xsl:stylesheet>
... """
```

Note that it needs to resolve two URIs: `honk:test` when compiling the XSLT document (i.e. when resolving `xsl:import` and `xsl:include` elements) and `hoi:test` at transformation time, when calls to the `document` function are resolved. If we now register different resolvers with two different parsers, we can parse our document twice in different resolver contexts:

```

>>> hoi_parser = etree.XMLParser()
>>> normal_doc = etree.parse(StringIO(xml_text), hoi_parser)

>>> hoi_parser.resolvers.add( PrefixResolver("hoi") )
>>> hoi_doc = etree.parse(StringIO(xml_text), hoi_parser)

>>> honk_parser = etree.XMLParser()
>>> honk_parser.resolvers.add( PrefixResolver("honk") )
>>> honk_doc = etree.parse(StringIO(xml_text), honk_parser)

```

These contexts are important for the further behaviour of the documents. They memorise their original parser so that the correct set of resolvers is used in subsequent lookups. To compile the stylesheet, XSLT must resolve the `honk:test` URI in the `xsl:include` element. The `hoi` resolver cannot do that:

```

>>> transform = etree.XSLT(normal_doc)
Traceback (most recent call last):
  [...]
XSLTParseError: Cannot resolve URI honk:test

>>> transform = etree.XSLT(hoi_doc)
Traceback (most recent call last):
  [...]
XSLTParseError: Cannot resolve URI honk:test

```

However, if we use the `honk` resolver associated with the respective document, everything works fine:

```

>>> transform = etree.XSLT(honk_doc)
Resolved url honk:test as prefix honk

```

Running the transform accesses the same parser context again, but since it now needs to resolve the `hoi` URI in the call to the document function, its `honk` resolver will fail to do so:

```

>>> result = transform(normal_doc)
Traceback (most recent call last):
  [...]
XSLTApplyError: Cannot resolve URI hoi:test

>>> result = transform(hoi_doc)
Traceback (most recent call last):
  [...]
XSLTApplyError: Cannot resolve URI hoi:test

>>> result = transform(honk_doc)
Traceback (most recent call last):
  [...]
XSLTApplyError: Cannot resolve URI hoi:test

```

This can only be solved by adding a `hoi` resolver to the original parser:

```

>>> honk_parser.resolvers.add( PrefixResolver("hoi") )
>>> result = transform(honk_doc)
Resolved url hoi:test as prefix hoi
>>> print str(result),
<?xml version="1.0"?>
<test>hoi-TEST</test>

```

We can see that the `hoi` resolver was called to generate a document that was then inserted into the result document by the XSLT transformation. Note that this is completely independent of the XML file you transform, as the URI is resolved from within the stylesheet context:

```
>>> result = transform(normal_doc)
Resolved url hoi:test as prefix hoi
>>> print str(result),
<?xml version="1.0"?>
<test>hoi-TEST</test>
```

It may be seen as a matter of taste what resolvers the generated document inherits. For XSLT, the output document inherits the resolvers of the input document and not those of the stylesheet. Therefore, the last result does not inherit any resolvers at all.

## 17.3 I/O access control in XSLT

By default, XSLT supports all extension functions from `libxslt` and `libexslt` as well as Python regular expressions through `EXSLT`. Some extensions enable style sheets to read and write files on the local file system.

XSLT has a mechanism to control the access to certain I/O operations during the transformation process. This is most interesting where XSL scripts come from potentially insecure sources and must be prevented from modifying the local file system. Note, however, that there is no way to keep them from eating up your precious CPU time, so this should not stop you from thinking about what XSLT you execute.

Access control is configured using the `XSLTAccessControl` class. It can be called with a number of keyword arguments that allow or deny specific operations:

```
>>> transform = etree.XSLT(honk_doc)
Resolved url honk:test as prefix honk
>>> result = transform(normal_doc)
Resolved url hoi:test as prefix hoi

>>> ac = etree.XSLTAccessControl(read_network=False)
>>> transform = etree.XSLT(honk_doc, access_control=ac)
Resolved url honk:test as prefix honk
>>> result = transform(normal_doc)
Traceback (most recent call last):
  [...]
XSLTApplyError: xsltLoadDocument: read rights for hoi:test denied
```

There are a few things to keep in mind:

- XSL parsing (`xsl:import`, etc.) is not affected by this mechanism
- `read_file=False` does not imply `write_file=False`, all controls are independent.
- `read_file` only applies to files in the file system. Any other scheme for URLs is controlled by the `*_network` keywords.
- If you need more fine-grained control than switching access on and off, you should consider writing a custom document loader that returns empty documents or raises exceptions if access is denied.

## Chapter 18

# Extension functions for XPath and XSLT

This document describes how to use Python extension functions in XPath and XSLT. They allow you to do things like this:

```
<xsl:value-of select="f:myPythonFunction(../sometag)" />
```

Here is how such a function looks like. As the first argument, it always receives a context object (see below). The other arguments are provided by the respective call in the XPath expression, one in the following examples. Any number of arguments is allowed:

```
>>> def hello(dummy, a):
...     return "Hello %s" % a
>>> def ola(dummy, a):
...     return "Ola %s" % a
>>> def loadsofargs(dummy, *args):
...     return "Got %d arguments." % len(args)
```

### 18.1 The FunctionNamespace

In order to use a function in XPath/XSLT, it needs to have a (namespaced) name by which it can be called during evaluation. This is done using the `FunctionNamespace` class. For simplicity, we choose the empty namespace (`None`):

```
>>> from lxml import etree
>>> ns = etree.FunctionNamespace(None)
>>> ns['hello'] = hello
>>> ns['countargs'] = loadsofargs
```

This registers the function `hello` with the name `hello` in the default namespace (`None`), and the function `loadsofargs` with the name `countargs`. Now we're going to create a document that we can run XPath expressions against:

```
>>> root = etree.XML('<a><b>Haegar</b></a>')
>>> doc = etree.ElementTree(root)
```

Done. Now we can have XPath expressions call our new function:

```

>>> print root.xpath("hello('world')")
Hello world
>>> print root.xpath('hello(local-name(*))')
Hello b
>>> print root.xpath('hello(string(b))')
Hello Haegar
>>> print root.xpath('countargs(., b, ./*)')
Got 3 arguments.

```

Note how we call both a Python function (*hello*) and an XPath built-in function (*string*) in exactly the same way. Normally, however, you would want to separate the two in different namespaces. The `FunctionNamespace` class allows you to do this:

```

>>> ns = etree.FunctionNamespace('http://mydomain.org/myfunctions')
>>> ns['hello'] = hello
>>> prefixmap = {'f' : 'http://mydomain.org/myfunctions'}
>>> print root.xpath('f:hello(local-name(*))', namespaces=prefixmap)
Hello b

```

## 18.2 Global prefix assignment

In the last example, you had to specify a prefix for the function namespace. If you always use the same prefix for a function namespace, you can also register it with the namespace:

```

>>> ns = etree.FunctionNamespace('http://mydomain.org/myother/functions')
>>> ns.prefix = 'es'
>>> ns['hello'] = ola
>>> print root.xpath('es:hello(local-name(*))')
Ola b

```

This is a global assignment, so take care not to assign the same prefix to more than one namespace. The resulting behaviour in that case is completely undefined. It is always a good idea to consistently use the same meaningful prefix for each namespace throughout your application.

The prefix assignment only works with functions and `FunctionNamespace` objects, not with the general `Namespace` object that registers element classes. The reasoning is that elements in lxml do not care about prefixes anyway, so it would rather complicate things than be of any help.

## 18.3 The XPath context

Functions get a context object as first parameter. In lxml 1.x, this value was `None`, but since lxml 2.0 it provides two properties: `eval_context` and `context_node`. The context node is the Element where the current function is called:

```

>>> def print_tag(context, nodes):
...     print context.context_node.tag, [ n.tag for n in nodes ]

>>> ns = etree.FunctionNamespace('http://mydomain.org/printtag')
>>> ns.prefix = "pt"
>>> ns["print_tag"] = print_tag

>>> ignore = root.xpath("//*[pt:print_tag(./*)]")

```

```
a ['b']
b []
```

The `eval_context` is a dictionary that is local to the evaluation. It allows functions to keep state:

```
>>> def print_context(context):
...     context.eval_context[context.context_node.tag] = "done"
...     entries = context.eval_context.items()
...     entries.sort()
...     print entries
>>> ns["print_context"] = print_context

>>> ignore = root.xpath("//*[pt:print_context()]")
[('a', 'done')]
[('a', 'done'), ('b', 'done')]
```

## 18.4 Evaluators and XSLT

Extension functions work for all ways of evaluating XPath expressions and for XSL transformations:

```
>>> e = etree.XPathEvaluator(doc)
>>> print e.evaluate('es:hello(local-name(/a))')
Ola a

>>> namespaces = {'f' : 'http://mydomain.org/myfunctions'}
>>> e = etree.XPathEvaluator(doc, namespaces=namespaces)
>>> print e.evaluate('f:hello(local-name(/a))')
Hello a

>>> xslt = etree.XSLT(etree.XML('''
... <stylesheet version="1.0"
...     xmlns="http://www.w3.org/1999/XSL/Transform"
...     xmlns:es="http://mydomain.org/myother/functions">
...     <output method="text" encoding="ASCII"/>
...     <template match="/">
...         <value-of select="es:hello(string(/b))"/>
...     </template>
... </stylesheet>
... '''))
>>> print xslt(doc)
Ola Haegar
```

It is also possible to register namespaces with a single evaluator after its creation. While the following example involves no functions, the idea should still be clear:

```
>>> from StringIO import StringIO
>>> f = StringIO('<a xmlns="http://mydomain.org/myfunctions" />')
>>> ns_doc = etree.parse(f)
>>> e = etree.XPathEvaluator(ns_doc)
>>> e.evaluate('/a')
[]
```

This returns nothing, as we did not ask for the right namespace. When we register the namespace with the evaluator, however, we can access it via a prefix:

```
>>> e.registerNamespace('foo', 'http://mydomain.org/myfunctions')
>>> e.evaluate('/foo:a')[0].tag
'{http://mydomain.org/myfunctions}a'
```

Note that this prefix mapping is only known to this evaluator, as opposed to the global mapping of the `FunctionNamespace` objects:

```
>>> e2 = etree.XPathEvaluator(ns_doc)
>>> e2.evaluate('/foo:a')
Traceback (most recent call last):
...
XPathEvalError: Undefined namespace prefix
```

## 18.5 Evaluator-local extensions

Apart from the global registration of extension functions, there is also a way of making extensions known to a single Evaluator or XSLT. All evaluators and the XSLT object accept a keyword argument `extensions` in their constructor. The value is a dictionary mapping (namespace, name) tuples to functions:

```
>>> extensions = (('local-ns', 'local-hello') : hello}
>>> namespaces = {'l' : 'local-ns'}

>>> e = etree.XPathEvaluator(doc, namespaces=namespaces, extensions=extensions)
>>> print e.evaluate('l:local-hello(string(b))')
Hello Haegar
```

For larger numbers of extension functions, you can define classes or modules and use the `Extension` helper:

```
>>> class MyExt:
...     def function1(self, _, arg):
...         return '1'+arg
...     def function2(self, _, arg):
...         return '2'+arg
...     def function3(self, _, arg):
...         return '3'+arg

>>> ext_module = MyExt()
>>> functions = ('function1', 'function2')
>>> extensions = etree.Extension( ext_module, functions, ns='local-ns' )

>>> e = etree.XPathEvaluator(doc, namespaces=namespaces, extensions=extensions)
>>> print e.evaluate('l:function1(string(b))')
1Haegar
```

The optional second argument to `Extension` can either be a sequence of names to select from the module, a dictionary that explicitly maps function names to their XPath alter-ego or `None` (explicitly passed) to take all available functions under their original name (if their name does not start with `'_'`).

The additional `ns` keyword argument takes a namespace URI or `None` (also if left out) for the default namespace. The following examples will therefore all do the same thing:

```
>>> functions = ('function1', 'function2', 'function3')
>>> extensions = etree.Extension( ext_module, functions )
```

```

>>> e = etree.XPathEvaluator(doc, extensions=extensions)
>>> print e.evaluate('function1(function2(function3(string(b))))')
123Haegar

>>> extensions = etree.Extension( ext_module, functions, ns=None )
>>> e = etree.XPathEvaluator(doc, extensions=extensions)
>>> print e.evaluate('function1(function2(function3(string(b))))')
123Haegar

>>> extensions = etree.Extension(ext_module)
>>> e = etree.XPathEvaluator(doc, extensions=extensions)
>>> print e.evaluate('function1(function2(function3(string(b))))')
123Haegar

>>> functions = {
...     'function1' : 'function1',
...     'function2' : 'function2',
...     'function3' : 'function3'
...     }
>>> extensions = etree.Extension(ext_module, functions)
>>> e = etree.XPathEvaluator(doc, extensions=extensions)
>>> print e.evaluate('function1(function2(function3(string(b))))')
123Haegar

```

For convenience, you can also pass a sequence of extensions:

```

>>> extensions1 = etree.Extension(ext_module)
>>> extensions2 = etree.Extension(ext_module, ns='local-ns')
>>> e = etree.XPathEvaluator(doc, extensions=[extensions1, extensions2],
...                               namespaces=namespaces)
>>> print e.evaluate('function1(1:function2(function3(string(b))))')
123Haegar

```

## 18.6 What to return from a function

Extension functions can return any data type for which there is an XPath equivalent (see the documentation on *XPath return values*). This includes numbers, boolean values, elements and lists of elements. Note that integers will also be returned as floats:

```

>>> def returnsFloat(_):
...     return 1.7
>>> def returnsInteger(_):
...     return 1
>>> def returnsBool(_):
...     return True
>>> def returnFirstNode(_, nodes):
...     return nodes[0]

>>> ns = etree.FunctionNamespace(None)
>>> ns['float'] = returnsFloat
>>> ns['int']   = returnsInteger
>>> ns['bool']  = returnsBool
>>> ns['first'] = returnFirstNode

```

```

>>> e = etree.XPathEvaluator(doc)
>>> e.evaluate("float()")
1.7
>>> e.evaluate("int()")
1.0
>>> int( e.evaluate("int()") )
1
>>> e.evaluate("bool()")
True
>>> e.evaluate("count(first(//b))")
1.0

```

As the last example shows, you can pass the results of functions back into the XPath expression. Elements and sequences of elements are treated as XPath node-sets:

```

>>> def returnsNodeSet(_):
...     results1 = etree.Element('results1')
...     etree.SubElement(results1, 'result').text = "Alpha"
...     etree.SubElement(results1, 'result').text = "Beta"
...
...     results2 = etree.Element('results2')
...     etree.SubElement(results2, 'result').text = "Gamma"
...     etree.SubElement(results2, 'result').text = "Delta"
...
...     results3 = etree.SubElement(results2, 'subresult')
...     return [results1, results2, results3]

>>> ns['new-node-set'] = returnsNodeSet

>>> e = etree.XPathEvaluator(doc)

>>> r = e.evaluate("new-node-set()/result")
>>> print [ t.text for t in r ]
['Alpha', 'Beta', 'Gamma', 'Delta']

>>> r = e.evaluate("new-node-set()")
>>> print [ t.tag for t in r ]
['results1', 'results2', 'subresult']
>>> print [ len(t) for t in r ]
[2, 3, 0]
>>> r[0][0].text
'Alpha'

>>> print etree.tostring(r[0])
<results1><result>Alpha</result><result>Beta</result></results1>

>>> print etree.tostring(r[1])
<results2><result>Gamma</result><result>Delta</result><subresult/></results2>

>>> print etree.tostring(r[2])
<subresult/>

```

The current implementation deep-copies newly created elements in node-sets. Only the elements and their children are passed on, no outlying parents or tail texts will be available in the result. This also

means that in the above example, the *subresult* elements in *results2* and *results3* are no longer identical within the node-set, they belong to independent trees:

```
>>> print r[1][-1].tag, r[2].tag
subresult subresult
>>> print r[1][-1] == r[2]
False
>>> print r[1][-1].getparent().tag
results2
>>> print r[2].getparent()
None
```

This is an implementation detail that you should be aware of, but you should avoid relying on it in your code. Note that elements taken from the source document (the most common case) do not suffer from this restriction. They will always be passed unchanged.

## Chapter 19

# Using custom Element classes in lxml

lxml has very sophisticated support for custom Element classes. You can provide your own classes for Elements and have lxml use them by default, for all elements generated by a specific parser, for a specific tag name in a specific namespace or for an exact element at a specific position in the tree.

Custom Elements must inherit from the `lxml.etree.ElementBase` class, which provides the Element interface for subclasses:

```
>>> from lxml import etree
>>> class HonkElement(etree.ElementBase):
...     def honking(self):
...         return self.get('honking') == 'true'
...     honking = property(honking)
```

This defines a new Element class `HonkElement` with a property `honking`.

Note that you cannot (or rather *must not*) instantiate this class yourself. `lxml.etree` will do that for you through its normal ElementTree API.

### 19.1 Element initialization

There is one thing to know up front. Element classes *must not* have a constructor, neither must there be any internal state (except for the data stored in the underlying XML tree). Element instances are created and garbage collected at need, so there is no way to predict when and how often a constructor would be called. Even worse, when the `__init__` method is called, the object may not even be initialized yet to represent the XML tag, so there is not much use in providing an `__init__` method in subclasses.

Most use cases will not require any class initialisation, so you can content yourself with skipping to the next section for now. However, if you really need to set up your element class on instantiation, there is one possible way to do so. `ElementBase` classes have an `_init()` method that can be overridden. It can be used to modify the XML tree, e.g. to construct special children or verify and update attributes.

The semantics of `_init()` are as follows:

- It is called at least once on element instantiation time. That is, when a Python representation

of the element is created by lxml. At that time, the element object is completely initialized to represent a specific XML element within the tree.

- The method has complete access to the XML tree. Modifications can be done in exactly the same way as anywhere else in the program.
- Python representations of elements may be created multiple times during the lifetime of an XML element in the underlying tree. The `_init()` code provided by subclasses must take special care by itself that multiple executions either are harmless or that they are prevented by some kind of flag in the XML tree. The latter can be achieved by modifying an attribute value or by removing or adding a specific child node and then verifying this before running through the init process.
- Any exceptions raised in `_init()` will be propagated through the API call that lead to the creation of the Element. So be careful with the code you write here as its exceptions may turn up in various unexpected places.

## 19.2 Setting up a class lookup scheme

The first thing to do when deploying custom element classes is to register a class lookup scheme on a parser. lxml.etree provides quite a number of different schemes, that also support class lookup based on namespaces or attribute values. Most lookups support fallback chaining, which allows the next lookup mechanism to take over when the previous one fails to find a class.

For example, setting a different default element class for a parser works as follows:

```
>>> parser_lookup = etree.ElementDefaultClassLookup(element=HonkElement)
>>> parser = etree.XMLParser()
>>> parser.set_element_class_lookup(parser_lookup)
```

There is one drawback of the parser based scheme: the `Element()` factory does not know about your specialised parser and creates a new document that deploys the default parser:

```
>>> e1 = etree.Element("root")
>>> print isinstance(e1, HonkElement)
False
```

You should therefore avoid using this function in code that uses custom classes. The `makeelement()` method of parsers provides a simple replacement:

```
>>> e1 = parser.makeelement("root")
>>> print isinstance(e1, HonkElement)
True
```

If you use a parser at the module level, you can easily redirect a module level `Element()` factory to the parser method by adding code like this:

```
>>> MODULE_PARSER = etree.XMLParser()
>>> Element = MODULE_PARSER.makeelement
```

While the `XML()` and `HTML()` factories also depend on the default parser, you can pass them a different parser as second argument:

```
>>> element = etree.XML("<test/>")
>>> print isinstance(element, HonkElement)
False
```

```
>>> element = etree.XML("<test/>", parser)
>>> print isinstance(element, HonkElement)
True
```

Whenever you create a document with a parser, it will inherit the lookup scheme and all subsequent element instantiations for this document will use it:

```
>>> element = etree.fromstring("<test/>", parser)
>>> print isinstance(element, HonkElement)
True
>>> el = etree.SubElement(element, "subel")
>>> print isinstance(el, HonkElement)
True
```

For small projects, you may also consider setting a lookup scheme on the default parser. To avoid interfering with other modules, however, it is usually a better idea to use a dedicated parser for each module (or a parser pool when using threads) and then register the required lookup scheme only for this parser.

### 19.2.1 Default class lookup

This is the most simple lookup mechanism. It always returns the default element class. Consequently, no further fallbacks are supported, but this scheme is a good fallback for other custom lookup mechanisms.

Usage:

```
>>> lookup = etree.ElementDefaultClassLookup()
>>> parser = etree.XMLParser()
>>> parser.set_element_class_lookup(lookup)
```

Note that the default for new parsers is to use the global fallback, which is also the default lookup (if not configured otherwise).

To change the default element implementation, you can pass your new class to the constructor. While it accepts classes for `element`, `comment` and `pi` nodes, most use cases will only override the `element` class:

```
>>> el = parser.makeelement("myelement")
>>> print isinstance(el, HonkElement)
False

>>> lookup = etree.ElementDefaultClassLookup(element=HonkElement)
>>> parser.set_element_class_lookup(lookup)

>>> el = parser.makeelement("myelement")
>>> print isinstance(el, HonkElement)
True
>>> el.honking
False
>>> el = parser.makeelement("myelement", honking='true')
>>> print etree.tostring(el)
<myelement honking="true"/>
>>> el.honking
True
```

## 19.2.2 Namespace class lookup

This is an advanced lookup mechanism that supports namespace/tag-name specific element classes. You can select it by calling:

```
>>> lookup = etree.ElementNamespaceClassLookup()
>>> parser = etree.XMLParser()
>>> parser.set_element_class_lookup(lookup)
```

See the separate section on [implementing namespaces](#) below to learn how to make use of it.

This scheme supports a fallback mechanism that is used in the case where the namespace is not found or no class was registered for the element name. Normally, the default class lookup is used here. To change it, pass the desired fallback lookup scheme to the constructor:

```
>>> fallback = etree.ElementDefaultClassLookup(element=HonkElement)
>>> lookup = etree.ElementNamespaceClassLookup(fallback)
>>> parser.set_element_class_lookup(lookup)
```

## 19.2.3 Attribute based lookup

This scheme uses a mapping from attribute values to classes. An attribute name is set at initialisation time and is then used to find the corresponding value. It is set up as follows:

```
>>> id_class_mapping = {} # maps attribute values to element classes
>>> lookup = etree.AttributeBasedElementClassLookup(
...     'id', id_class_mapping)
>>> parser = etree.XMLParser()
>>> parser.set_element_class_lookup(lookup)
```

Instead of a global setup of this scheme, you should consider using a per-parser setup.

This class uses its fallback if the attribute is not found or its value is not in the mapping. Normally, the default class lookup is used here. If you want to use the namespace lookup, for example, you can use this code:

```
>>> fallback = etree.ElementNamespaceClassLookup()
>>> lookup = etree.AttributeBasedElementClassLookup(
...     'id', id_class_mapping, fallback)
>>> parser = etree.XMLParser()
>>> parser.set_element_class_lookup(lookup)
```

## 19.2.4 Custom element class lookup

This is the most customisable way of finding element classes on a per-element basis. It allows you to implement a custom lookup scheme in a subclass:

```
>>> class MyLookup(etree.CustomElementClassLookup):
...     def lookup(self, node_type, document, namespace, name):
...         return MyElementClass # defined elsewhere

>>> parser = etree.XMLParser()
>>> parser.set_element_class_lookup(MyLookup())
```

The `lookup()` method must either return `None` (which triggers the fallback mechanism) or a subclass of `lxml.etree.ElementBase`. It can take any decision it wants based on the node type (one of “element”, “comment”, “PI”), the XML document of the element, or its namespace or tag name.

Instead of a global setup of this scheme, you should consider using a per-parser setup.

### 19.2.5 Tree based element class lookup in Python

Taking more elaborate decisions than allowed by the custom scheme is difficult to achieve in pure Python. It would require access to the tree - before the elements in the tree have been instantiated as Python Element objects.

Luckily, there is a way to do this. The separate module `lxml.pyclasslookup` provides a lookup class called `PythonElementClassLookup` that works similar to the custom lookup scheme:

```
>>> from lxml.pyclasslookup import PythonElementClassLookup
>>> class MyLookup(PythonElementClassLookup):
...     def lookup(self, document, element):
...         return MyElementClass # defined elsewhere

>>> parser = etree.XMLParser()
>>> parser.set_element_class_lookup(MyLookup())
```

As before, the first argument to the `lookup()` method is the opaque document instance that contains the Element. The second arguments is a lightweight Element proxy implementation that is only valid during the lookup. Do not try to keep a reference to it. Once the lookup is finished, the proxy will become invalid. You will get an `AssertionError` if you access any of the properties or methods outside the scope of the lookup call where they were instantiated.

During the lookup, the element object behaves mostly like a normal Element instance. It provides the properties `tag`, `text`, `tail` etc. and supports indexing, slicing and the `getchildren()`, `getparent()` etc. methods. It does *not* support iteration, nor does it support any kind of modification. All of its properties are read-only and it cannot be removed or inserted into other trees. You can use it as a starting point to freely traverse the tree and collect any kind of information that its elements provide. Once you have taken the decision which class to use for this element, you can simply return it and have `lxml` take care of cleaning up the instantiated proxy classes.

## 19.3 Implementing namespaces

`lxml` allows you to implement namespaces, in a rather literal sense. After setting up the namespace class lookup mechanism as described above, you can build a new element namespace (or retrieve an existing one) by calling the `get_namespace(uri)` method of the lookup:

```
>>> lookup = etree.ElementNamespaceClassLookup()
>>> parser = etree.XMLParser()
>>> parser.set_element_class_lookup(lookup)

>>> namespace = lookup.get_namespace('http://hui.de/honk')
```

and then register the new element type with that namespace, say, under the tag name `honk`:

```
>>> namespace['honk'] = HonkElement
```

After this, you create and use your XML elements through the normal API of `lxml`:

```
>>> xml = '<honk xmlns="http://hui.de/honk" honking="true"/>'
>>> honk_element = etree.XML(xml, parser)
>>> print honk_element.honking
True
```

The same works when creating elements by hand:

```
>>> honk_element = parser.makeelement('{http://hui.de/honk}honk',
...                                     honking='true')
>>> print honk_element.honking
True
```

Essentially, what this allows you to do, is to give elements a custom API based on their namespace and tag name.

A somewhat related topic are [extension functions](#) which use a similar mechanism for registering extension functions in XPath and XSLT.

In the setup example above, we associated the HonkElement class only with the 'honk' element. If an XML tree contains different elements in the same namespace, they do not pick up the same implementation:

```
>>> xml = '<honk xmlns="http://hui.de/honk" honking="true"><bla/></honk>'
>>> honk_element = etree.XML(xml, parser)
>>> print honk_element.honking
True
>>> print honk_element[0].honking
Traceback (most recent call last):
...
AttributeError: 'lxml.etree._Element' object has no attribute 'honking'
```

You can therefore provide one implementation per element name in each namespace and have lxml select the right one on the fly. If you want one element implementation per namespace (ignoring the element name) or prefer having a common class for most elements except a few, you can specify a default implementation for an entire namespace by registering that class with the empty element name (None).

You may consider following an object oriented approach here. If you build a class hierarchy of element classes, you can also implement a base class for a namespace that is used if no specific element class is provided. Again, you can just pass None as an element name:

```
>>> class HonkNSElement(etree.ElementBase):
...     def honk(self):
...         return "HONK"
>>> namespace[None] = HonkNSElement

>>> class HonkElement(HonkNSElement):
...     def honking(self):
...         return self.get('honking') == 'true'
...     honking = property(honking)
>>> namespace['honk'] = HonkElement
```

Now you can rely on lxml to always return objects of type HonkNSElement or its subclasses for elements of this namespace:

```
>>> xml = '<honk xmlns="http://hui.de/honk" honking="true"><bla/></honk>'
>>> honk_element = etree.XML(xml, parser)

>>> print type(honk_element), type(honk_element[0])
```

```
<class 'HonkElement'> <class 'HonkNSElement'>

>>> print honk_element.honking
True
>>> print honk_element.honk()
HONK
>>> print honk_element[0].honk()
HONK
>>> print honk_element[0].honking
Traceback (most recent call last):
...
AttributeError: 'HonkNSElement' object has no attribute 'honking'
```

## Chapter 20

# Sax support

In this document we'll describe lxml's SAX support. lxml has support for producing SAX events for an `ElementTree` or `Element`. lxml can also turn SAX events into an `ElementTree`. The SAX API used by lxml is compatible with that in the Python core (`xml.sax`), so is useful for interfacing lxml with code that uses the Python core SAX facilities.

### 20.1 Building a tree from SAX events

First of all, lxml has support for building a new tree given SAX events. To do this, we use the special SAX content handler defined by lxml named `lxml.sax.ElementTreeContentHandler`:

```
>>> import lxml.sax
>>> handler = lxml.sax.ElementTreeContentHandler()
```

Now let's fire some SAX events at it:

```
>>> handler.startElementNS((None, 'a'), 'a', {})
>>> handler.startElementNS((None, 'b'), 'b', {(None, 'foo'): 'bar'})
>>> handler.characters('Hello world')
>>> handler.endElementNS((None, 'b'), 'b')
>>> handler.endElementNS((None, 'a'), 'a')
```

This constructs an equivalent tree. You can access it through the `etree` property of the handler:

```
>>> tree = handler.etree
>>> lxml.etree.tostring(tree.getroot())
'<a><b foo="bar">Hello world</b></a>'
```

By passing a `makeelement` function the constructor of `ElementTreeContentHandler`, e.g. the one of a parser you configured, you can determine which element class lookup scheme should be used.

### 20.2 Producing SAX events from an `ElementTree` or `Element`

Let's make a tree we can generate SAX events for:

```
>>> from StringIO import StringIO
```

```
>>> f = StringIO('<a><b>Text</b></a>')
>>> tree = lxml.etree.parse(f)
```

To see whether the correct SAX events are produced, we'll write a custom content handler.:

```
>>> from xml.sax.handler import ContentHandler
>>> class MyContentHandler(ContentHandler):
...     def __init__(self):
...         self.a_amount = 0
...         self.b_amount = 0
...         self.text = None
...
...     def startElementNS(self, name, qname, attributes):
...         uri, localname = name
...         if localname == 'a':
...             self.a_amount += 1
...         if localname == 'b':
...             self.b_amount += 1
...
...     def characters(self, data):
...         self.text = data
```

Note that it only defines the `startElementNS()` method and not `startElement()`. The SAX event generator in `lxml.sax` currently only supports namespace-aware processing.

To test the content handler, we can produce SAX events from the tree:

```
>>> handler = MyContentHandler()
>>> lxml.sax.saxify(tree, handler)
```

This is what we expect:

```
>>> handler.a_amount
1
>>> handler.b_amount
1
>>> handler.text
'Text'
```

## 20.3 Interfacing with pulldom/minidom

`lxml.sax` is a simple way to interface with the standard XML support in the Python library. Note, however, that this is a one-way solution, as Python's DOM implementation cannot generate SAX events from a DOM tree.

You can use `xml.dom.pulldom` to build a minidom from `lxml`:

```
>>> from xml.dom.pulldom import SAX2DOM
>>> handler = SAX2DOM()
>>> lxml.sax.saxify(tree, handler)
```

`PullDOM` makes the result available through the `document` attribute:

```
>>> dom = handler.document
>>> print dom.firstChild.localName
a
```

## Chapter 21

# The public C-API of lxml.etree

As of version 1.1, lxml.etree provides a public C-API. This allows external C extensions to efficiently access public functions and classes of lxml, without going through the Python API.

The API is described in the file [etreepublic.pxd](#), which is directly c-importable by extension modules implemented in [Pyrex](#) or [Cython](#).

### 21.1 Writing external modules in Cython

This is the easiest way of extending lxml at the C level. A [Cython](#) (or [Pyrex](#)) module should start like this:

```
# My Cython extension

# import the public functions and classes of lxml.etree
cimport etreepublic as cetree

# import the lxml.etree module in Python
cdef object etree
from lxml import etree

# initialize the access to the C-API of lxml.etree
cetree.import_lxml__etree()
```

From this line on, you can access all public functions of lxml.etree from the `cetree` namespace like this:

```
# build a tag name from namespace and element name
py_tag = cetree.namespacedNameFromNsName("http://some/url", "myelement")
```

Public lxml classes are easily subclassed. For example, to implement and set a new default element class, you can write Cython code like the following:

```
from etreepublic cimport ElementBase
cdef class NewElementClass(ElementBase):
    def set_value(self, myval):
        self.set("my_attribute", myval)

etree.set_element_class_lookup(
```

```
DefaultElementClassLookup(element=NewElementClass))
```

## 21.2 Writing external modules in C

If you really feel like it, you can also interface with `lxml.etree` straight from C code. All you have to do is include the header file for the public API, import the `lxml.etree` module and then call the import function:

```
/* My C extension */

/* common includes */
#include "Python.h"
#include "stdio.h"
#include "string.h"
#include "stdarg.h"
#include "libxml/xmlversion.h"
#include "libxml/encoding.h"
#include "libxml/hash.h"
#include "libxml/tree.h"
#include "libxml/xmlIO.h"
#include "libxml/xmlsave.h"
#include "libxml/globals.h"
#include "libxml/xmlstring.h"

/* lxml.etree specific includes */
#include "lxml-version.h"
#include "etree_defs.h"
#include "etree.h"

/* setup code */
import_lxml__etree()
```

Note that including `etree.h` does not automatically include the header files it requires. Note also that the above list of common includes may not be sufficient.

## Part IV

# Developing lxml

## Chapter 22

# How to build lxml from source

To build lxml from source, you need libxml2 and libxslt properly installed, *including the header files*. These are likely shipped in separate `-dev` or `-devel` packages like `libxml2-dev`, which you need to install. The build process also requires `setuptools`. The lxml source distribution comes with a script called `ez_setup.py` that can be used to install them.

### 22.1 Cython

The `lxml.etree` and `lxml.objectify` modules are written in [Cython](#). Since we distribute the Cython-generated `.c` files with lxml releases, however, you do not need Cython to build lxml from the normal release sources. We even encourage you to *not install Cython* for a normal release build, as the generated C code can vary quite heavily between Cython versions, which may or may not generate correct code for lxml. The pre-generated release sources were tested and therefore are known to work.

So, if you want a reliable build of lxml, we suggest to a) use a source release of lxml and b) disable or uninstall Cython for the build.

*Only* if you are interested in building lxml from a Subversion checkout (e.g. to test a bug fix that has not been release yet) or if want to be an lxml developer, then you do need a working Cython installation. You can use [EasyInstall](#) to install it:

```
easy_install Cython==0.9.6.12
```

lxml currently requires Cython 0.9.6.11b or 0.9.6.12, later versions were not tested.

### 22.2 Subversion

The lxml package is developed in a Subversion repository. You can retrieve the current developer version by calling:

```
svn co http://codespeak.net/svn/lxml/trunk lxml
```

This will create a directory `lxml` and download the source into it. You can also [browse the repository through the web](#) or use your favourite SVN client to access it.

## 22.3 Setuptools

Usually, building lxml is done through setuptools. Do a Subversion checkout (or download the source tar-ball and unpack it) and then type:

```
python setup.py build
```

or:

```
python setup.py bdist_egg
```

If you want to test lxml from the source directory, it is better to build it in-place like this:

```
python setup.py build_ext -i
```

or, in Unix-like environments:

```
make
```

If you get errors about missing header files (e.g. `libxml/xmlversion.h`) then you need to make sure the development packages of both `libxml2` and `libxslt` are properly installed. Try passing the following option to `setup.py` to make sure the right config is found:

```
python setup.py build --with-xslt-config=/path/to/xslt-config
```

If this doesn't help, you may have to add the location of the header files to the include path like:

```
python setup.py build_ext -i -I /usr/include/libxml2
```

where the file is in `/usr/include/libxml2/libxml/xmlversion.h`

To use `lxml.etree` in-place, you can place lxml's `src` directory on your Python module search path (`PYTHONPATH`) and then import `lxml.etree` to play with it:

```
# cd lxml
# PYTHONPATH=src python
Python 2.5.1
Type "help", "copyright", "credits" or "license" for more information.
>>> from lxml import etree
>>>
```

To recompile after changes, note that you may have to run `make clean` or delete the file `src/lxml/etree.c`. Distutils do not automatically pick up changes that affect files other than the main file `src/lxml/etree.pyx`.

## 22.4 Running the tests and reporting errors

The source distribution (`tgz`) and the Subversion repository contain a test suite for lxml. You can run it from the top-level directory:

```
python test.py
```

Note that the test script only tests the in-place build (see distutils building above), as it searches the `src` directory. You can use the following one-step command to trigger an in-place build and test it:

```
make test
```

This also runs the ElementTree and cElementTree compatibility tests. To call them separately, make sure you have lxml on your PYTHONPATH first, then run:

```
python selftest.py
```

and:

```
python selftest2.py
```

If the tests give failures, errors, or worse, segmentation faults, we'd really like to know. Please contact us on the [mailing list](#), and please specify the version of lxml, libxml2, libxslt and Python you were using, as well as your operating system type (Linux, Windows, MacOS, ...).

## 22.5 Contributing an egg

This is the procedure to make an lxml egg for your platform:

- Download the lxml-x.y.tar.gz release. This contains the pregenerated C so that you can be sure you build exactly from the release sources. Unpack them and cd into the resulting directory.
- `python setup.py build`
- If you're on a unixy platform, cd into `build/lib.your.platform` and strip any `.so` file you find there. This reduces the size of the egg considerably.
- `python setup.py bdist_egg upload`

The last 'upload' step only works if you have access to the lxml cheeseshop entry. If not, you can just make an egg with `bdist_egg` and mail it to the lxml maintainer.

## 22.6 Providing newer library versions on Mac-OS X

The Unix environment in Mac-OS X makes it relatively easy to install Unix/Linux style package management tools and new software. However, it seems to be hard to get libraries set up for exclusive usage that Mac-OS X ships in an older version. The result can be segfaults on this platform that are hard to track down.

To make sure the newer libxml2 and libxslt versions (e.g. those provided by fink or macports) are used at *build time*, you must take care that the script `xslt-config` is found from the newly installed version when running the build setup. The system libraries also provide this script, but the new one must come first in the PATH. The best way to make sure the right version is used is by passing the path to the script as an option to setup.py:

```
python setup.py build --with-xslt-config=/path/to/xslt-config \
                    --with-xml2-config=/path/to/xml2-config
```

To make sure the newer libxml2 and libxslt versions are used at *runtime*, you should add *all* directories where the newer libraries are installed (i.e. libxml2, libxslt and libexslt) to the DYLD\_LIBRARY\_PATH environment variable when you use lxml (i.e. not only at build time). This seems to fix a lot of problems for users.

Please read this thread about [experiences with MacOS-X](#) if you encounter problems. It also has a [buildout for lxml](#) that you can use.

## 22.7 Static linking on Windows

Most operating systems have proper package management that makes installing current versions of libxml2 and libxslt easy. The most famous exception is Microsoft Windows, which entirely lacks these capabilities. It can therefore be interesting to statically link the external libraries into lxml.etree to avoid having to install them separately.

Download lxml and all required libraries to the same directory. The iconv, libxml2, libxslt, and zlib libraries are all available from the ftp site <ftp://ftp.zlatkovic.com/pub/libxml/>.

Your directory should now have the following files in it (although most likely different versions):

```
iconv-1.9.1.win32.zip
libxml2-2.6.23.win32.zip
libxslt-1.1.15.win32.zip
lxml-1.0.0.tgz
zlib-1.2.3.win32.zip
```

Now extract each of those files in the *same* directory. This should give you something like this:

```
iconv-1.9.1.win32/
iconv-1.9.1.win32.zip
libxml2-2.6.23.win32/
libxml2-2.6.23.win32.zip
libxslt-1.1.15.win32/
libxslt-1.1.15.win32.zip
lxml-1.0.0/
lxml-1.0.0.tgz
zlib-1.2.3.win32/
zlib-1.2.3.win32.zip
```

Go to the lxml directory and edit the file `setup.py`. There should be a section near the top that looks like this:

```
STATIC_INCLUDE_DIRS = []
STATIC_LIBRARY_DIRS = []
STATIC_CFLAGS = []
```

Change this section to something like this, but take care to use the correct version numbers:

```
STATIC_INCLUDE_DIRS = [
    "..\\libxml2-2.6.23.win32\\include",
    "..\\libxslt-1.1.15.win32\\include",
    "..\\zlib-1.2.3.win32\\include",
    "..\\iconv-1.9.1.win32\\include"
]

STATIC_LIBRARY_DIRS = [
    "..\\libxml2-2.6.23.win32\\lib",
    "..\\libxslt-1.1.15.win32\\lib",
    "..\\zlib-1.2.3.win32\\lib",
    "..\\iconv-1.9.1.win32\\lib"
]

STATIC_CFLAGS = []
```

Add any CFLAGS you might consider useful to the third list. As [Ashish Kulkarni](#) notes, you might

have to add the standard Windows library `wsock32.dll` to the list of libraries to make `lxml.objectify` compile.

Now you should be able to pass the `--static` option to `setup.py` and everything should work well. Try calling:

```
python setup.py bdist_wininst --static
```

This will create a windows installer in the `pkg` directory.

## 22.8 Building Debian packages from SVN sources

[Andreas Pakulat](#) proposed the following approach.

- `apt-get source lxml`
- remove the unpacked directory
- `tar.gz` the `lxml` SVN version and replace the `orig.tar.gz` that lies in the directory
- check `md5sum` of created `tar.gz` file and place new sum and size in `dsc` file
- do `dpkg-source -x lxml-[VERSION].dsc` and `cd` into the newly created directory
- run `dch -i` and add a comment like “use trunk version”, this will increase the debian version number so `apt/dpkg` won't get confused
- run `dpkg-buildpackage -rfakeroot -us -uc` to build the package

In case `dpkg-buildpackage` tells you that some dependencies are missing, you can either install them manually or run `apt-get build-dep lxml`.

That will give you `.deb` packages in the parent directory which can be installed using `dpkg -i`.

## Chapter 23

# How to read the source of lxml

**Author:** Stefan Behnel

This document describes how to read the source code of `lxml` and how to start working on it. You might also be interested in the companion document that describes [how to build lxml from sources](#).

### 23.1 What is Cython?

`Cython` is the language that `lxml` is written in. It is a very Python-like language that was specifically designed for writing Python extension modules.

The reason why Cython (or actually its predecessor `Pyrex` at the time) was chosen as an implementation language for `lxml`, is that it makes it very easy to interface with both the Python world and external C code. Cython generates all the necessary glue code for the Python API, including Python types, calling conventions and reference counting. On the other side of the table, calling into C code is not more than declaring the signature of the function and maybe some variables as being C types, pointers or structs, and then calling it. The rest of the code is just plain Python code.

The Cython language is so close to Python that the Cython compiler can actually compile many, many Python programs to C without major modifications. But the real speed gains of a C compilation come from type annotations that were added to the language and that allow Cython to generate very efficient C code.

Even if you are not familiar with Cython, you should keep in mind that a slow implementation of a feature is better than none. So, if you want to contribute and have an idea what code you want to write, feel free to start with a pure Python implementation. Chances are, if you get the change officially accepted and integrated, others will take the time to optimise it so that it runs fast in Cython.

### 23.2 Where to start?

First of all, read [how to build lxml from sources](#) to learn how to retrieve the source code from the Subversion repository and how to build it. The source code lives in the subdirectory `src` of the checkout.

The main extension modules in `lxml` are `lxml.etree` and `lxml.objectify`. All main modules have the file extension `.pyx`, which shows the descendance from `Pyrex`. As usual in Python, the main files

start with a short description and a couple of imports. Cython distinguishes between the run-time `import` statement (as known from Python) and the compile-time `cimport` statement, which imports C declarations, either from external libraries or from other Cython modules.

### 23.2.1 Concepts

lxml's tree API is based on proxy objects. That means, every `Element` object (or rather `_Element` object) is a proxy for a libxml2 node structure. The class declaration is (mainly):

```
cdef class _Element:
    cdef _Document _doc
    cdef xmlNode* _c_node
```

It is a naming convention that C variables and C level class members that are passed into libxml2 start with a prefixed `c_` (commonly libxml2 struct pointers), and that C level class members are prefixed with an underscore. So you will often see names like `c_doc` for an `xmlDoc*` variable (or `c_node` for an `xmlNode*`), or the above `_c_node` for a class member that points to an `xmlNode` struct (or `_c_doc` for an `xmlDoc*`).

It is important to know that every proxy in lxml has a factory function that properly sets up C level members. Proxy objects must *never* be instantiated outside of that factory. For example, to instantiate an `_Element` object or its subclasses, you must always call its factory function:

```
cdef xmlNode* c_node
cdef _Document doc
cdef _Element element
...
element = _elementFactory(doc, c_node)
```

A good place to see how this factory is used are the `Element` methods `getparent()`, `getnext()` and `getprevious()`.

### 23.2.2 The documentation

An important part of lxml is the documentation that lives in the `doc` directory. It describes a large part of the API and comprises a lot of example code in the form of doctests.

The documentation is written in the [ReStructured Text](#) format, a very powerful text markup language that looks almost like plain text. It is part of the [docutils](#) package.

The project web site of [lxml](#) is completely generated from these text documents. Even the side menu is just collected from the table of contents that the ReST processor writes into each HTML page. Obviously, we use lxml for this.

The easiest way to generate the HTML pages is by calling:

```
make html
```

This will call the script `doc/mkhtml.py` to run the ReST processor on the files. After generating an HTML page the script parses it back in to build the side menu, and injects the complete menu into each page at the very end.

Running the `make` command will also generate the API documentation if you have [epydoc](#) installed. The `epydoc` package will import and introspect the extension modules and also introspect and parse the Python modules of lxml. The aggregated information will then be written out into an HTML

documentation site.

## 23.3 lxml.etree

The main module, `lxml.etree`, is in the file `lxml.etree.pyx`. It implements the main functions and types of the ElementTree API, as well as all the factory functions for proxies. It is the best place to start if you want to find out how a specific feature is implemented.

At the very end of the file, it contains a series of `include` statements that merge the rest of the implementation into the generated C code. Yes, you read right: no importing, no source file namespacing, just plain good old `include` and a huge C code result of more than 100,000 lines that we throw right into the C compiler.

The main include files are:

**apihelpers.pxi** Private C helper functions. Except for the factory functions, most of the little functions that are used all over the place are defined here. This includes things like reading out the text content of a libxml2 tree node, checking input from the API level, creating a new Element node or handling attribute values. If you want to work on the lxml code, you should keep these functions in the back of your head, as they will definitely make your life easier.

**classlookup.pxi** Element class lookup mechanisms. The main API and engines for those who want to define custom Element classes and inject them into lxml.

**docloader.pxi** Support for custom document loaders. Base class and registry for custom document resolvers.

**extensions.pxi** Infrastructure for extension functions in XPath/XSLT, including XPath value conversion and function registration.

**iterparse.pxi** Incremental XML parsing. An iterator class that builds iterparse events while parsing.

**nsclasses.pxi** Namespace implementation and registry. The registry and engine for Element classes that use the ElementNamespaceClassLookup scheme.

**parser.pxi** Parsers for XML and HTML. This is the main parser engine. It's the reason why you can parse a document from various sources in two lines of Python code. It's definitely not the right place to start reading lxml's source code.

**parsertarget.pxi** An ElementTree compatible parser target implementation based on the SAX2 interface of libxml2.

**proxy.pxi** Very low-level functions for memory allocation/deallocation and Element proxy handling. Ignoring this for the beginning will save your head from exploding.

**public-api.pxi** The set of C functions that are exported to other extension modules at the C level. For example, `lxml.objectify` makes use of these. See the *C-level API* documentation.

**saxparser.pxi** SAX-like parser interfaces as known from ElementTree's TreeBuilder.

**serializer.pxi** XML output functions. Basically everything that creates byte sequences from XML trees.

**xinclude.pxi** XInclude support.

**xmlerror.pxi** Error log handling. All error messages that libxml2 generates internally walk through the code in this file to end up in lxml's Python level error logs.

At the end of the file, you will find a long list of named error codes. It is generated from the libxml2 HTML documentation (using lxml, of course). See the script `update-error-constants.py` for this.

**xmliid.pxi** XMLID and IDDict, a dictionary-like way to find Elements by their XML-ID attribute.

**xpath.pxi** XPath evaluators.

**xslt.pxi** XSL transformations, including the XSLT class, document lookup handling and access control.

The different schema languages (DTD, RelaxNG, XML Schema and Schematron) are implemented in the following include files:

- `dtd.pxi`
- `relaxng.pxi`
- `schematron.pxi`
- `xmlschema.pxi`

## 23.4 Python modules

The `lxml` package also contains a number of pure Python modules:

**builder.py** The E-factory and the `ElementBuilder` class. These provide a simple interface to XML tree generation.

**cssselect.py** A CSS selector implementation based on XPath. The main class is called `CSSSelector`.

**doctestcompare.py** A relaxed comparison scheme for XML/HTML markup in doctest.

**ElementInclude.py** XInclude-like document inclusion, compatible with `ElementTree`.

**\_elementpath.py** XPath-like path language, compatible with `ElementTree`.

**sax.py** SAX2 compatible interfaces to copy lxml trees from/to SAX compatible tools.

**usedoctest.py** Wrapper module for `doctestcompare.py` that simplifies its usage from inside a doctest.

## 23.5 lxml.objectify

A Cython implemented extension module that uses the public C-API of `lxml.etree`. It provides a Python object-like interface to XML trees.

## 23.6 lxml.pyclasslookup

A Cython implemented extension module that uses the public C-API of `lxml.etree`. It provides a class lookup scheme that duplicates lxml's `ElementTree` API in a very simple way to provide Python access to the tree *before* instantiating the real Python proxies in `lxml.etree`.

## **23.7** `lxml.html`

A specialised toolkit for HTML handling, based on `lxml.etree`. This is implemented in pure Python.

# Chapter 24

## Credits

Stefan Behnel - main developer and maintainer

Martijn Faassen - creator of lxml and initial main developer

Ian Bicking - creator and maintainer of lxml.html

Holger Joukl - bug reports, feedback and development on lxml.objectify

Sidnei da Silva - official MS Windows builds

Marc-Antoine Parent - XPath extension function help and patches

**Olivier Grisel - improved (c)ElementTree compatibility patches**, website improvements.

Kasimier Buchcik - help with specs and libxml2

Florian Wagner - help with copy.deepcopy support, bug reporting

Emil Kroymann - help with encoding support, bug reporting

Slou - help with index() support, bug reporting

Paul Everitt - bug reporting, feedback on API design

Victor Ng - Discussions on memory management strategies, vlibxml2

Robert Kern - feedback on API design

Andreas Pakulat - rpath linking support, doc improvements

Steve Howe - Windows builds

David Sankel - building statically on Windows

... and lots of other people who contributed to lxml by reporting bugs, discussing its functionality or blaming the docs for the bugs in their code. Thank you all, user feedback and discussions form a very important part of an Open Source project!

### 24.1 Special thanks goes to:

- the libxml2 project and especially Daniel Veillard for a great XML library.

- Fredrik Lundh for the ElementTree API.
- Greg Ewing (Pyrex) and Robert Bradshaw (Cython) for the binding technology.
- the codespeak crew, in particular Philipp von Weitershausen and Holger Krekel for hosting lxml on codespeak.net
- Infrae for initiating the project.

# Appendix A

## Changes

### A.1 Changes in version 2.0.11, released 2008-12-12

#### Bugs fixed

- Crash when using an XPath evaluator in multiple threads.

### A.2 Changes in version 2.0.10, released 2008-11-17

#### Bugs fixed

- Ref-count leaks when lxml enters a try-except statement while an outside exception lives in `sys.exc_*`. This was due to a problem in Cython, not lxml itself.

### A.3 Changes in version 2.0.9, released 2008-09-05

#### Bugs fixed

- Memory problem when passing documents between threads.
- Target parser did not honour the `recover` option and raised an exception instead of calling `.close()` on the target.

### A.4 Changes in version 2.0.8, released 2008-07-24

#### Features added

- `lxml.html.rewrite_links()` strips links to work around documents with whitespace in URL attributes.

## **Bugs fixed**

- Crash when parsing XSLT stylesheets in a thread and using them in another.
- CSS selector parser dropped remaining expression after a function with parameters.

## **Other changes**

## **A.5 Changes in version 2.0.7, released 2008-06-20**

### **Features added**

- Pickling `ElementTree` objects in `lxml.objectify`.

### **Bugs fixed**

- Descending dot-separated classes in CSS selectors were not resolved correctly.
- `ElementTree.parse()` didn't handle target parser result.
- Potential threading problem in `XInclude`.
- Crash in `Element` class lookup classes when the `__init__()` method of the super class is not called from Python subclasses.

### **Other changes**

- Non-ASCII characters in attribute values are no longer escaped on serialisation.

## **A.6 Changes in version 2.0.6, released 2008-05-31**

### **Features added**

### **Bugs fixed**

- Incorrect evaluation of `e1.find("tag[child]")`.
- Windows build was broken.
- Moving a subtree from a document created in one thread into a document of another thread could crash when the rest of the source document is deleted while the subtree is still in use.
- Rare crash when serialising to a file object with certain encodings.

### **Other changes**

- `lxml` should now build without problems on MacOS-X.

## A.7 Changes in version 2.0.5, released 2008-05-01

### Features added

### Bugs fixed

- Resolving to a filename in custom resolvers didn't work.
- lxml did not honour libxslt's second error state "STOPPED", which let some XSLT errors pass silently.
- Memory leak in Schematron with libxml2  $\geq$  2.6.31.

### Other changes

## A.8 Changes in version 2.0.4, released 2008-04-13

### Features added

### Bugs fixed

- Hanging thread in conjunction with GTK threading.
- Crash bug in iterparse when moving elements into other documents.
- HTML elements' `.cssselect()` method was broken.
- `ElementTree.find*()` didn't accept QName objects.

### Other changes

## A.9 Changes in version 2.0.3, released 2008-03-26

### Features added

- `soupparser.parse()` allows passing keyword arguments on to BeautifulSoup.
- `fromstring()` method in `lxml.html.soupparser`.

### Bugs fixed

- `lxml.html.diff` didn't treat empty tags properly (e.g., `<br>`).
- Handle entity replacements correctly in target parser.
- Crash when using `iterparse()` with XML Schema validation.
- The BeautifulSoup parser (`soupparser.py`) did not replace entities, which made them turn up in text content.

- Attribute assignment of custom PyTypes in `objectify` could fail to correctly serialise the value to a string.

## Other changes

- `lxml.html.ElementSoup` was replaced by a new module `lxml.html.soupparser` with a more consistent API. The old module remains for compatibility with `ElementTree`'s own `ElementSoup` module.
- Setting the `XSLT_CONFIG` and `XML2_CONFIG` environment variables at build time will let `setup.py` pick up the `xml2-config` and `xslt-config` scripts from the supplied path name.
- Passing `--with-xml2-config=/path/to/xml2-config` to `setup.py` will override the `xml2-config` script that is used to determine the C compiler options. The same applies for the `--with-xslt-config` option.

## A.10 Changes in version 2.0.2, released 2008-02-22

### Features added

- Support passing `base_url` to file parser functions to override the filename of the file(-like) object.

### Bugs fixed

- The prefix for `objectify`'s `pytype` namespace was missing from the set of default prefixes.
- Memory leak in `Schematron` (fixed only for `libxml2 2.6.31+`).
- Error type names in `RelaxNG` were reported incorrectly.
- Slice deletion bug fixed in `objectify`.

### Other changes

- Enabled doctests for some Python modules (especially `lxml.html`).
- Add a `method` argument to `lxml.html.tostring()` (`method="xml"` for XHTML output).
- Make it clearer that methods like `lxml.html.fromstring()` take a `base_url` argument.

## A.11 Changes in version 2.0.1, released 2008-02-13

### Features added

- Child iteration in `lxml.pyclasslookup`.
- Loads of new docstrings reflect the signature of functions and methods to make them visible in API docs and `help()`

## Bugs fixed

- The module `lxml.html.builder` was duplicated as `lxml.htmlbuilder`
- Form elements would return `None` for `form.fields.keys()` if there was an unnamed input field. Now unnamed input fields are completely ignored.
- Setting an element slice in `objectify` could insert slice-overlapping elements at the wrong position.

## Other changes

- The generated API documentation was cleaned up and disburdened from non-public classes etc.
- The previously public module `lxml.html.setmixin` was renamed to `lxml.html._setmixin` as it is not an official part of `lxml`. If you want to use it, feel free to copy it over to your own source base.
- Passing `--with-xslt-config=/path/to/xslt-config` to `setup.py` will override the `xslt-config` script that is used to determine the C compiler options.

## A.12 Changes in version 2.0, released 2008-02-01

### Features added

- Passing the `unicode` type as `encoding` to `tostring()` will serialise to unicode. The `tounicode()` function is now deprecated.
- `XMLSchema()` and `RelaxNG()` can parse from `StringIO`.
- `makeparser()` function in `lxml.objectify` to create a new parser with the usual `objectify` setup.

### Bugs fixed

### Other changes

## A.13 Changes in version 2.0beta2, released 2008-01-26

### Features added

- Plain ASCII XPath string results are no longer forced into unicode objects as in 2.0beta1, but are returned as plain strings as before.
- All XPath string results are 'smart' objects that have a `getparent()` method to retrieve their parent Element.
- `with_tail` option in serialiser functions.
- More accurate exception messages in validator creation.

## Bugs fixed

- Missing import in `lxml.html.clean`.
- Some Python 2.4-isms prevented lxml from building/running under Python 2.3.

## Other changes

- Exceptions carry only the part of the error log that is related to the operation that caused the error.
- `XMLSchema()` and `RelaxNG()` now enforce passing the source file/filename through the `file` keyword argument.
- The test suite now skips most doctests under Python 2.3.
- `make clean` no longer removes the `.c` files (use `make realclean` instead)

## A.14 Changes in version 2.0beta1, released 2008-01-11

### Features added

- Parse-time XML schema validation (`schema` parser keyword).
- XPath string results of the `text()` function and attribute selection make their Element container accessible through a `getparent()` method. As a side-effect, they are now always unicode objects (even ASCII strings).
- XSLT objects are usable in any thread - at the cost of a deep copy if they were not created in that thread.
- Invalid entity names and character references will be rejected by the `Entity()` factory.
- `entity.text` returns the textual representation of the entity, e.g. `&amp;`.

### Bugs fixed

- XPath on ElementTrees could crash when selecting the virtual root node of the ElementTree.
- Compilation `--without-threading` was buggy in alpha5/6.

### Other changes

- Minor performance tweaks for Element instantiation and subelement creation

## A.15 Changes in version 2.0alpha6, released 2007-12-19

### Features added

- New properties `position` and `code` on `ParseError` exception (as in ET 1.3)

### Bugs fixed

- Memory leak in the `parse()` function.
- Minor bugs in XSLT error message formatting.
- Result document memory leak in target parser.

### Other changes

- Various places in the XPath, XSLT and iteration APIs now require keyword-only arguments.
- The argument order in `element.itorsiblings()` was changed to match the order used in all other iteration methods. The second argument ('preceding') is now a keyword-only argument.
- The `getiterator()` method on `Elements` and `ElementTrees` was reverted to return an iterator as it did in lxml 1.x. The ET API specification allows it to return either a sequence or an iterator, and it traditionally returned a sequence in ET and an iterator in lxml. However, it is now deprecated in favour of the `iter()` method, which should be used in new code wherever possible.
- The 'pretty printed' serialisation of `ElementTree` objects now inserts newlines at the root level between processing instructions, comments and the root tag.
- A 'pretty printed' serialisation is now terminated with a newline.
- Second argument to `lxml.etree.Extension()` helper is no longer required, third argument is now a keyword-only argument `ns`.
- `lxml.html.tostring` takes an `encoding` argument.

## A.16 Changes in version 2.0alpha5, released 2007-11-24

### Features added

- Rich comparison of `element.attrib` proxies.
- `ElementTree` compatible `TreeBuilder` class.
- Use default prefixes for some common XML namespaces.
- `lxml.html.clean.Cleaner` now allows for a `host_whitelist`, and two overridable methods: `allow_embedded_url(url)` and the more general `allow_element(e1)`.
- Extended slicing of `Elements` as in `element[1:-1:2]`, both in `etree` and in `objectify`
- Resolvers can now provide a `base_url` keyword argument when resolving a document as string

data.

- When using `lxml.doctestcompare` you can give the doctest option `NOPARSE_MARKUP` (like `# doctest: +NOPARSE_MARKUP`) to suppress the special checking for one test.

## Bugs fixed

- Target parser failed to report comments.
- In the `lxml.html.iter_links` method, links in `<object>` tags weren't recognized. (Note: plugin-specific link parameters still aren't recognized.) Also, the `<embed>` tag, though not standard, is now included in `lxml.html.defs.special_inline_tags`.
- Using custom resolvers on XSLT stylesheets parsed from a string could request ill-formed URLs.
- With `lxml.doctestcompare` if you do `<tag xmlns="...">` in your output, it will then be namespace-neutral (before the ellipsis was treated as a real namespace).

## Other changes

- The module source files were renamed to “`lxml*.pyx`”, such as “`lxml.etree.pyx`”. This was changed for consistency with the way Pyrex commonly handles package imports. The main effect is that classes now know about their fully qualified class name, including the package name of their module.
- Keyword-only arguments in some API functions, especially in the parsers and serialisers.

## A.17 Changes in version 1.3.6, released 2007-10-29

### Bugs fixed

- Backported `decref` crash fix from 2.0
- Well hidden free-while-in-use crash bug in `ObjectPath`

### Other changes

- The test suites now run `gc.collect()` in the `tearDown()` methods. While this makes them take a lot longer to run, it also makes it easier to link a specific test to garbage collection problems that would otherwise appear in later tests.

## A.18 Changes in version 1.3.5, released 2007-10-22

### Features added

### Bugs fixed

- `lxml.etree` could crash when adding more than 10000 namespaces to a document

- lxml failed to serialise namespace declarations of elements other than the root node of a tree

## A.19 Changes in version 2.0alpha4, released 2007-10-07

### Features added

### Bugs fixed

- `AttributeError` in feed parser on parse errors

### Other changes

- Tag name validation in `lxml.etree` (and `lxml.html`) now distinguishes between HTML tags and XML tags based on the parser that was used to parse or create them. HTML tags no longer reject any non-ASCII characters in tag names but only spaces and the special characters `<>&/'`.

## A.20 Changes in version 2.0alpha3, released 2007-09-26

### Features added

- Separate `feed_error_log` property for the feed parser interface. The normal parser interface and `iterparse` continue to use `error_log`.
- The normal parsers and the feed parser interface are now separated and can be used concurrently on the same parser instance.
- `fromstringlist()` and `tostringlist()` functions as in `ElementTree 1.3`
- `iterparse()` accepts an `html` boolean keyword argument for parsing with the HTML parser (note that this interface may be subject to change)
- Parsers accept an `encoding` keyword argument that overrides the encoding of the parsed documents.
- New C-API function `hasChild()` to test for children
- `annotate()` function in `objectify` can annotate with Python types and XSI types in one step. Accompanied by `xsiannotate()` and `pyannotate()`.

### Bugs fixed

- XML feed parser setup problem
- Type annotation for unicode strings in `DataElement()`

### Other changes

- `lxml.etree` now emits a warning if you use `XPath` with `libxml2 2.6.27` (which can crash on certain `XPath` errors)

- Type annotation in `objectify` now preserves the already annotated type by default to prevent losing type information that is already there.

## A.21 Changes in version 2.0alpha2, released 2007-09-15

### Features added

- `ET.write()`, `tostring()` and `tounicode()` now accept a keyword argument `method` that can be one of 'xml' (or None), 'html' or 'text' to serialise as XML, HTML or plain text content.
- `iterfind()` method on Elements returns an iterator equivalent to `findall()`
- `itertext()` method on Elements
- Setting a QName object as value of the `.text` property or as an attribute will resolve its prefix in the respective context
- ElementTree-like parser target interface as described in <http://effbot.org/elementtree/elementtree-xmlparser.htm>
- ElementTree-like feed parser interface on XMLParser and HTMLParser (`feed()` and `close()` methods)

### Bugs fixed

- lxml failed to serialise namespace declarations of elements other than the root node of a tree
- Race condition in XSLT where the resolver context leaked between concurrent XSLT calls

### Other changes

- `element.getiterator()` returns a list, use `element.iter()` to retrieve an iterator (ElementTree 1.3 compatible behaviour)

## A.22 Changes in version 2.0alpha1, released 2007-09-02

### Features added

- Reimplemented `objectify.E` for better performance and improved integration with `objectify`. Provides extended type support based on registered PyTypes.
- XSLT objects now support deep copying
- New `makeSubElement()` C-API function that allows creating a new subelement straight with text, tail and attributes.
- XPath extension functions can now access the current context node (`context.context_node`) and use a context dictionary (`context.eval_context`) from the context provided in their first parameter

- HTML tag soup parser based on BeautifulSoup in `lxml.html.ElementSoup`
- New module `lxml.doctestcompare` by Ian Bicking for writing simplified doctests based on XML/HTML output. Use by importing `lxml.usedoctest` or `lxml.html.usedoctest` from within a doctest.
- New module `lxml.cssselect` by Ian Bicking for selecting Elements with CSS selectors.
- New package `lxml.html` written by Ian Bicking for advanced HTML treatment.
- Namespace class setup is now local to the `ElementNamespaceClassLookup` instance and no longer global.
- Schematron validation (incomplete in libxml2)
- Additional `stringify` argument to `objectify.PyType()` takes a conversion function to strings to support setting text values from arbitrary types.
- Entity support through an `Entity` factory and element classes. XML parsers now have a `resolve_entities` keyword argument that can be set to `False` to keep entities in the document.
- `column` field on error log entries to accompany the `line` field
- Error specific messages in XPath parsing and evaluation NOTE: for evaluation errors, you will now get an `XPathEvalError` instead of an `XPathSyntaxError`. To catch both, you can except on `XPathError`
- The regular expression functions in XPath now support passing a node-set instead of a string
- Extended type annotation in objectify: new `xsiannotate()` function
- EXSLT RegExp support in standard XPath (not only XSLT)

## Bugs fixed

- `lxml.etree` did not check tag/attribute names
- The XML parser did not report undefined entities as error
- The text in exceptions raised by XML parsers, validators and XPath evaluators now reports the first error that occurred instead of the last
- Passing `”` as XPath namespace prefix did not raise an error
- Thread safety in XPath evaluators

## Other changes

- `objectify.PyType` for `None` is now called “`NoneType`”
- `el.getiterator()` renamed to `el.iter()`, following `ElementTree 1.3` - original name is still available as alias
- In the public C-API, `findOrBuildNodeNs()` was replaced by the more generic `findOrBuildNodeNsPrefix`
- Major refactoring in XPath/XSLT extension function code
- Network access in parsers disabled by default

## A.23 Changes in version 1.3.4, released 2007-08-30

### Features added

- The `ElementMaker` in `lxml.builder` now accepts the keyword arguments `namespace` and `nsmap` to set a namespace and nsmap for the Elements it creates.
- The `docinfo` on `ElementTree` objects has new properties `internalDTD` and `externalDTD` that return a DTD object for the internal or external subset of the document respectively.
- Serialising an `ElementTree` now includes any internal DTD subsets that are part of the document, as well as comments and PIs that are siblings of the root node.

### Bugs fixed

- Parsing with the `no_network` option could fail

### Other changes

- `lxml` now raises a `TagNameWarning` about tag names containing `'` instead of an `Error` as 1.3.3 did. The reason is that a number of projects currently misuse the previous lack of tag name validation to generate namespace prefixes without declaring namespaces. Apart from the danger of generating broken XML this way, it also breaks most of the namespace-aware tools in XML, including `XPath`, `XSLT` and validation. `lxml 1.3.x` will continue to support this bug with a `Warning`, while `lxml 2.0` will be strict about well-formed tag names (not only regarding `'`).
- Serialising an `Element` no longer includes its comment and PI siblings (only `ElementTree` serialisation includes them).

## A.24 Changes in version 1.3.3, released 2007-07-26

### Features added

- `ElementTree` compatible parser `ETCompatXMLParser` strips processing instructions and comments while parsing XML
- Parsers now support stripping PIs (keyword argument `'remove_pis'`)
- `etree.fromstring()` now supports parsing both HTML and XML, depending on the parser you pass.
- Support `base_url` keyword argument in `HTML()` and `XML()`

### Bugs fixed

- Parsing from Python Unicode strings failed on some platforms
- `Element()` did not raise an exception on tag names containing `'`

- `Element.getiterator(tag)` did not accept `Comment` and `ProcessingInstruction` as tags. It also accepts `Element` now.

## A.25 Changes in version 1.3.2, released 2007-07-03

### Features added

### Bugs fixed

- “deallocating None” crash bug

## A.26 Changes in version 1.3.1, released 2007-07-02

### Features added

- `objectify.DataElement` now supports setting values from existing data elements (not just plain Python types) and reuses defined namespaces etc.
- E-factory support for `lxml.objectify` (`objectify.E`)

### Bugs fixed

- Better way to prevent crashes in `Element` proxy cleanup code
- `objectify.DataElement` didn't set up `None` value correctly
- `objectify.DataElement` didn't check the value against the provided type hints
- Reference-counting bug in `Element.attrib.pop()`

## A.27 Changes in version 1.3, released 2007-06-24

### Features added

- Module `lxml.pyclasslookup` module implements an `Element` class lookup scheme that can access the entire tree in read-only mode to help determining a suitable `Element` class
- Parsers take a `remove_comments` keyword argument that skips over comments
- `parse()` function in `objectify`, corresponding to `XML()` etc.
- `Element.addnext(e1)` and `Element.addprevious(e1)` methods to support adding processing instructions and comments around the root node
- `Element.attrib` was missing `clear()` and `pop()` methods
- Extended type annotation in `objectify`: cleaner annotation namespace setup plus new `deannotate()` function

- Support for custom Element class instantiation in `lxml.sax`: passing a `makeelement` function to the `ElementTreeContentHandler` will reuse the lookup context of that function
- `'.'` represents empty `ObjectPath` (identity)
- `Element.values()` to accompany the existing `.keys()` and `.items()`
- `collectAttributes()` C-function to build a list of attribute keys/values/items for a `libxml2` node
- DTD validator class (like `RelaxNG` and `XMLSchema`)
- HTML generator helpers by Fredrik Lundh in `lxml.htmlbuilder`
- `ElementMaker` XML generator by Fredrik Lundh in `lxml.builder.E`
- Support for pickeling `objectify.ObjectifiedElement` objects to XML
- `update()` method on `Element.attrib`
- Optimised replacement for `libxml2`'s `_xmlReconsiliateNs()`. This allows `lxml` a better handling of namespaces when moving elements between documents.

## Bugs fixed

- Removing Elements from a tree could make them loose their namespace declarations
- `ElementInclude` didn't honour base URL of original document
- Replacing the children slice of an Element would cut off the tails of the original children
- `Element.getiterator(tag)` did not accept `Comment` and `ProcessingInstruction` as tags
- API functions now check incoming strings for XML conformity. Zero bytes or low ASCII characters are no longer accepted (`AssertionError`).
- XSLT parsing failed to pass resolver context on to imported documents
- passing `”` as namespace prefix in `nsmap` could be passed through to `libxml2`
- `Objectify` couldn't handle prefixed XSD type names in `xsi:type`
- More ET compatible behaviour when writing out XML declarations or not
- More robust error handling in `iterparse()`
- Documents lost their top-level PIs and comments on serialisation
- `lxml.sax` failed on comments and PIs. Comments are now properly ignored and PIs are copied.
- Possible memory leaks in namespace handling when moving elements between documents

## Other changes

- major restructuring in the documentation

## A.28 Changes in version 1.2.1, released 2007-02-27

### Bugs fixed

- Build fixes for MS compiler
- Item assignments to special names like `element["text"]` failed
- Renamed `ObjectifiedDataElement.__setText()` to `_setText()` to make it easier to access
- The pattern for attribute names in `ObjectPath` was too restrictive

## A.29 Changes in version 1.2, released 2007-02-20

### Features added

- Rich comparison of `QName` objects
- Support for regular expressions in benchmark selection
- `get/set` emulation (not `.attrib!`) for attributes on processing instructions
- `ElementInclude` Python module for `ElementTree` compatible `XInclude` processing that honours custom resolvers registered with the source document
- `ElementTree.parser` property holds the parser used to parse the document
- `setup.py` has been refactored for greater readability and flexibility
- `--rpath` flag to `setup.py` to induce automatic linking-in of dynamic library runtime search paths has been renamed to `--auto-rpath`. This makes it possible to pass an `--rpath` directly to `distutils`; previously this was being shadowed.

### Bugs fixed

- `Element` instantiation now uses locks to prevent race conditions with threads
- `ElementTree.write()` did not raise an exception when the file was not writable
- Error handling could crash under Python  $\leq 2.4.1$  - fixed by disabling thread support in these environments
- `Element.find*()` did not accept `QName` objects as path

### Other changes

- code cleanup: redundant `_NodeBase` super class merged into `_Element` class Note: although the impact should be zero in most cases, this change breaks the compatibility of the public C-API

## A.30 Changes in version 1.1.2, released 2006-10-30

### Features added

- Data elements in objectify support `repr()`, which is now used by `dump()`
- Source distribution now ships with a patched Pyrex
- New C-API function `makeElement()` to create new elements with text, tail, attributes and namespaces
- Reuse original parser flags for XInclude
- Simplified support for handling XSLT processing instructions

### Bugs fixed

- Parser resources were not freed before the next parser run
- Open files and XML strings returned by Python resolvers were not closed/freed
- Crash in the IDDict returned by XMLDTDID
- Copying Comments and ProcessingInstructions failed
- Memory leak for external URLs in `_XSLTProcessingInstruction.parseXSL()`
- Memory leak when garbage collecting tailed root elements
- HTML script/style content was not propagated to `.text`
- Show text xincluded between text nodes correctly in `.text` and `.tail`
- `'integer * objectify.StringElement'` operation was not supported

## A.31 Changes in version 1.1.1, released 2006-09-21

### Features added

- XSLT profiling support (`profile_run` keyword)
- `countchildren()` method on `objectify.ObjectifiedElement`
- Support custom elements for tree nodes in `lxml.objectify`

### Bugs fixed

- `lxml.objectify` failed to support long data values (e.g., “123L”)
- Error messages from XSLT did not reach `XSLT.error_log`
- Factories `objectify.Element()` and `objectify.DataElement()` were missing `attrib` and `nsmmap` keyword arguments

- Changing the default parser in `lxml.objectify` did not update the factories `Element()` and `DataElement()`
- Let `lxml.objectify.Element()` always generate tree elements (not data elements)
- Build under Windows failed ('0' bug in patched Pyrex version)

## A.32 Changes in version 1.1, released 2006-09-13

### Features added

- Comments and processing instructions return '`<!-- coment -->`' and '`<?pi-target content?>`' for `repr()`
- Parsers are now the preferred (and default) place where element class lookup schemes should be registered. Namespace lookup is no longer supported by default.
- Support for Python 2.5 beta
- Unlock the GIL for deep copying documents and for `XPath()`
- New `compact` keyword argument for parsing read-only documents
- Support for parser options in `iterparse()`
- The `namespace` axis is supported in `XPath` and returns (prefix, URI) tuples
- The `XPath` expression `"/"` now returns an empty list instead of raising an exception
- XML-Object API on top of `lxml` (`lxml.objectify`)
- Customizable Element class lookup:
  - different pre-implemented lookup mechanisms
  - support for externally provided lookup functions
- Support for processing instructions (ET-like, not compatible)
- Public C-level API for independent extension modules
- Module level `iterwalk()` function as 'iterparse' for trees
- Module level `iterparse()` function similar to `ElementTree` (see documentation for differences)
- `Element.nsmap` property returns a mapping of all namespace prefixes known at the Element to their namespace URI
- Reentrant threading support in RelaxNG, XMLSchema and XSLT
- Threading support in parsers and serializers:
  - All in-memory operations (`tostring`, `parse(StringIO)`, etc.) free the GIL
  - File operations (on file names) free the GIL
  - Reading from file-like objects frees the GIL and reacquires it for reading
  - Serialisation to file-like objects is single-threaded (high lock overhead)

- Element iteration over XPath axes:
  - Element.iterdescendants() iterates over the descendants of an element
  - Element.iterancestors() iterates over the ancestors of an element (from parent to parent)
  - Element.itorsiblings() iterates over either the following or preceding siblings of an element
  - Element.iterchildren() iterates over the children of an element in either direction
  - All iterators support the `tag` keyword argument to restrict the generated elements
- Element.getnext() and Element.getprevious() return the direct siblings of an element

## Bugs fixed

- filenames with local 8-bit encoding were not supported
- 1.1beta did not compile under Python 2.3
- ignore unknown 'pyval' attribute values in objectify
- objectify.ObjectifiedElement.addattr() failed to accept Elements and Lists
- objectify.ObjectPath.setattr() failed to accept Elements and Lists
- XPathSyntaxError now inherits from XPathError
- Threading race conditions in RelaxNG and XMLSchema
- Crash when mixing elements from XSLT results into other trees, concurrent XSLT is only allowed when the stylesheet was parsed in the main thread
- The EXSLT `regexp:match` function now works as defined (except for some differences in the regular expression syntax)
- Setting element.text to "" returned None on request, not the empty string
- `iterparse()` could crash on long XML files
- Creating documents no longer copies the parser for later URL resolving. For performance reasons, only a reference is kept. Resolver updates on the parser will now be reflected by documents that were parsed before the change. Although this should rarely become visible, it is a behavioral change from 1.0.

## A.33 Changes in version 1.0.4, released 2006-09-09

### Features added

- List-like `Element.extend()` method

### Bugs fixed

- Crash in tail handling in `Element.replace()`

## **A.34 Changes in version 1.0.3, released 2006-08-08**

### **Features added**

- `Element.replace(old, new)` method to replace a subelement by another one

### **Bugs fixed**

- Crash when mixing elements from XSLT results into other trees
- Copying/deepcopying did not work for `ElementTree` objects
- Setting an attribute to a non-string value did not raise an exception
- `Element.remove()` deleted the tail text from the removed `Element`

## **A.35 Changes in version 1.0.2, released 2006-06-27**

### **Features added**

- Support for setting a custom default `Element` class as opposed to namespace specific classes (which still override the default class)

### **Bugs fixed**

- Rare exceptions in Python list functions were not handled
- Parsing accepted unicode strings with XML encoding declaration in certain cases
- Parsing 8-bit encoded strings from `StringIO` objects raised an exception
- Module function `initThread()` was removed - useless (and never worked)
- XSLT and parser exception messages include the error line number

## **A.36 Changes in version 1.0.1, released 2006-06-09**

### **Features added**

- Repeated calls to `Element.attrib` now efficiently return the same instance

### **Bugs fixed**

- Document deallocation could crash in certain garbage collection scenarios
- Extension function calls in XSLT variable declarations could break the stylesheet and crash on repeated calls

- Deep copying Elements could loose namespaces declared in parents
- Deep copying Elements did not copy tail
- Parsing file(-like) objects failed to load external entities
- Parsing 8-bit strings from file(-like) objects raised an exception
- xsl:include failed when the stylesheet was parsed from a file-like object
- lxml.sax.ElementTreeProducer did not call startDocument() / endDocument()
- MSVC compiler complained about long strings (supports only 2048 bytes)

## A.37 Changes in version 1.0, released 2006-06-01

### Features added

- Element.getiterator() and the findall() methods support finding arbitrary elements from a namespace (pattern {namespace}\*)
- Another speedup in tree iteration code
- General speedup of Python Element object creation and deallocation
- Writing C14N no longer serializes in memory (reduced memory footprint)
- PyErrLog for error logging through the Python logging module
- Element.getroottree() returns an ElementTree for the root node of the document that contains the element.
- ElementTree.getpath(element) returns a simple, absolute XPath expression to find the element in the tree structure
- Error logs have a last\_error attribute for convenience
- Comment texts can be changed through the API
- Formatted output via pretty\_print keyword in serialization functions
- XSLT can block access to file system and network via XSLTAccessControl
- ElementTree.write() no longer serializes in memory (reduced memory footprint)
- Speedup of Element.findall(tag) and Element.getiterator(tag)
- Support for writing the XML representation of Elements and ElementTrees to Python unicode strings via etree.tounicode()
- Support for writing XSLT results to Python unicode strings via unicode()
- Parsing a unicode string no longer copies the string (reduced memory footprint)
- Parsing file-like objects reads chunks rather than the whole file (reduced memory footprint)
- Parsing StringIO objects from the start avoids copying the string (reduced memory footprint)
- Read-only 'docinfo' attribute in ElementTree class holds DOCTYPE information, original encoding

and XML version as seen by the parser

- etree module can be compiled without libxslt by commenting out the line `include "xslt.pxi"` near the end of the `etree.pyx` source file
- Better error messages in parser exceptions
- Error reporting also works in XSLT
- Support for custom document loaders (URI resolvers) in parsers and XSLT, resolvers are registered at parser level
- Implementation of `exslt:regexp` for XSLT based on the Python `'re'` module, enabled by default, can be switched off with `'regexp=False'` keyword argument
- Support for `exslt` extensions (`libexslt`) and `libxslt` extra functions (`node-set`, `document`, `write`, `output`)
- Substantial speedup in `XPath.evaluate()`
- `HTMLParser` for parsing (broken) HTML
- `XMLDTDID` function parses XML into tuple (root node, ID dict) based on `xml:id` implementation of `libxml2` (as opposed to ET compatible XMLID)

## Bugs fixed

- Memory leak in `Element.__setitem__`
- Memory leak in `Element.attrib.items()` and `Element.attrib.values()`
- Memory leak in `XPath` extension functions
- Memory leak in unicode related setup code
- `Element` now raises `ValueError` on empty tag names
- Namespace fixing after moving elements between documents could fail if the source document was freed too early
- Setting namespace-less tag names on namespaced elements (`'{ns}t' -> 't'`) didn't reset the namespace
- Unknown constants from newer `libxml2` versions could raise exceptions in the error handlers
- `lxml.etree` compiles much faster
- On `libxml2 <= 2.6.22`, parsing strings with encoding declaration could fail in certain cases
- Document reference in `ElementTree` objects was not updated when the root element was moved to a different document
- Running absolute `XPath` expressions on an `Element` now evaluates against the root tree
- Evaluating absolute `XPath` expressions (`/*`) on an `ElementTree` could fail
- Crashes when calling XSLT, RelaxNG, etc. with uninitialized `ElementTree` objects
- Removed public function `initThreadLogging()`, replaced by more general `initThread()` which fixes a number of setup problems in threads

- Memory leak when using iconv encoders in tostring/write
- Deep copying Elements and ElementTrees maintains the document information
- Serialization functions raise LookupError for unknown encodings
- Memory deallocation crash resulting from deep copying elements
- Some ElementTree methods could crash if the root node was not initialized (neither file nor element passed to the constructor)
- Element/SubElement failed to set attribute namespaces from passed `attrib` dictionary
- `tostring()` adds an XML declaration for non-ASCII encodings
- `tostring()` failed to serialize encodings that contain 0-bytes
- `ElementTree.xpath()` and `XPathDocumentEvaluator` were not using the `ElementTree` root node as reference point
- Calling `document('')` in XSLT failed to return the stylesheet

## A.38 Changes in version 0.9.2, released 2006-05-10

### Features added

- Speedup for `Element.makeelement()`: the new element reuses the original `libxml2` document instead of creating a new empty one
- Speedup for `reversed()` iteration over element children (Py2.4+ only)
- `ElementTree` compatible `QName` class
- `RelaxNG` and `XMLSchema` accept any `Element`, not only `ElementTrees`

### Bugs fixed

- `str(xslt_result)` was broken for XSLT output other than UTF-8
- Memory leak if `write_c14n` fails to write the file after conversion
- Crash in `XMLSchema` and `RelaxNG` when passing non-schema documents
- Memory leak in `RelaxNG()` when `RelaxNGParseError` is raised

## A.39 Changes in version 0.9.1, released 2006-03-30

### Features added

- `lxml.sax.ElementTreeContentHandler` checks closing elements and raises `SaxError` on mismatch
- `lxml.sax.ElementTreeContentHandler` supports namespace-less SAX events (`startElement`, `endElement`) and defaults to empty attributes (keyword argument)

- Speedup for repeatedly accessing element tag names
- Minor API performance improvements

### Bugs fixed

- Memory deallocation bug when using XSLT output method “html”
- sax.py was handling UTF-8 encoded tag names where it shouldn't
- lxml.tests package will no longer be installed (is still in source tar)

## A.40 Changes in version 0.9, released 2006-03-20

### Features added

- Error logging API for libxml2 error messages
- Various performance improvements
- Benchmark script for lxml, ElementTree and cElementTree
- Support for registering extension functions through new FunctionNamespace class (see doc/extensions.txt)
- ETXPath class for XPath expressions in ElementTree notation ('//{ns}tag')
- Support for variables in XPath expressions (also in XPath class)
- XPath class for compiled XPath expressions
- XMLID module level function (ElementTree compatible)
- XMLParser API for customized libxml2 parser configuration
- Support for custom Element classes through new Namespace API (see doc/namespace\_extensions.txt)
- Common exception base class LxmlError for module exceptions
- real iterator support in iter(Element), Element.getiterator()
- XSLT objects are callable, result trees support str()
- Added MANIFEST.in for easier creation of RPM files.
- 'getparent' method on elements allows navigation to an element's parent element.
- Python core compatible SAX tree builder and SAX event generator. See doc/sax.txt for more information.

### Bugs fixed

- Segfaults and memory leaks in various API functions of Element
- Segfault in XSLT.tostring()

- ElementTree objects no longer interfere, Elements can be root of different ElementTrees at the same time
- document("") works in XSLT documents read from files (in-memory documents cannot support this due to libxslt deficiencies)

## A.41 Changes in version 0.8, released 2005-11-03

### Features added

- Support for copy.deepcopy() on elements. copy.copy() works also, but does the same thing, and does *not* create a shallow copy, as that makes no sense in the context of libxml2 trees. This means a potential incompatibility with ElementTree, but there's more chance that it works than if copy.copy() isn't supported at all.
- Increased compatibility with (c)ElementTree; .parse() on ElementTree is supported and parsing of gzipped XML files works.
- implemented index() on elements, allowing one to find the index of a SubElement.

### Bugs fixed

- Use xslt-config instead of xml2-config to find out libxml2 directories to take into account a case where libxslt is installed in a different directory than libxml2.
- Eliminate crash condition in iteration when text nodes are changed.
- Passing 'None' to tostring() does not result in a segfault anymore, but an AssertionError.
- Some test fixes for Windows.
- Raise XMLSyntaxError and XPathSyntaxError instead of plain python syntax errors. This should be less confusing.
- Fixed error with uncaught exception in Pyrex code.
- Calling lxml.etree.fromstring("") throws XMLSyntaxError instead of a segfault.
- has\_key() works on attrib. 'in' tests also work correctly on attrib.
- INSTALL.txt was saying 2.2.16 instead of 2.6.16 as a supported libxml2 version, as it should.
- Passing a UTF-8 encoded string to the XML() function would fail; fixed.

## A.42 Changes in version 0.7, released 2005-06-15

### Features added

- parameters (XPath expressions) can be passed to XSLT using keyword parameters.
- Simple XInclude support. Calling the xinclude() method on a tree will process any XInclude statements in the document.

- XMLSchema support. Use the XMLSchema class or the convenience `xmlschema()` method on a tree to do XML Schema (XSD) validation.
- Added convenience `xslt()` method on tree. This is less efficient than the XSLT object, but makes it easier to write quick code.
- Added convenience `relaxng()` method on tree. This is less efficient than the RelaxNG object, but makes it easier to write quick code.
- Make it possible to use `XPathEvaluator` with elements as well. The `XPathEvaluator` in this case will retain the element so multiple XPath queries can be made against one element efficiently. This replaces the second argument to the `.evaluate()` method that existed previously.
- Allow `registerNamespace()` to be called on an `XPathEvaluator`, after creation, to add additional namespaces. Also allow `registerNamespaces()`, which does the same for a namespace dictionary.
- Add 'prefix' attribute to element to be able to read prefix information. This is entirely read-only.
- It is possible to supply an extra `nsmap` keyword parameter to the `Element()` and `SubElement()` constructors, which supplies a prefix to namespace URI mapping. This will create namespace prefix declarations on these elements and these prefixes will show up in XML serialization.

## Bugs fixed

- Killed yet another memory management related bug: trees created using `newDoc` would not get a `libxml2`-level dictionary, which caused problems when deallocating these documents later if they contained a node that came from a document with a dictionary.
- Moving namespaced elements between documents was problematic as references to the original document would remain. This has been fixed by applying `xmlReconciliateNs()` after each move operation.
- Can pass `None` to `'dump()'` without segfaults.
- `tostring()` works properly for non-root elements as well.
- Cleaned out the `tostring()` method so it should handle encoding correctly.
- Cleaned out the `ElementTree.write()` method so it should handle encoding correctly. Writing directly to a file should also be faster, as there is no need to go through a Python string in that case. Made sure the test cases test both serializing to `StringIO` as well as serializing to a real file.

## A.43 Changes in version 0.6, released 2005-05-14

### Features added

- Changed `setup.py` so that `library_dirs` is also guessed. This should help with compilation on the Mac OS X platform, where otherwise the wrong library (shipping with the OS) could be picked up.
- Tweaked `setup.py` so that it picks up the version from `version.txt`.

## **Bugs fixed**

- Do the right thing when handling namespaced attributes.
- fix bug where tostring() moved nodes into new documents. tostring() had very nasty side-effects before this fix, sorry!

## **A.44 Changes in version 0.5.1, released 2005-04-09**

- Python 2.2 compatibility fixes.
- unicode fixes in Element() and Comment() as well as XML(); unicode input wasn't properly being UTF-8 encoded.

## **A.45 Changes in version 0.5, released 2005-04-08**

Initial public release.

## Appendix B

# Generated API documentation

## B.1 Package lxml

### B.1.1 Modules

- **ElementInclude**: Limited XInclude support for the ElementTree package.  
(Section B.2, p. 194)
- **builder**: The E Element factory for generating XML documents.  
(Section B.3, p. 197)
- **cssselect**: CSS Selectors based on XPath.  
(Section B.4, p. 200)
- **doctestcompare**: lxml-based doctest output comparison.  
(Section B.5, p. 204)
- **etree**: The `lxml.etree` module implements the extended ElementTree API for XML.  
(Section B.6, p. 208)
- **html**: The `lxml.html` tool set for HTML handling.  
(Section B.7, p. 335)
  - **ElementSoup**: Legacy interface to the BeautifulSoup HTML parser.  
(Section B.8, p. 339)
  - **\_dictmixin** (Section ??, p. ??)
  - **\_setmixin** (Section ??, p. ??)
  - **builder**: A set of HTML generator tags for building HTML documents.  
(Section B.9, p. 340)
  - **clean**: A cleanup tool for HTML.  
(Section B.10, p. 344)
  - **defs** (Section B.11, p. 348)
  - **diff** (Section B.12, p. 349)
  - **formfill** (Section B.13, p. 350)
  - **soupparser**: External interface to the BeautifulSoup HTML parser.  
(Section B.14, p. 353)
  - **usedoctest**: Doctest module for HTML comparison.  
(Section B.15, p. 354)
- **objectify**: The `lxml.objectify` module implements a Python object API for XML.  
(Section B.16, p. 355)
- **pyclasslookup**: A whole-tree Element class lookup scheme for `lxml.etree`.  
(Section B.17, p. 387)
- **sax**: SAX-based adapter to copy trees from/to the Python standard library.  
(Section B.18, p. 389)
- **usedoctest**: Doctest module for XML comparison.  
(Section B.19, p. 396)

## B.2 Module `lxml.ElementInclude`

Limited XInclude support for the ElementTree package.

While `lxml.etree` has full support for XInclude (see `etree.ElementTree.xinclude()`), this module provides a simpler, pure Python, ElementTree compatible implementation that supports a simple form of custom URL resolvers.

### B.2.1 Functions

```
default_loader(href, parse, encoding=None)
```

```
include(elem, loader=None, base_url=None)
```

### B.2.2 Variables

Name	Description
XINCLUDE	<b>Value:</b> <code>'{http://www.w3.org/2001/XInclude}'</code>
XINCLUDE_INCLUDE	<b>Value:</b> <code>'{http://www.w3.org/2001/XInclude}include'</code>
XINCLUDE_FALLBACK	<b>Value:</b> <code>'{http://www.w3.org/2001/XInclude}fallback'</code>



---

Name	Description
<i>Inherited from exceptions.SyntaxError</i>	filename, lineno, message, msg, offset, print_file_and_line, text
<i>Inherited from exceptions.BaseException</i>	args
<i>Inherited from object</i>	__class__

## B.3 Module `lxml.builder`

The `E` Element factory for generating XML documents.

### B.3.1 Variables

Name	Description
<code>E</code>	<b>Value:</b> <code>ElementMaker()</code>

### B.3.2 Class `ElementMaker`



Element generator factory.

Unlike the ordinary `Element` factory, the `E` factory allows you to pass in more than just a tag and some optional attributes; you can also pass in text and other elements. The text is added as either text or tail attributes, and elements are inserted at the right spot. Some small examples:

```

>>> from lxml import etree as ET
>>> from lxml.builder import E

>>> ET.tostring(E("tag"))
'<tag/>'
>>> ET.tostring(E("tag", "text"))
'<tag>text</tag>'
>>> ET.tostring(E("tag", "text", key="value"))
'<tag key="value">text</tag>'
>>> ET.tostring(E("tag", E("subtag", "text"), "tail"))
'<tag><subtag>text</subtag>tail</tag>'
  
```

For simple tags, the factory also allows you to write `E.tag(...)` instead of `E('tag', ...)`:

```

>>> ET.tostring(E.tag())
'<tag/>'
>>> ET.tostring(E.tag("text"))
'<tag>text</tag>'
>>> ET.tostring(E.tag(E.subtag("text"), "tail"))
'<tag><subtag>text</subtag>tail</tag>'
  
```

Here's a somewhat larger example; this shows how to generate HTML documents, using a mix of prepared factory functions for inline elements, nested `E.tag` calls, and embedded

XHTML fragments:

```
# some common inline elements
A = E.a
I = E.i
B = E.b

def CLASS(v):
    # helper function, 'class' is a reserved word
    return {'class': v}

page = (
    E.html(
        E.head(
            E.title("This is a sample document")
        ),
        E.body(
            E.h1("Hello!", CLASS("title")),
            E.p("This is a para-
graph with ", B("bold"), " text in it!"),
            E.p("This is another paragraph, with a ",
                A("link", href="http://www.python.org"), "."),
            E.p("Here are some reserved charac-
ters: <spam&egg>."),
            ET.XML("<p>And finally, here is an embed-
ded XHTML fragment.</p>"),
        )
    )
)

print ET.tostring(page)
```

Here's a prettyprinted version of the output from the above script:

```
<html>
  <head>
    <title>This is a sample document</title>
  </head>
  <body>
    <h1 class="title">Hello!</h1>
    <p>This is a paragraph with <b>bold</b> text in it!</p>
    <p>This is another para-
graph, with <a href="http://www.python.org">link</a>.</p>
    <p>Here are some reserved charac-
ters: &lt;spam&amp;egg&gt;.</p>
    <p>And finally, here is an embedded XHTML fragment.</p>
  </body>
</html>
```

**Methods**

<b><code>__init__</code></b> ( <i>self</i> , <i>typemap</i> =None, <i>namespace</i> =None, <i>nsmap</i> =None, <i>makeelement</i> =None)
---

<i>x</i> . <b><code>__init__</code></b> (...) initializes <i>x</i> ; see <i>x</i> . <b><code>__class__</code></b> . <b><code>__doc__</code></b> for signature Overrides: object. <b><code>__init__</code></b> <b><code>__exit__</code></b> (inherited documentation)
---

<b><code>__call__</code></b> ( <i>self</i> , <i>tag</i> , * <i>children</i> , ** <i>attrib</i> )
--

<b><code>__getattr__</code></b> ( <i>self</i> , <i>tag</i> )
--

***Inherited from object***

**`__delattr__`**(*self*), **`__getattr__`**(*self*), **`__hash__`**(*self*), **`__new__`**(*self*), **`__reduce__`**(*self*), **`__reduce_ex__`**(*self*),  
**`__repr__`**(*self*), **`__setattr__`**(*self*), **`__str__`**(*self*)

**Properties**

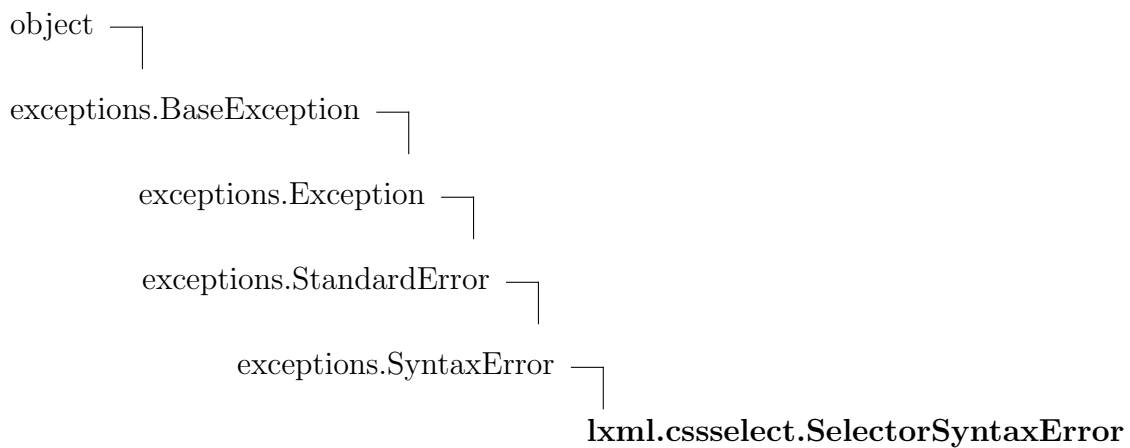
Name	Description
<i>Inherited from object</i>	
<b><code>__class__</code></b>	

## B.4 Module `lxml.cssselect`

CSS Selectors based on XPath.

This module supports selecting XML/HTML tags based on CSS selectors. See the `CSS-Selector` class for details.

### B.4.1 Class `SelectorSyntaxError`



#### Methods

##### *Inherited from `exceptions.SyntaxError`*

`__init__()`, `__new__()`, `__str__()`

##### *Inherited from `exceptions.BaseException`*

`__delattr__()`, `__getattr__()`, `__getitem__()`, `__getslice__()`, `__reduce__()`, `__repr__()`, `__setattr__()`, `__setstate__()`

##### *Inherited from `object`*

`__hash__()`, `__reduce_ex__()`

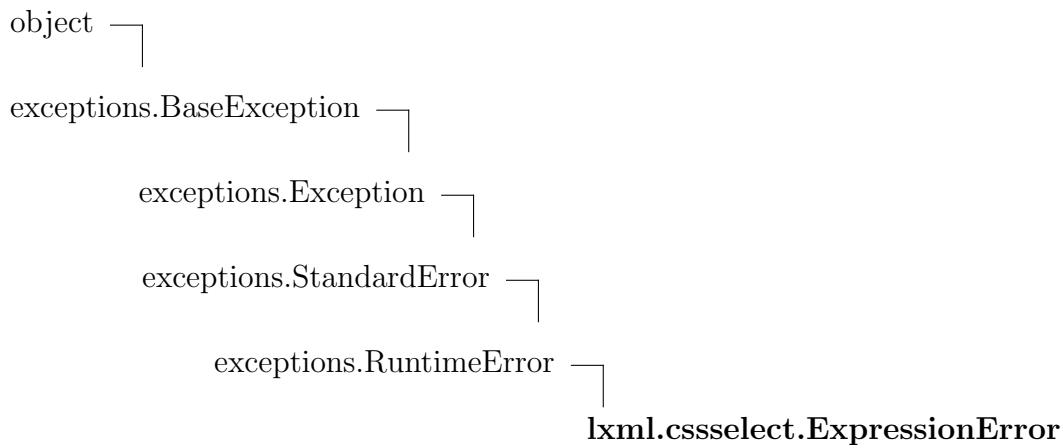
#### Properties

Name	Description
<i>Inherited from <code>exceptions.SyntaxError</code></i>	
<code>filename</code> , <code>lineno</code> , <code>message</code> , <code>msg</code> , <code>offset</code> , <code>print_file_and_line</code> , <code>text</code>	
<i>Inherited from <code>exceptions.BaseException</code></i>	
<code>args</code>	

*continued on next page*

Name	Description
<i>Inherited from object</i>	
<code>__class__</code>	

### B.4.2 Class `ExpressionError`



#### Methods

##### *Inherited from exceptions.RuntimeError*

`__init__()`, `__new__()`

##### *Inherited from exceptions.BaseException*

`__delattr__()`, `__getattr__()`, `__getitem__()`, `__getslice__()`, `__reduce__()`, `__repr__()`,  
`__setattr__()`, `__setstate__()`, `__str__()`

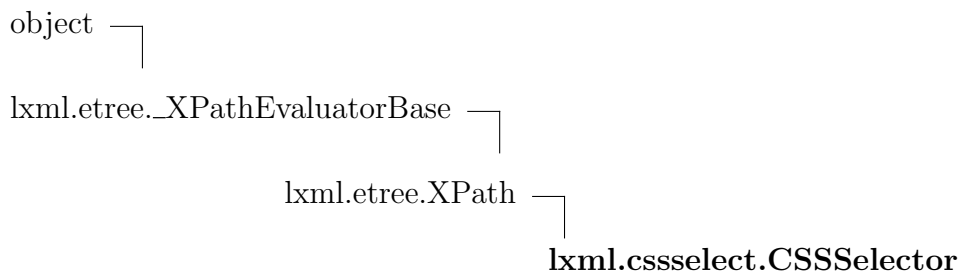
##### *Inherited from object*

`__hash__()`, `__reduce_ex__()`

#### Properties

Name	Description
<i>Inherited from exceptions.BaseException</i>	
args, message	
<i>Inherited from object</i>	
<code>__class__</code>	

### B.4.3 Class CSSSelector



A CSS selector.

Usage:

```

>>> from lxml import etree, cssselect
>>> select = cssselect.CSSSelector("a tag > child")

>>> root = etree.XML("<a><b><c></tag><child>TEXT</child></tag></b></a>")
>>> [ el.tag for el in select(root) ]
['child']
  
```

#### Methods

<p><b><code>__init__(self, css)</code></b>  <code>x.__init__(...)</code> initializes <code>x</code>; see <code>x.__class__.__doc__</code> for signature. Overrides: <code>object.__init__</code> extit(inherited documentation)</p>
---

<p><b><code>__repr__(self)</code></b>  <code>repr(x)</code> Overrides: <code>object.__repr__</code> extit(inherited documentation)</p>
--

**Inherited from `lxml.etree.XPath` (Section [B.6.60](#))**

`__call__()`, `__new__()`

**Inherited from `lxml.etree.XPathEvaluatorBase`**

`evaluate()`

**Inherited from `object`**

`__delattr__()`, `__getattr__()`, `__hash__()`, `__reduce__()`, `__reduce_ex__()`, `__setattr__()`, `__str__()`

#### Properties

---

Name	Description
<i>Inherited from lxml.etree.XPath (Section <a href="#">B.6.60</a>)</i> path	
<i>Inherited from lxml.etree._XPathEvaluatorBase</i> error_log	
<i>Inherited from object</i> __class__	

## B.5 Module `lxml.doctestcompare`

lxml-based doctest output comparison.

Note: normally, you should just import the `lxml.usedoctest` and `lxml.html.usedoctest` modules from within a doctest, instead of this one:

```
>>> import lxml.usedoctest # for XML output
```

```
>>> import lxml.html.usedoctest # for HTML output
```

To use this module directly, you must call `lxml.doctest.install()`, which will cause doctest to use this in all subsequent calls.

This changes the way output is checked and comparisons are made for XML or HTML-like content.

XML or HTML content is noticed because the example starts with `<` (it's HTML if it starts with `<html>`). You can also use the `PARSE_HTML` and `PARSE_XML` flags to force parsing.

Some rough wildcard-like things are allowed. Whitespace is generally ignored (except in attributes). In text (attributes and text in the body) you can use `...` as a wildcard. In an example it also matches any trailing tags in the element, though it does not match leading tags. You may create a tag `<any>` or include an `any` attribute in the tag. An `any` tag matches any tag, while the attribute matches any and all attributes.

When a match fails, the reformatted example and gotten text is displayed (indented), and a rough diff-like output is given. Anything marked with `-` is in the output but wasn't supposed to be, and similarly `+` means its in the example but wasn't in the output.

You can disable parsing on one line with `# doctest:+NOPARSE_MARKUP`

### B.5.1 Functions

<b><code>install(html=False)</code></b>
Install doctestcompare for all future doctests.
If <code>html</code> is true, then by default the HTML parser will be used; otherwise the XML parser is used.

```
temp_install(html=False, del_module=None)
```

Use this *inside* a doctest to enable this checker for this doctest only.

If `html` is true, then by default the HTML parser will be used; otherwise the XML parser is used.

### B.5.2 Variables

Name	Description
<code>PARSE_HTML</code>	<b>Value:</b> 1024
<code>PARSE_XML</code>	<b>Value:</b> 2048
<code>NOPARSE_MARKUP</code>	<b>Value:</b> 4096

### B.5.3 Class `LXMLOutputChecker`

```
doctest.OutputChecker └─
                        |
                        └─ lxml.doctestcompare.LXMLOutputChecker
```

**Known Subclasses:** `lxml.doctestcompare.LHTMLOutputChecker`

#### Methods

```
get_default_parser(self)
```

```
check_output(self, want, got, optionflags)
```

Return True iff the actual output from an example (`got`) matches the expected output (`want`). These strings are always considered to match if they are identical; but depending on what option flags the test runner is using, several non-exact match types are also possible. See the documentation for `TestRunner` for more information about option flags. Overrides: `doctest.OutputChecker.check_output` `exitit`(inherited documentation)

```
get_parser(self, want, got, optionflags)
```

```
compare_docs(self, want, got)
```

```
text_compare(self, want, got, strip)
```

```
tag_compare(self, want, got)
```

```
output_difference(self, example, got, optionflags)
```

Return a string describing the differences between the expected output for a given example (*example*) and the actual output (*got*). *optionflags* is the set of option flags used to compare *want* and *got*. Overrides: `doctest.OutputChecker.output_difference` `exitit`(inherited documentation)

```
html_empty_tag(self, el, html=True)
```

```
format_doc(self, doc, html, indent, prefix='')
```

```
format_text(self, text, strip=True)
```

```
format_tag(self, el)
```

```
format_end_tag(self, el)
```

```
collect_diff(self, want, got, html, indent)
```

```
collect_diff_tag(self, want, got)
```

```
collect_diff_end_tag(self, want, got)
```

```
collect_diff_text(self, want, got, strip=True)
```

#### Class Variables

Name	Description
<code>empty_tags</code>	<b>Value:</b> ('param', 'img', 'area', 'br', 'basefont', 'input', 'base...')

#### B.5.4 Class LHTMLOutputChecker

```
doctest.OutputChecker
```

```
lxml.doctestcompare.LXMLOutputChecker
```

```
lxml.doctestcompare.LHTMLOutputChecker
```

**Methods**

<code>get_default_parser(<i>self</i>)</code>
--

Overrides: <code>lxml.doctestcompare.LXMLOutputChecker.get_default_parser</code>
--

*Inherited from `lxml.doctestcompare.LXMLOutputChecker` (Section [B.5.3](#))*

`check_output()`, `collect_diff()`, `collect_diff_end_tag()`, `collect_diff_tag()`, `collect_diff_text()`, `compare_docs()`, `format_doc()`, `format_end_tag()`, `format_tag()`, `format_text()`, `get_parser()`, `html_empty_tag()`, `output_difference()`, `tag_compare()`, `text_compare()`

**Class Variables**

Name	Description
<i>Inherited from <code>lxml.doctestcompare.LXMLOutputChecker</code> (Section <a href="#">B.5.3</a>)</i>	
<code>empty_tags</code>	

## B.6 Module `lxml.etree`

The `lxml.etree` module implements the extended ElementTree API for XML. **Version:** 2.0.10-60473

### B.6.1 Functions

**Comment**(*text=None*)

Comment element factory. This factory function creates a special element that will be serialized as an XML comment.

**Element**(*\_tag*, *attrib=None*, *nsmmap=None*, *\*\*\_extra*)

Element factory. This function returns an object implementing the Element interface.

**ElementTree**(*element=None*, *file=None*, *parser=None*)

ElementTree wrapper class.

**Entity**(*name*)

Entity factory. This factory function creates a special element that will be serialized as an XML entity reference or character reference. Note, however, that entities will not be automatically declared in the document. A document that uses entity references requires a DTD to define the entities.

**Extension**(*module*, *function\_mapping=None*, *ns=None*)

Build a dictionary of extension functions from the functions defined in a module or the methods of an object.

As second argument, you can pass an additional mapping of attribute names to XPath function names, or a list of function names that should be taken.

The `ns` keyword argument accepts a namespace URI for the XPath functions.

**FunctionNamespace**(*ns\_uri*)

Retrieve the function namespace object associated with the given URI.

Creates a new one if it does not yet exist. A function namespace can only be used to register extension functions.

**HTML**(*text*, *parser=None*, *base\_url=None*)

Parses an HTML document from a string constant. This function can be used to embed “HTML literals” in Python code.

To override the parser with a different `HTMLParser` you can pass it to the `parser` keyword argument.

The `base_url` keyword argument allows to set the original base URL of the document to support relative Paths when looking up external entities (DTD, XInclude, ...).

**PI**(*target*, *text=None*)

ProcessingInstruction element factory. This factory function creates a special element that will be serialized as an XML processing instruction.

**ProcessingInstruction**(*target*, *text=None*)

ProcessingInstruction element factory. This factory function creates a special element that will be serialized as an XML processing instruction.

**SubElement**(*\_parent*, *\_tag*, *attrib=None*, *nsmmap=None*, *\*\*\_extra*)

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

---

**XML**(*text*, *parser*=None, *base\_url*=None)

Parses an XML document from a string constant. This function can be used to embed “XML literals” in Python code, like in

```
>>> root = etree.XML("<root><test/></root>")
```

To override the parser with a different `XMLParser` you can pass it to the `parser` keyword argument.

The `base_url` keyword argument allows to set the original base URL of the document to support relative Paths when looking up external entities (DTD, XInclude, ...).

---

**XMLDTDID**(*text*)

Parse the text and return a tuple (root node, ID dictionary). The root node is the same as returned by the `XML()` function. The dictionary contains string-element pairs. The dictionary keys are the values of ID attributes as defined by the DTD. The elements referenced by the ID are stored as dictionary values.

Note that you must not modify the XML tree if you use the ID dictionary. The results are undefined.

---

**XMLID**(*text*)

Parse the text and return a tuple (root node, ID dictionary). The root node is the same as returned by the `XML()` function. The dictionary contains string-element pairs. The dictionary keys are the values of 'id' attributes. The elements referenced by the ID are stored as dictionary values.

---

**XPathEvaluator**(*etree\_or\_element*, *namespaces*=None, *extensions*=None, *regexp*=True)

Creates an XPath evaluator for an `ElementTree` or an `Element`.

The resulting object can be called with an XPath expression as argument and XPath variables provided as keyword arguments.

Additional namespace declarations can be passed with the 'namespace' keyword argument. EXSLT regular expression support can be disabled with the 'regexp' boolean keyword (defaults to True).

**clearErrorLog(...)**

Clear the global error log. Note that this log is already bound to a fixed size. **Deprecated:** use `clear_error_log()` instead.

**clear\_error\_log()**

Clear the global error log. Note that this log is already bound to a fixed size.

**dump(*elem*, *pretty\_print*=True, *with\_tail*=True)**

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

**fromstring(*text*, *parser*=None, *base\_url*=None)**

Parses an XML document from a string.

To override the default parser with a different parser you can pass it to the `parser` keyword argument.

The `base_url` keyword argument allows to set the original base URL of the document to support relative Paths when looking up external entities (DTD, XInclude, ...).

**fromstringlist(*strings*, *parser*=None)**

Parses an XML document from a sequence of strings.

To override the default parser with a different parser you can pass it to the `parser` keyword argument.

**getDefaultParser(...)**

**Deprecated:** please use `get_default_parser` instead.

**get\_default\_parser()**

**iselement**(*element*)

Checks if an object appears to be a valid element object.

**parse**(*source*, *parser=None*, *base\_url=None*)

Return an `ElementTree` object loaded with source elements. If no parser is provided as second argument, the default parser is used.

The `base_url` keyword allows setting a URL for the document when parsing from a file-like object. This is needed when looking up external entities (DTD, XInclude, ...) with relative paths.

**parseid**(*source*, *parser=None*)

Parses the source into a tuple containing an `ElementTree` object and an ID dictionary. If no parser is provided as second argument, the default parser is used.

Note that you must not modify the XML tree if you use the ID dictionary. The results are undefined.

**setDefaultParser**(...)

**Deprecated:** please use `set_default_parser` instead.

**setElementClassLookup**(...)

**Deprecated:** use `set_element_class_lookup(lookup)` instead

**set\_default\_parser**(*parser=None*)

Set a default parser for the current thread. This parser is used globally whenever no parser is supplied to the various parse functions of the `lxml` API. If this function is called without a parser (or if it is `None`), the default parser is reset to the original configuration.

Note that the pre-installed default parser is not thread-safe. Avoid the default parser in multi-threaded environments. You can create a separate parser for each thread explicitly or use a parser pool.

```
set_element_class_lookup(lookup=None)
```

Set the global default element class lookup method.

```
tostring(element_or_tree, encoding=None, method="xml",  
xml_declaration=None, pretty_print=False, with_tail=True)
```

Serialize an element to an encoded string representation of its XML tree.

Defaults to ASCII encoding without XML declaration. This behaviour can be configured with the keyword arguments 'encoding' (string) and 'xml\_declaration' (bool). Note that changing the encoding to a non UTF-8 compatible encoding will enable a declaration by default.

You can also serialise to a Unicode string without declaration by passing the `unicode` function as encoding.

The keyword argument 'pretty\_print' (bool) enables formatted XML.

The keyword argument 'method' selects the output method: 'xml', 'html' or plain 'text'.

You can prevent the tail text of the element from being serialised by passing the boolean `with_tail` option. This has no impact on the tail text of children, which will always be serialised.

```
tostringlist(element_or_tree, *args, **kwargs)
```

Serialize an element to an encoded string representation of its XML tree, stored in a list of partial strings.

This is purely for ElementTree 1.3 compatibility. The result is a single string wrapped in a list.

**tounicode**(*element\_or\_tree*, *method*="xml", *pretty\_print*=False, *with\_tail*=True)

Serialize an element to the Python unicode representation of its XML tree.

Note that the result does not carry an XML encoding declaration and is therefore not necessarily suited for serialization to byte streams without further treatment.

The boolean keyword argument 'pretty\_print' enables formatted XML.

The keyword argument 'method' selects the output method: 'xml', 'html' or plain 'text'.

You can prevent the tail text of the element from being serialised by passing the boolean `with_tail` option. This has no impact on the tail text of children, which will always be serialised. **Deprecated:** use `tostring(el, encoding=unicode)` instead.

**useGlobalPythonLog**(...)

Replace the global error log by an `etree.PyErrorLog` that uses the standard Python logging package.

Note that this disables access to the global error log from exceptions. Parsers, XSLT etc. will continue to provide their normal local error log. **Deprecated:** use `use_global_python_log()` instead.

**use\_global\_python\_log**(*log*)

Replace the global error log by an `etree.PyErrorLog` that uses the standard Python logging package.

Note that this disables access to the global error log from exceptions. Parsers, XSLT etc. will continue to provide their normal local error log.

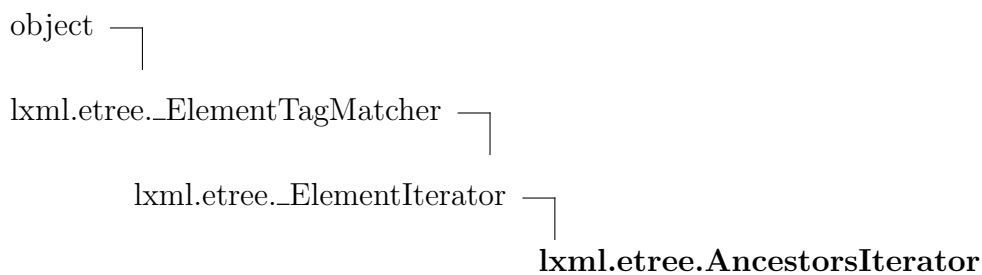
## B.6.2 Variables

Name	Description
DEBUG	<b>Value:</b> 1
LIBXML_COMPILED_VERSION	<b>Value:</b> (2, 6, 30)

*continued on next page*

Name	Description
LIBXML_VERSION	<b>Value:</b> (2, 6, 30)
LIBXSLT_COMPILED_VERSION	<b>Value:</b> (1, 1, 21)
LIBXSLT_VERSION	<b>Value:</b> (1, 1, 21)
LXML_VERSION	<b>Value:</b> (2, 0, 10, 60473)
<code>__pyx_capi__</code>	<b>Value:</b> {'appendChild': <PyCObject object at 0x849d758>, 'attribu...

### B.6.3 Class *AncestorsIterator*



*AncestorsIterator*(self, node, tag=None) Iterates over the ancestors of an element (from parent to parent).

#### Methods

<code>__init__(self, node, tag=None)</code>
<hr/>
x. <code>__init__</code> (...) initializes x; see x. <code>__class__</code> . <code>__doc__</code> for signature Overrides: object. <code>__init__</code>

<code>__new__(T, S, ...)</code>
<b>Return Value</b>
a new object with type S, a subtype of T
Overrides: object. <code>__new__</code>

#### *Inherited from lxml.etree.\_ElementIterator*

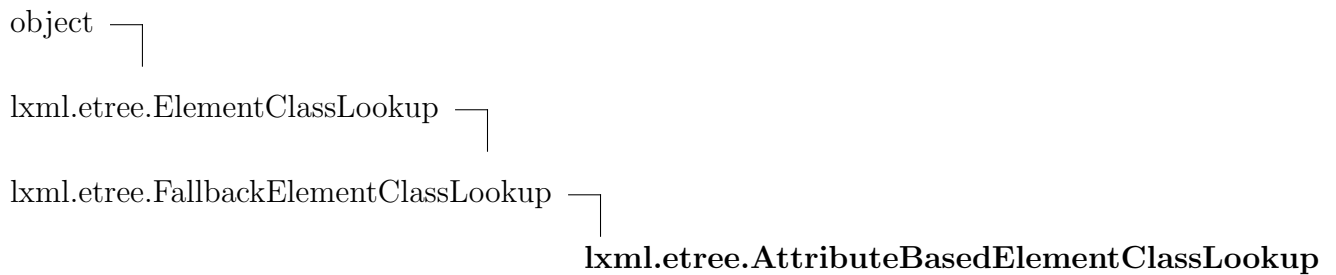
`__iter__`(), `__next__`(), `next`()

#### *Inherited from object*

`__delattr__`(), `__getattr__`(), `__hash__`(), `__reduce__`(), `__reduce_ex__`(), `__repr__`(), `__setattr__`(), `__str__`()

**Properties**

Name	Description
<i>Inherited from object</i>	
<code>__class__</code>	

**B.6.4 Class *AttributeBasedElementClassLookup***

`AttributeBasedElementClassLookup(self, attribute_name, class_mapping, fallback=None)`  
 Checks an attribute of an `Element` and looks up the value in a class dictionary.

**Arguments:** • attribute name - '{ns}name' style string

- class mapping - Python dict mapping attribute values to `Element` classes
- fallback - optional fallback lookup mechanism

A `None` key in the class mapping will be checked if the attribute is missing.

**Methods**

<code>__init__(self, attribute_name, class_mapping, fallback=None)</code>
<code>x.__init__(...)</code> initializes x; see <code>x.__class__.__doc__</code> for signature Overrides: <code>object.__init__</code>

<code>__new__(T, S, ...)</code>
<b>Return Value</b> a new object with type S, a subtype of T
Overrides: <code>object.__new__</code>

*Inherited from `lxml.etree.FallbackElementClassLookup` (Section [B.6.28](#))*

`setFallback()`, `set_fallback()`

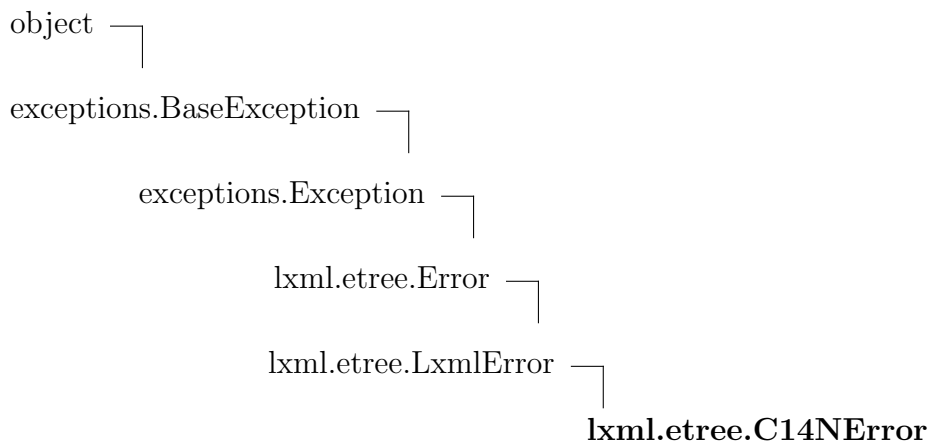
*Inherited from `object`*

`__delattr__()`, `__getattr__()`, `__hash__()`, `__reduce__()`, `__reduce_ex__()`, `__repr__()`,  
`__setattr__()`, `__str__()`

### Properties

Name	Description
<code>fallback</code>	<i>Inherited from <code>lxml.etree.FallbackElementClassLookup</code> (Section <a href="#">B.6.28</a>)</i>
<code>__class__</code>	<i>Inherited from <code>object</code></i>

### B.6.5 Class C14NError



Error during C14N serialisation.

### Methods

*Inherited from `lxml.etree.LxmlError`(Section [B.6.30](#))*

`__init__()`

*Inherited from `exceptions.Exception`*

`__new__()`

*Inherited from `exceptions.BaseException`*

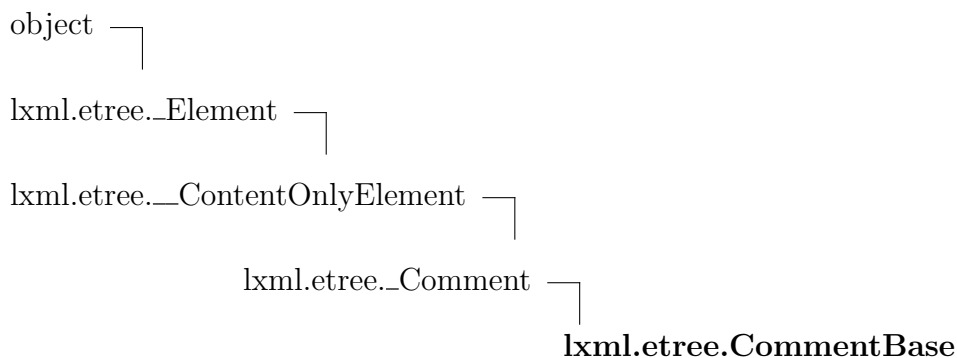
`__delattr__()`, `__getattr__()`, `__getitem__()`, `__getslice__()`, `__reduce__()`, `__repr__()`,  
`__setattr__()`, `__setstate__()`, `__str__()`

*Inherited from `object`*

`__hash__()`, `__reduce_ex__()`

**Properties**

Name	Description
	<i>Inherited from <code>exceptions.BaseException</code></i> args, message
	<i>Inherited from object</i> <code>__class__</code>

**B.6.6 Class `CommentBase`**

**Known Subclasses:** `lxml.html.HtmlComment`

All custom Comment classes must inherit from this one.

Note that you cannot (and must not) instantiate this class or its subclasses.

Subclasses *must not* override `__init__` or `__new__` as it is absolutely undefined when these objects will be created or destroyed. All persistent state of Comments must be stored in the underlying XML. If you really need to initialize the object after creation, you can implement an `_init(self)` method that will be called after object creation.

**Methods**

<code>__new__(T, S, ...)</code>
<b>Return Value</b> a new object with type S, a subtype of T
Overrides: <code>object.__new__</code>

*Inherited from `lxml.etree._Comment`*

`__repr__()`

*Inherited from `lxml.etree._ContentOnlyElement`*

`__delitem__()`, `__getitem__()`, `__len__()`, `__setitem__()`, `append()`, `get()`, `insert()`,

items(), keys(), set(), values()

### ***Inherited from lxml.etree.\_Element***

`__contains__()`, `__copy__()`, `__deepcopy__()`, `__iter__()`, `__nonzero__()`, `__reversed__()`, `addnext()`, `addprevious()`, `clear()`, `extend()`, `find()`, `findall()`, `findtext()`, `getchildren()`, `getiterator()`, `getnext()`, `getparent()`, `getprevious()`, `getroottree()`, `index()`, `iter()`, `iterancestors()`, `iterchildren()`, `iterdescendants()`, `iterfind()`, `itersiblings()`, `itertext()`, `makeelement()`, `remove()`, `replace()`, `xpath()`

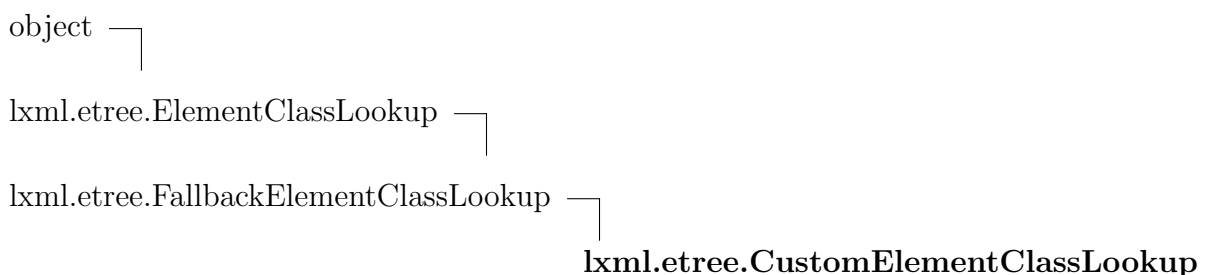
### ***Inherited from object***

`__delattr__()`, `__getattr__()`, `__hash__()`, `__init__()`, `__reduce__()`, `__reduce_ex__()`, `__setattr__()`, `__str__()`

### **Properties**

Name	Description
<i>Inherited from lxml.etree._Comment</i> tag	
<i>Inherited from lxml.etree._ContentOnlyElement</i> attrib, text	
<i>Inherited from lxml.etree._Element</i> nsmmap, prefix, sourceline, tail	
<i>Inherited from object</i> __class__	

## **B.6.7 Class CustomElementClassLookup**



### **Known Subclasses:** lxml.html.HtmlElementClassLookup

CustomElementClassLookup(self, fallback=None) Element class lookup based on a subclass method.

You can inherit from this class and override the method:

```
lookup(self, type, doc, namespace, name)
```

to lookup the element class for a node. Arguments of the method: \* type: one of 'element', 'comment', 'PI', 'entity' \* doc: document that the node is in \* namespace: namespace URI of the node (or None for comments/Pis/entities) \* name: name of the element/entity, None for comments, target for PIs

If you return None from this method, the fallback will be called.

## Methods

```
__new__(T, S, ...)
```

### Return Value

a new object with type S, a subtype of T

Overrides: object.\_\_new\_\_

```
lookup(self, type, doc, namespace, name)
```

*Inherited from lxml.etree.FallbackElementClassLookup (Section B.6.28)*

\_\_init\_\_(), setFallback(), set\_fallback()

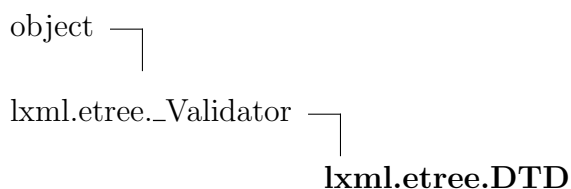
*Inherited from object*

\_\_delattr\_\_(), \_\_getattr\_\_(), \_\_hash\_\_(), \_\_reduce\_\_(), \_\_reduce\_ex\_\_(), \_\_repr\_\_(),  
\_\_setattr\_\_(), \_\_str\_\_()

## Properties

Name	Description
<i>Inherited from lxml.etree.FallbackElementClassLookup (Section B.6.28)</i> fallback	
<i>Inherited from object</i> __class__	

## B.6.8 Class DTD



DTD(self, file=None, external\_id=None) A DTD validator.

Can load from filesystem directly given a filename or file-like object. Alternatively, pass

the keyword parameter `external_id` to load from a catalog.

## Methods

<code>__call__(self, etree)</code>
Validate doc using the DTD. Returns true if the document is valid, false if not.

<code>__init__(self, file=None, external_id=None)</code>
<code>x.__init__(...)</code> initializes x; see <code>x.__class__.__doc__</code> for signature Overrides: <code>object.__init__</code>

<code>__new__(T, S, ...)</code>
<b>Return Value</b> a new object with type S, a subtype of T Overrides: <code>object.__new__</code>

### *Inherited from `lxml.etree._Validator`*

`assertValid()`, `assert_()`, `validate()`

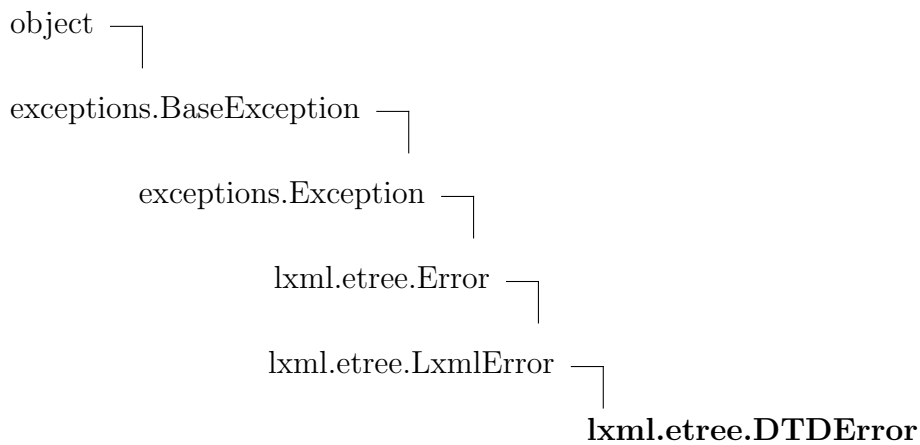
### *Inherited from `object`*

`__delattr__()`, `__getattr__()`, `__hash__()`, `__reduce__()`, `__reduce_ex__()`, `__repr__()`, `__setattr__()`, `__str__()`

## Properties

Name	Description
<i>Inherited from <code>lxml.etree._Validator</code></i> <code>error_log</code>	
<i>Inherited from <code>object</code></i> <code>__class__</code>	

### B.6.9 Class DTDError



**Known Subclasses:** lxml.etree.DTDParseError, lxml.etree.DTDValidateError

Base class for DTD errors.

#### Methods

*Inherited from lxml.etree.LxmlError(Section B.6.30)*

`__init__()`

*Inherited from exceptions.Exception*

`__new__()`

*Inherited from exceptions.BaseException*

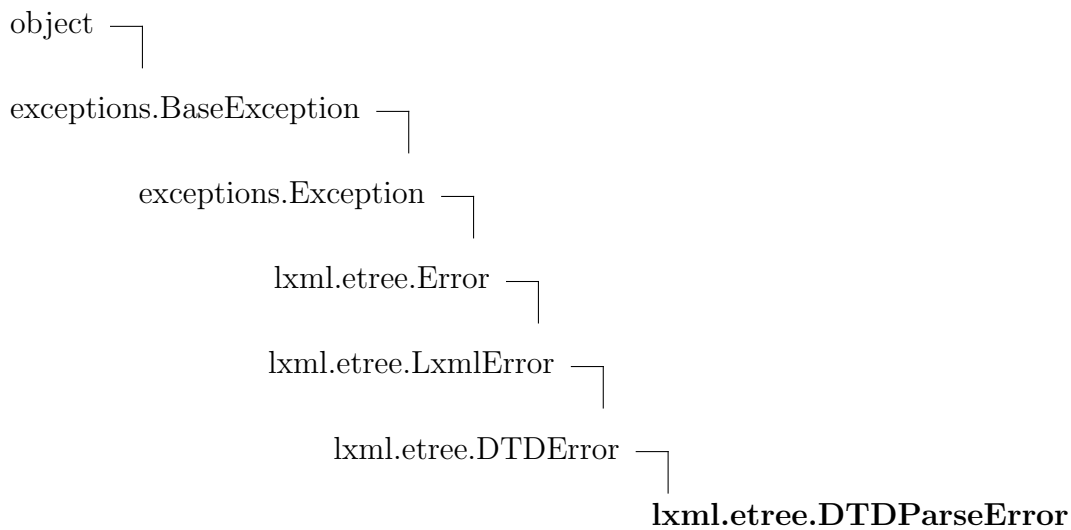
`__delattr__()`, `__getattr__()`, `__getitem__()`, `__getslice__()`, `__reduce__()`, `__repr__()`,  
`__setattr__()`, `__setstate__()`, `__str__()`

*Inherited from object*

`__hash__()`, `__reduce_ex__()`

#### Properties

Name	Description
<i>Inherited from exceptions.BaseException</i>	
args, message	
<i>Inherited from object</i>	
<code>__class__</code>	

**B.6.10 Class *DTDParseError***

Error while parsing a DTD.

**Methods**

*Inherited from [lxml.etree.LxmlError](#) (Section [B.6.30](#))*

`__init__()`

*Inherited from [exceptions.Exception](#)*

`__new__()`

*Inherited from [exceptions.BaseException](#)*

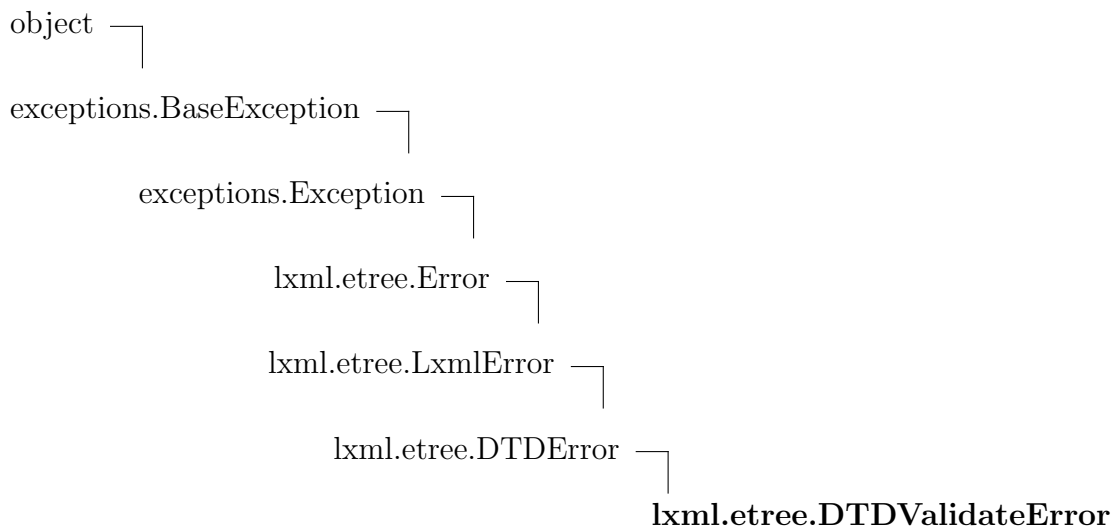
`__delattr__()`, `__getattr__()`, `__getitem__()`, `__getslice__()`, `__reduce__()`, `__repr__()`,  
`__setattr__()`, `__setstate__()`, `__str__()`

*Inherited from [object](#)*

`__hash__()`, `__reduce_ex__()`

**Properties**

Name	Description
<i>Inherited from <a href="#">exceptions.BaseException</a></i>	
	args, message
<i>Inherited from <a href="#">object</a></i>	
<code>__class__</code>	

**B.6.11 Class `DTDValidateError`**

Error while validating an XML document with a DTD.

**Methods**

*Inherited from `lxml.etree.LxmlError` (Section [B.6.30](#))*

`__init__()`

*Inherited from `exceptions.Exception`*

`__new__()`

*Inherited from `exceptions.BaseException`*

`__delattr__()`, `__getattr__()`, `__getitem__()`, `__getslice__()`, `__reduce__()`, `__repr__()`,  
`__setattr__()`, `__setstate__()`, `__str__()`

*Inherited from `object`*

`__hash__()`, `__reduce_ex__()`

**Properties**

Name	Description
<i>Inherited from <code>exceptions.BaseException</code></i>	
args, message	
<i>Inherited from <code>object</code></i>	
<code>__class__</code>	

### B.6.12 Class DocInfo



Document information provided by parser and DTD.

#### Methods

<b><code>__init__</code></b> (...)
<code>x.__init__</code> (...) initializes x; see <code>x.__class__.__doc__</code> for signature Overrides: <code>object.__init__</code>

<b><code>__new__</code></b> ( <i>T</i> , <i>S</i> , ...)
<b>Return Value</b> a new object with type <i>S</i> , a subtype of <i>T</i>
Overrides: <code>object.__new__</code>

#### Inherited from object

`__delattr__`(), `__getattr__`(), `__hash__`(), `__reduce__`(), `__reduce_ex__`(), `__repr__`(), `__setattr__`(), `__str__`()

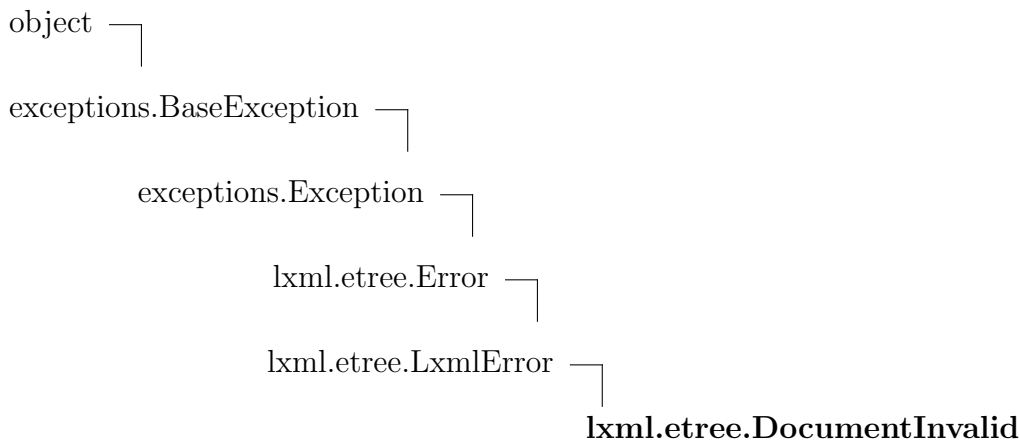
#### Properties

Name	Description
URL	Returns the source URL of the document (or None if unknown).
doctype	Returns a DOCTYPE declaration string for the document.
encoding	Returns the encoding name as declared by the document.
externalDTD	Returns a DTD validator based on the external subset of the document.
internalDTD	Returns a DTD validator based on the internal subset of the document.
public_id	Returns the public ID of the DOCTYPE.
root_name	Returns the name of the root node as defined by the DOCTYPE.
system_url	Returns the system ID of the DOCTYPE.

*continued on next page*

Name	Description
xml_version	Returns the XML version as declared by the document.
<i>Inherited from object</i>	
__class__	

### B.6.13 Class DocumentInvalid



Validation error.

Raised by all document validators when their `assertValid(tree)` method fails.

#### Methods

*Inherited from `lxml.etree.LxmlError` (Section [B.6.30](#))*

`__init__()`

*Inherited from `exceptions.Exception`*

`__new__()`

*Inherited from `exceptions.BaseException`*

`__delattr__()`, `__getattr__()`, `__getitem__()`, `__getslice__()`, `__reduce__()`, `__repr__()`,  
`__setattr__()`, `__setstate__()`, `__str__()`

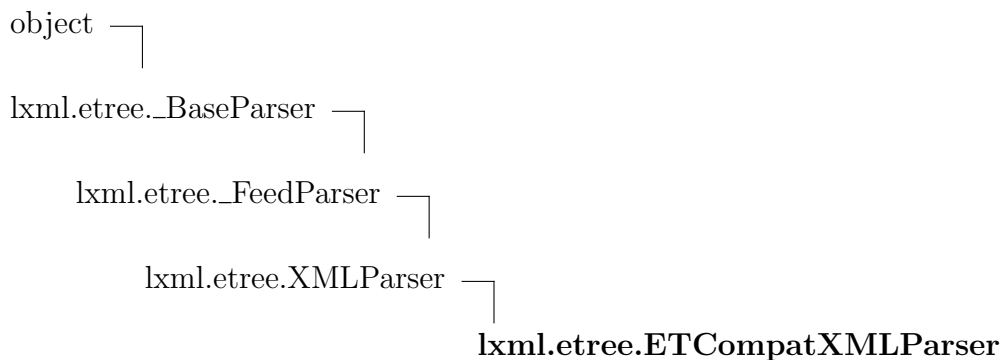
*Inherited from `object`*

`__hash__()`, `__reduce_ex__()`

#### Properties

Name	Description
<i>Inherited from</i> <code>exceptions.BaseException</code>	
	args, message
<i>Inherited from</i> <code>object</code>	
<code>__class__</code>	

### B.6.14 Class `ETCompatXMLParser`



`ETCompatXMLParser(self, attribute_defaults=False, dtd_validation=False, load_dtd=False, no_network=True, ns_clean=False, recover=False, remove_blank_text=False, compact=True, resolve_entities=True, remove_comments=True, remove_pis=True, target=None, encoding=None, schema=None)` An XML parser with an `ElementTree` compatible default setup.

See the `XMLParser` class for details.

This parser has `remove_comments` and `remove_pis` enabled by default and thus ignores comments and processing instructions.

#### Methods

```

__init__(self, attribute_defaults=False, dtd_validation=False,
load_dtd=False, no_network=True, ns_clean=False, recover=False,
remove_blank_text=False, compact=True, resolve_entities=True,
remove_comments=True, remove_pis=True, target=None, encoding=None,
schema=None)

```

`x.__init__(...)` initializes x; see `x.__class__.__doc__` for signature. Overrides: `object.__init__`

<p><code>__new__(T, S, ...)</code></p> <p><b>Return Value</b> a new object with type S, a subtype of T</p> <p>Overrides: <code>object.__new__</code></p>
--

***Inherited from `lxml.etree._FeedParser`***

`close()`, `feed()`

***Inherited from `lxml.etree._BaseParser`***

`copy()`, `makeelement()`, `setElementClassLookup()`, `set_element_class_lookup()`

***Inherited from `object`***

`__delattr__()`, `__getattr__()`, `__hash__()`, `__reduce__()`, `__reduce_ex__()`, `__repr__()`,  
`__setattr__()`, `__str__()`

**Properties**

Name	Description
<i>Inherited from <code>lxml.etree._FeedParser</code></i> <code>feed_error_log</code>	
<i>Inherited from <code>lxml.etree._BaseParser</code></i> <code>error_log</code> , <code>resolvers</code> , <code>version</code>	
<i>Inherited from <code>object</code></i> <code>__class__</code>	

**B.6.15 Class *ETXPath***

object └─

`lxml.etree._XPathEvaluatorBase` └─

`lxml.etree.XPath` └─

**`lxml.etree.ETXPath`**

`ETXPath(self, path, extensions=None, regexp=True)` Special XPath class that supports the `ElementTree {uri}` notation for namespaces.

Note that this class does not accept the `namespace` keyword argument. All namespaces must be passed as part of the path string.

**Methods**

<code>__init__(self, path, extensions=None, regep=True)</code>
--

<code>x.__init__(...)</code> initializes x; see <code>x.__class__.__doc__</code> for signature Overrides: <code>object.__init__</code>
--

<code>__new__(T, S, ...)</code>
---------------------------------

**Return Value**

a new object with type S, a subtype of T

Overrides: `object.__new__`

*Inherited from `lxml.etree.XPath` (Section [B.6.60](#))*

`__call__()`, `__repr__()`

*Inherited from `lxml.etree._XPathEvaluatorBase`*

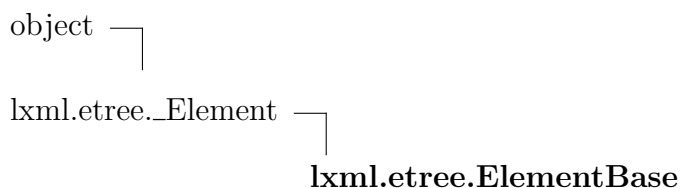
`evaluate()`

*Inherited from `object`*

`__delattr__()`, `__getattr__()`, `__hash__()`, `__reduce__()`, `__reduce_ex__()`, `__setattr__()`, `__str__()`

**Properties**

Name	Description
<i>Inherited from <code>lxml.etree.XPath</code> (Section <a href="#">B.6.60</a>)</i> <code>path</code>	
<i>Inherited from <code>lxml.etree._XPathEvaluatorBase</code></i> <code>error_log</code>	
<i>Inherited from <code>object</code></i> <code>__class__</code>	

**B.6.16 Class `ElementBase`**

**Known Subclasses:** `lxml.objectify.ObjectifiedElement`, `lxml.html.HtmlElement`

All custom *Element* classes must inherit from this one.

Note that you cannot (and must not) instantiate this class or its subclasses.

Subclasses *must not* override `__init__` or `__new__` as it is absolutely undefined when these objects will be created or destroyed. All persistent state of *Elements* must be stored in the underlying XML. If you really need to initialize the object after creation, you can implement an `__init__(self)` method that will be called after object creation.

## Methods

<code>__new__(T, S, ...)</code>
---------------------------------

**Return Value**

a new object with type *S*, a subtype of *T*

Overrides: `object.__new__`

### *Inherited from lxml.etree.\_Element*

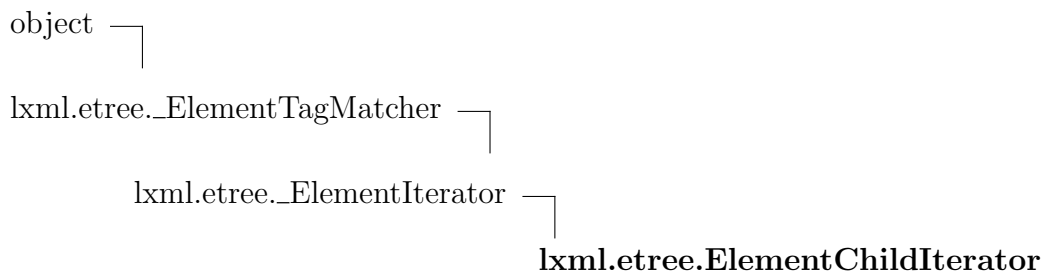
`__contains__()`, `__copy__()`, `__deepcopy__()`, `__delitem__()`, `__getitem__()`, `__iter__()`, `__len__()`, `__nonzero__()`, `__repr__()`, `__reversed__()`, `__setitem__()`, `addnext()`, `addprevious()`, `append()`, `clear()`, `extend()`, `find()`, `findall()`, `findtext()`, `get()`, `getchildren()`, `getiterator()`, `getnext()`, `getparent()`, `getprevious()`, `getroottree()`, `index()`, `insert()`, `items()`, `iter()`, `iterancestors()`, `iterchildren()`, `iterdescendants()`, `iterfind()`, `itersiblings()`, `itertext()`, `keys()`, `makeelement()`, `remove()`, `replace()`, `set()`, `values()`, `xpath()`

### *Inherited from object*

`__delattr__()`, `__getattr__()`, `__hash__()`, `__init__()`, `__reduce__()`, `__reduce_ex__()`, `__setattr__()`, `__str__()`

## Properties

Name	Description
<i>Inherited from lxml.etree._Element</i>	
attrib, nsmap, prefix, sourceline, tag, tail, text	
<i>Inherited from object</i>	
<code>__class__</code>	

**B.6.17 Class `ElementChildIterator`**

`ElementChildIterator(self, node, tag=None, reversed=False)` Iterates over the children of an element.

**Methods**

<code>__init__(self, node, tag=None, reversed=False)</code>
<code>x.__init__(...)</code> initializes x; see <code>x.__class__.__doc__</code> for signature Overrides: <code>object.__init__</code>

<code>__new__(T, S, ...)</code>
<b>Return Value</b> a new object with type S, a subtype of T
Overrides: <code>object.__new__</code>

***Inherited from `lxml.etree._ElementIterator`***

`__iter__()`, `__next__()`, `next()`

***Inherited from `object`***

`__delattr__()`, `__getattr__()`, `__hash__()`, `__reduce__()`, `__reduce_ex__()`, `__repr__()`, `__setattr__()`, `__str__()`

**Properties**

Name	Description
<i>Inherited from <code>object</code></i>	
<code>__class__</code>	

**B.6.18 Class `ElementClassLookup`**

**Known Subclasses:** `lxml.etree.FallbackElementClassLookup`, `lxml.etree.ElementDefaultClassLookup`, `lxml.objectify.ObjectifyElementClassLookup`

`ElementClassLookup(self)`

Superclass of Element class lookups.

**Methods**

<p><code>__new__(T, S, ...)</code></p> <p><b>Return Value</b> a new object with type S, a subtype of T</p> <p>Overrides: <code>object.__new__</code></p>
--

***Inherited from `object`***

`__delattr__()`, `__getattr__()`, `__hash__()`, `__init__()`, `__reduce__()`, `__reduce_ex__()`, `__repr__()`, `__setattr__()`, `__str__()`

**Properties**

Name	Description
<i>Inherited from <code>object</code></i>	
<code>__class__</code>	

**B.6.19 Class `ElementDefaultClassLookup`**

`ElementDefaultClassLookup(self, element=None, comment=None, pi=None, entity=None)`  
Element class lookup scheme that always returns the default Element class.

The keyword arguments `element`, `comment`, `pi` and `entity` accept the respective Element classes.

**Methods**

<code>__init__(self, element=None, comment=None, pi=None, entity=None)</code>
---

<code>x.__init__(...)</code> initializes <code>x</code> ; see <code>x.__class__.__doc__</code> for signature. Overrides: <code>object.__init__</code>
---

<code>__new__(T, S, ...)</code>
---------------------------------

**Return Value**

a new object with type `S`, a subtype of `T`

Overrides: `object.__new__`

**Inherited from object**

`__delattr__()`, `__getattr__()`, `__hash__()`, `__reduce__()`, `__reduce_ex__()`, `__repr__()`, `__setattr__()`, `__str__()`

**Properties**

Name	Description
<code>comment_class</code>	
<code>element_class</code>	
<code>entity_class</code>	
<code>pi_class</code>	
<i>Inherited from object</i>	
<code>__class__</code>	

**B.6.20 Class `ElementDepthFirstIterator`**

object └─

`lxml.etree.ElementTagMatcher` └─

**`lxml.etree.ElementDepthFirstIterator`**

`ElementDepthFirstIterator(self, node, tag=None, inclusive=True)` Iterates over an element and its sub-elements in document order (depth first pre-order).

Note that this also includes comments, entities and processing instructions. To filter them out, check if the `tag` property of the returned element is a string (i.e. not `None` and not a factory function), or pass the `Element` factory for the `tag` keyword.

If the optional `tag` argument is not `None`, the iterator returns only the elements that

match the respective name and namespace.

The optional boolean argument 'inclusive' defaults to True and can be set to False to exclude the start element itself.

Note that the behaviour of this iterator is completely undefined if the tree it traverses is modified during iteration.

## Methods

<code>__init__(self, node, tag=None, inclusive=True)</code>
---

<code>x.__init__(...)</code> initializes x; see <code>x.__class__.__doc__</code> for signature Overrides: <code>object.__init__</code>
---

<code>__iter__(...)</code>
----------------------------

<code>__new__(T, S, ...)</code>
---------------------------------

**Return Value**

a new object with type S, a subtype of T

Overrides: `object.__new__`

<code>__next__(...)</code>
----------------------------

<code>next(x)</code>
----------------------

**Return Value**

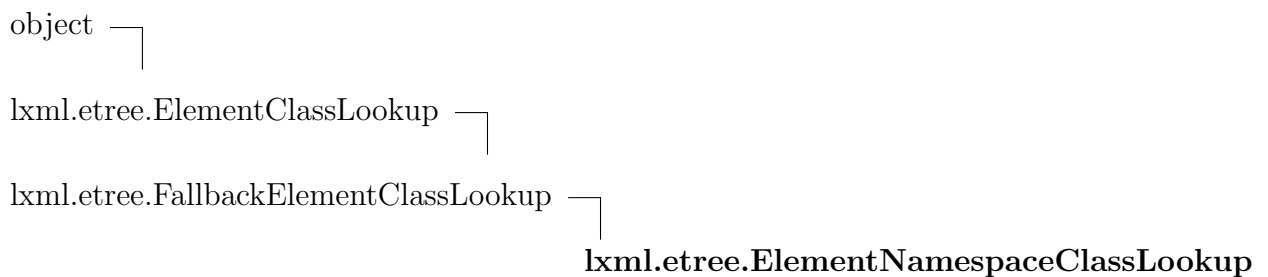
the next value, or raise `StopIteration`

## *Inherited from object*

`__delattr__()`, `__getattr__()`, `__hash__()`, `__reduce__()`, `__reduce_ex__()`, `__repr__()`,  
`__setattr__()`, `__str__()`

## Properties

Name	Description
<i>Inherited from object</i>	
<code>__class__</code>	

**B.6.21 Class `ElementNamespaceClassLookup`**

`ElementNamespaceClassLookup(self, fallback=None)`

Element class lookup scheme that searches the Element class in the Namespace registry.

**Methods**

<b><code>__init__(self, fallback=None)</code></b> <hr/> x. <code>__init__</code> (...) initializes x; see x. <code>__class__.__doc__</code> for signature Overrides: object. <code>__init__</code>
---

<b><code>__new__(T, S, ...)</code></b> <b>Return Value</b> a new object with type S, a subtype of T Overrides: object. <code>__new__</code>
--

<b><code>get_namespace(self, ns_uri)</code></b> <hr/> Retrieve the namespace object associated with the given URI. Creates a new one if it does not yet exist.
--

*Inherited from `lxml.etree.FallbackElementClassLookup` (Section [B.6.28](#))*

`setFallback()`, `set_fallback()`

*Inherited from `object`*

`__delattr__()`, `__getattr__()`, `__hash__()`, `__reduce__()`, `__reduce_ex__()`, `__repr__()`,  
`__setattr__()`, `__str__()`

**Properties**

Name	Description
<code>fallback</code>	<i>Inherited from <code>lxml.etree.FallbackElementClassLookup</code> (Section <a href="#">B.6.28</a>)</i>
<code>__class__</code>	<i>Inherited from <code>object</code></i>

## B.6.22 Class `ElementTextIterator`



`ElementTextIterator(self, element, tag=None, with_tail=True)` Iterates over the text content of a subtree.

You can pass the `tag` keyword argument to restrict text content to a specific tag name.

You can set the `with_tail` keyword argument to `False` to skip over tail text.

### Methods

<code>__init__(self, element, tag=None, with_tail=True)</code>
<code>x.__init__(...)</code> initializes <code>x</code> ; see <code>x.__class__.__doc__</code> for signature. Overrides: <code>object.__init__</code>

<code>__iter__(...)</code>
----------------------------

<code>__new__(T, S, ...)</code>
<b>Return Value</b> a new object with type <code>S</code> , a subtype of <code>T</code>
Overrides: <code>object.__new__</code>

<code>__next__(...)</code>
----------------------------

<code>next(x)</code>
<b>Return Value</b> the next value, or raise <code>StopIteration</code>

### *Inherited from `object`*

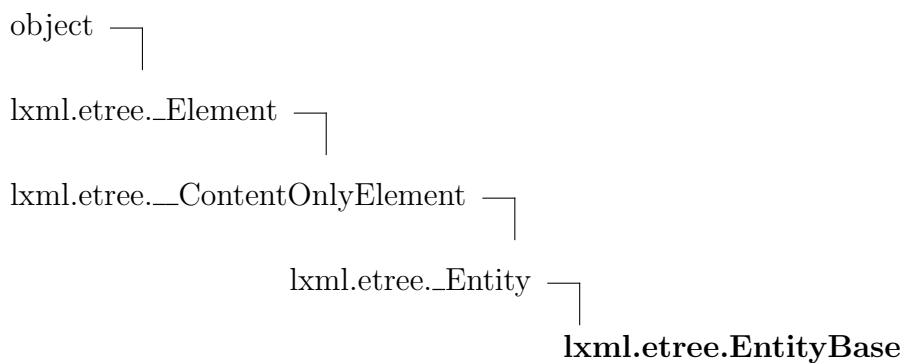
`__delattr__()`, `__getattr__()`, `__hash__()`, `__reduce__()`, `__reduce_ex__()`, `__repr__()`,

`__setattr__()`, `__str__()`

### Properties

Name	Description
<i>Inherited from object</i>	
<code>__class__</code>	

### B.6.23 Class *EntityBase*



**Known Subclasses:** `lxml.html.HtmlEntity`

All custom Entity classes must inherit from this one.

Note that you cannot (and must not) instantiate this class or its subclasses.

Subclasses *must not* override `__init__` or `__new__` as it is absolutely undefined when these objects will be created or destroyed. All persistent state of Entities must be stored in the underlying XML. If you really need to initialize the object after creation, you can implement an `_init(self)` method that will be called after object creation.

### Methods

`__new__(T, S, ...)`

**Return Value**

a new object with type S, a subtype of T

Overrides: `object.__new__`

*Inherited from `lxml.etree._Entity`*

`__repr__()`

*Inherited from `lxml.etree._ContentOnlyElement`*

`__delitem__()`, `__getitem__()`, `__len__()`, `__setitem__()`, `append()`, `get()`, `insert()`,

items(), keys(), set(), values()

### *Inherited from lxml.etree.\_Element*

`__contains__()`, `__copy__()`, `__deepcopy__()`, `__iter__()`, `__nonzero__()`, `__reversed__()`, `addnext()`, `addprevious()`, `clear()`, `extend()`, `find()`, `findall()`, `findtext()`, `getchildren()`, `getiterator()`, `getnext()`, `getparent()`, `getprevious()`, `getroottree()`, `index()`, `iter()`, `iterancestors()`, `iterchildren()`, `iterdescendants()`, `iterfind()`, `itersiblings()`, `itertext()`, `makeelement()`, `remove()`, `replace()`, `xpath()`

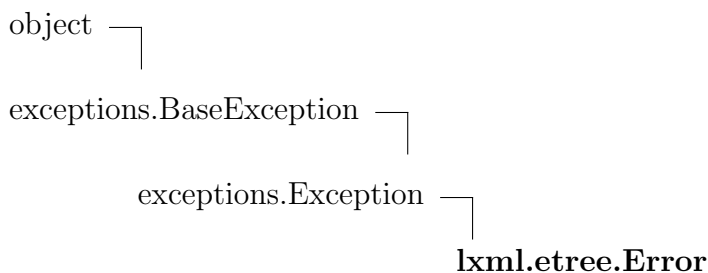
### *Inherited from object*

`__delattr__()`, `__getattr__()`, `__hash__()`, `__init__()`, `__reduce__()`, `__reduce_ex__()`, `__setattr__()`, `__str__()`

### Properties

Name	Description
<i>Inherited from lxml.etree._Entity</i>	name, tag, text
<i>Inherited from lxml.etree._ContentOnlyElement</i>	attrib
<i>Inherited from lxml.etree._Element</i>	nsmmap, prefix, sourceline, tail
<i>Inherited from object</i>	<code>__class__</code>

## B.6.24 Class Error



**Known Subclasses:** lxml.etree.LxmlError

### Methods

### *Inherited from exceptions.Exception*

`__init__()`, `__new__()`

***Inherited from exceptions.BaseException***

`__delattr__()`, `__getattr__()`, `__getitem__()`, `__getslice__()`, `__reduce__()`, `__repr__()`,  
`__setattr__()`, `__setstate__()`, `__str__()`

***Inherited from object***

`__hash__()`, `__reduce_ex__()`

**Properties**

Name	Description
<i>Inherited from exceptions.BaseException</i>	args, message
<i>Inherited from object</i>	<code>__class__</code>

**B.6.25 Class ErrorDomains**

Libxml2 error domains

**Class Variables**

Name	Description
C14N	<b>Value:</b> 21
CATALOG	<b>Value:</b> 20
CHECK	<b>Value:</b> 24
DATATYPE	<b>Value:</b> 15
DTD	<b>Value:</b> 4
FTP	<b>Value:</b> 9
HTML	<b>Value:</b> 5
HTTP	<b>Value:</b> 10
I18N	<b>Value:</b> 27
IO	<b>Value:</b> 8
MEMORY	<b>Value:</b> 6
MODULE	<b>Value:</b> 26
NAMESPACE	<b>Value:</b> 3
NONE	<b>Value:</b> 0
OUTPUT	<b>Value:</b> 7
PARSER	<b>Value:</b> 1
REGEXP	<b>Value:</b> 14
RELAXNGP	<b>Value:</b> 18

*continued on next page*

Name	Description
RELAXNGV	Value: 19
SCHEMASP	Value: 16
SCHEMASV	Value: 17
SCHEMATRONV	Value: 28
TREE	Value: 2
VALID	Value: 23
WRITER	Value: 25
XINCLUDE	Value: 11
XPATH	Value: 12
XPOINTER	Value: 13
XSLT	Value: 22

### B.6.26 Class ErrorLevels

Libxml2 error levels

Class Variables

Name	Description
ERROR	Value: 2
FATAL	Value: 3
NONE	Value: 0
WARNING	Value: 1

### B.6.27 Class ErrorTypes

Libxml2 error types

Class Variables

Name	Description
C14N_CREATE_CTXT	Value: 1950
C14N_CREATE_STACK	Value: 1952
C14N_INVALID_NODE	Value: 1953
C14N_RELATIVE_NAMESPACE	Value: 1955
C14N_REQUIRES_UTF8	Value: 1951
C14N_UNKNOW_NODE	Value: 1954

*continued on next page*

<b>Name</b>	<b>Description</b>
CATALOG_ENTRY_BROKEN	<b>Value:</b> 1651
CATALOG_MISSING_ATTR	<b>Value:</b> 1650
CATALOG_NOT_CATALOG	<b>Value:</b> 1653
CATALOG_PREFER_VALUE	<b>Value:</b> 1652
CATALOG_RECURSION	<b>Value:</b> 1654
CHECK_	<b>Value:</b> 6005
CHECK_ENTITY_TYPE	<b>Value:</b> 5012
CHECK_FOUND_ATTRIBUTE	<b>Value:</b> 5001
CHECK_FOUND_CDATA	<b>Value:</b> 5003
CHECK_FOUND_COMMENT	<b>Value:</b> 5007
CHECK_FOUND_DOCTYPE	<b>Value:</b> 5008
CHECK_FOUND_ELEMENT	<b>Value:</b> 5000
CHECK_FOUND_ENTITY	<b>Value:</b> 5005
CHECK_FOUND_ENTITYREF	<b>Value:</b> 5004
CHECK_FOUND_FRAGMENT	<b>Value:</b> 5009
CHECK_FOUND_NOTATION	<b>Value:</b> 5010
CHECK_FOUND_PI	<b>Value:</b> 5006
CHECK_FOUND_TEXT	<b>Value:</b> 5002
CHECK_NAME_NOT_NULL	<b>Value:</b> 5037
CHECK_NOT_ATTR	<b>Value:</b> 5023
CHECK_NOT_ATTR_DECL	<b>Value:</b> 5024
CHECK_NOT_DTD	<b>Value:</b> 5022
CHECK_NOT_ELEM_DECL	<b>Value:</b> 5025

*continued on next page*

<b>Name</b>	<b>Description</b>
CHECK_NOT_ENTITY_DECL	<b>Value:</b> 5026
CHECK_NOT_NCNAME	<b>Value:</b> 5034
CHECK_NOT_NS_DECL	<b>Value:</b> 5027
CHECK_NOT_UTF8	<b>Value:</b> 5032
CHECK_NO_DICT	<b>Value:</b> 5033
CHECK_NO_DOC	<b>Value:</b> 5014
CHECK_NO_ELEM	<b>Value:</b> 5016
CHECK_NO_HREF	<b>Value:</b> 5028
CHECK_NO_NAME	<b>Value:</b> 5015
CHECK_NO_NEXT	<b>Value:</b> 5020
CHECK_NO_PARENT	<b>Value:</b> 5013
CHECK_NO_PREV	<b>Value:</b> 5018
CHECK_NS_ANCESTOR	<b>Value:</b> 5031
CHECK_NS_SCOPE	<b>Value:</b> 5030
CHECK_OUTSIDE_DICT	<b>Value:</b> 5035
CHECK_UNKNOWN_NODE	<b>Value:</b> 5011
CHECK_WRONG_DOC	<b>Value:</b> 5017
CHECK_WRONG_NAME	<b>Value:</b> 5036
CHECK_WRONG_NEXT	<b>Value:</b> 5021
CHECK_WRONG_PARENT	<b>Value:</b> 5029
CHECK_WRONG_PREV	<b>Value:</b> 5019
CHECK_X	<b>Value:</b> 6006
DTD_ATTRIBUTE_DEFAULT	<b>Value:</b> 500
DTD_ATTRIBUTE_REDEFINED	<b>Value:</b> 501
DTD_ATTRIBUTE_VALUE	<b>Value:</b> 502
DTD_CONTENT_ERROR	<b>Value:</b> 503
DTD_CONTENT_MODEL	<b>Value:</b> 504

*continued on next page*

<b>Name</b>	<b>Description</b>
DTD_CONTENT_NOT_DETERMINIST	<b>Value:</b> 505
DTD_DIFFERENT_PREFIX	<b>Value:</b> 506
DTD_ELEM_DEFAULT_NAMESPACE	<b>Value:</b> 507
DTD_ELEM_NAMESPACE	<b>Value:</b> 508
DTD_ELEM_REDEFINED	<b>Value:</b> 509
DTD_EMPTY_NOTATION	<b>Value:</b> 510
DTD_ENTITY_TYPE	<b>Value:</b> 511
DTD_ID_FIXED	<b>Value:</b> 512
DTD_ID_REDEFINED	<b>Value:</b> 513
DTD_ID_SUBSET	<b>Value:</b> 514
DTD_INVALID_CHILD	<b>Value:</b> 515
DTD_INVALID_DEFAULT	<b>Value:</b> 516
DTD_LOAD_ERROR	<b>Value:</b> 517
DTD_MISSING_ATTRIBUTE	<b>Value:</b> 518
DTD_MIXED_CORRUPT	<b>Value:</b> 519
DTD_MULTIPLE_ID	<b>Value:</b> 520
DTD_NOTATION_REDEFINED	<b>Value:</b> 526
DTD_NOTATION_VALUE	<b>Value:</b> 527
DTD_NOT_EMPTY	<b>Value:</b> 528
DTD_NOT_PCDATA	<b>Value:</b> 529
DTD_NOT_STANDALONE	<b>Value:</b> 530
DTD_NO_DOC	<b>Value:</b> 521
DTD_NO_DTD	<b>Value:</b> 522
DTD_NO_ELEM_NAME	<b>Value:</b> 523
DTD_NO_PREFIX	<b>Value:</b> 524
DTD_NO_ROOT	<b>Value:</b> 525
DTD_ROOT_NAME	<b>Value:</b> 531
DTD_STANDALONE_DEFAULTED	<b>Value:</b> 538

*continued on next page*

<b>Name</b>	<b>Description</b>
DTD_STANDALONE_WHITE_SPACE	<b>Value:</b> 532
DTD_UNKNOWN_ATTRIBUTE	<b>Value:</b> 533
DTD_UNKNOWN_ELEMENT	<b>Value:</b> 534
DTD_UNKNOWN_ENTITY	<b>Value:</b> 535
DTD_UNKNOWN_ID	<b>Value:</b> 536
DTD_UNKNOWN_NOTATION	<b>Value:</b> 537
DTD_XMLID_TYPE	<b>Value:</b> 540
DTD_XMLID_VALUE	<b>Value:</b> 539
ERR_ATTLIST_NOT_FINISHED	<b>Value:</b> 51
ERR_ATTLIST_NOT_STARTED	<b>Value:</b> 50
ERR_ATTRIBUTE_NOT_FINISHED	<b>Value:</b> 40
ERR_ATTRIBUTE_NOT_STARTED	<b>Value:</b> 39
ERR_ATTRIBUTE_REDEFINED	<b>Value:</b> 42
ERR_ATTRIBUTE_WITHOUT_VALUE	<b>Value:</b> 41
ERR_CDATA_NOT_FINISHED	<b>Value:</b> 63
ERR_CHARREF_AT_EOF	<b>Value:</b> 10
ERR_CHARREF_IN_DTD	<b>Value:</b> 13
ERR_CHARREF_IN_EPILOG	<b>Value:</b> 12
ERR_CHARREF_IN_PROLOG	<b>Value:</b> 11
ERR_COMMENT_NOT_FINISHED	<b>Value:</b> 45
ERR_CONDSEC_INVALID_ID	<b>Value:</b> 83
ERR_CONDSEC_INVALID_KEYWORD	<b>Value:</b> 95

*continued on next page*

<b>Name</b>	<b>Description</b>
ERR_CONDSEC_NOT_FINISHED	<b>Value:</b> 59
ERR_CONDSEC_NOT_STARTED	<b>Value:</b> 58
ERR_DOCTYPE_NOT_FINISHED	<b>Value:</b> 61
ERR_DOCUMENT_EMPTY	<b>Value:</b> 4
ERR_DOCUMENT_END	<b>Value:</b> 5
ERR_DOCUMENT_START	<b>Value:</b> 3
ERR_ELEMCONTENT_NOT_FINISHED	<b>Value:</b> 55
ERR_ELEMCONTENT_NOT_STARTED	<b>Value:</b> 54
ERR_ENCODING_NAME	<b>Value:</b> 79
ERR_ENTITYREF_AT_EOF	<b>Value:</b> 14
ERR_ENTITYREF_IN_DTD	<b>Value:</b> 17
ERR_ENTITYREF_IN_EPILOG	<b>Value:</b> 16
ERR_ENTITYREF_IN_PROLOG	<b>Value:</b> 15
ERR_ENTITYREF_NO_NAME	<b>Value:</b> 22
ERR_ENTITYREF_SEMICOLON_MISSING	<b>Value:</b> 23
ERR_ENTITY_BOUNDARY	<b>Value:</b> 90
ERR_ENTITY_CHAR_ERROR	<b>Value:</b> 87
ERR_ENTITY_IS_EXTERNAL	<b>Value:</b> 29
ERR_ENTITY_IS_PARAMETER	<b>Value:</b> 30
ERR_ENTITY_LOOP	<b>Value:</b> 89
ERR_ENTITY_NOT_FINISHED	<b>Value:</b> 37

*continued on next page*

<b>Name</b>	<b>Description</b>
ERR_ENTITY_NOT_STARTED	<b>Value:</b> 36
ERR_ENTITY_PE_INTERNAL	<b>Value:</b> 88
ERR_ENTITY_PROCESSING	<b>Value:</b> 104
ERR_EQUAL_REQUIRED	<b>Value:</b> 75
ERR_EXTRA_CONTENT	<b>Value:</b> 86
ERR_EXT_ENTITY_STANDALONE	<b>Value:</b> 82
ERR_EXT_SUBSET_NOT_FINISHED	<b>Value:</b> 60
ERR_GT_REQUIRED	<b>Value:</b> 73
ERR_HYPHEN_IN_COMMENT	<b>Value:</b> 80
ERR_INTERNAL_ERROR	<b>Value:</b> 1
ERR_INVALID_CHAR	<b>Value:</b> 9
ERR_INVALID_CHARRREF	<b>Value:</b> 8
ERR_INVALID_DEC_CHARREF	<b>Value:</b> 7
ERR_INVALID_ENCODING	<b>Value:</b> 81
ERR_INVALID_HEX_CHARREF	<b>Value:</b> 6
ERR_INVALID_URI	<b>Value:</b> 91
ERR_LITERAL_NOT_FINISHED	<b>Value:</b> 44
ERR_LITERAL_NOT_STARTED	<b>Value:</b> 43
ERR_LTSLASH_REQUIRED	<b>Value:</b> 74
ERR_LT_IN_ATTRIBUTE	<b>Value:</b> 38
ERR_LT_REQUIRED	<b>Value:</b> 72
ERR_MISPLACEDCDATA_END	<b>Value:</b> 62
ERR_MISSING_ENCODING	<b>Value:</b> 101

*continued on next page*

<b>Name</b>	<b>Description</b>
ERR_MIXED_NOT_FINISHED	<b>Value:</b> 53
ERR_MIXED_NOT_STARTED	<b>Value:</b> 52
ERR_NAME_REQUIRED	<b>Value:</b> 68
ERR_NMTOKEN_REQUIRED	<b>Value:</b> 67
ERR_NOTATION_NOT_FINISHED	<b>Value:</b> 49
ERR_NOTATION_NOT_STARTED	<b>Value:</b> 48
ERR_NOTATION_PROCESSING	<b>Value:</b> 105
ERR_NOT_STANDALONE	<b>Value:</b> 103
ERR_NOT_WELL_BALANCED	<b>Value:</b> 85
ERR_NO_DTD	<b>Value:</b> 94
ERR_NO_MEMORY	<b>Value:</b> 2
ERR_NS_DECL_ERROR	<b>Value:</b> 35
ERR_OK	<b>Value:</b> 0
ERR_PCDATA_REQUIRED	<b>Value:</b> 69
ERR_PEREF_AT_EOF	<b>Value:</b> 18
ERR_PEREF_IN_EPILOG	<b>Value:</b> 20
ERR_PEREF_IN_INT_SUBSET	<b>Value:</b> 21
ERR_PEREF_IN_PROLOG	<b>Value:</b> 19
ERR_PEREF_NO_NAME	<b>Value:</b> 24
ERR_PEREF_SEMICOLON_MISSING	<b>Value:</b> 25
ERR_PL_NOT_FINISHED	<b>Value:</b> 47
ERR_PL_NOT_STARTED	<b>Value:</b> 46
ERR_PUBID_REQUIRED	<b>Value:</b> 71

*continued on next page*

<b>Name</b>	<b>Description</b>
ERR_RESERVED_XML_NAME	<b>Value:</b> 64
ERR_SEPARATOR_REQUIRED	<b>Value:</b> 66
ERR_SPACE_REQUIRED	<b>Value:</b> 65
ERR_STANDALONE_VALUE	<b>Value:</b> 78
ERR_STRING_NOT_CLOSED	<b>Value:</b> 34
ERR_STRING_NOT_STARTED	<b>Value:</b> 33
ERR_TAG_NAME_MISMATCH	<b>Value:</b> 76
ERR_TAG_NOT_FINISHED	<b>Value:</b> 77
ERR_UNDECLARED_ENTITY	<b>Value:</b> 26
ERR_UNKNOWN_ENCODING	<b>Value:</b> 31
ERR_UNPARSED_ENTITY	<b>Value:</b> 28
ERR_UNSUPPORTED_ENCODING	<b>Value:</b> 32
ERR_URLFRAGMENT	<b>Value:</b> 92
ERR_URLREQUIRED	<b>Value:</b> 70
ERR_VALUE_REQUIRED	<b>Value:</b> 84
ERR_VERSION_MISSING	<b>Value:</b> 96
ERR_XMLDECL_NOT_FINISHED	<b>Value:</b> 57
ERR_XMLDECL_NOT_STARTED	<b>Value:</b> 56
FTP_ACCNT	<b>Value:</b> 2002
FTP_EPSV_ANSWER	<b>Value:</b> 2001
FTP_PASV_ANSWER	<b>Value:</b> 2000
FTP_URLSYNTAX	<b>Value:</b> 2003
HTML_STRUCTURE_ERROR	<b>Value:</b> 800
HTML_UNKNOWN_TAG	<b>Value:</b> 801

*continued on next page*

<b>Name</b>	<b>Description</b>
HTTP_UNKNOWN_HOST	<b>Value:</b> 2022
HTTP_URL_SYNTAX	<b>Value:</b> 2020
HTTP_USE_IP	<b>Value:</b> 2021
I18N_CONV_FAILED	<b>Value:</b> 6003
I18N_EXCESS_HANDLER	<b>Value:</b> 6002
I18N_NO_HANDLER	<b>Value:</b> 6001
I18N_NO_NAME	<b>Value:</b> 6000
I18N_NO_OUTPUT	<b>Value:</b> 6004
IO_BUFFER_FULL	<b>Value:</b> 1548
IO_EACCES	<b>Value:</b> 1501
IO_EADDRINUSE	<b>Value:</b> 1554
IO_EAFNOSUPPORT	<b>Value:</b> 1556
IO_EAGAIN	<b>Value:</b> 1502
IO_EALREADY	<b>Value:</b> 1555
IO_EBADF	<b>Value:</b> 1503
IO_EBADMSG	<b>Value:</b> 1504
IO_EBUSY	<b>Value:</b> 1505
IO_ECANCELED	<b>Value:</b> 1506
IO_ECHILD	<b>Value:</b> 1507
IO_ECONNREFUSED	<b>Value:</b> 1552
IO_EDEADLK	<b>Value:</b> 1508
IO_EDOM	<b>Value:</b> 1509
IO_EXIST	<b>Value:</b> 1510
IO_EFAULT	<b>Value:</b> 1511
IO_EFBIG	<b>Value:</b> 1512
IO_EINPROGRESS	<b>Value:</b> 1513
IO_EINTR	<b>Value:</b> 1514
IO_EINVAL	<b>Value:</b> 1515
IO_EIO	<b>Value:</b> 1516
IO_EISCONN	<b>Value:</b> 1551
IO_EISDIR	<b>Value:</b> 1517
IO_EMFILE	<b>Value:</b> 1518
IO_EMLINK	<b>Value:</b> 1519
IO_MSGSIZE	<b>Value:</b> 1520
IO_ENAMETOOLONG	<b>Value:</b> 1521
IO_ENCODER	<b>Value:</b> 1544
IO_ENETUNREACH	<b>Value:</b> 1553

*continued on next page*

<b>Name</b>	<b>Description</b>
IO_ENFILE	<b>Value:</b> 1522
IO_ENODEV	<b>Value:</b> 1523
IO_ENOENT	<b>Value:</b> 1524
IO_ENOEXEC	<b>Value:</b> 1525
IO_ENOLCK	<b>Value:</b> 1526
IO_ENOMEM	<b>Value:</b> 1527
IO_ENOSPC	<b>Value:</b> 1528
IO_ENOSYS	<b>Value:</b> 1529
IO_ENOTDIR	<b>Value:</b> 1530
IO_ENOTEMPTY	<b>Value:</b> 1531
IO_ENOTSOCK	<b>Value:</b> 1550
IO_ENOTSUP	<b>Value:</b> 1532
IO_ENOTTY	<b>Value:</b> 1533
IO_ENXIO	<b>Value:</b> 1534
IO_EPERM	<b>Value:</b> 1535
IO_EPIPE	<b>Value:</b> 1536
IO_ERANGE	<b>Value:</b> 1537
IO_EROFS	<b>Value:</b> 1538
IO_ESPIPE	<b>Value:</b> 1539
IO_ESRCH	<b>Value:</b> 1540
IO_ETIMEDOUT	<b>Value:</b> 1541
IO_EXDEV	<b>Value:</b> 1542
IO_FLUSH	<b>Value:</b> 1545
IO_LOAD_ERROR	<b>Value:</b> 1549
IO_NETWORK_ATTEMPT	<b>Value:</b> 1543
IO_NO_INPUT	<b>Value:</b> 1547
IO_UNKNOWN	<b>Value:</b> 1500
IO_WRITE	<b>Value:</b> 1546
MODULE_CLOSE	<b>Value:</b> 4901
MODULE_OPEN	<b>Value:</b> 4900
NS_ERR_ATTRIBUTE_- REDEFINED	<b>Value:</b> 203
NS_ERR_EMPTY	<b>Value:</b> 204
NS_ERR_QNAME	<b>Value:</b> 202
NS_ERR_UNDEFINED_- NAMESPACE	<b>Value:</b> 201
NS_ERR_XML_NAMES- PACE	<b>Value:</b> 200

*continued on next page*

<b>Name</b>	<b>Description</b>
REGEXP_COMPILE_ERROR	<b>Value:</b> 1450
RNGP_ANYNAME_ATTR_ANCESTOR	<b>Value:</b> 1000
RNGP_ATTRIBUTE_CHILDREN	<b>Value:</b> 1002
RNGP_ATTRIBUTE_CONTENT	<b>Value:</b> 1003
RNGP_ATTRIBUTE_EMPTY	<b>Value:</b> 1004
RNGP_ATTRIBUTE_NOOP	<b>Value:</b> 1005
RNGP_ATTR_CONFLICT	<b>Value:</b> 1001
RNGP_CHOICE_CONTENT	<b>Value:</b> 1006
RNGP_CHOICE_EMPTY	<b>Value:</b> 1007
RNGP_CREATE_FAILURE	<b>Value:</b> 1008
RNGP_DATA_CONTENT	<b>Value:</b> 1009
RNGP_DEFINE_CREATE_FAILED	<b>Value:</b> 1011
RNGP_DEFINE_EMPTY	<b>Value:</b> 1012
RNGP_DEFINE_MISSING	<b>Value:</b> 1013
RNGP_DEFINE_NAME_MISSING	<b>Value:</b> 1014
RNGP_DEF_CHOICE_AND_INTERLEAVE	<b>Value:</b> 1010
RNGP_ELEMENT_CONTENT	<b>Value:</b> 1018
RNGP_ELEMENT_EMPTY	<b>Value:</b> 1017
RNGP_ELEMENT_NAME	<b>Value:</b> 1019
RNGP_ELEMENT_NO_CONTENT	<b>Value:</b> 1020
RNGP_ELEM_CONTENT_EMPTY	<b>Value:</b> 1015

*continued on next page*

<b>Name</b>	<b>Description</b>
RNGP_ELEM_CONTENT_ERROR	<b>Value:</b> 1016
RNGP_ELEM_TEXT_CONFLICT	<b>Value:</b> 1021
RNGP_EMPTY	<b>Value:</b> 1022
RNGP_EMPTY_CONSTRUCT	<b>Value:</b> 1023
RNGP_EMPTY_CONTENT	<b>Value:</b> 1024
RNGP_EMPTY_NOT_EMPTY	<b>Value:</b> 1025
RNGP_ERROR_TYPE_LIB	<b>Value:</b> 1026
RNGP_EXCEPT_EMPTY	<b>Value:</b> 1027
RNGP_EXCEPT_MISSING	<b>Value:</b> 1028
RNGP_EXCEPT_MULTIPLE	<b>Value:</b> 1029
RNGP_EXCEPT_NO_CONTENT	<b>Value:</b> 1030
RNGP_EXTERNALREF_EMPTY	<b>Value:</b> 1031
RNGP_EXTERNALREF_RECURSE	<b>Value:</b> 1033
RNGP_EXTERNALREF_FAILURE	<b>Value:</b> 1032
RNGP_FORBIDDEN_ATTRIBUTE	<b>Value:</b> 1034
RNGP_FOREIGN_ELEMENT	<b>Value:</b> 1035
RNGP_GRAMMAR_CONTENT	<b>Value:</b> 1036
RNGP_GRAMMAR_EMPTY	<b>Value:</b> 1037
RNGP_GRAMMAR_MISSING	<b>Value:</b> 1038
RNGP_GRAMMAR_NO_START	<b>Value:</b> 1039
RNGP_GROUP_ATTR_CONFLICT	<b>Value:</b> 1040
RNGP_HREF_ERROR	<b>Value:</b> 1041

*continued on next page*

<b>Name</b>	<b>Description</b>
RNGP_INCLUDE_EMPTY	<b>Value:</b> 1042
RNGP_INCLUDE_FAILURE	<b>Value:</b> 1043
RNGP_INCLUDE_RECURSE	<b>Value:</b> 1044
RNGP_INTERLEAVE_ADD	<b>Value:</b> 1045
RNGP_INTERLEAVE_CREATE_FAILED	<b>Value:</b> 1046
RNGP_INTERLEAVE_EMPTY	<b>Value:</b> 1047
RNGP_INTERLEAVE_NO_CONTENT	<b>Value:</b> 1048
RNGP_INVALID_DEFINE_NAME	<b>Value:</b> 1049
RNGP_INVALID_URI	<b>Value:</b> 1050
RNGP_INVALID_VALUE	<b>Value:</b> 1051
RNGP_MISSING_HREF	<b>Value:</b> 1052
RNGP_NAME_MISSING	<b>Value:</b> 1053
RNGP_NEED_COMBINE	<b>Value:</b> 1054
RNGP_NOTALLOWED_NOT_EMPTY	<b>Value:</b> 1055
RNGP_NSNAME_ATTR_ANCESTOR	<b>Value:</b> 1056
RNGP_NSNAME_NO_NS	<b>Value:</b> 1057
RNGP_PARAM_FORBIDDEN	<b>Value:</b> 1058
RNGP_PARAM_NAME_MISSING	<b>Value:</b> 1059
RNGP_PARENTREF_CREATE_FAILED	<b>Value:</b> 1060
RNGP_PARENTREF_NAME_INVALID	<b>Value:</b> 1061
RNGP_PARENTREF_NOT_EMPTY	<b>Value:</b> 1064
RNGP_PARENTREF_NO_NAME	<b>Value:</b> 1062

*continued on next page*

Name	Description
RNGP_PARENTREF_N-O_PARENT	<b>Value:</b> 1063
RNGP_PARSE_ERROR	<b>Value:</b> 1065
RNGP_PAT_ANYNAME_EXCEPT_ANYNAME	<b>Value:</b> 1066
RNGP_PAT_ATTR_ATTR	<b>Value:</b> 1067
RNGP_PAT_ATTR_ELEM	<b>Value:</b> 1068
RNGP_PAT_DATA_EXCEPT_ATTR	<b>Value:</b> 1069
RNGP_PAT_DATA_EXCEPT_ELEM	<b>Value:</b> 1070
RNGP_PAT_DATA_EXCEPT_EMPTY	<b>Value:</b> 1071
RNGP_PAT_DATA_EXCEPT_GROUP	<b>Value:</b> 1072
RNGP_PAT_DATA_EXCEPT_INTERLEAVE	<b>Value:</b> 1073
RNGP_PAT_DATA_EXCEPT_LIST	<b>Value:</b> 1074
RNGP_PAT_DATA_EXCEPT_ONEMORE	<b>Value:</b> 1075
RNGP_PAT_DATA_EXCEPT_REF	<b>Value:</b> 1076
RNGP_PAT_DATA_EXCEPT_TEXT	<b>Value:</b> 1077
RNGP_PAT_LIST_ATTR	<b>Value:</b> 1078
RNGP_PAT_LIST_ELEM	<b>Value:</b> 1079
RNGP_PAT_LIST_INTERLEAVE	<b>Value:</b> 1080
RNGP_PAT_LIST_LIST	<b>Value:</b> 1081
RNGP_PAT_LIST_REF	<b>Value:</b> 1082
RNGP_PAT_LIST_TEXT	<b>Value:</b> 1083
RNGP_PAT_NSNAME_EXCEPT_ANYNAME	<b>Value:</b> 1084
RNGP_PAT_NSNAME_EXCEPT_NSNAME	<b>Value:</b> 1085

*continued on next page*

<b>Name</b>	<b>Description</b>
RNGP_PAT_ONEMOR-E_GROUP_ATTR	<b>Value:</b> 1086
RNGP_PAT_ONEMOR-E_INTERLEAVE_ATTR	<b>Value:</b> 1087
RNGP_PAT_START_A-TTR	<b>Value:</b> 1088
RNGP_PAT_START_D-ATA	<b>Value:</b> 1089
RNGP_PAT_START_E-MPTY	<b>Value:</b> 1090
RNGP_PAT_START_G-ROUP	<b>Value:</b> 1091
RNGP_PAT_START_I-NTERLEAVE	<b>Value:</b> 1092
RNGP_PAT_START_LI-ST	<b>Value:</b> 1093
RNGP_PAT_START_O-NEMORE	<b>Value:</b> 1094
RNGP_PAT_START_T-E-X-T	<b>Value:</b> 1095
RNGP_PAT_START_V-ALUE	<b>Value:</b> 1096
RNGP_PREFIX_UNDEF-INED	<b>Value:</b> 1097
RNGP_REF_CREATE_F-AILED	<b>Value:</b> 1098
RNGP_REF_CYCLE	<b>Value:</b> 1099
RNGP_REF_NAME_I-NV-ALID	<b>Value:</b> 1100
RNGP_REF_NOT_E-MPTY	<b>Value:</b> 1103
RNGP_REF_NO_DEF	<b>Value:</b> 1101
RNGP_REF_NO_NAME	<b>Value:</b> 1102
RNGP_START_CHOI-C-E_AND_INTERLEAVE	<b>Value:</b> 1104
RNGP_START_CONTE-NT	<b>Value:</b> 1105
RNGP_START_EMPTY	<b>Value:</b> 1106
RNGP_START_MISSI-N-G	<b>Value:</b> 1107
RNGP_TEXT_EXPECT-ED	<b>Value:</b> 1108

*continued on next page*

<b>Name</b>	<b>Description</b>
RNGP_TEXT_HAS_CHILD	<b>Value:</b> 1109
RNGP_TYPE_MISSING	<b>Value:</b> 1110
RNGP_TYPE_NOT_FOUND	<b>Value:</b> 1111
RNGP_TYPE_VALUE	<b>Value:</b> 1112
RNGP_UNKNOWN_ATTRIBUTE	<b>Value:</b> 1113
RNGP_UNKNOWN_COMBINE	<b>Value:</b> 1114
RNGP_UNKNOWN_CONSTRUCT	<b>Value:</b> 1115
RNGP_UNKNOWN_TYPE_LIB	<b>Value:</b> 1116
RNGP_URL_FRAGMENT	<b>Value:</b> 1117
RNGP_URL_NOT_ABSOLUTE	<b>Value:</b> 1118
RNGP_VALUE_EMPTY	<b>Value:</b> 1119
RNGP_VALUE_NO_CONTENT	<b>Value:</b> 1120
RNGP_XMLNS_NAME	<b>Value:</b> 1121
RNGP_XML_NS	<b>Value:</b> 1122
SAVE_CHAR_INVALID	<b>Value:</b> 1401
SAVE_NOT_UTF8	<b>Value:</b> 1400
SAVE_NO_DOCTYPE	<b>Value:</b> 1402
SAVE_UNKNOWN_ENCODING	<b>Value:</b> 1403
SCHEMAP_AG_PROPS_CORRECT	<b>Value:</b> 3087
SCHEMAP_ATTRFORMDEFAULT_VALUE	<b>Value:</b> 1701
SCHEMAP_ATTRGRP_NONAME_NOREF	<b>Value:</b> 1702
SCHEMAP_ATTR_NONAME_NOREF	<b>Value:</b> 1703
SCHEMAP_AU_PROPS_CORRECT	<b>Value:</b> 3089
SCHEMAP_AU_PROPS_CORRECT_2	<b>Value:</b> 3078

*continued on next page*

Name	Description
SCHEMAP_A_PROPS_CORRECT_2	<b>Value:</b> 3079
SCHEMAP_A_PROPS_CORRECT_3	<b>Value:</b> 3090
SCHEMAP_COMPLEX_TYPE_NONAME_NOREF	<b>Value:</b> 1704
SCHEMAP_COS_ALLLIMITED	<b>Value:</b> 3091
SCHEMAP_COS_CT_EXTENDS_1_1	<b>Value:</b> 3063
SCHEMAP_COS_CT_EXTENDS_1_2	<b>Value:</b> 3088
SCHEMAP_COS_CT_EXTENDS_1_3	<b>Value:</b> 1800
SCHEMAP_COS_ST_DERIVED_OK_2_1	<b>Value:</b> 3031
SCHEMAP_COS_ST_DERIVED_OK_2_2	<b>Value:</b> 3032
SCHEMAP_COS_ST_RESTRICTS_1_1	<b>Value:</b> 3011
SCHEMAP_COS_ST_RESTRICTS_1_2	<b>Value:</b> 3012
SCHEMAP_COS_ST_RESTRICTS_1_3_1	<b>Value:</b> 3013
SCHEMAP_COS_ST_RESTRICTS_1_3_2	<b>Value:</b> 3014
SCHEMAP_COS_ST_RESTRICTS_2_1	<b>Value:</b> 3015
SCHEMAP_COS_ST_RESTRICTS_2_3_1_1	<b>Value:</b> 3016
SCHEMAP_COS_ST_RESTRICTS_2_3_1_2	<b>Value:</b> 3017
SCHEMAP_COS_ST_RESTRICTS_2_3_2_1	<b>Value:</b> 3018
SCHEMAP_COS_ST_RESTRICTS_2_3_2_2	<b>Value:</b> 3019
SCHEMAP_COS_ST_RESTRICTS_2_3_2_3	<b>Value:</b> 3020
SCHEMAP_COS_ST_RESTRICTS_2_3_2_4	<b>Value:</b> 3021

*continued on next page*

Name	Description
SCHEMAP_COS_ST_RESTRICTS_2_3_2_5	<b>Value:</b> 3022
SCHEMAP_COS_ST_RESTRICTS_3_1	<b>Value:</b> 3023
SCHEMAP_COS_ST_RESTRICTS_3_3_1	<b>Value:</b> 3024
SCHEMAP_COS_ST_RESTRICTS_3_3_1_2	<b>Value:</b> 3025
SCHEMAP_COS_ST_RESTRICTS_3_3_2_1	<b>Value:</b> 3027
SCHEMAP_COS_ST_RESTRICTS_3_3_2_2	<b>Value:</b> 3026
SCHEMAP_COS_ST_RESTRICTS_3_3_2_3	<b>Value:</b> 3028
SCHEMAP_COS_ST_RESTRICTS_3_3_2_4	<b>Value:</b> 3029
SCHEMAP_COS_ST_RESTRICTS_3_3_2_5	<b>Value:</b> 3030
SCHEMAP_COS_VALID_DEFAULT_1	<b>Value:</b> 3058
SCHEMAP_COS_VALID_DEFAULT_2_1	<b>Value:</b> 3059
SCHEMAP_COS_VALID_DEFAULT_2_2_1	<b>Value:</b> 3060
SCHEMAP_COS_VALID_DEFAULT_2_2_2	<b>Value:</b> 3061
SCHEMAP_CT_PROPS_CORRECT_1	<b>Value:</b> 1782
SCHEMAP_CT_PROPS_CORRECT_2	<b>Value:</b> 1783
SCHEMAP_CT_PROPS_CORRECT_3	<b>Value:</b> 1784
SCHEMAP_CT_PROPS_CORRECT_4	<b>Value:</b> 1785
SCHEMAP_CT_PROPS_CORRECT_5	<b>Value:</b> 1786
SCHEMAP_CVC_SIMPLE_TYPE	<b>Value:</b> 3062
SCHEMAP_C_PROPS_CORRECT	<b>Value:</b> 3080
SCHEMAP_DEF_AND_PREFIX	<b>Value:</b> 1768

*continued on next page*

Name	Description
SCHEMAP_DERIVATION_OK_RESTRICTION_1	<b>Value:</b> 1787
SCHEMAP_DERIVATION_OK_RESTRICTION_2.1.1	<b>Value:</b> 1788
SCHEMAP_DERIVATION_OK_RESTRICTION_2.1.2	<b>Value:</b> 1789
SCHEMAP_DERIVATION_OK_RESTRICTION_2.1.3	<b>Value:</b> 3077
SCHEMAP_DERIVATION_OK_RESTRICTION_2.2	<b>Value:</b> 1790
SCHEMAP_DERIVATION_OK_RESTRICTION_3	<b>Value:</b> 1791
SCHEMAP_DERIVATION_OK_RESTRICTION_4.1	<b>Value:</b> 1797
SCHEMAP_DERIVATION_OK_RESTRICTION_4.2	<b>Value:</b> 1798
SCHEMAP_DERIVATION_OK_RESTRICTION_4.3	<b>Value:</b> 1799
SCHEMAP_ELEMFORMDEFAULT_VALUE	<b>Value:</b> 1705
SCHEMAP_ELEM_DEFAULT_FIXED	<b>Value:</b> 1755
SCHEMAP_ELEM_NONAME_NOREF	<b>Value:</b> 1706
SCHEMAP_EXTENSION_NO_BASE	<b>Value:</b> 1707
SCHEMAP_E_PROPS_CORRECT_2	<b>Value:</b> 3045
SCHEMAP_E_PROPS_CORRECT_3	<b>Value:</b> 3046
SCHEMAP_E_PROPS_CORRECT_4	<b>Value:</b> 3047
SCHEMAP_E_PROPS_CORRECT_5	<b>Value:</b> 3048

*continued on next page*

<b>Name</b>	<b>Description</b>
SCHEMAP_E_PROPS_CORRECT_6	<b>Value:</b> 3049
SCHEMAP_FACET_NO_VALUE	<b>Value:</b> 1708
SCHEMAP_FAILED_BUILD_IMPORT	<b>Value:</b> 1709
SCHEMAP_FAILED_LOAD	<b>Value:</b> 1757
SCHEMAP_FAILED_PARSE	<b>Value:</b> 1766
SCHEMAP_GROUP_NO_NAME_NOREF	<b>Value:</b> 1710
SCHEMAP_IMPORT_NAMESPACE_NOT_URI	<b>Value:</b> 1711
SCHEMAP_IMPORT_REDEFINE_NSNAME	<b>Value:</b> 1712
SCHEMAP_IMPORT_SCHEMA_NOT_URI	<b>Value:</b> 1713
SCHEMAP_INCLUDE_SCHEMA_NOT_URI	<b>Value:</b> 1770
SCHEMAP_INCLUDE_SCHEMA_NO_URI	<b>Value:</b> 1771
SCHEMAP_INTERNAL	<b>Value:</b> 3069
SCHEMAP_INTERSECTION_NOT_EXPRESSIBLE	<b>Value:</b> 1793
SCHEMAP_INVALID_ATTR_COMBINATION	<b>Value:</b> 1777
SCHEMAP_INVALID_ATTR_INLINE_COMBINATION	<b>Value:</b> 1778
SCHEMAP_INVALID_ATTR_NAME	<b>Value:</b> 1780
SCHEMAP_INVALID_ATTR_USE	<b>Value:</b> 1774
SCHEMAP_INVALID_BOOLEAN	<b>Value:</b> 1714
SCHEMAP_INVALID_ENUM	<b>Value:</b> 1715
SCHEMAP_INVALID_FACET	<b>Value:</b> 1716

*continued on next page*

<b>Name</b>	<b>Description</b>
SCHEMAP_INVALID_F- ACET_VALUE	<b>Value:</b> 1717
SCHEMAP_INVALID_M- AXOCCURS	<b>Value:</b> 1718
SCHEMAP_INVALID_M- INOCCURS	<b>Value:</b> 1719
SCHEMAP_INVALID_R- EF_AND_SUBTYPE	<b>Value:</b> 1720
SCHEMAP_INVALID_- WHITE_SPACE	<b>Value:</b> 1721
SCHEMAP_MG_PROPS- _CORRECT_1	<b>Value:</b> 3074
SCHEMAP_MG_PROPS- _CORRECT_2	<b>Value:</b> 3075
SCHEMAP_MISSING_SI- MPLETYPE_CHILD	<b>Value:</b> 1779
SCHEMAP_NOATTR_N- OREF	<b>Value:</b> 1722
SCHEMAP_NOROOT	<b>Value:</b> 1759
SCHEMAP_NOTATION- _NO_NAME	<b>Value:</b> 1723
SCHEMAP_NOTHING_- TO_PARSE	<b>Value:</b> 1758
SCHEMAP_NOTYPE_N- OREF	<b>Value:</b> 1724
SCHEMAP_NOT_DETE- RMINISTIC	<b>Value:</b> 3070
SCHEMAP_NOT_SCHE- MA	<b>Value:</b> 1772
SCHEMAP_NO_XMLNS	<b>Value:</b> 3056
SCHEMAP_NO_XSI	<b>Value:</b> 3057
SCHEMAP_PREFIX_UN- DEFINED	<b>Value:</b> 1700
SCHEMAP_P_PROPS_C- ORRECT_1	<b>Value:</b> 3042
SCHEMAP_P_PROPS_C- ORRECT_2_1	<b>Value:</b> 3043
SCHEMAP_P_PROPS_C- ORRECT_2_2	<b>Value:</b> 3044
SCHEMAP_RECURSIV- E	<b>Value:</b> 1775

*continued on next page*

Name	Description
SCHEMAP_REDEFINE-D_ATTR	<b>Value:</b> 1764
SCHEMAP_REDEFINE-D_ATTRGROUP	<b>Value:</b> 1763
SCHEMAP_REDEFINE-D_ELEMENT	<b>Value:</b> 1762
SCHEMAP_REDEFINE-D_GROUP	<b>Value:</b> 1760
SCHEMAP_REDEFINE-D_NOTATION	<b>Value:</b> 1765
SCHEMAP_REDEFINE-D_TYPE	<b>Value:</b> 1761
SCHEMAP_REF_AND-CONTENT	<b>Value:</b> 1781
SCHEMAP_REF_AND-SUBTYPE	<b>Value:</b> 1725
SCHEMAP_REGEXP_I-NVALID	<b>Value:</b> 1756
SCHEMAP_RESTRICTION_NONAME_NOREF	<b>Value:</b> 1726
SCHEMAP_S4S_ATTR_I-NVALID_VALUE	<b>Value:</b> 3037
SCHEMAP_S4S_ATTR_-MISSING	<b>Value:</b> 3036
SCHEMAP_S4S_ATTR_-NOT_ALLOWED	<b>Value:</b> 3035
SCHEMAP_S4S_ELEM_-MISSING	<b>Value:</b> 3034
SCHEMAP_S4S_ELEM_-NOT_ALLOWED	<b>Value:</b> 3033
SCHEMAP_SIMPLETYPE_NONAME	<b>Value:</b> 1727
SCHEMAP_SRC_ATTRIBUTE_1	<b>Value:</b> 3051
SCHEMAP_SRC_ATTRIBUTE_2	<b>Value:</b> 3052
SCHEMAP_SRC_ATTRIBUTE_3_1	<b>Value:</b> 3053
SCHEMAP_SRC_ATTRIBUTE_3_2	<b>Value:</b> 3054
SCHEMAP_SRC_ATTRIBUTE_4	<b>Value:</b> 3055

*continued on next page*

Name	Description
SCHEMAP_SRC_ATTRIBUTE_GROUP_1	<b>Value:</b> 3071
SCHEMAP_SRC_ATTRIBUTE_GROUP_2	<b>Value:</b> 3072
SCHEMAP_SRC_ATTRIBUTE_GROUP_3	<b>Value:</b> 3073
SCHEMAP_SRC_CT_1	<b>Value:</b> 3076
SCHEMAP_SRC_ELEMENT_1	<b>Value:</b> 3038
SCHEMAP_SRC_ELEMENT_2_1	<b>Value:</b> 3039
SCHEMAP_SRC_ELEMENT_2_2	<b>Value:</b> 3040
SCHEMAP_SRC_ELEMENT_3	<b>Value:</b> 3041
SCHEMAP_SRC_IMPORT	<b>Value:</b> 3082
SCHEMAP_SRC_IMPORT_1_1	<b>Value:</b> 3064
SCHEMAP_SRC_IMPORT_1_2	<b>Value:</b> 3065
SCHEMAP_SRC_IMPORT_2	<b>Value:</b> 3066
SCHEMAP_SRC_IMPORT_2_1	<b>Value:</b> 3067
SCHEMAP_SRC_IMPORT_2_2	<b>Value:</b> 3068
SCHEMAP_SRC_IMPORT_3_1	<b>Value:</b> 1795
SCHEMAP_SRC_IMPORT_3_2	<b>Value:</b> 1796
SCHEMAP_SRC_INCLUDE	<b>Value:</b> 3050
SCHEMAP_SRC_LIST_ITEMTYPE_OR_SIMPLETYPE	<b>Value:</b> 3006
SCHEMAP_SRC_REDEFINE	<b>Value:</b> 3081
SCHEMAP_SRC_RESOLVE	<b>Value:</b> 3004

*continued on next page*

Name	Description
SCHEMAP_SRC_RESTRICTION_BASE_OR_SIMPLETYPE	<b>Value:</b> 3005
SCHEMAP_SRC_SIMPLETYPE_1	<b>Value:</b> 3000
SCHEMAP_SRC_SIMPLETYPE_2	<b>Value:</b> 3001
SCHEMAP_SRC_SIMPLETYPE_3	<b>Value:</b> 3002
SCHEMAP_SRC_SIMPLETYPE_4	<b>Value:</b> 3003
SCHEMAP_SRC_UNION_MEMBERTYPES_OR_SIMPLETYPES	<b>Value:</b> 3007
SCHEMAP_ST_PROPS_CORRECT_1	<b>Value:</b> 3008
SCHEMAP_ST_PROPS_CORRECT_2	<b>Value:</b> 3009
SCHEMAP_ST_PROPS_CORRECT_3	<b>Value:</b> 3010
SCHEMAP_SUPERNUMEROUS_LIST_ITEM_TYPE	<b>Value:</b> 1776
SCHEMAP_TYPE_AND_SUBTYPE	<b>Value:</b> 1728
SCHEMAP_UNION_NOT_EXPRESSIBLE	<b>Value:</b> 1794
SCHEMAP_UNKNOWN_ALL_CHILD	<b>Value:</b> 1729
SCHEMAP_UNKNOWN_ANYATTRIBUTE_CHILD	<b>Value:</b> 1730
SCHEMAP_UNKNOWN_ATTRGRP_CHILD	<b>Value:</b> 1732
SCHEMAP_UNKNOWN_ATTRIBUTE_GROUP	<b>Value:</b> 1733
SCHEMAP_UNKNOWN_ATTR_CHILD	<b>Value:</b> 1731
SCHEMAP_UNKNOWN_BASE_TYPE	<b>Value:</b> 1734
SCHEMAP_UNKNOWN_CHOICE_CHILD	<b>Value:</b> 1735

*continued on next page*

<b>Name</b>	<b>Description</b>
SCHEMAP_UNKNOWN-_COMPLEXCONTENT-_CHILD	<b>Value:</b> 1736
SCHEMAP_UNKNOWN-_COMPLEXTYPE-_CHILD	<b>Value:</b> 1737
SCHEMAP_UNKNOWN-_ELEM_CHILD	<b>Value:</b> 1738
SCHEMAP_UNKNOWN-_EXTENSION_CHILD	<b>Value:</b> 1739
SCHEMAP_UNKNOWN-_FACET_CHILD	<b>Value:</b> 1740
SCHEMAP_UNKNOWN-_FACET_TYPE	<b>Value:</b> 1741
SCHEMAP_UNKNOWN-_GROUP_CHILD	<b>Value:</b> 1742
SCHEMAP_UNKNOWN-_IMPORT_CHILD	<b>Value:</b> 1743
SCHEMAP_UNKNOWN-_INCLUDE_CHILD	<b>Value:</b> 1769
SCHEMAP_UNKNOWN-_LIST_CHILD	<b>Value:</b> 1744
SCHEMAP_UNKNOWN-_MEMBER_TYPE	<b>Value:</b> 1773
SCHEMAP_UNKNOWN-_NOTATION_CHILD	<b>Value:</b> 1745
SCHEMAP_UNKNOWN-_PREFIX	<b>Value:</b> 1767
SCHEMAP_UNKNOWN-_PROCESSCONTENT-_CHILD	<b>Value:</b> 1746
SCHEMAP_UNKNOWN-_REF	<b>Value:</b> 1747
SCHEMAP_UNKNOWN-_RESTRICTION_CHILD	<b>Value:</b> 1748
SCHEMAP_UNKNOWN-_SCHEMAS_CHILD	<b>Value:</b> 1749
SCHEMAP_UNKNOWN-_SEQUENCE_CHILD	<b>Value:</b> 1750
SCHEMAP_UNKNOWN-_SIMPLECONTENT-_CHILD	<b>Value:</b> 1751

*continued on next page*

<b>Name</b>	<b>Description</b>
SCHEMAP_UNKNOWN- _SIMPLETYPE_CHILD	<b>Value:</b> 1752
SCHEMAP_UNKNOWN- _TYPE	<b>Value:</b> 1753
SCHEMAP_UNKNOWN- _UNION_CHILD	<b>Value:</b> 1754
SCHEMAP_WARN_AT- TR_POINTLESS_PROH	<b>Value:</b> 3086
SCHEMAP_WARN_AT- TR_REDECL_PROH	<b>Value:</b> 3085
SCHEMAP_WARN_SKI- P_SCHEMA	<b>Value:</b> 3083
SCHEMAP_WARN_UN- LOCATED_SCHEMA	<b>Value:</b> 3084
SCHEMAP_WILDCARD- _INVALID_NS_MEMBE- R	<b>Value:</b> 1792
SCHEMATRONV_ASSE- RT	<b>Value:</b> 4000
SCHEMATRONV_REP- ORT	<b>Value:</b> 4001
SCHEMAV_ATTRINVA- LID	<b>Value:</b> 1821
SCHEMAV_ATTRUNK- NOWN	<b>Value:</b> 1820
SCHEMAV_CONSTRU- CT	<b>Value:</b> 1817
SCHEMAV_CVC_ATTR- IBUTE_1	<b>Value:</b> 1861
SCHEMAV_CVC_ATTR- IBUTE_2	<b>Value:</b> 1862
SCHEMAV_CVC_ATTR- IBUTE_3	<b>Value:</b> 1863
SCHEMAV_CVC_ATTR- IBUTE_4	<b>Value:</b> 1864
SCHEMAV_CVC_AU	<b>Value:</b> 1874
SCHEMAV_CVC_COMP- LEX_TYPE_1	<b>Value:</b> 1873
SCHEMAV_CVC_COMP- LEX_TYPE_2_1	<b>Value:</b> 1841
SCHEMAV_CVC_COMP- LEX_TYPE_2_2	<b>Value:</b> 1842

*continued on next page*

<b>Name</b>	<b>Description</b>
SCHEMAV_CVC_COMPLEX_TYPE_2_3	<b>Value:</b> 1843
SCHEMAV_CVC_COMPLEX_TYPE_2_4	<b>Value:</b> 1844
SCHEMAV_CVC_COMPLEX_TYPE_3_1	<b>Value:</b> 1865
SCHEMAV_CVC_COMPLEX_TYPE_3_2_1	<b>Value:</b> 1866
SCHEMAV_CVC_COMPLEX_TYPE_3_2_2	<b>Value:</b> 1867
SCHEMAV_CVC_COMPLEX_TYPE_4	<b>Value:</b> 1868
SCHEMAV_CVC_COMPLEX_TYPE_5_1	<b>Value:</b> 1869
SCHEMAV_CVC_COMPLEX_TYPE_5_2	<b>Value:</b> 1870
SCHEMAV_CVC_DATA_TYPE_VALID_1_2_1	<b>Value:</b> 1824
SCHEMAV_CVC_DATA_TYPE_VALID_1_2_2	<b>Value:</b> 1825
SCHEMAV_CVC_DATA_TYPE_VALID_1_2_3	<b>Value:</b> 1826
SCHEMAV_CVC_ELT_1	<b>Value:</b> 1845
SCHEMAV_CVC_ELT_2	<b>Value:</b> 1846
SCHEMAV_CVC_ELT_3_1	<b>Value:</b> 1847
SCHEMAV_CVC_ELT_3_2_1	<b>Value:</b> 1848
SCHEMAV_CVC_ELT_3_2_2	<b>Value:</b> 1849
SCHEMAV_CVC_ELT_4_1	<b>Value:</b> 1850
SCHEMAV_CVC_ELT_4_2	<b>Value:</b> 1851
SCHEMAV_CVC_ELT_4_3	<b>Value:</b> 1852
SCHEMAV_CVC_ELT_5_1_1	<b>Value:</b> 1853
SCHEMAV_CVC_ELT_5_1_2	<b>Value:</b> 1854
SCHEMAV_CVC_ELT_5_2_1	<b>Value:</b> 1855

*continued on next page*

Name	Description
SCHEMAV_CVC_ELT_5- _2_2_1	<b>Value:</b> 1856
SCHEMAV_CVC_ELT_5- _2_2_2_1	<b>Value:</b> 1857
SCHEMAV_CVC_ELT_5- _2_2_2_2	<b>Value:</b> 1858
SCHEMAV_CVC_ELT_6	<b>Value:</b> 1859
SCHEMAV_CVC_ELT_7	<b>Value:</b> 1860
SCHEMAV_CVC_ENUM- ERATION_VALID	<b>Value:</b> 1840
SCHEMAV_CVC_FACE- T_VALID	<b>Value:</b> 1829
SCHEMAV_CVC_FRAC- TIONDIGITS_VALID	<b>Value:</b> 1838
SCHEMAV_CVC_IDC	<b>Value:</b> 1877
SCHEMAV_CVC LENG- TH_VALID	<b>Value:</b> 1830
SCHEMAV_CVC_MAXE- XCLUSIVE_VALID	<b>Value:</b> 1836
SCHEMAV_CVC_MAXI- NCLUSIVE_VALID	<b>Value:</b> 1834
SCHEMAV_CVC_MAXL- LENGTH_VALID	<b>Value:</b> 1832
SCHEMAV_CVC_MINE- XCLUSIVE_VALID	<b>Value:</b> 1835
SCHEMAV_CVC_MINI- NCLUSIVE_VALID	<b>Value:</b> 1833
SCHEMAV_CVC_MINL- LENGTH_VALID	<b>Value:</b> 1831
SCHEMAV_CVC_PATT- ERN_VALID	<b>Value:</b> 1839
SCHEMAV_CVC_TOTA- LDIGITS_VALID	<b>Value:</b> 1837
SCHEMAV_CVC_TYPE- _1	<b>Value:</b> 1875
SCHEMAV_CVC_TYPE- _2	<b>Value:</b> 1876
SCHEMAV_CVC_TYPE- _3_1_1	<b>Value:</b> 1827
SCHEMAV_CVC_TYPE- _3_1_2	<b>Value:</b> 1828

*continued on next page*

<b>Name</b>	<b>Description</b>
SCHEMAV_CVC_WILDCARD	<b>Value:</b> 1878
SCHEMAV_DOCUMENT_ELEMENT_MISSING	<b>Value:</b> 1872
SCHEMAV_ELEMCONTENT	<b>Value:</b> 1810
SCHEMAV_ELEMENT_CONTENT	<b>Value:</b> 1871
SCHEMAV_EXTRACONTENT	<b>Value:</b> 1813
SCHEMAV_FACET	<b>Value:</b> 1823
SCHEMAV_HAVEDEFAULT	<b>Value:</b> 1811
SCHEMAV_INTERNAL	<b>Value:</b> 1818
SCHEMAV_INVALIDATTR	<b>Value:</b> 1814
SCHEMAV_INVALIDELEMENT	<b>Value:</b> 1815
SCHEMAV_ISABSTRACT	<b>Value:</b> 1808
SCHEMAV_MISC	<b>Value:</b> 1879
SCHEMAV_MISSING	<b>Value:</b> 1804
SCHEMAV_NOROLLBACK	<b>Value:</b> 1807
SCHEMAV_NOROOT	<b>Value:</b> 1801
SCHEMAV_NOTDETERMINIST	<b>Value:</b> 1816
SCHEMAV_NOTEMPTY	<b>Value:</b> 1809
SCHEMAV_NOTNILLABLE	<b>Value:</b> 1812
SCHEMAV_NOTSIMPLE	<b>Value:</b> 1819
SCHEMAV_NOTTOPELLEVEL	<b>Value:</b> 1803
SCHEMAV_NOTYPE	<b>Value:</b> 1806
SCHEMAV_UNDECLARED_ELEM	<b>Value:</b> 1802
SCHEMAV_VALUE	<b>Value:</b> 1822
SCHEMAV_WRONGELEMENT	<b>Value:</b> 1805

*continued on next page*

<b>Name</b>	<b>Description</b>
TREE_INVALID_DEC	<b>Value:</b> 1301
TREE_INVALID_HEX	<b>Value:</b> 1300
TREE_NOT_UTF8	<b>Value:</b> 1303
TREE_UNTERMINATED_ENTITY	<b>Value:</b> 1302
WAR_CATALOG_PI	<b>Value:</b> 93
WAR_ENTITY_REDEFINED	<b>Value:</b> 107
WAR_LANG_VALUE	<b>Value:</b> 98
WAR_NS_COLUMN	<b>Value:</b> 106
WAR_NS_URI	<b>Value:</b> 99
WAR_NS_URL_RELATIVE	<b>Value:</b> 100
WAR_SPACE_VALUE	<b>Value:</b> 102
WAR_UNDECLARED_ENTITY	<b>Value:</b> 27
WAR_UNKNOWN_VERSION	<b>Value:</b> 97
XINCLUDE_BUILD_FAILED	<b>Value:</b> 1609
XINCLUDE_DEPRECATED_NS	<b>Value:</b> 1617
XINCLUDE_ENTITY_DEF_MISMATCH	<b>Value:</b> 1602
XINCLUDE_FALLBACKS_IN_INCLUDE	<b>Value:</b> 1615
XINCLUDE_FALLBACK_NOT_IN_INCLUDE	<b>Value:</b> 1616
XINCLUDE_FRAGMENT_ID	<b>Value:</b> 1618
XINCLUDE_HREF_URI	<b>Value:</b> 1605
XINCLUDE_INCLUDE_IN_INCLUDE	<b>Value:</b> 1614
XINCLUDE_INVALID_CHAR	<b>Value:</b> 1608
XINCLUDE_MULTIPLE_ROOT	<b>Value:</b> 1611
XINCLUDE_NO_FALLBACK	<b>Value:</b> 1604
XINCLUDE_NO_HREF	<b>Value:</b> 1603

*continued on next page*

Name	Description
XINCLUDE_PARSE_VALUE	<b>Value:</b> 1601
XINCLUDE_RECURSION	<b>Value:</b> 1600
XINCLUDE_TEXT_DOCUMENT	<b>Value:</b> 1607
XINCLUDE_TEXT_FRAGMENT	<b>Value:</b> 1606
XINCLUDE_UNKNOWN_ENCODING	<b>Value:</b> 1610
XINCLUDE_XPTR_FAILED	<b>Value:</b> 1612
XINCLUDE_XPTR_RESULT	<b>Value:</b> 1613
XPATH_ENCODING_ERROR	<b>Value:</b> 1220
XPATH_EXPRESSION_OK	<b>Value:</b> 1200
XPATH_EXPR_ERROR	<b>Value:</b> 1207
XPATH_INVALID_ARITY	<b>Value:</b> 1212
XPATH_INVALID_CHAR_ERROR	<b>Value:</b> 1221
XPATH_INVALID_CONTEXT_POSITION	<b>Value:</b> 1214
XPATH_INVALID_CONTEXT_SIZE	<b>Value:</b> 1213
XPATH_INVALID_OPERAND	<b>Value:</b> 1210
XPATH_INVALID_PREDICATE_ERROR	<b>Value:</b> 1206
XPATH_INVALID_TYPE	<b>Value:</b> 1211
XPATH_MEMORY_ERROR	<b>Value:</b> 1215
XPATH_NUMBER_ERROR	<b>Value:</b> 1201
XPATH_START_LITERAL_ERROR	<b>Value:</b> 1203
XPATH_UNCLOSED_ERROR	<b>Value:</b> 1208

*continued on next page*

Name	Description
XPATH_UNDEF_PREFIX_ERROR	<b>Value:</b> 1219
XPATH_UNDEF_VARIABLE_ERROR	<b>Value:</b> 1205
XPATH_UNFINISHED_LITERAL_ERROR	<b>Value:</b> 1202
XPATH_UNKNOWN_FUNCTION_ERROR	<b>Value:</b> 1209
XPATH_VARIABLE_REF_ERROR	<b>Value:</b> 1204
XPTR_CHILDSEQ_START	<b>Value:</b> 1901
XPTR_EVAL_FAILED	<b>Value:</b> 1902
XPTR_EXTRA_OBJECTS	<b>Value:</b> 1903
XPTR_RESOURCE_ERROR	<b>Value:</b> 1217
XPTR_SUB_RESOURCE_ERROR	<b>Value:</b> 1218
XPTR_SYNTAX_ERROR	<b>Value:</b> 1216
XPTR_UNKNOWN_SCHEME	<b>Value:</b> 1900

### B.6.28 Class *FallbackElementClassLookup*

object

`lxml.etree.ElementClassLookup`

**`lxml.etree.FallbackElementClassLookup`**

**Known Subclasses:** `lxml.etree.AttributeBasedElementClassLookup`, `lxml.etree.CustomElementClassLookup`, `lxml.etree.ElementNamespaceClassLookup`, `lxml.etree.ParserBasedElementClassLookup`, `lxml.pyclasslookup.PythonElementClassLookup`

`FallbackElementClassLookup(self, fallback=None)`

Superclass of Element class lookups with additional fallback.

**Methods**

**`__init__(self, fallback=None)`**

`x.__init__(...)` initializes `x`; see `x.__class__.__doc__` for signature Overrides: `object.__init__`

**`__new__(T, S, ...)`**

**Return Value**

a new object with type `S`, a subtype of `T`

Overrides: `object.__new__`

**`setFallback(...)`**

Sets the fallback scheme for this lookup method. **Deprecated:** use `set_fallback()` instead.

**`set_fallback(self, lookup)`**

Sets the fallback scheme for this lookup method.

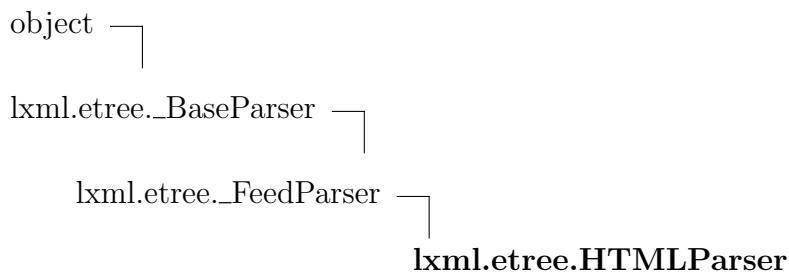
***Inherited from object***

`__delattr__()`, `__getattr__()`, `__hash__()`, `__reduce__()`, `__reduce_ex__()`, `__repr__()`, `__setattr__()`, `__str__()`

**Properties**

Name	Description
<code>fallback</code>	
<i>Inherited from object</i>	
<code>__class__</code>	

### B.6.29 Class `HTMLParser`



**Known Subclasses:** `lxml.html.HTMLParser`

`HTMLParser(self, recover=True, no_network=True, remove_blank_text=False, compact=True, remove_comments=False, remove_pis=False, target=None, encoding=None, schema=None)`  
The HTML parser.

This parser allows reading HTML into a normal XML tree. By default, it can read broken (non well-formed) HTML, depending on the capabilities of libxml2. Use the 'recover' option to switch this off.

Available boolean keyword arguments:

- `recover` - try hard to parse through broken HTML (default: `True`)
- `no_network` - prevent network access for related files (default: `True`)
- `remove_blank_text` - discard empty text nodes
- `remove_comments` - discard comments
- `remove_pis` - discard processing instructions
- `compact` - safe memory for short text content (default: `True`)

Other keyword arguments:

- `encoding` - override the document encoding
- `target` - a parser target object that will receive the parse events
- `schema` - an `XMLSchema` to validate against

Note that you should avoid sharing parsers between threads for performance reasons.

**Methods**

<b><code>__init__</code></b> ( <i>self</i> , <i>recover</i> =True, <i>no_network</i> =True, <i>remove_blank_text</i> =False, <i>compact</i> =True, <i>remove_comments</i> =False, <i>remove_pis</i> =False, <i>target</i> =None, <i>encoding</i> =None, <i>schema</i> =None)
--

<i>x</i> . <b><code>__init__</code></b> (...) initializes <i>x</i> ; see <i>x</i> . <b><code>__class__</code></b> . <b><code>__doc__</code></b> for signature Overrides: <i>object</i> . <b><code>__init__</code></b>
---

<b><code>__new__</code></b> ( <i>T</i> , <i>S</i> , ...)
--

**Return Value**

a new object with type *S*, a subtype of *T*

Overrides: *object*.**`__new__`**

***Inherited from `lxml.etree._FeedParser`***

`close()`, `feed()`

***Inherited from `lxml.etree._BaseParser`***

`copy()`, `makeelement()`, `setElementClassLookup()`, `set_element_class_lookup()`

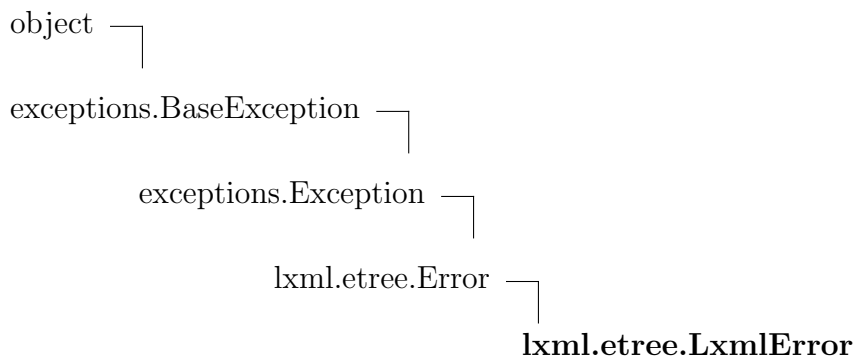
***Inherited from `object`***

`__delattr__()`, `__getattr__()`, `__hash__()`, `__reduce__()`, `__reduce_ex__()`, `__repr__()`, `__setattr__()`, `__str__()`

**Properties**

Name	Description
<i>Inherited from <code>lxml.etree._FeedParser</code></i> <code>feed_error_log</code>	
<i>Inherited from <code>lxml.etree._BaseParser</code></i> <code>error_log</code> , <code>resolvers</code> , <code>version</code>	
<i>Inherited from <code>object</code></i> <code>__class__</code>	

### B.6.30 Class LxmlError



**Known Subclasses:** lxml.etree.C14NError, lxml.etree.DTDError, lxml.etree.DocumentInvalid, lxml.etree.LxmlRegistryError, lxml.etree.LxmlSyntaxError, lxml.etree.ParserError, lxml.etree.RelaxNGError, lxml.etree.SchematronError, lxml.etree.XIncludeError, lxml.etree.XMLSchemaError, lxml.etree.XPathError, lxml.etree.XSLTError, lxml.sax.SaxError

Main exception base class for lxml. All other exceptions inherit from this one.

#### Methods

`__init__(...)`

`x.__init__(...)` initializes x; see `x.__class__.__doc__` for signature. Overrides: `object.__init__`, `exitit` (inherited documentation)

#### *Inherited from exceptions.Exception*

`__new__()`

#### *Inherited from exceptions.BaseException*

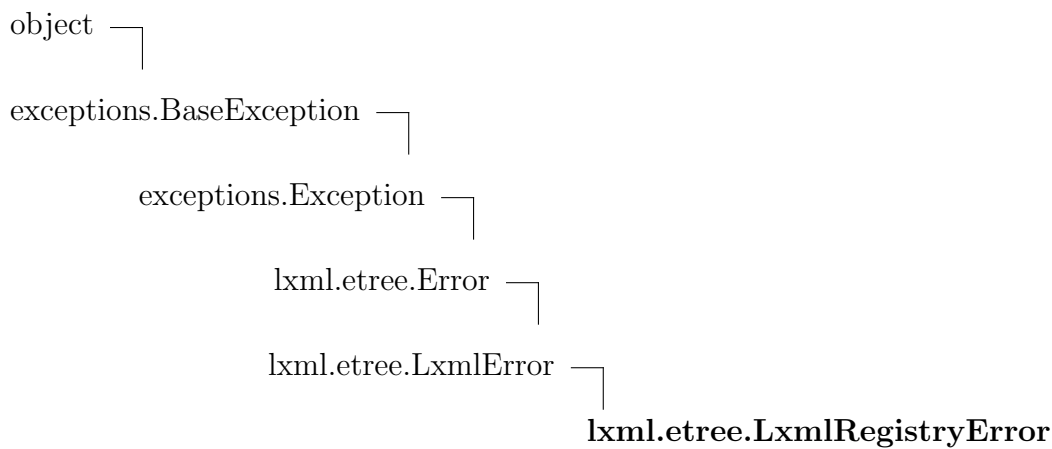
`__delattr__()`, `__getattr__()`, `__getitem__()`, `__getslice__()`, `__reduce__()`, `__repr__()`, `__setattr__()`, `__setstate__()`, `__str__()`

#### *Inherited from object*

`__hash__()`, `__reduce_ex__()`

#### Properties

Name	Description
<i>Inherited from exceptions.BaseException</i>	
	args, message
<i>Inherited from object</i>	
<code>__class__</code>	

**B.6.31 Class `LxmlRegistryError`**

**Known Subclasses:** `lxml.etree.NamespaceRegistryError`

Base class of lxml registry errors.

**Methods**

*Inherited from `lxml.etree.LxmlError` (Section [B.6.30](#))*

`__init__()`

*Inherited from `exceptions.Exception`*

`__new__()`

*Inherited from `exceptions.BaseException`*

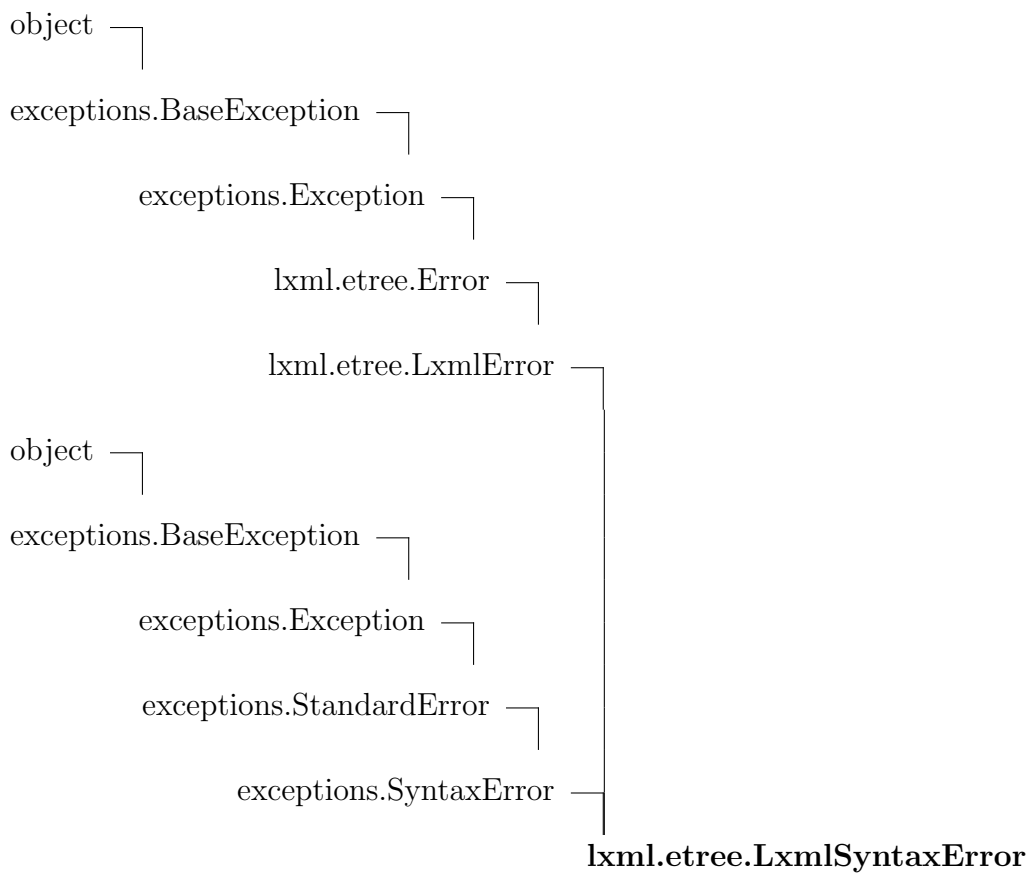
`__delattr__()`, `__getattr__()`, `__getitem__()`, `__getslice__()`, `__reduce__()`, `__repr__()`,  
`__setattr__()`, `__setstate__()`, `__str__()`

*Inherited from `object`*

`__hash__()`, `__reduce_ex__()`

**Properties**

Name	Description
<i>Inherited from <code>exceptions.BaseException</code></i>	
	args, message
<i>Inherited from <code>object</code></i>	
<code>__class__</code>	

**B.6.32 Class `LxmlSyntaxError`**

**Known Subclasses:** `lxml.etree.ParseError`, `lxml.etree.XPathSyntaxError`, `lxml.ElementInclude.FatalI`  
 Base class for all syntax errors.

**Methods**

**Inherited from `lxml.etree.LxmlError` (Section [B.6.30](#))**

`__init__()`

**Inherited from `exceptions.SyntaxError`**

`__new__()`, `__str__()`

**Inherited from `exceptions.BaseException`**

`__delattr__()`, `__getattr__()`, `__getitem__()`, `__getslice__()`, `__reduce__()`, `__repr__()`,  
`__setattr__()`, `__setstate__()`

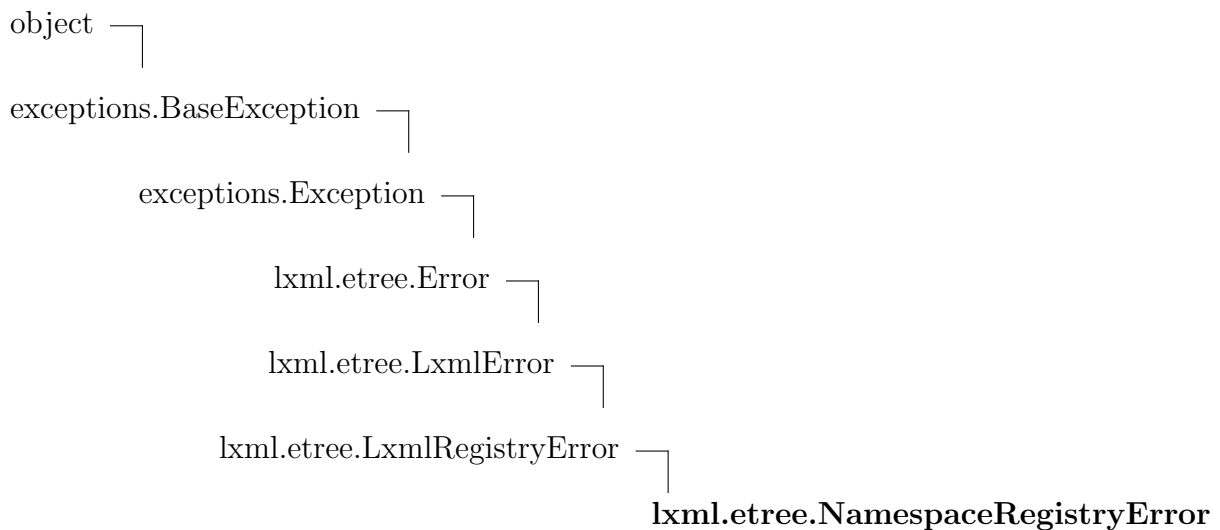
**Inherited from `object`**

`__hash__()`, `__reduce_ex__()`

## Properties

Name	Description
	<i>Inherited from exceptions.SyntaxError</i> filename, lineno, message, msg, offset, print_file_and_line, text
	<i>Inherited from exceptions.BaseException</i> args
	<i>Inherited from object</i> __class__

## B.6.33 Class NamespaceRegistryError



Error registering a namespace extension.

## Methods

*Inherited from lxml.etree.LxmlError(Section B.6.30)*

`__init__()`

*Inherited from exceptions.Exception*

`__new__()`

*Inherited from exceptions.BaseException*

`__delattr__()`, `__getattr__()`, `__getitem__()`, `__getslice__()`, `__reduce__()`, `__repr__()`,  
`__setattr__()`, `__setstate__()`, `__str__()`

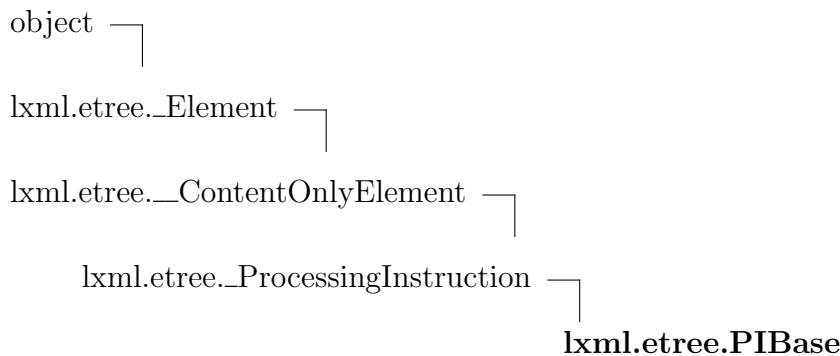
*Inherited from object*

`__hash__()`, `__reduce_ex__()`

### Properties

Name	Description
<i>Inherited from exceptions.BaseException</i>	
args, message	
<i>Inherited from object</i>	
<code>__class__</code>	

### B.6.34 Class PIBase



**Known Subclasses:** `lxml.etree._XSLTProcessingInstruction`, `lxml.html.HtmlProcessingInstruction`

All custom Processing Instruction classes must inherit from this one.

Note that you cannot (and must not) instantiate this class or its subclasses.

Subclasses *must not* override `__init__` or `__new__` as it is absolutely undefined when these objects will be created or destroyed. All persistent state of PIs must be stored in the underlying XML. If you really need to initialize the object after creation, you can implement an `__init__(self)` method that will be called after object creation.

### Methods

`__new__(T, S, ...)`

**Return Value**

a new object with type S, a subtype of T

Overrides: `object.__new__`

*Inherited from lxml.etree.\_ProcessingInstruction*

`__repr__()`

***Inherited from lxml.etree.\_ContentOnlyElement***

`__delitem__()`, `__getitem__()`, `__len__()`, `__setitem__()`, `append()`, `get()`, `insert()`, `items()`, `keys()`, `set()`, `values()`

***Inherited from lxml.etree.\_Element***

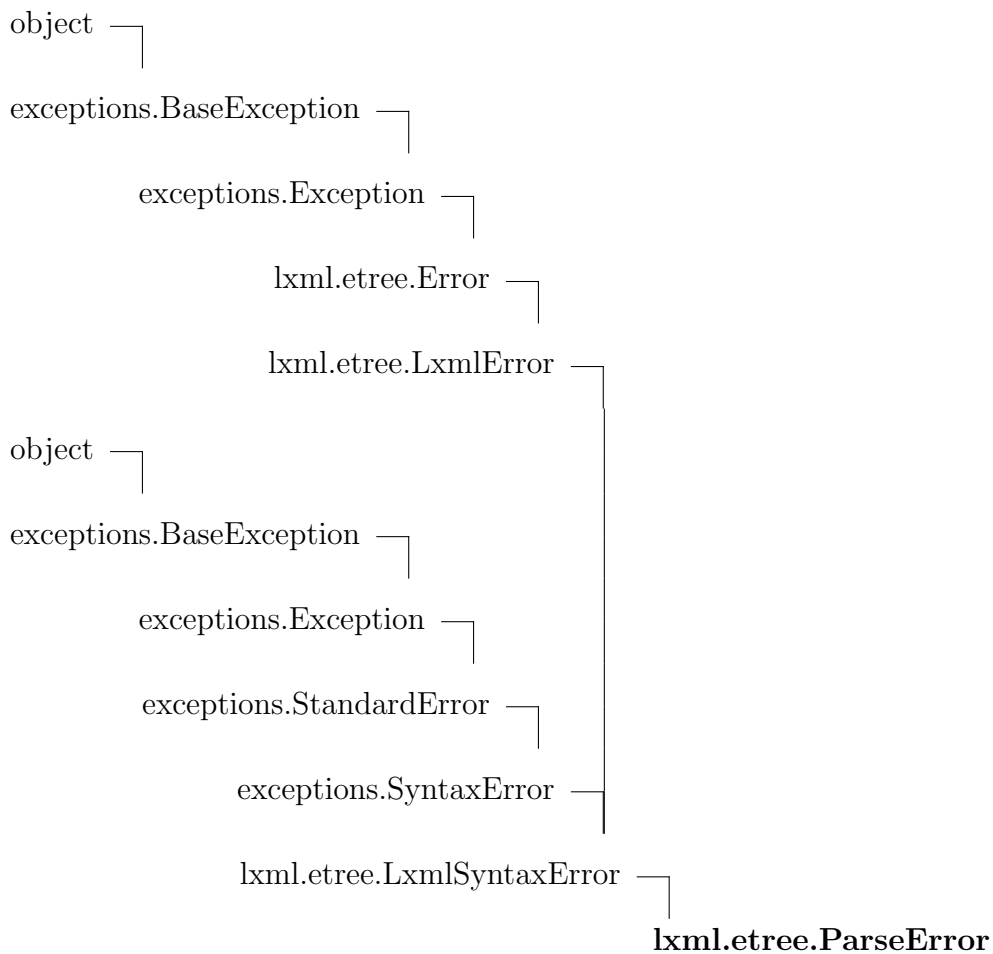
`__contains__()`, `__copy__()`, `__deepcopy__()`, `__iter__()`, `__nonzero__()`, `__reversed__()`, `addnext()`, `addprevious()`, `clear()`, `extend()`, `find()`, `findall()`, `findtext()`, `getchildren()`, `getiterator()`, `getnext()`, `getparent()`, `getprevious()`, `getroottree()`, `index()`, `iter()`, `iterancestors()`, `iterchildren()`, `iterdescendants()`, `iterfind()`, `itersiblings()`, `itertext()`, `makeelement()`, `remove()`, `replace()`, `xpath()`

***Inherited from object***

`__delattr__()`, `__getattr__()`, `__hash__()`, `__init__()`, `__reduce__()`, `__reduce_ex__()`, `__setattr__()`, `__str__()`

**Properties**

Name	Description
<i>Inherited from lxml.etree._ProcessingInstruction</i>	tag, target
<i>Inherited from lxml.etree._ContentOnlyElement</i>	attrib, text
<i>Inherited from lxml.etree._Element</i>	nsmmap, prefix, sourceline, tail
<i>Inherited from object</i>	<code>__class__</code>

**B.6.35 Class ParseError**

**Known Subclasses:** `lxml.etree.XMLSyntaxError`

Syntax error while parsing an XML document.

For compatibility with ElementTree 1.3 and later.

**Methods**

**Inherited from `lxml.etree.LxmlError` (Section [B.6.30](#))**

`__init__()`

**Inherited from `exceptions.SyntaxError`**

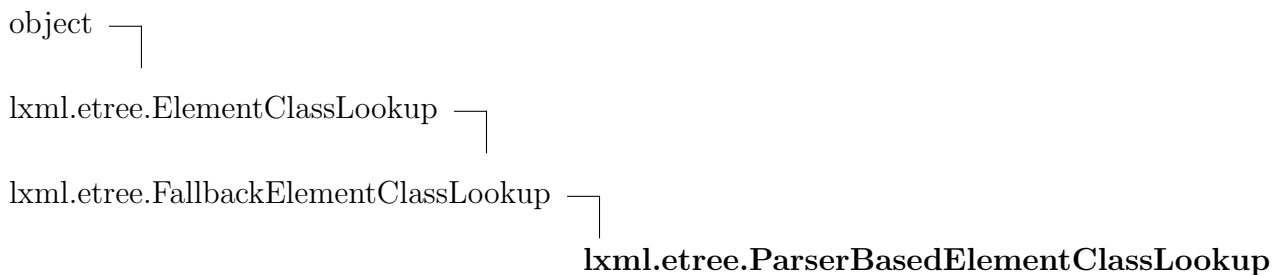
`__new__()`, `__str__()`

**Inherited from `exceptions.BaseException`**

`__delattr__()`, `__getattr__()`, `__getitem__()`, `__getslice__()`, `__reduce__()`, `__repr__()`, `__setattr__()`, `__setstate__()`

**Inherited from object**`__hash__()`, `__reduce_ex__()`**Properties**

Name	Description
<i>Inherited from exceptions.SyntaxError</i>	<code>filename</code> , <code>lineno</code> , <code>message</code> , <code>msg</code> , <code>offset</code> , <code>print_file_and_line</code> , <code>text</code>
<i>Inherited from exceptions.BaseException</i>	<code>args</code>
<i>Inherited from object</i>	<code>__class__</code>

**B.6.36 Class ParserBasedElementClassLookup**

`ParserBasedElementClassLookup(self, fallback=None)` Element class lookup based on the XML parser.

**Methods**

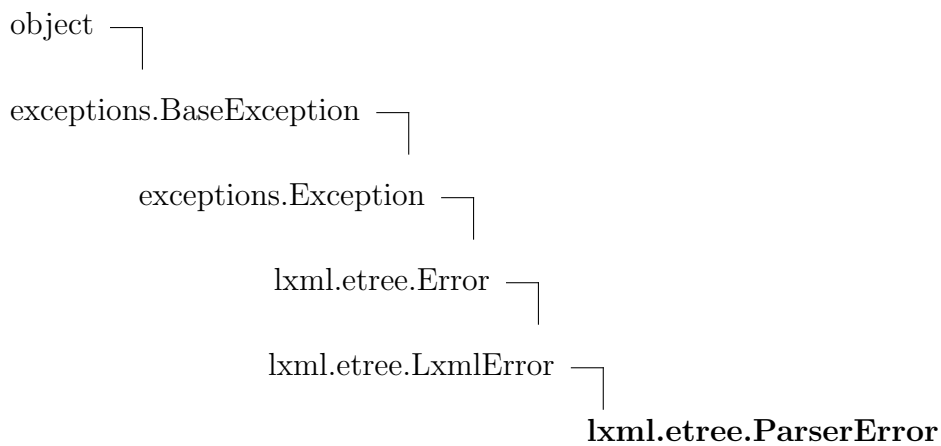
<b><code>__new__(T, S, ...)</code></b> <b>Return Value</b> a new object with type S, a subtype of T Overrides: <code>object.__new__</code>
---

*Inherited from `lxml.etree.FallbackElementClassLookup` (Section [B.6.28](#))*

`__init__()`, `setFallback()`, `set_fallback()`**Inherited from object**`__delattr__()`, `__getattr__()`, `__hash__()`, `__reduce__()`, `__reduce_ex__()`, `__repr__()`, `__setattr__()`, `__str__()`

**Properties**

Name	Description
<code>fallback</code>	<i>Inherited from <code>lxml.etree.FallbackElementClassLookup</code> (Section <a href="#">B.6.28</a>)</i>
<code>__class__</code>	<i>Inherited from <code>object</code></i>

**B.6.37 Class `ParserError`**

Internal lxml parser error.

**Methods**

*Inherited from `lxml.etree.LxmlError` (Section [B.6.30](#))*

`__init__()`

*Inherited from `exceptions.Exception`*

`__new__()`

*Inherited from `exceptions.BaseException`*

`__delattr__()`, `__getattr__()`, `__getitem__()`, `__getslice__()`, `__reduce__()`, `__repr__()`,  
`__setattr__()`, `__setstate__()`, `__str__()`

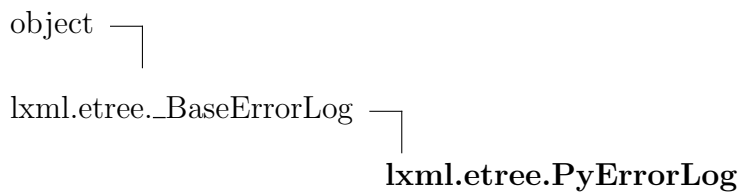
*Inherited from `object`*

`__hash__()`, `__reduce_ex__()`

**Properties**

Name	Description
<i>Inherited from exceptions.BaseException</i>	
args, message	
<i>Inherited from object</i>	
__class__	

### B.6.38 Class PyErrorLog



PyErrorLog(self, logger\_name=None) A global error log that connects to the Python stdlib logging package.

The constructor accepts an optional logger name.

If you want to change the mapping between libxml2's ErrorLevels and Python logging levels, you can modify the level\_map dictionary from a subclass.

The default mapping is:

```

ErrorLevels.WARNING = logging.WARNING
ErrorLevels.ERROR    = logging.ERROR
ErrorLevels.FATAL    = logging.CRITICAL
  
```

You can also override the method receive() that takes a LogEntry object and calls self.log(log\_entry, format\_string, arg1, arg2, ...) with appropriate data.

#### Methods

<b>__init__(self, logger_name=None)</b> <hr/> x.__init__(...) initializes x; see x.__class__.__doc__ for signature Overrides: object.__init__
--

<b>__new__(T, S, ...)</b> <b>Return Value</b> a new object with type S, a subtype of T Overrides: object.__new__
---

**copy(...)**

Dummy method that returns an empty error log. Overrides:  
lxml.etree.\_BaseErrorLog.copy

**log(...)****receive(...)*****Inherited from lxml.etree.\_BaseErrorLog***`__repr__()`***Inherited from object***

`__delattr__()`, `__getattr__()`, `__hash__()`, `__reduce__()`, `__reduce_ex__()`, `__setattr__()`, `__str__()`

**Properties**

Name	Description
level_map	
<i>Inherited from lxml.etree._BaseErrorLog</i>	
last_error	
<i>Inherited from object</i>	
__class__	

**B.6.39 Class QName**

object └─  
          **lxml.etree.QName**

QName(text\_or\_uri, tag=None)

QName wrapper.

Pass a tag name by itself or a namespace URI and a tag name to create a qualified name. The `text` property holds the qualified name in `{namespace}tagname` notation.

You can pass QName objects wherever a tag name is expected. Also, setting Element text from a QName will resolve the namespace prefix and set a qualified text value.

## Methods

**`__eq__(x, y)`**  
-----  
`x==y`

**`__ge__(x, y)`**  
-----  
`x>=y`

**`__gt__(x, y)`**  
-----  
`x>y`

**`__hash__(x)`**  
-----  
`hash(x)` Overrides: `object.__hash__`

**`__init__(text_or_uri, tag=None)`**  
-----  
`x.__init__(...)` initializes x; see `x.__class__.__doc__` for signature Overrides:  
`object.__init__`

**`__le__(x, y)`**  
-----  
`x<=y`

**`__lt__(x, y)`**  
-----  
`x<y`

**`__ne__(x, y)`**  
-----  
`x!=y`

```
__new__(T, S, ...)
```

**Return Value**

a new object with type S, a subtype of T

Overrides: object.\_\_new\_\_

```
__str__(...)
```

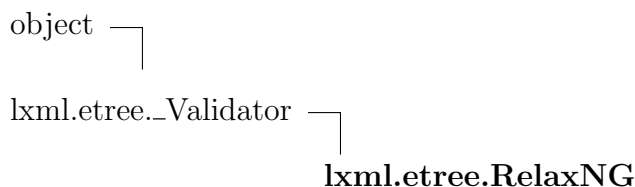
str(x) Overrides: object.\_\_str\_\_ exitit(inherited documentation)

**Inherited from object**

`__delattr__()`, `__getattr__()`, `__reduce__()`, `__reduce_ex__()`, `__repr__()`, `__setattr__()`

**Properties**

Name	Description
text	
<i>Inherited from object</i>	
<code>__class__</code>	

**B.6.40 Class RelaxNG**

RelaxNG(self, etree=None, file=None) Turn a document into a Relax NG validator.

Either pass a schema as Element or ElementTree, or pass a file or filename through the file keyword argument.

**Methods**

```
__call__(self, etree)
```

Validate doc using Relax NG.

Returns true if document is valid, false if not.

<code>__init__(self, etree=None, file=None)</code>
<code>x.__init__(...)</code> initializes x; see <code>x.__class__.__doc__</code> for signature Overrides: <code>object.__init__</code>

<code>__new__(T, S, ...)</code>
<b>Return Value</b> a new object with type S, a subtype of T
Overrides: <code>object.__new__</code>

**Inherited from *lxml.etree.\_Validator***

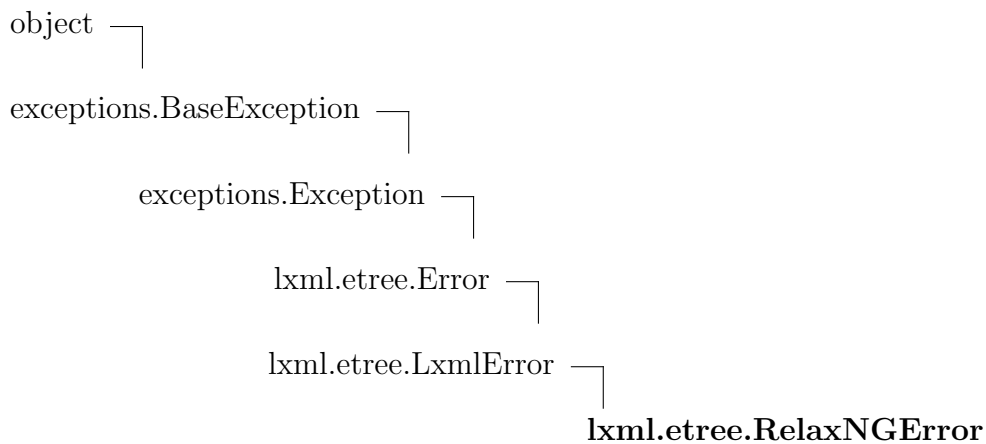
`assertValid()`, `assert_()`, `validate()`

**Inherited from *object***

`__delattr__()`, `__getattr__()`, `__hash__()`, `__reduce__()`, `__reduce_ex__()`, `__repr__()`, `__setattr__()`, `__str__()`

**Properties**

Name	Description
<i>Inherited from <code>lxml.etree._Validator</code></i>	
<code>error_log</code>	
<i>Inherited from <code>object</code></i>	
<code>__class__</code>	

**B.6.41 Class RelaxNGError**

**Known Subclasses:** `lxml.etree.RelaxNGParseError`, `lxml.etree.RelaxNGValidateError`

Base class for RelaxNG errors.

#### Methods

*Inherited from lxml.etree.LxmlError(Section B.6.30)*

`__init__()`

*Inherited from exceptions.Exception*

`__new__()`

*Inherited from exceptions.BaseException*

`__delattr__()`, `__getattr__()`, `__getitem__()`, `__getslice__()`, `__reduce__()`, `__repr__()`, `__setattr__()`, `__setstate__()`, `__str__()`

*Inherited from object*

`__hash__()`, `__reduce_ex__()`

#### Properties

Name	Description
<i>Inherited from exceptions.BaseException</i>	
args, message	
<i>Inherited from object</i>	
<code>__class__</code>	

### B.6.42 Class RelaxNGErrorTypes

Libxml2 RelaxNG error types

#### Class Variables

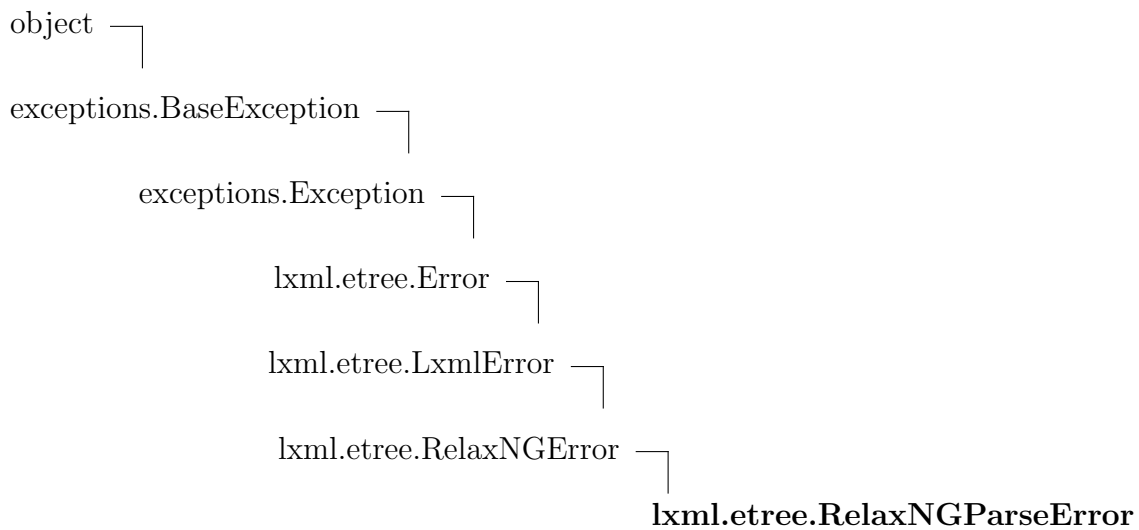
Name	Description
RELAXNG_ERR_ATTR-EXTRANS	<b>Value:</b> 20
RELAXNG_ERR_ATTR-NAME	<b>Value:</b> 14
RELAXNG_ERR_ATTR-NONS	<b>Value:</b> 16
RELAXNG_ERR_ATTR-VALID	<b>Value:</b> 24

*continued on next page*

Name	Description
RELAXNG_ERR_ATTR-WRONGNS	<b>Value:</b> 18
RELAXNG_ERR_CONTENTVALID	<b>Value:</b> 25
RELAXNG_ERR_DATA-ELEM	<b>Value:</b> 28
RELAXNG_ERR_DATA-TYPE	<b>Value:</b> 31
RELAXNG_ERR_DUPI-D	<b>Value:</b> 4
RELAXNG_ERR_ELEM-EXTRANS	<b>Value:</b> 19
RELAXNG_ERR_ELEM-NAME	<b>Value:</b> 13
RELAXNG_ERR_ELEM-NONS	<b>Value:</b> 15
RELAXNG_ERR_ELEM-NOTEMPTY	<b>Value:</b> 21
RELAXNG_ERR_ELEM-WRONG	<b>Value:</b> 38
RELAXNG_ERR_ELEM-WRONGNS	<b>Value:</b> 17
RELAXNG_ERR_EXTR-ACONTENT	<b>Value:</b> 26
RELAXNG_ERR_EXTR-ADATA	<b>Value:</b> 35
RELAXNG_ERR_INTE-REXTRA	<b>Value:</b> 12
RELAXNG_ERR_INTE-RNAL	<b>Value:</b> 37
RELAXNG_ERR_INTE-RNODATA	<b>Value:</b> 10
RELAXNG_ERR_INTE-RSEQ	<b>Value:</b> 11
RELAXNG_ERR_INVA-LIDATTR	<b>Value:</b> 27
RELAXNG_ERR_LACK-DATA	<b>Value:</b> 36
RELAXNG_ERR_LIST	<b>Value:</b> 33
RELAXNG_ERR_LISTE-LEM	<b>Value:</b> 30

*continued on next page*

<b>Name</b>	<b>Description</b>
RELAXNG_ERR_LISTE-MPTY	<b>Value: 9</b>
RELAXNG_ERR_LISTE-XTRA	<b>Value: 8</b>
RELAXNG_ERR_MEMORY	<b>Value: 1</b>
RELAXNG_ERR_NODE-FINE	<b>Value: 7</b>
RELAXNG_ERR_NOEL-EM	<b>Value: 22</b>
RELAXNG_ERR_NOGR-AMMAR	<b>Value: 34</b>
RELAXNG_ERR_NOST-ATE	<b>Value: 6</b>
RELAXNG_ERR_NOTE-LEM	<b>Value: 23</b>
RELAXNG_ERR_TEXT-WRONG	<b>Value: 39</b>
RELAXNG_ERR_TYPE	<b>Value: 2</b>
RELAXNG_ERR_TYPE-CMP	<b>Value: 5</b>
RELAXNG_ERR_TYPE-VAL	<b>Value: 3</b>
RELAXNG_ERR_VALE-LEM	<b>Value: 29</b>
RELAXNG_ERR_VALU-E	<b>Value: 32</b>
RELAXNG_OK	<b>Value: 0</b>

**B.6.43 Class RelaxNGParseError**

Error while parsing an XML document as RelaxNG.

**Methods**

*Inherited from `lxml.etree.LxmlError` (Section [B.6.30](#))*

`__init__()`

*Inherited from `exceptions.Exception`*

`__new__()`

*Inherited from `exceptions.BaseException`*

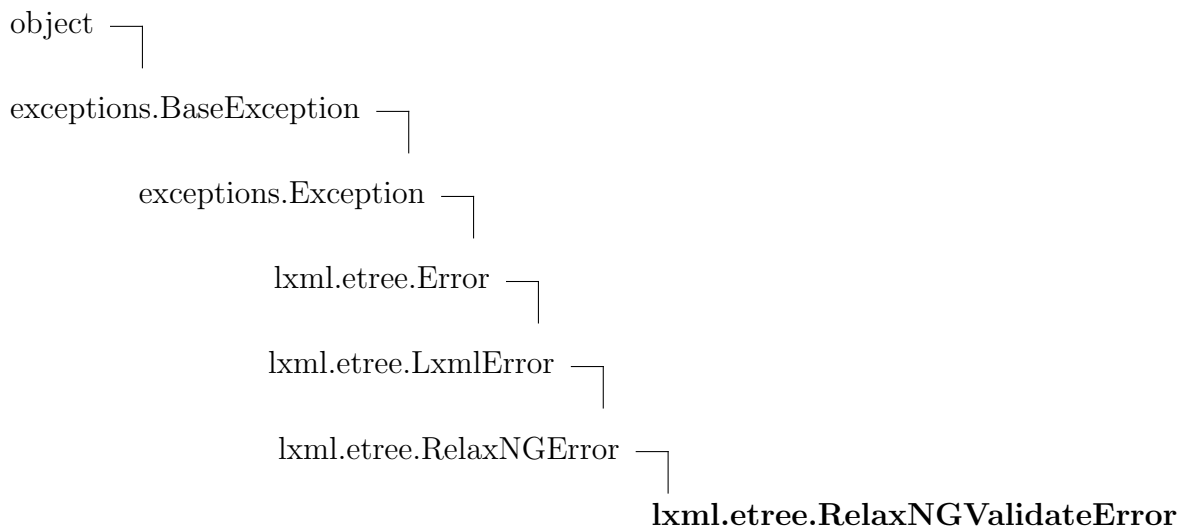
`__delattr__()`, `__getattr__()`, `__getitem__()`, `__getslice__()`, `__reduce__()`, `__repr__()`,  
`__setattr__()`, `__setstate__()`, `__str__()`

*Inherited from `object`*

`__hash__()`, `__reduce_ex__()`

**Properties**

Name	Description
<i>Inherited from <code>exceptions.BaseException</code></i>	
args, message	
<i>Inherited from <code>object</code></i>	
<code>__class__</code>	

**B.6.44 Class `RelaxNGValidateError`**

Error while validating an XML document with a RelaxNG schema.

**Methods**

*Inherited from `lxml.etree.LxmlError` (Section [B.6.30](#))*

`__init__()`

*Inherited from `exceptions.Exception`*

`__new__()`

*Inherited from `exceptions.BaseException`*

`__delattr__()`, `__getattr__()`, `__getitem__()`, `__getslice__()`, `__reduce__()`, `__repr__()`,  
`__setattr__()`, `__setstate__()`, `__str__()`

*Inherited from `object`*

`__hash__()`, `__reduce_ex__()`

**Properties**

Name	Description
<i>Inherited from <code>exceptions.BaseException</code></i>	
args, message	
<i>Inherited from <code>object</code></i>	
<code>__class__</code>	

### B.6.45 Class Resolver



This is the base class of all resolvers.

#### Methods

**\_\_new\_\_**(*T, S, ...*)

**Return Value**

a new object with type S, a subtype of T

Overrides: object.\_\_new\_\_

**resolve**(*self, system\_url, public\_id, context*)

Override this method to resolve an external source by `system_url` and `public_id`. The third argument is an opaque context object.

Return the result of one of the `resolve_*()` methods.

**resolve\_empty**(*self, context*)

Return an empty input document.

Pass context as parameter.

**resolve\_file**(*self, f, context, base\_url=None*)

Return an open file-like object as input document.

Pass open file and context as parameters.

**resolve\_filename**(*self, filename, context*)

Return the name of a parsable file as input document.

Pass filename and context as parameters.

<b>resolve_string</b> ( <i>self</i> , <i>string</i> , <i>context</i> , <i>base_url</i> =None)
---

Return a parsable string as input document.

Pass data string and context as parameters.

You can pass the source URL as 'base\_url' keyword.

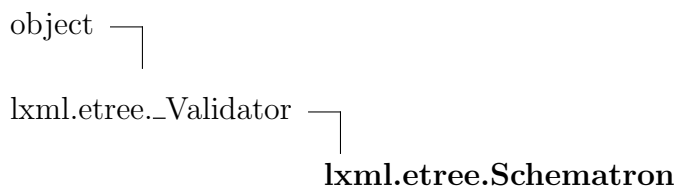
### *Inherited from object*

`__delattr__()`, `__getattr__()`, `__hash__()`, `__init__()`, `__reduce__()`, `__reduce_ex__()`,  
`__repr__()`, `__setattr__()`, `__str__()`

### Properties

Name	Description
<i>Inherited from object</i>	
<code>__class__</code>	

## B.6.46 Class Schematron



`Schematron(self, etree=None, file=None)` A Schematron validator.

Pass a root Element or an ElementTree to turn it into a validator. Alternatively, pass a filename as keyword argument 'file' to parse from the file system.

### Methods

<b>__call__</b> ( <i>self</i> , <i>etree</i> )
--

Validate doc using Schematron.

Returns true if document is valid, false if not.

<code>__init__(self, etree=None, file=None)</code>
<code>x.__init__(...)</code> initializes x; see <code>x.__class__.__doc__</code> for signature Overrides: <code>object.__init__</code>

<code>__new__(T, S, ...)</code>
<b>Return Value</b> a new object with type S, a subtype of T
Overrides: <code>object.__new__</code>

**Inherited from *lxml.etree.\_Validator***

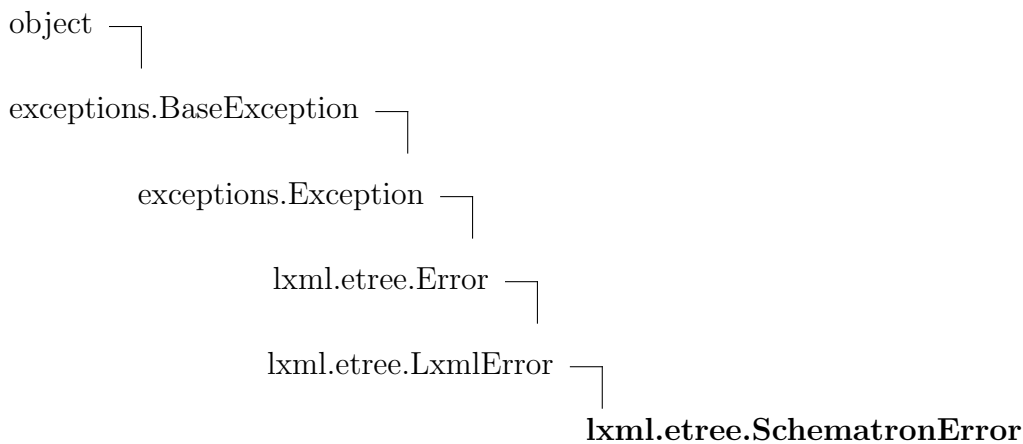
`assertValid()`, `assert_()`, `validate()`

**Inherited from *object***

`__delattr__()`, `__getattr__()`, `__hash__()`, `__reduce__()`, `__reduce_ex__()`, `__repr__()`, `__setattr__()`, `__str__()`

**Properties**

Name	Description
<i>Inherited from <code>lxml.etree._Validator</code></i>	
<code>error_log</code>	
<i>Inherited from <code>object</code></i>	
<code>__class__</code>	

**B.6.47 Class SchematronError**

**Known Subclasses:** `lxml.etree.SchematronParseError`, `lxml.etree.SchematronValidateError`

Base class of all Schematron errors.

### Methods

*Inherited from lxml.etree.LxmlError(Section B.6.30)*

`__init__()`

*Inherited from exceptions.Exception*

`__new__()`

*Inherited from exceptions.BaseException*

`__delattr__()`, `__getattr__()`, `__getitem__()`, `__getslice__()`, `__reduce__()`, `__repr__()`,  
`__setattr__()`, `__setstate__()`, `__str__()`

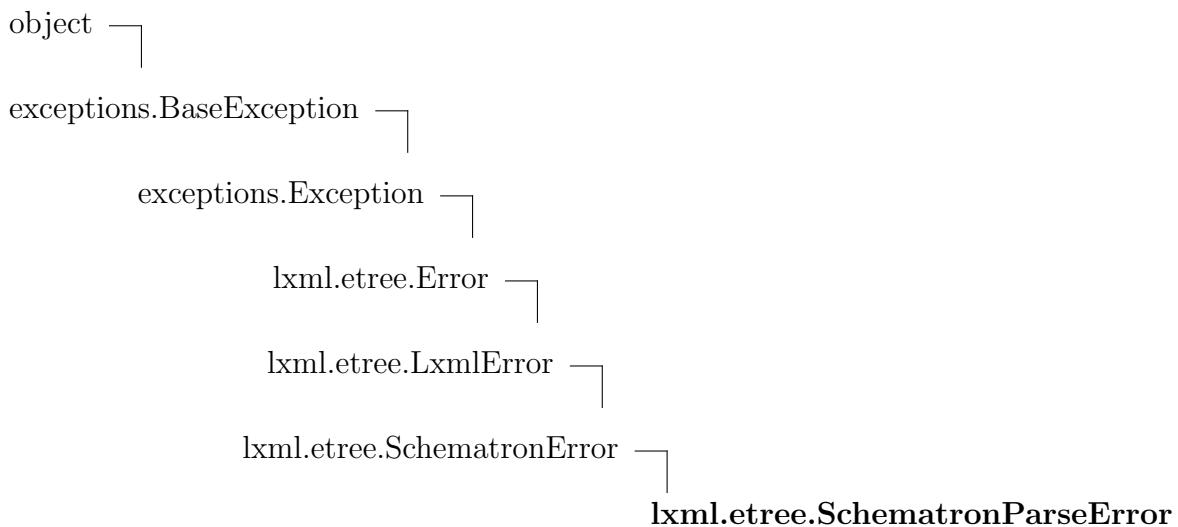
*Inherited from object*

`__hash__()`, `__reduce_ex__()`

### Properties

Name	Description
<i>Inherited from exceptions.BaseException</i>	
args, message	
<i>Inherited from object</i>	
<code>__class__</code>	

## B.6.48 Class SchematronParseError



Error while parsing an XML document as Schematron schema.

#### Methods

*Inherited from lxml.etree.LxmlError(Section B.6.30)*

`__init__()`

*Inherited from exceptions.Exception*

`__new__()`

*Inherited from exceptions.BaseException*

`__delattr__()`, `__getattr__()`, `__getitem__()`, `__getslice__()`, `__reduce__()`, `__repr__()`,  
`__setattr__()`, `__setstate__()`, `__str__()`

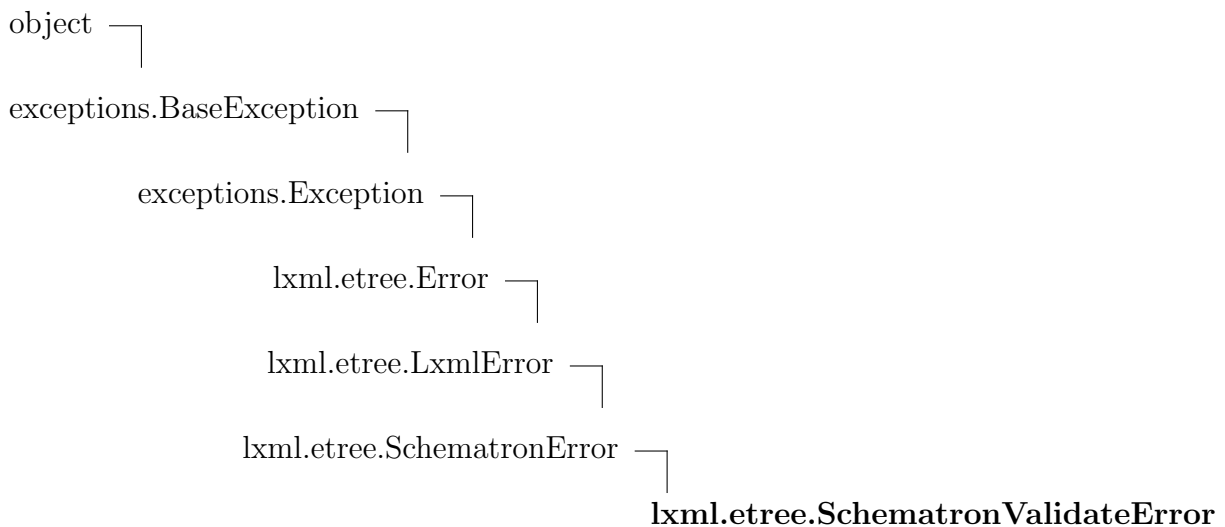
*Inherited from object*

`__hash__()`, `__reduce_ex__()`

#### Properties

Name	Description
<i>Inherited from exceptions.BaseException</i>	args, message
<i>Inherited from object</i>	<code>__class__</code>

### B.6.49 Class SchematronValidateError



Error while validating an XML document with a Schematron schema.

## Methods

*Inherited from `lxml.etree.LxmlError` (Section [B.6.30](#))*

`__init__()`

*Inherited from `exceptions.Exception`*

`__new__()`

*Inherited from `exceptions.BaseException`*

`__delattr__()`, `__getattr__()`, `__getitem__()`, `__getslice__()`, `__reduce__()`, `__repr__()`, `__setattr__()`, `__setstate__()`, `__str__()`

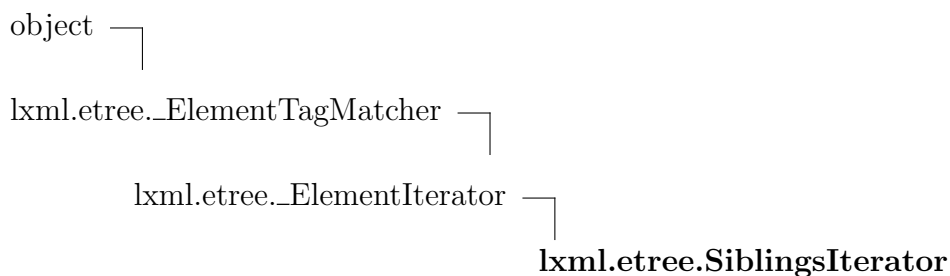
*Inherited from `object`*

`__hash__()`, `__reduce_ex__()`

## Properties

Name	Description
<i>Inherited from <code>exceptions.BaseException</code></i>	<i>args, message</i>
<i>Inherited from <code>object</code></i>	<i>__class__</i>

## B.6.50 Class SiblingsIterator



`SiblingsIterator(self, node, tag=None, preceding=False)` Iterates over the siblings of an element.

You can pass the boolean keyword `preceding` to specify the direction.

**Methods**

<code>__init__(self, node, tag=None, preceding=False)</code>
--

<code>x.__init__(...)</code> initializes x; see <code>x.__class__.__doc__</code> for signature Overrides: <code>object.__init__</code>
---

<code>__new__(T, S, ...)</code>
---------------------------------

**Return Value**

a new object with type S, a subtype of T

Overrides: `object.__new__`

**Inherited from *lxml.etree.\_ElementIterator***

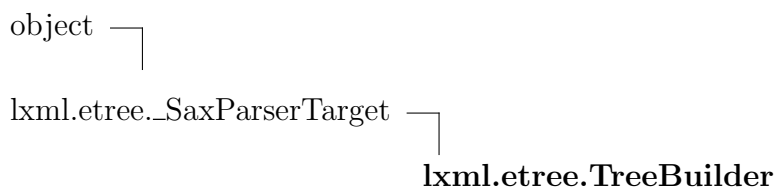
`__iter__()`, `__next__()`, `next()`

**Inherited from *object***

`__delattr__()`, `__getattr__()`, `__hash__()`, `__reduce__()`, `__reduce_ex__()`, `__repr__()`,  
`__setattr__()`, `__str__()`

**Properties**

Name	Description
<i>Inherited from object</i>	
<code>__class__</code>	

**B.6.51 Class *TreeBuilder***

`TreeBuilder(self, element_factory=None, parser=None)` Parser target that builds a tree.

The final tree is returned by the `close()` method.

**Methods**

**`__init__(self, element_factory=None, parser=None)`**

`x.__init__(...)` initializes `x`; see `x.__class__.__doc__` for signature Overrides: `object.__init__`

**`__new__(T, S, ...)`**

**Return Value**

a new object with type `S`, a subtype of `T`

Overrides: `object.__new__`

**`close(self)`**

Flushes the builder buffers, and returns the toplevel document element.

**`comment(self, comment)`**

**`data(self, data)`**

Adds text to the current element. The value should be either an 8-bit string containing ASCII text, or a Unicode string.

**`end(self, tag)`**

Closes the current element.

**`pi(self, target, data)`**

**`start(self, tag, attrs, nsmmap=None)`**

Opens a new element.

***Inherited from object***

`__delattr__()`, `__getattr__()`, `__hash__()`, `__reduce__()`, `__reduce_ex__()`, `__repr__()`, `__setattr__()`, `__str__()`

**Properties**

Name	Description
<i>Inherited from object</i>	
<code>__class__</code>	

**B.6.52 Class XInclude**

XInclude(self) XInclude processor.

Create an instance and call it on an Element to run XInclude processing.

**Methods**

<code>__call__(self, node)</code>
-----------------------------------

<code>__init__(self)</code>
<hr/>
<code>x.__init__(...)</code> initializes x; see <code>x.__class__.__doc__</code> for signature Overrides: <code>object.__init__</code>

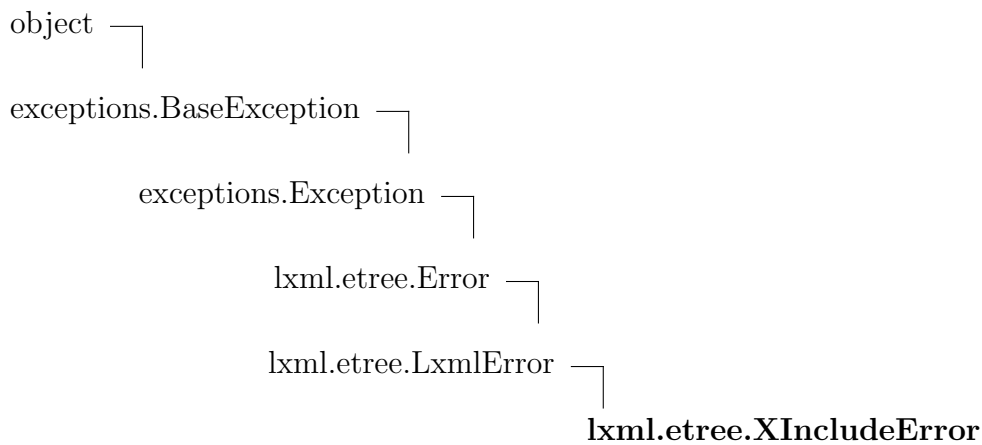
<code>__new__(T, S, ...)</code>
<b>Return Value</b>
a new object with type S, a subtype of T
Overrides: <code>object.__new__</code>

***Inherited from object***

`__delattr__()`, `__getattr__()`, `__hash__()`, `__reduce__()`, `__reduce_ex__()`, `__repr__()`, `__setattr__()`, `__str__()`

**Properties**

Name	Description
<code>error_log</code>	
<i>Inherited from object</i>	
<code>__class__</code>	

**B.6.53 Class XIncludeError**

Error during XInclude processing.

**Methods**

*Inherited from lxml.etree.LxmlError(Section B.6.30)*

`__init__()`

*Inherited from exceptions.Exception*

`__new__()`

*Inherited from exceptions.BaseException*

`__delattr__()`, `__getattr__()`, `__getitem__()`, `__getslice__()`, `__reduce__()`, `__repr__()`,  
`__setattr__()`, `__setstate__()`, `__str__()`

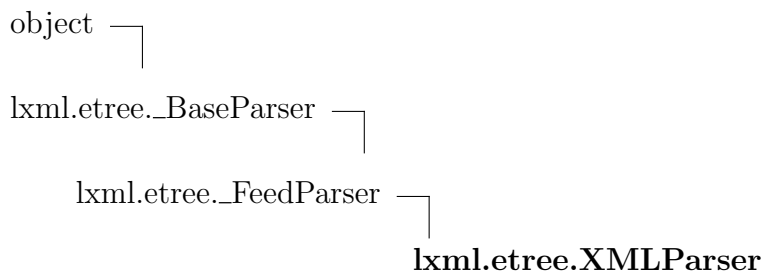
*Inherited from object*

`__hash__()`, `__reduce_ex__()`

**Properties**

Name	Description
<i>Inherited from exceptions.BaseException</i>	
	args, message
<i>Inherited from object</i>	
<code>__class__</code>	

### B.6.54 Class XMLParser



**Known Subclasses:** lxml.etree.ETCompatXMLParser

XMLParser(self, attribute\_defaults=False, dtd\_validation=False, load\_dtd=False, no\_network=True, ns\_clean=False, recover=False, remove\_blank\_text=False, compact=True, resolve\_entities=True, remove\_comments=False, remove\_pis=False, target=None, encoding=None, schema=None)

The XML parser.

Parsers can be supplied as additional argument to various parse functions of the lxml API. A default parser is always available and can be replaced by a call to the global function 'set\_default\_parser'. New parsers can be created at any time without a major run-time overhead.

The keyword arguments in the constructor are mainly based on the libxml2 parser configuration. A DTD will also be loaded if validation or attribute default values are requested.

Available boolean keyword arguments:

- attribute\_defaults - read default attributes from DTD
- dtd\_validation - validate (if DTD is available)
- load\_dtd - use DTD for parsing
- no\_network - prevent network access for related files (default: True)
- ns\_clean - clean up redundant namespace declarations
- recover - try hard to parse through broken XML
- remove\_blank\_text - discard blank text nodes
- remove\_comments - discard comments
- remove\_pis - discard processing instructions
- compact - safe memory for short text content (default: True)
- resolve\_entities - replace entities by their text value (default: True)

Other keyword arguments:

- encoding - override the document encoding

- target - a parser target object that will receive the parse events
- schema - an XMLSchema to validate against

Note that you should avoid sharing parsers between threads. While this is not harmful, it is more efficient to use separate parsers. This does not apply to the default parser.

## Methods

```
__init__(self, attribute_defaults=False, dtd_validation=False,
load_dtd=False, no_network=True, ns_clean=False, recover=False,
remove_blank_text=False, compact=True, resolve_entities=True,
remove_comments=False, remove_pis=False, target=None,
encoding=None, schema=None)
```

x.\_\_init\_\_(...) initializes x; see x.\_\_class\_\_.\_\_doc\_\_ for signature Overrides: object.\_\_init\_\_

```
__new__(T, S, ...)
```

### Return Value

a new object with type S, a subtype of T

Overrides: object.\_\_new\_\_

### Inherited from *lxml.etree.\_FeedParser*

close(), feed()

### Inherited from *lxml.etree.\_BaseParser*

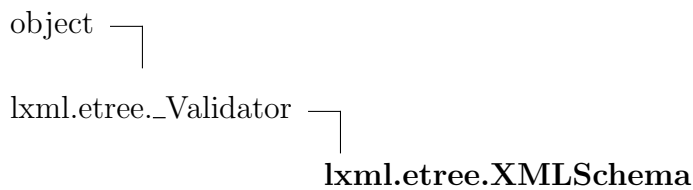
copy(), makeelement(), setElementClassLookup(), set\_element\_class\_lookup()

### Inherited from *object*

\_\_delattr\_\_(), \_\_getattr\_\_(), \_\_hash\_\_(), \_\_reduce\_\_(), \_\_reduce\_ex\_\_(), \_\_repr\_\_(), \_\_setattr\_\_(), \_\_str\_\_()

## Properties

Name	Description
<i>Inherited from lxml.etree._FeedParser</i> feed_error_log	
<i>Inherited from lxml.etree._BaseParser</i> error_log, resolvers, version	
<i>Inherited from object</i> __class__	

**B.6.55 Class XMLSchema**

XMLSchema(self, etree=None, file=None) Turn a document into an XML Schema validator.

Either pass a schema as Element or ElementTree, or pass a file or filename through the `file` keyword argument.

**Methods**

<p><code>__call__(self, etree)</code></p> <hr/> <p>Validate doc using XML Schema.</p> <p>Returns true if document is valid, false if not.</p>
---

<p><code>__init__(self, etree=None, file=None)</code></p> <hr/> <p><code>x.__init__(...)</code> initializes x; see <code>x.__class__.__doc__</code> for signature Overrides: <code>object.__init__</code></p>
---

<p><code>__new__(T, S, ...)</code></p> <p><b>Return Value</b> a new object with type S, a subtype of T</p> <p>Overrides: <code>object.__new__</code></p>
--

**Inherited from lxml.etree.\_Validator**

`assertValid()`, `assert_()`, `validate()`

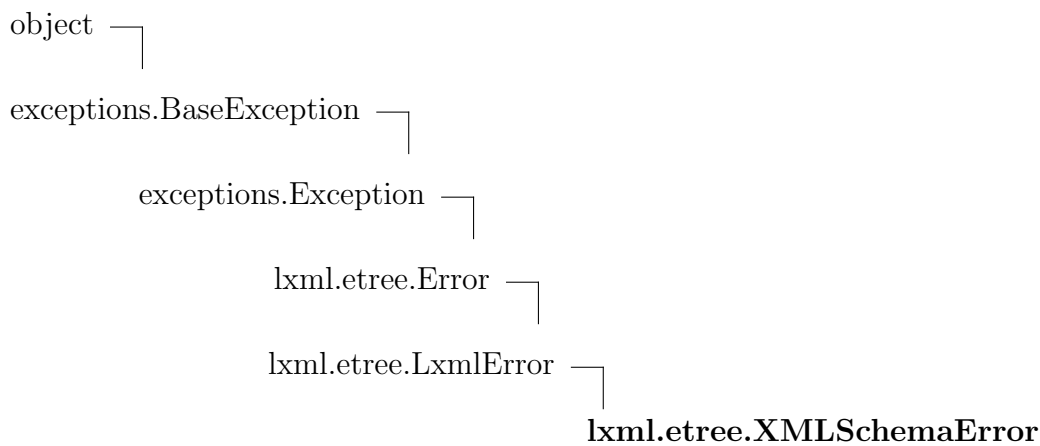
**Inherited from object**

`__delattr__()`, `__getattr__()`, `__hash__()`, `__reduce__()`, `__reduce_ex__()`, `__repr__()`, `__setattr__()`, `__str__()`

**Properties**

Name	Description
<i>Inherited from lxml.etree._Validator</i>	
error_log	
<i>Inherited from object</i>	
__class__	

### B.6.56 Class XMLSchemaError



**Known Subclasses:** lxml.etree.XMLSchemaParseError, lxml.etree.XMLSchemaValidateError

Base class of all XML Schema errors

#### Methods

*Inherited from lxml.etree.LxmlError(Section B.6.30)*

`__init__()`

*Inherited from exceptions.Exception*

`__new__()`

*Inherited from exceptions.BaseException*

`__delattr__()`, `__getattr__()`, `__getitem__()`, `__getslice__()`, `__reduce__()`, `__repr__()`,  
`__setattr__()`, `__setstate__()`, `__str__()`

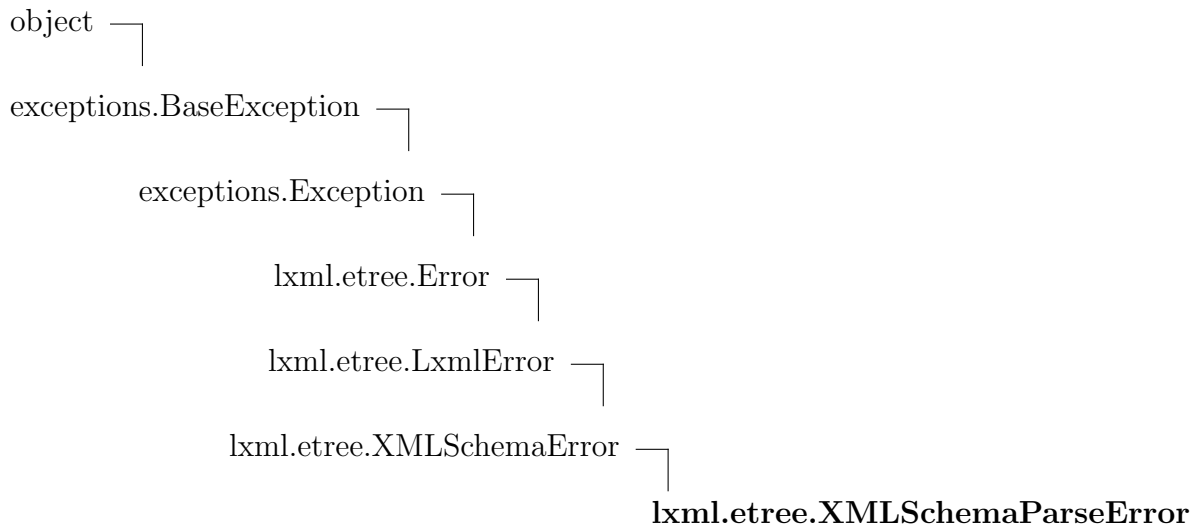
*Inherited from object*

`__hash__()`, `__reduce_ex__()`

#### Properties

Name	Description
	<i>Inherited from exceptions.BaseException</i> args, message
	<i>Inherited from object</i> __class__

### B.6.57 Class XMLSchemaParseError



Error while parsing an XML document as XML Schema.

#### Methods

*Inherited from lxml.etree.LxmlError(Section B.6.30)*

`__init__()`

*Inherited from exceptions.Exception*

`__new__()`

*Inherited from exceptions.BaseException*

`__delattr__()`, `__getattr__()`, `__getitem__()`, `__getslice__()`, `__reduce__()`, `__repr__()`,  
`__setattr__()`, `__setstate__()`, `__str__()`

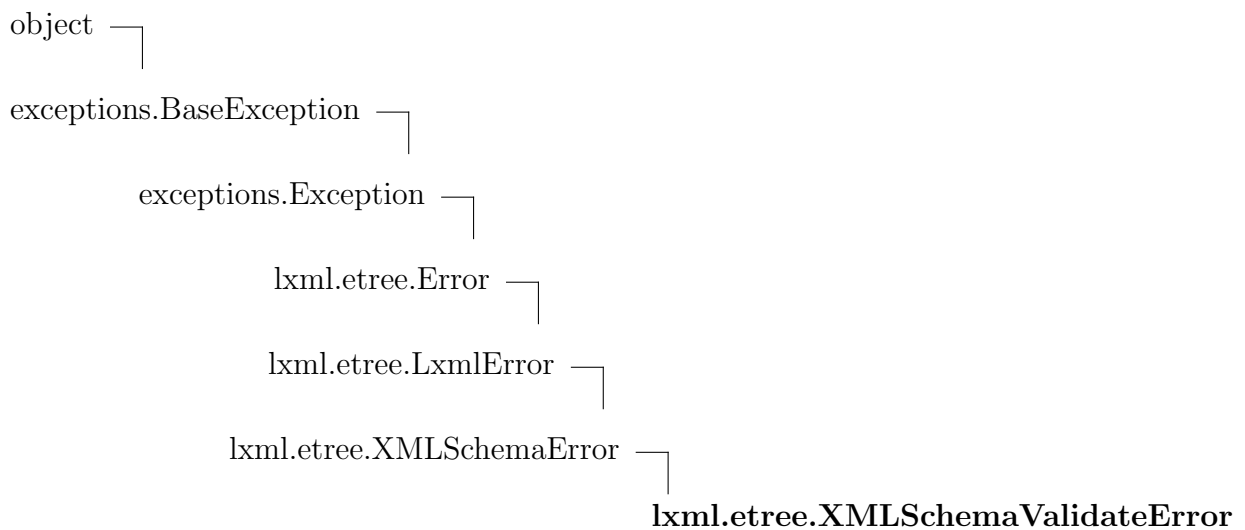
*Inherited from object*

`__hash__()`, `__reduce_ex__()`

#### Properties

Name	Description
	<i>Inherited from exceptions.BaseException</i> args, message
	<i>Inherited from object</i> __class__

### B.6.58 Class XMLSchemaValidateError



Error while validating an XML document with an XML Schema.

#### Methods

*Inherited from lxml.etree.LxmlError(Section B.6.30)*

`__init__()`

*Inherited from exceptions.Exception*

`__new__()`

*Inherited from exceptions.BaseException*

`__delattr__()`, `__getattr__()`, `__getitem__()`, `__getslice__()`, `__reduce__()`, `__repr__()`,  
`__setattr__()`, `__setstate__()`, `__str__()`

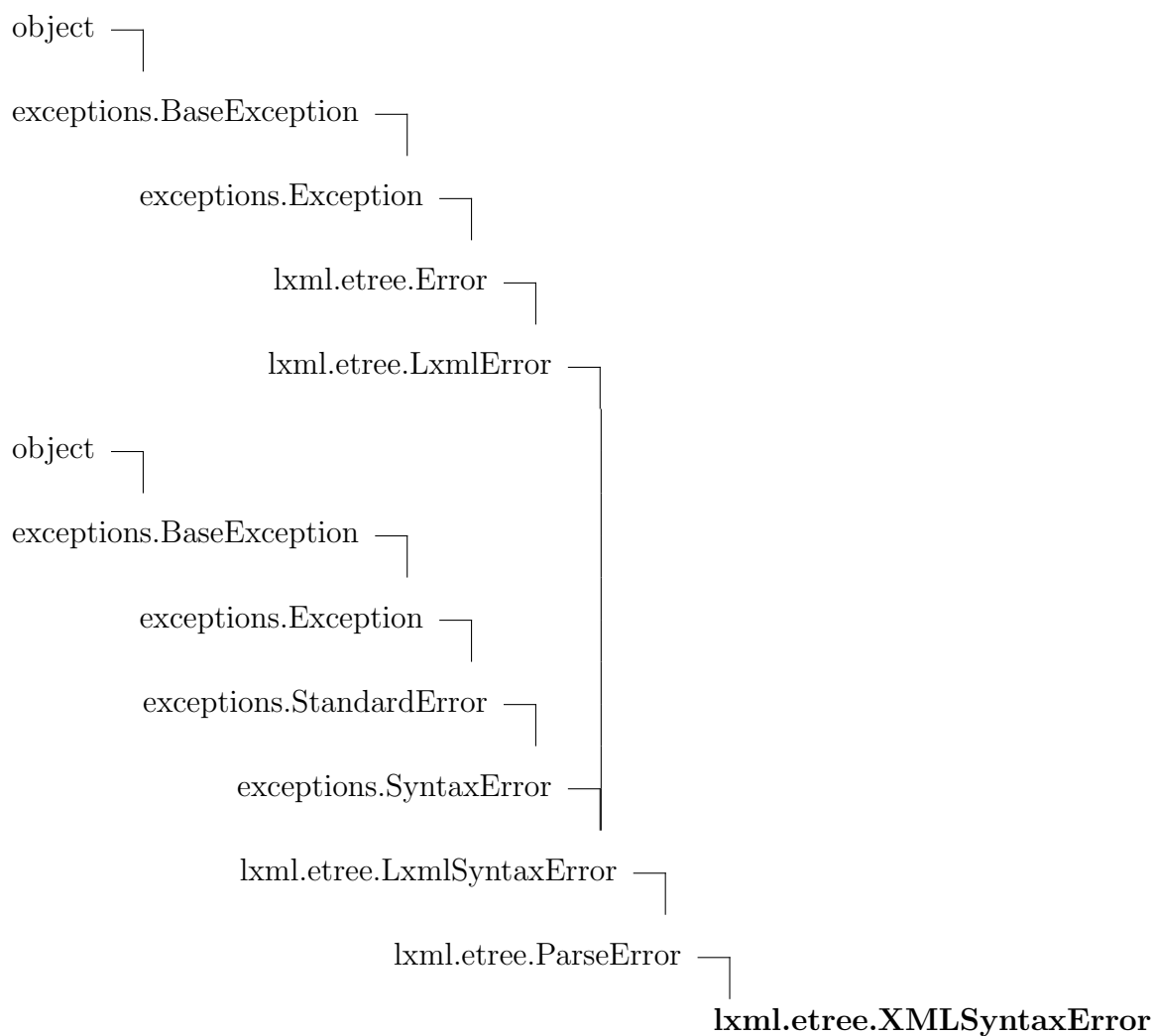
*Inherited from object*

`__hash__()`, `__reduce_ex__()`

#### Properties

Name	Description
	<i>Inherited from exceptions.BaseException</i>
	args, message
	<i>Inherited from object</i>
<code>__class__</code>	

### B.6.59 Class XMLSyntaxError



Syntax error while parsing an XML document.

## Methods

`__init__(...)`

`x.__init__(...)` initializes x; see `x.__class__.__doc__` for signature. Overrides: `object.__init__` `exitit` (inherited documentation)

**Inherited from exceptions.SyntaxError**

`__new__()`, `__str__()`

**Inherited from exceptions.BaseException**

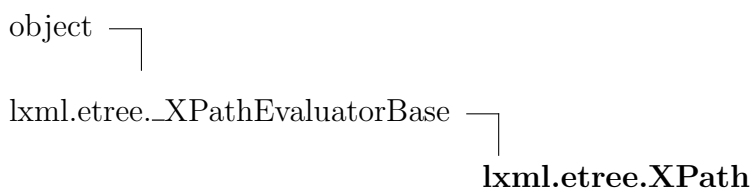
`__delattr__()`, `__getattr__()`, `__getitem__()`, `__getslice__()`, `__reduce__()`, `__repr__()`, `__setattr__()`, `__setstate__()`

**Inherited from object**

`__hash__()`, `__reduce_ex__()`

## Properties

Name	Description
<i>Inherited from exceptions.SyntaxError</i>	
	<code>filename</code> , <code>lineno</code> , <code>message</code> , <code>msg</code> , <code>offset</code> , <code>print_file_and_line</code> , <code>text</code>
<i>Inherited from exceptions.BaseException</i>	
	<code>args</code>
<i>Inherited from object</i>	
	<code>__class__</code>

**B.6.60 Class XPath**

**Known Subclasses:** `lxml.etree.ETXPath`, `lxml.cssselect.CSSSelector`

`XPath(self, path, namespaces=None, extensions=None, regexp=True)` A compiled XPath expression that can be called on `Elements` and `ElementTrees`.

Besides the XPath expression, you can pass prefix-namespace mappings and extension functions to the constructor through the keyword arguments `namespaces` and `extensions`. EXSLT regular expression support can be disabled with the `'regexp'` boolean keyword (defaults to `True`).

## Methods

<code>__call__(self, _etree_or_element, **_variables)</code>
--

<code>__init__(self, path, namespaces=None, extensions=None, regexp=True)</code>
--

x.__init__(...) initializes x; see x.__class__.__doc__ for signature Overrides: object.__init__
--

<code>__new__(T, S, ...)</code>
---------------------------------

**Return Value**

a new object with type S, a subtype of T

Overrides: object.\_\_new\_\_

<code>__repr__(...)</code>
----------------------------

repr(x) Overrides: object.__repr__ exitit(inherited documentation)
--

**Inherited from lxml.etree.\_XPathEvaluatorBase**

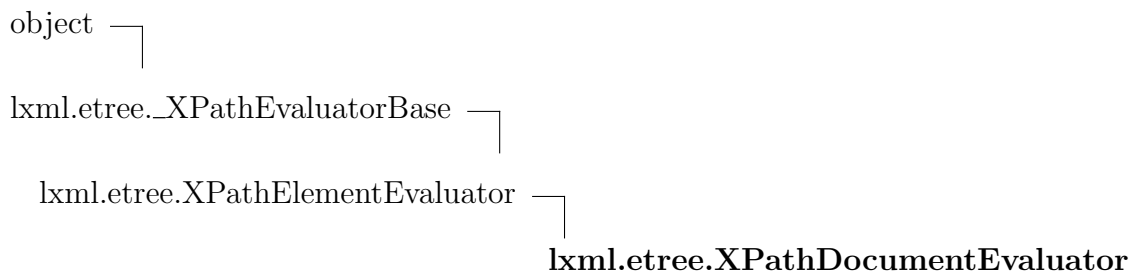
evaluate()

**Inherited from object**

`__delattr__()`, `__getattr__()`, `__hash__()`, `__reduce__()`, `__reduce_ex__()`, `__setattr__()`, `__str__()`

## Properties

Name	Description
path	
<i>Inherited from lxml.etree._XPathEvaluatorBase</i>	
error_log	
<i>Inherited from object</i>	
__class__	

**B.6.61 Class XPathDocumentEvaluator**

XPathDocumentEvaluator(self, etree, namespaces=None, extensions=None, regexp=True)  
 Create an XPath evaluator for an ElementTree.

Additional namespace declarations can be passed with the 'namespace' keyword argument. EXSLT regular expression support can be disabled with the 'regexp' boolean keyword (defaults to True).

**Methods**

**\_\_call\_\_**(self, \_path, \*\*\_variables)

Evaluate an XPath expression on the document.

Variables may be provided as keyword arguments. Note that namespaces are currently not supported for variables. Overrides:

`lxml.etree.XPathElementEvaluator.__call__`

**\_\_init\_\_**(self, etree, namespaces=None, extensions=None, regexp=True)

`x.__init__(...)` initializes x; see `x.__class__.__doc__` for signature Overrides: `object.__init__`

**\_\_new\_\_**(T, S, ...)

**Return Value**

a new object with type S, a subtype of T

Overrides: `object.__new__`

*Inherited from `lxml.etree.XPathElementEvaluator` (Section [B.6.62](#))*

`registerNamespace()`, `registerNamespaces()`, `register_namespace()`, `register_namespaces()`

*Inherited from `lxml.etree.XPathEvaluatorBase`*

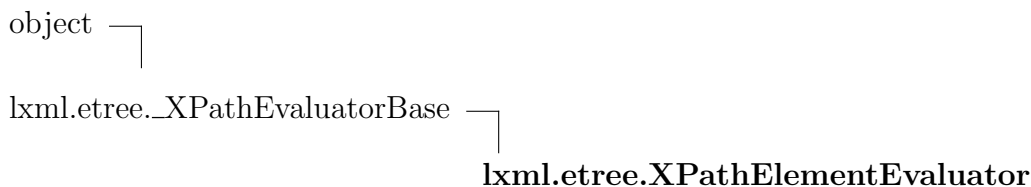
`evaluate()`

**Inherited from object**

`__delattr__()`, `__getattr__()`, `__hash__()`, `__reduce__()`, `__reduce_ex__()`, `__repr__()`,  
`__setattr__()`, `__str__()`

**Properties**

Name	Description
<i>Inherited from lxml.etree._XPathEvaluatorBase</i>	
<code>error_log</code>	
<i>Inherited from object</i>	
<code>__class__</code>	

**B.6.62 Class XPathElementEvaluator**

**Known Subclasses:** `lxml.etree.XPathDocumentEvaluator`

`XPathElementEvaluator(self, element, namespaces=None, extensions=None, regexp=True)`  
 Create an XPath evaluator for an element.

Absolute XPath expressions (starting with '/') will be evaluated against the `ElementTree` as returned by `getroottree()`.

Additional namespace declarations can be passed with the 'namespace' keyword argument. EXSLT regular expression support can be disabled with the 'regexp' boolean keyword (defaults to True).

**Methods**

<code>__call__(self, _path, **_variables)</code>
Evaluate an XPath expression on the document.
Variables may be provided as keyword arguments. Note that namespaces are currently not supported for variables.
Absolute XPath expressions (starting with '/') will be evaluated against the <code>ElementTree</code> as returned by <code>getroottree()</code> .

`__init__(self, element, namespaces=None, extensions=None, regexp=True)`

`x.__init__(...)` initializes x; see `x.__class__.__doc__` for signature Overrides: `object.__init__`

`__new__(T, S, ...)`

**Return Value**

a new object with type S, a subtype of T

Overrides: `object.__new__`

`registerNamespace(...)`

Register a namespace with the XPath context. **Deprecated:** use `register_namespace()` instead

`registerNamespaces(...)`

Register a prefix -> uri dict. **Deprecated:** use `register_namespaces()` instead

`register_namespace(...)`

Register a namespace with the XPath context.

`register_namespaces(...)`

Register a prefix -> uri dict.

**Inherited from `lxml.etree.XPathEvaluatorBase`**

`evaluate()`

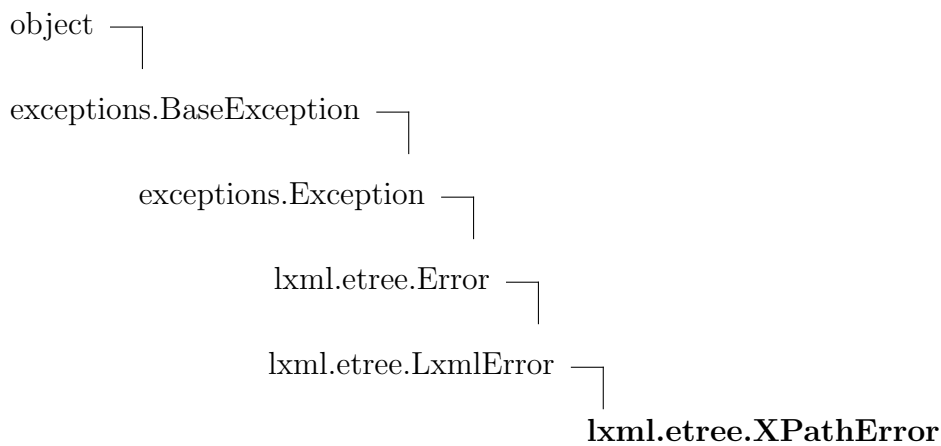
**Inherited from `object`**

`__delattr__()`, `__getattr__()`, `__hash__()`, `__reduce__()`, `__reduce_ex__()`, `__repr__()`, `__setattr__()`, `__str__()`

**Properties**

Name	Description
	<i>Inherited from lxml.etree._XPathEvaluatorBase</i>
	error_log
	<i>Inherited from object</i>
	__class__

### B.6.63 Class XPathError



**Known Subclasses:** lxml.etree.XPathEvalError, lxml.etree.XPathSyntaxError

Base class of all XPath errors.

#### Methods

*Inherited from lxml.etree.LxmlError(Section B.6.30)*

\_\_init\_\_()

*Inherited from exceptions.Exception*

\_\_new\_\_()

*Inherited from exceptions.BaseException*

\_\_delattr\_\_(), \_\_getattr\_\_(), \_\_getitem\_\_(), \_\_getslice\_\_(), \_\_reduce\_\_(), \_\_repr\_\_(),  
\_\_setattr\_\_(), \_\_setstate\_\_(), \_\_str\_\_()

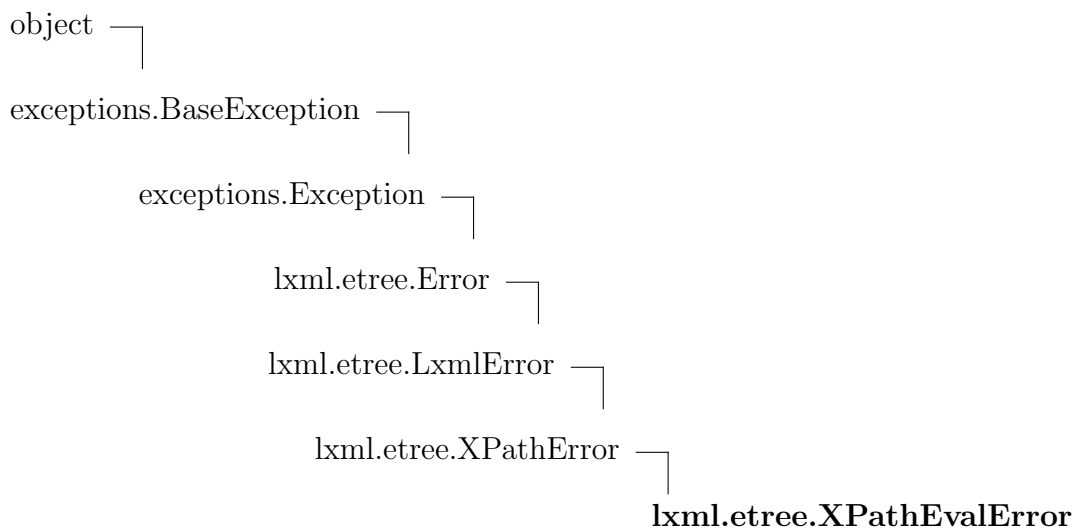
*Inherited from object*

\_\_hash\_\_(), \_\_reduce\_ex\_\_()

#### Properties

Name	Description
	<i>Inherited from exceptions.BaseException</i> args, message
	<i>Inherited from object</i> __class__

### B.6.64 Class XPathEvalError



**Known Subclasses:** lxml.etree.XPathFunctionError, lxml.etree.XPathResultError

Error during XPath evaluation.

#### Methods

*Inherited from lxml.etree.LxmlError(Section B.6.30)*

`__init__()`

*Inherited from exceptions.Exception*

`__new__()`

*Inherited from exceptions.BaseException*

`__delattr__()`, `__getattr__()`, `__getitem__()`, `__getslice__()`, `__reduce__()`, `__repr__()`,  
`__setattr__()`, `__setstate__()`, `__str__()`

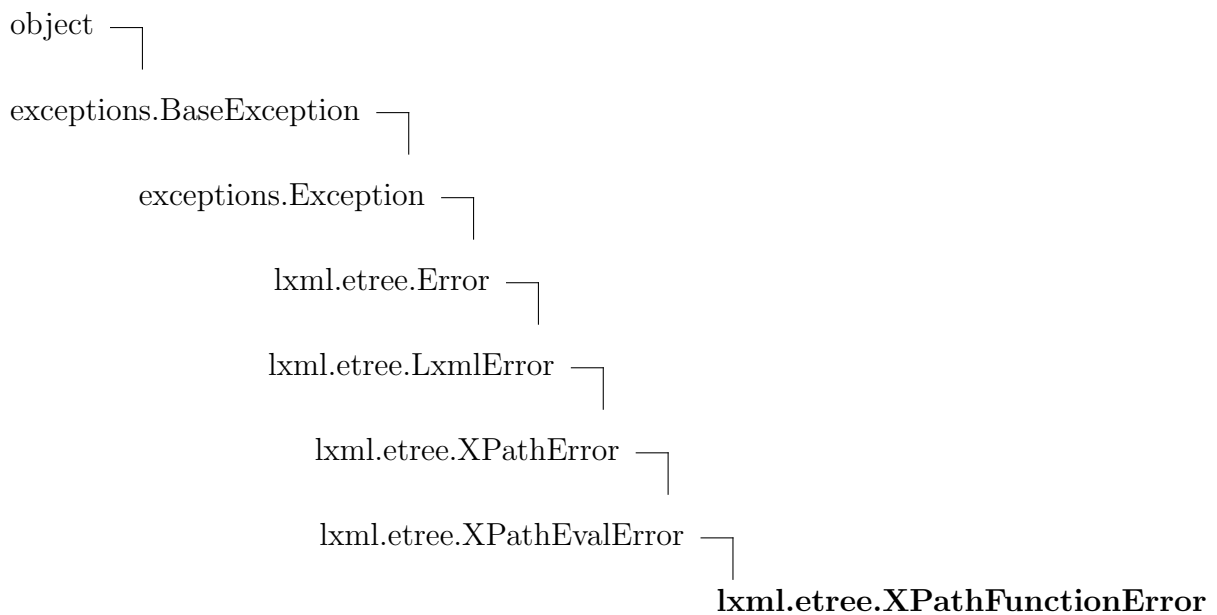
*Inherited from object*

`__hash__()`, `__reduce_ex__()`

## Properties

Name	Description
<i>Inherited from exceptions.BaseException</i>	
args, message	
<i>Inherited from object</i>	
__class__	

## B.6.65 Class XPathFunctionError



Internal error looking up an XPath extension function.

## Methods

*Inherited from lxml.etree.LxmlError(Section [B.6.30](#))*

`__init__()`

*Inherited from exceptions.Exception*

`__new__()`

*Inherited from exceptions.BaseException*

`__delattr__()`, `__getattr__()`, `__getitem__()`, `__getslice__()`, `__reduce__()`, `__repr__()`,  
`__setattr__()`, `__setstate__()`, `__str__()`

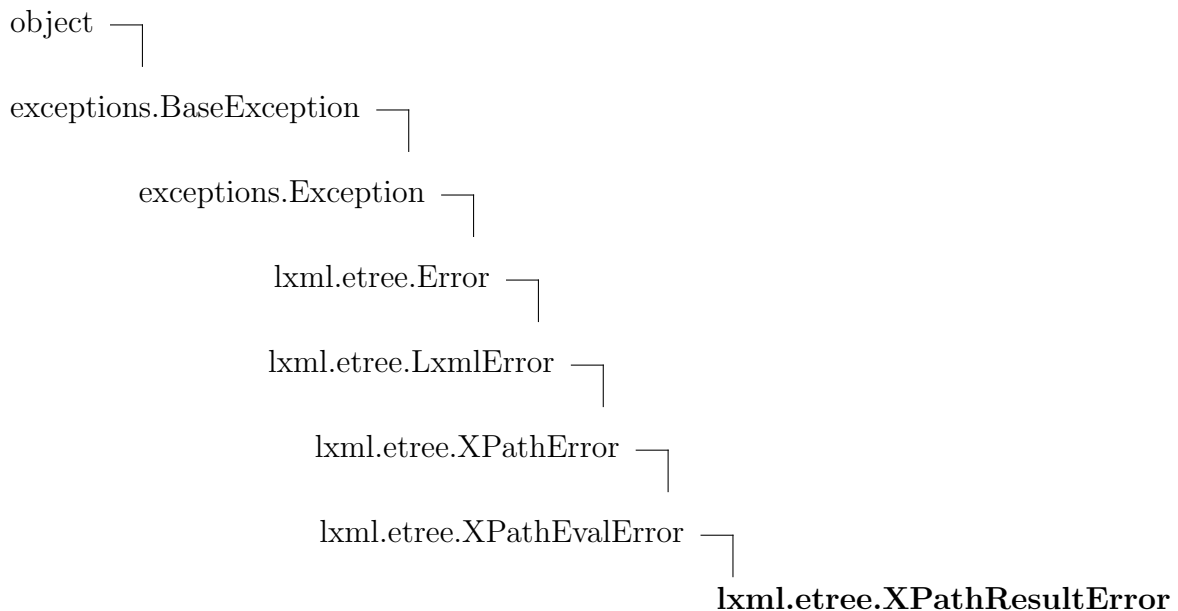
*Inherited from object*

`__hash__()`, `__reduce_ex__()`

### Properties

Name	Description
	<i>Inherited from exceptions.BaseException</i>
	args, message
	<i>Inherited from object</i>
<code>__class__</code>	

### B.6.66 Class XPathResultError



Error handling an XPath result.

### Methods

*Inherited from lxml.etree.LxmlError(Section B.6.30)*

`__init__()`

*Inherited from exceptions.Exception*

`__new__()`

*Inherited from exceptions.BaseException*

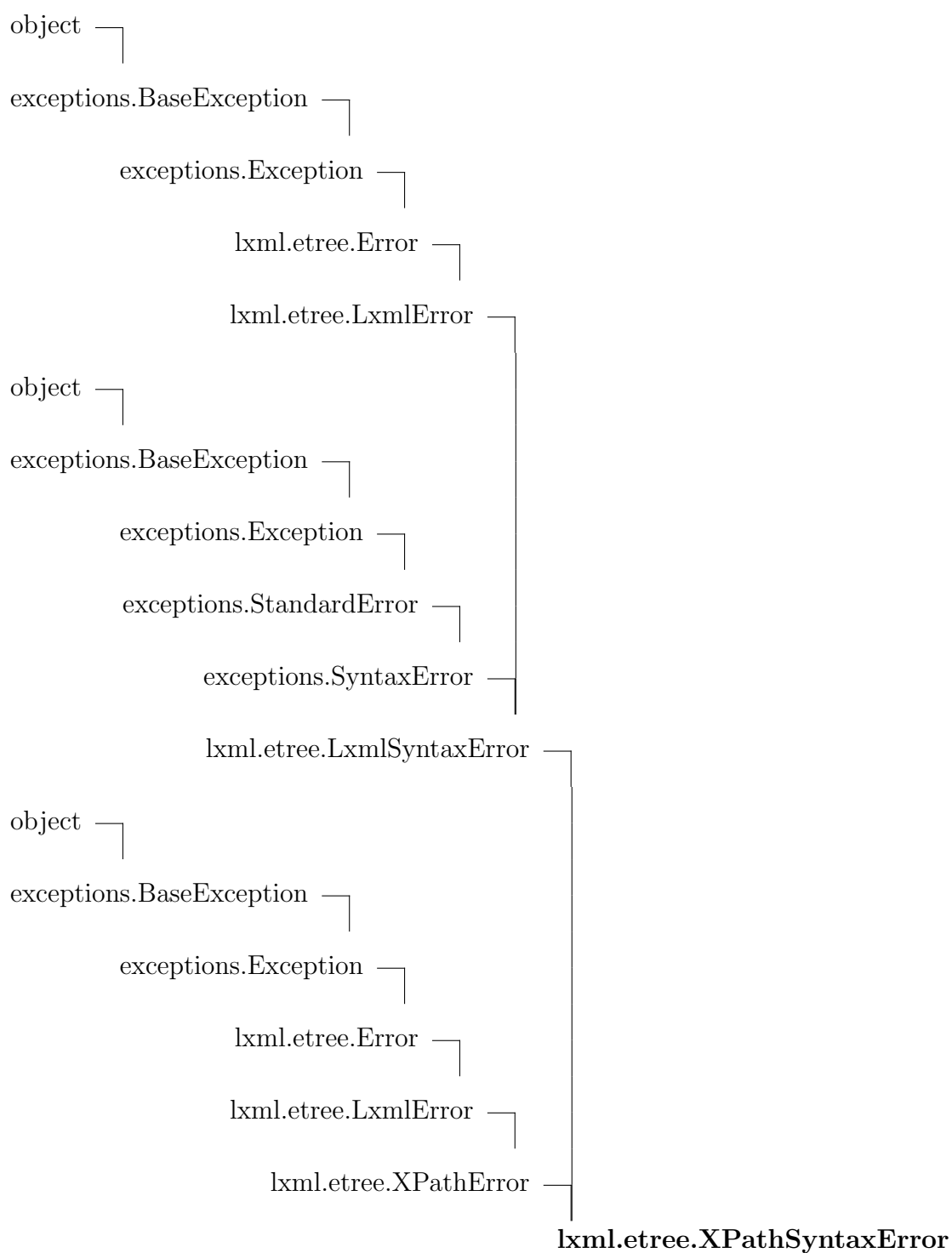
`__delattr__()`, `__getattr__()`, `__getitem__()`, `__getslice__()`, `__reduce__()`, `__repr__()`,  
`__setattr__()`, `__setstate__()`, `__str__()`

***Inherited from object***

`__hash__()`, `__reduce_ex__()`

**Properties**

Name	Description
<i>Inherited from exceptions.BaseException</i> args, message	
<i>Inherited from object</i> __class__	

**B.6.67 Class XPathSyntaxError****Methods**

*Inherited from `lxml.etree.LxmlError` (Section [B.6.30](#))*

`__init__()`

*Inherited from `exceptions.SyntaxError`*

`__new__()`, `__str__()`

*Inherited from `exceptions.BaseException`*

`__delattr__()`, `__getattr__()`, `__getitem__()`, `__getslice__()`, `__reduce__()`, `__repr__()`,  
`__setattr__()`, `__setstate__()`

*Inherited from `object`*

`__hash__()`, `__reduce_ex__()`

### Properties

Name	Description
<i>Inherited from <code>exceptions.SyntaxError</code></i>	<code>filename</code> , <code>lineno</code> , <code>message</code> , <code>msg</code> , <code>offset</code> , <code>print_file_and_line</code> , <code>text</code>
<i>Inherited from <code>exceptions.BaseException</code></i>	<code>args</code>
<i>Inherited from <code>object</code></i>	<code>__class__</code>

### B.6.68 Class XSLT

```
object └─
         └─ lxml.etree.XSLT
```

`XSLT(self, xslt_input, extensions=None, regexp=True, access_control=None)`

Turn an XSL document into an XSLT object.

Calling this object on a tree or Element will execute the XSLT:

```
>>> transform = etree.XSLT(xsl_tree)
>>> result = transform(xml_tree)
```

Keyword arguments of the constructor:

- `regexp`: enable exslt regular expression support in XPath (default: `True`)
- `access_control`: access restrictions for network or file system (see `XSLTAccessControl`)

Keyword arguments of the XSLT call:

- `profile_run`: enable XSLT profiling (default: `False`)

Other keyword arguments of the call are passed to the stylesheet as parameters.

## Methods

**`__call__(self, _input, profile_run=False, **_kw)`**

Execute the XSL transformation on a tree or Element.

Pass the `profile_run` option to get profile information about the XSLT. The result of the XSLT will have a property `xslt_profile` that holds an XML tree with profiling data.

**`__copy__(...)`**

**`__deepcopy__(...)`**

**`__init__(self, xslt_input, extensions=None, regexp=True, access_control=None)`**

`x.__init__(...)` initializes x; see `x.__class__.__doc__` for signature Overrides: `object.__init__`

**`__new__(T, S, ...)`**

### Return Value

a new object with type S, a subtype of T

Overrides: `object.__new__`

**`apply(self, _input, profile_run=False, **_kw)`**

**Deprecated:** call the object, not this method.

**`tostring(self, result_tree)`**

Save result doc to string based on stylesheet output method.

**Deprecated:** use `str(result_tree)` instead.

## Inherited from *object*

`__delattr__()`, `__getattr__()`, `__hash__()`, `__reduce__()`, `__reduce_ex__()`, `__repr__()`, `__setattr__()`, `__str__()`

**Properties**

Name	Description
error_log	The log of errors and warnings of an XSLT execution.
<i>Inherited from object</i>	
__class__	

**B.6.69 Class XSLTAccessControl**

XSLTAccessControl(self, read\_file=True, write\_file=True, create\_dir=True, read\_network=True, write\_network=True)

Access control for XSLT: reading/writing files, directories and network I/O. Access to a type of resource is granted or denied by passing any of the following boolean keyword arguments. All of them default to True to allow access.

- read\_file
- write\_file
- create\_dir
- read\_network
- write\_network

See XSLT.

**Methods**

```
__init__(self, read_file=True, write_file=True, create_dir=True,
read_network=True, write_network=True)
```

x.\_\_init\_\_(...) initializes x; see x.\_\_class\_\_.\_\_doc\_\_ for signature Overrides: object.\_\_init\_\_

```
__new__(T, S, ...)
```

**Return Value**

a new object with type S, a subtype of T

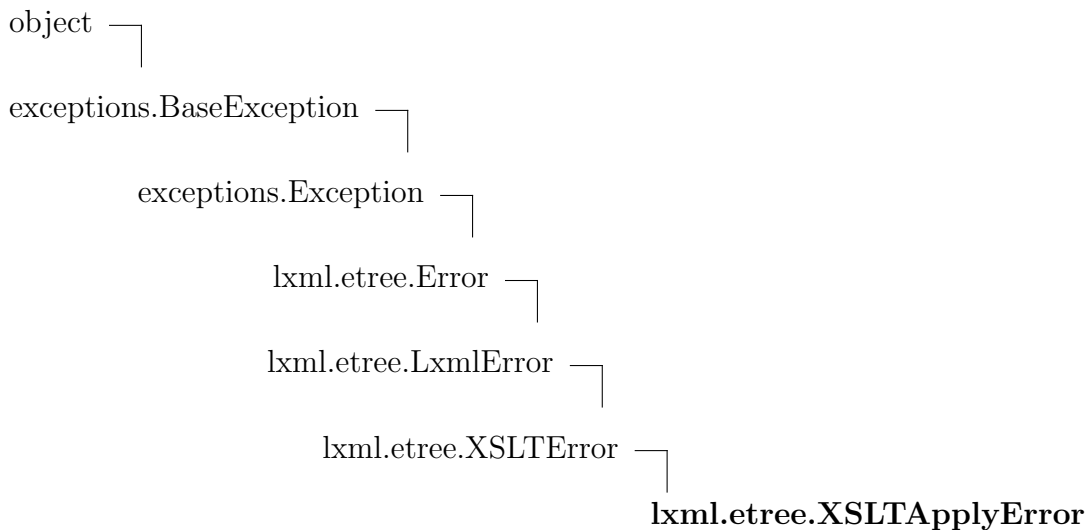
Overrides: object.\_\_new\_\_

**Inherited from object**

`__delattr__()`, `__getattr__()`, `__hash__()`, `__reduce__()`, `__reduce_ex__()`, `__repr__()`,  
`__setattr__()`, `__str__()`

**Properties**

Name	Description
<i>Inherited from object</i>	
<code>__class__</code>	

**B.6.70 Class XSLTApplyError**

Error running an XSL transformation.

**Methods*****Inherited from lxml.etree.LxmlError(Section B.6.30)***

`__init__()`

***Inherited from exceptions.Exception***

`__new__()`

***Inherited from exceptions.BaseException***

`__delattr__()`, `__getattr__()`, `__getitem__()`, `__getslice__()`, `__reduce__()`, `__repr__()`,  
`__setattr__()`, `__setstate__()`, `__str__()`

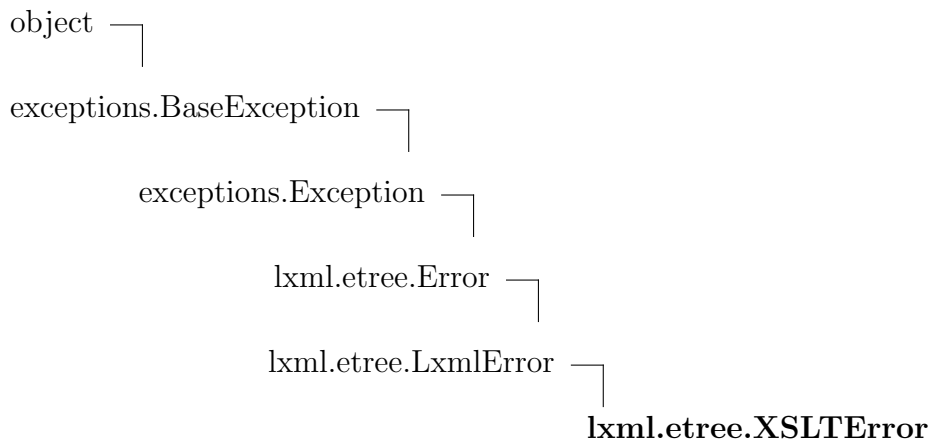
***Inherited from object***

`__hash__()`, `__reduce_ex__()`

### Properties

Name	Description
	<i>Inherited from exceptions.BaseException</i>
	args, message
	<i>Inherited from object</i>
<code>__class__</code>	

### B.6.71 Class XSLTError



**Known Subclasses:** `lxml.etree.XSLTApplyError`, `lxml.etree.XSLTExtensionError`, `lxml.etree.XSLTParseError`, `lxml.etree.XSLTSaveError`

Base class of all XSLT errors.

### Methods

*Inherited from lxml.etree.LxmlError(Section [B.6.30](#))*

`__init__()`

*Inherited from exceptions.Exception*

`__new__()`

*Inherited from exceptions.BaseException*

`__delattr__()`, `__getattr__()`, `__getitem__()`, `__getslice__()`, `__reduce__()`, `__repr__()`,  
`__setattr__()`, `__setstate__()`, `__str__()`

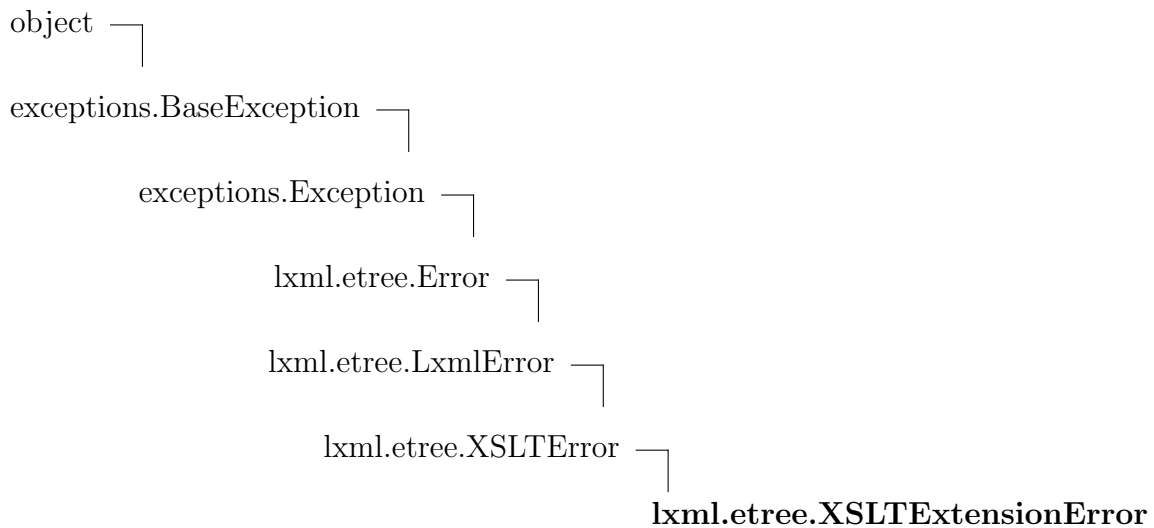
*Inherited from object*

`__hash__()`, `__reduce_ex__()`

### Properties

Name	Description
<i>Inherited from exceptions.BaseException</i>	
args, message	
<i>Inherited from object</i>	
<code>__class__</code>	

### B.6.72 Class XSLTExtensionError



Error registering an XSLT extension.

### Methods

*Inherited from lxml.etree.LxmlError(Section [B.6.30](#))*

`__init__()`

*Inherited from exceptions.Exception*

`__new__()`

*Inherited from exceptions.BaseException*

`__delattr__()`, `__getattr__()`, `__getitem__()`, `__getslice__()`, `__reduce__()`, `__repr__()`,  
`__setattr__()`, `__setstate__()`, `__str__()`

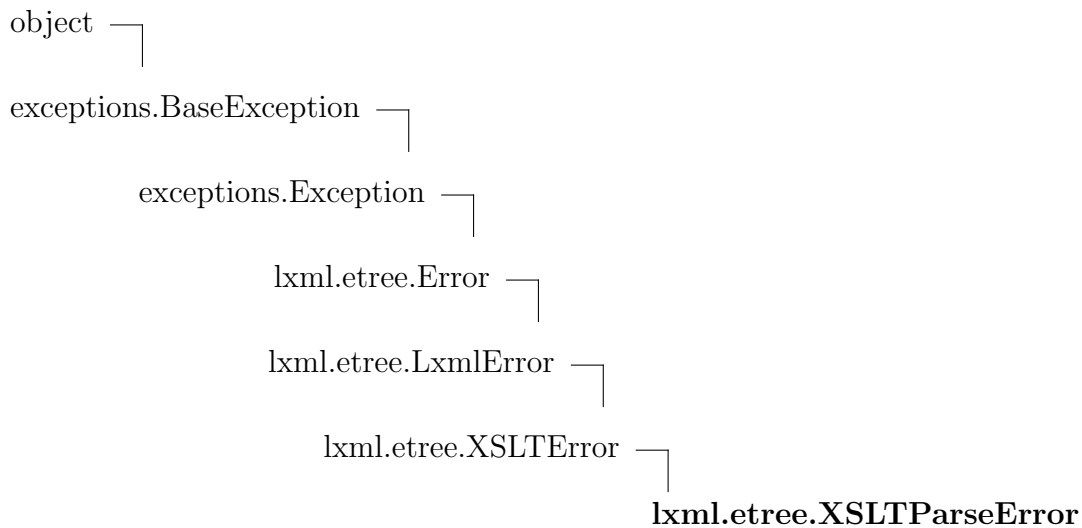
*Inherited from object*

`__hash__()`, `__reduce_ex__()`

### Properties

Name	Description
	<i>Inherited from exceptions.BaseException</i>
	args, message
	<i>Inherited from object</i>
<code>__class__</code>	

### B.6.73 Class XSLTParseError



Error parsing a stylesheet document.

### Methods

*Inherited from lxml.etree.LxmlError(Section [B.6.30](#))*

`__init__()`

*Inherited from exceptions.Exception*

`__new__()`

*Inherited from exceptions.BaseException*

`__delattr__()`, `__getattr__()`, `__getitem__()`, `__getslice__()`, `__reduce__()`, `__repr__()`,  
`__setattr__()`, `__setstate__()`, `__str__()`

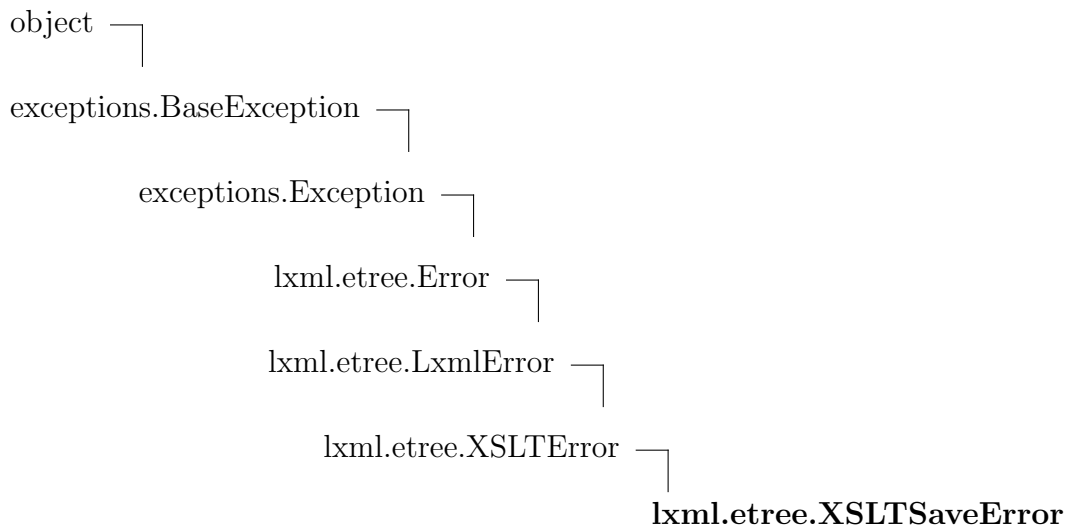
*Inherited from object*

`__hash__()`, `__reduce_ex__()`

### Properties

Name	Description
<i>Inherited from exceptions.BaseException</i>	
args, message	
<i>Inherited from object</i>	
<code>__class__</code>	

### B.6.74 Class XSLTSaveError



Error serialising an XSLT result.

### Methods

*Inherited from lxml.etree.LxmlError(Section [B.6.30](#))*

`__init__()`

*Inherited from exceptions.Exception*

`__new__()`

*Inherited from exceptions.BaseException*

`__delattr__()`, `__getattr__()`, `__getitem__()`, `__getslice__()`, `__reduce__()`, `__repr__()`,  
`__setattr__()`, `__setstate__()`, `__str__()`

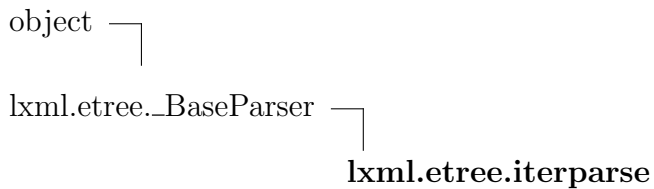
*Inherited from object*

`__hash__()`, `__reduce_ex__()`

### Properties

Name	Description
	<i>Inherited from <code>exceptions.BaseException</code></i>
	<code>args</code> , <code>message</code>
	<i>Inherited from object</i>
<code>__class__</code>	

### B.6.75 Class `iterparse`



`iterparse(self, source, events=("end",), tag=None, attribute_defaults=False, dtd_validation=False, load_dtd=False, no_network=True, remove_blank_text=False, remove_comments=False, remove_pis=False, encoding=None, html=False, schema=None)` Incremental parser.

Parses XML into a tree and generates tuples (`event`, `element`) in a SAX-like fashion. `event` is any of `'start'`, `'end'`, `'start-ns'`, `'end-ns'`.

For `'start'` and `'end'`, `element` is the `Element` that the parser just found opening or closing. For `'start-ns'`, it is a tuple (`prefix`, `URI`) of a new namespace declaration. For `'end-ns'`, it is simply `None`. Note that all start and end events are guaranteed to be properly nested.

The keyword argument `events` specifies a sequence of event type names that should be generated. By default, only `'end'` events will be generated.

The additional `tag` argument restricts the `'start'` and `'end'` events to those elements that match the given tag. By default, events are generated for all elements. Note that the `'start-ns'` and `'end-ns'` events are not impacted by this restriction.

The other keyword arguments in the constructor are mainly based on the `libxml2` parser configuration. A DTD will also be loaded if validation or attribute default values are requested.

- Available boolean keyword arguments:**
- `attribute_defaults` - read default attributes from DTD
  - `dtd_validation` - validate (if DTD is available)
  - `load_dtd` - use DTD for parsing
  - `no_network` - prevent network access for related files

- `remove_blank_text` - discard blank text nodes
- `remove_comments` - discard comments
- `remove_pis` - discard processing instructions

**Other keyword arguments:**

- `encoding` - override the document encoding
- `schema` - an XMLSchema to validate against

## Methods

```
__init__(self, source, events=("end", ), tag=None,
attribute_defaults=False, dtd_validation=False, load_dtd=False,
no_network=True, remove_blank_text=False, remove_comments=False,
remove_pis=False, encoding=None, html=False, schema=None)
```

`x.__init__(...)` initializes `x`; see `x.__class__.__doc__` for signature. Overrides: `object.__init__`

```
__iter__(...)
```

```
__new__(T, S, ...)
```

### Return Value

a new object with type `S`, a subtype of `T`

Overrides: `object.__new__`

```
__next__(...)
```

```
copy(self)
```

Create a new parser with the same configuration. Overrides: `lxml.etree._BaseParser.copy` `exitit` (inherited documentation)

```
next(x)
```

### Return Value

the next value, or raise `StopIteration`

## Inherited from *lxml.etree.\_BaseParser*

`makeelement()`, `setElementClassLookup()`, `set_element_class_lookup()`

## Inherited from *object*

`__delattr__()`, `__getattr__()`, `__hash__()`, `__reduce__()`, `__reduce_ex__()`, `__repr__()`, `__setattr__()`, `__str__()`

**Properties**

Name	Description
<code>error_log</code>	The error log of the last (or current) parser run.
<code>root</code>	
<i>Inherited from <code>lxml.etree._BaseParser</code></i>	
<code>resolvers</code> , <code>version</code>	
<i>Inherited from <code>object</code></i>	
<code>__class__</code>	

**B.6.76 Class *iterwalk***

`iterwalk(self, element_or_tree, events=("end",), tag=None)`

A tree walker that generates events from an existing tree as if it was parsing XML data with `iterparse()`.

**Methods**

<code>__init__(self, element_or_tree, events=("end",), tag=None)</code>
<code>x.__init__(...)</code> initializes x; see <code>x.__class__.__doc__</code> for signature. Overrides: <code>object.__init__</code>

<code>__iter__(...)</code>
----------------------------

<code>__new__(T, S, ...)</code>
<b>Return Value</b> a new object with type S, a subtype of T
Overrides: <code>object.__new__</code>

<code>__next__(...)</code>
----------------------------

<code>next(x)</code>
<b>Return Value</b> the next value, or raise <code>StopIteration</code>

***Inherited from object***

`__delattr__()`, `__getattr__()`, `__hash__()`, `__reduce__()`, `__reduce_ex__()`, `__repr__()`,  
`__setattr__()`, `__str__()`

**Properties**

Name	Description
<i>Inherited from object</i>	
<code>__class__</code>	

## B.7 Package lxml.html

The `lxml.html` tool set for HTML handling.

### B.7.1 Modules

- **ElementSoup**: Legacy interface to the BeautifulSoup HTML parser.  
(Section B.8, p. 339)
- **\_dictmixin** (Section ??, p. ??)
- **\_setmixin** (Section ??, p. ??)
- **builder**: A set of HTML generator tags for building HTML documents.  
(Section B.9, p. 340)
- **clean**: A cleanup tool for HTML.  
(Section B.10, p. 344)
- **defs** (Section B.11, p. 348)
- **diff** (Section B.12, p. 349)
- **formfill** (Section B.13, p. 350)
- **soupparser**: External interface to the BeautifulSoup HTML parser.  
(Section B.14, p. 353)
- **usedoctest**: Doctest module for HTML comparison.  
(Section B.15, p. 354)

### B.7.2 Functions

<b>document_fromstring</b> ( <i>html</i> , <i>**kw</i> )
--

<b>fragments_fromstring</b> ( <i>html</i> , <i>no_leading_text=False</i> , <i>base_url=None</i> , <i>**kw</i> )
---

Parses several HTML elements, returning a list of elements.

The first item in the list may be a string (though leading whitespace is removed). If `no_leading_text` is true, then it will be an error if there is leading text, and it will always be a list of only elements.

`base_url` will set the document's `base_url` attribute (and the tree's `docinfo.URL`)

**fragment\_fromstring**(*html*, *create\_parent=False*, *base\_url=None*, *\*\*kw*)

Parses a single HTML element; it is an error if there is more than one element, or if anything but whitespace precedes or follows the element.

If *create\_parent* is true (or is a tag name) then a parent node will be created to encapsulate the HTML in a single element.

*base\_url* will set the document's *base\_url* attribute (and the tree's *docinfo.URL*)

**fromstring**(*html*, *base\_url=None*, *\*\*kw*)

Parse the *html*, returning a single element/document.

This tries to minimally parse the chunk of text, without knowing if it is a fragment or a document.

*base\_url* will set the document's *base\_url* attribute (and the tree's *docinfo.URL*)

**submit\_form**(*form*, *extra\_values=None*, *open\_http=None*)

Helper function to submit a form. Returns a file-like object, as from `urllib.urlopen()`. This object also has a `.geturl()` function, which shows the URL if there were any redirects.

You can use this like:

```
form = doc.forms[0]
form.inputs['foo'].value = 'bar' # etc
response = form.submit()
doc = parse(response)
doc.make_links_absolute(response.geturl())
```

To change the HTTP requester, pass a function as *open\_http* keyword argument that opens the URL for you. The function must have the following signature:

```
open_http(method, URL, values)
```

The action is one of 'GET' or 'POST', the URL is the target URL as a string, and the values are a sequence of (name, value) tuples with the form data.

```
tostring(doc, pretty_print=False, include_meta_content_type=False,
encoding=None, method='html')
```

Return an HTML string representation of the document.

Note: if `include_meta_content_type` is true this will create a `<meta http-equiv="Content-Type" ...>` tag in the head; regardless of the value of `include_meta_content_type` any existing `<meta http-equiv="Content-Type" ...>` tag will be removed

The `encoding` argument controls the output encoding (defaults to ASCII, with `&#...;` character references for any characters outside of ASCII).

The `method` argument defines the output method. It defaults to 'html', but can also be 'xml' for xhtml output, or 'text' to serialise to plain text without markup. Note that you can pass the builtin `unicode` type as `encoding` argument to serialise to a unicode string.

Example:

```
>>> from lxml import html
>>> root = html.fragment_fromstring('<p>Hello<br>world!</p>')

>>> html.tostring(root)
'<p>Hello<br>world!</p>'
>>> html.tostring(root, method='html')
'<p>Hello<br>world!</p>'

>>> html.tostring(root, method='xml')
'<p>Hello<br/>world!</p>'

>>> html.tostring(root, method='text')
'Hello world!'

>>> html.tostring(root, method='text', encoding=unicode)
u'Hello world!'
```

```
open_in_browser(doc)
```

Open the HTML document in a web browser (saving it to a temporary file to open it).

```
Element(*args, **kw)
```

### B.7.3 Variables

Name	Description
find_rel_links	<b>Value:</b> <code>_MethodFunc('find_rel_links', copy= False)</code>
find_class	<b>Value:</b> <code>_MethodFunc('find_class', copy= False)</code>
make_links_absolute	<b>Value:</b> <code>_MethodFunc('make_links_absolute', copy= True)</code>
resolve_base_href	<b>Value:</b> <code>_MethodFunc('resolve_base_href', copy= True)</code>
iterlinks	<b>Value:</b> <code>_MethodFunc('iterlinks', copy= False)</code>
rewrite_links	<b>Value:</b> <code>_MethodFunc('rewrite_links', copy= True)</code>

## B.8 Module `lxml.html.ElementSoup`

Legacy interface to the BeautifulSoup HTML parser.

### B.8.1 Functions

```
convert_tree(beautiful_soup_tree, makeelement=None)
```

Convert a BeautifulSoup tree to a list of Element trees.

Returns a list instead of a single root Element to support HTML-like soup with more than one root element.

You can pass a different Element factory through the `makeelement` keyword.

```
parse(file, beautifulsoup=None, makeelement=None)
```

## B.9 Module lxml.html.builder

A set of HTML generator tags for building HTML documents.

Usage:

```
>>> from lxml.html.builder import *
>>> html = HTML(
...     HEAD( TITLE("Hello World") ),
...     BODY( CLASS("main"),
...           H1("Hello World !")
...         )
...     )

>>> import lxml.etree
>>> print lxml.etree.tostring(html, pretty_print=True)
<html>
  <head>
    <title>Hello World</title>
  </head>
  <body class="main">
    <h1>Hello World !</h1>
  </body>
</html>
```

### B.9.1 Functions

<b>CLASS</b> ( <i>v</i> )
---------------------------

<b>FOR</b> ( <i>v</i> )
-------------------------

### B.9.2 Variables

Name	Description
E	<b>Value:</b> ElementMaker(makeelement=html_parser.makeelement)
A	<b>Value:</b> E.a
ABBR	<b>Value:</b> E.abbr
ACRONYM	<b>Value:</b> E.acronym
ADDRESS	<b>Value:</b> E.address
APPLET	<b>Value:</b> E.applet
AREA	<b>Value:</b> E.area
B	<b>Value:</b> E.b

*continued on next page*

Name	Description
BASE	Value: <code>E.base</code>
BASEFONT	Value: <code>E.basefont</code>
BDO	Value: <code>E.bdo</code>
BIG	Value: <code>E.big</code>
BLOCKQUOTE	Value: <code>E.blockquote</code>
BODY	Value: <code>E.body</code>
BR	Value: <code>E.br</code>
BUTTON	Value: <code>E.button</code>
CAPTION	Value: <code>E.caption</code>
CENTER	Value: <code>E.center</code>
CITE	Value: <code>E.cite</code>
CODE	Value: <code>E.code</code>
COL	Value: <code>E.col</code>
COLGROUP	Value: <code>E.colgroup</code>
DD	Value: <code>E.dd</code>
DEL	Value: <code>getattr(E, 'del')</code>
DFN	Value: <code>E.dfn</code>
DIR	Value: <code>E.dir</code>
DIV	Value: <code>E.div</code>
DL	Value: <code>E.dl</code>
DT	Value: <code>E.dt</code>
EM	Value: <code>E.em</code>
FIELDSET	Value: <code>E.fieldset</code>
FONT	Value: <code>E.font</code>
FORM	Value: <code>E.form</code>
FRAME	Value: <code>E.frame</code>
FRAMESET	Value: <code>E.frameset</code>
H1	Value: <code>E.h1</code>
H2	Value: <code>E.h2</code>
H3	Value: <code>E.h3</code>
H4	Value: <code>E.h4</code>
H5	Value: <code>E.h5</code>
H6	Value: <code>E.h6</code>
HEAD	Value: <code>E.head</code>
HR	Value: <code>E.hr</code>
HTML	Value: <code>E.html</code>
I	Value: <code>E.i</code>
IFRAME	Value: <code>E.iframe</code>
IMG	Value: <code>E.img</code>

*continued on next page*

<b>Name</b>	<b>Description</b>
INPUT	<b>Value:</b> <code>E.input</code>
INS	<b>Value:</b> <code>E.ins</code>
ISINDEX	<b>Value:</b> <code>E.isindex</code>
KBD	<b>Value:</b> <code>E.kbd</code>
LABEL	<b>Value:</b> <code>E.label</code>
LEGEND	<b>Value:</b> <code>E.legend</code>
LI	<b>Value:</b> <code>E.li</code>
LINK	<b>Value:</b> <code>E.link</code>
MAP	<b>Value:</b> <code>E.map</code>
MENU	<b>Value:</b> <code>E.menu</code>
META	<b>Value:</b> <code>E.meta</code>
NOFRAMES	<b>Value:</b> <code>E.noframes</code>
NOSCRIPT	<b>Value:</b> <code>E.noscript</code>
OBJECT	<b>Value:</b> <code>E.object</code>
OL	<b>Value:</b> <code>E.ol</code>
OPTGROUP	<b>Value:</b> <code>E.optgroup</code>
OPTION	<b>Value:</b> <code>E.option</code>
P	<b>Value:</b> <code>E.p</code>
PARAM	<b>Value:</b> <code>E.param</code>
PRE	<b>Value:</b> <code>E.pre</code>
Q	<b>Value:</b> <code>E.q</code>
S	<b>Value:</b> <code>E.s</code>
SAMP	<b>Value:</b> <code>E.samp</code>
SCRIPT	<b>Value:</b> <code>E.script</code>
SELECT	<b>Value:</b> <code>E.select</code>
SMALL	<b>Value:</b> <code>E.small</code>
SPAN	<b>Value:</b> <code>E.span</code>
STRIKE	<b>Value:</b> <code>E.strike</code>
STRONG	<b>Value:</b> <code>E.strong</code>
STYLE	<b>Value:</b> <code>E.style</code>
SUB	<b>Value:</b> <code>E.sub</code>
SUP	<b>Value:</b> <code>E.sup</code>
TABLE	<b>Value:</b> <code>E.table</code>
TBODY	<b>Value:</b> <code>E.tbody</code>
TD	<b>Value:</b> <code>E.td</code>
TEXTAREA	<b>Value:</b> <code>E.textarea</code>
TFOOT	<b>Value:</b> <code>E.tfoot</code>
TH	<b>Value:</b> <code>E.th</code>
THEAD	<b>Value:</b> <code>E.thead</code>

*continued on next page*

---

Name	Description
TITLE	Value: E.title
TR	Value: E.tr
TT	Value: E.tt
U	Value: E.u
UL	Value: E.ul
VAR	Value: E.var

## B.10 Module `lxml.html.clean`

A cleanup tool for HTML.

Removes unwanted tags and content. See the `Cleaner` class for details.

### B.10.1 Functions

```
autolink(el, link_regexes=_link_regexes,
          avoid_elements=_avoid_elements, avoid_hosts=_avoid_hosts,
          avoid_classes=_avoid_classes)
```

Turn any URLs into links.

It will search for links identified by the given regular expressions (by default `mailto` and `http(s)` links).

It won't link text in an element in `avoid_elements`, or an element with a class in `avoid_classes`. It won't link to anything with a host that matches one of the regular expressions in `avoid_hosts` (default `localhost` and `127.0.0.1`).

If you pass in an element, the elements tail will not be substituted, only the contents of the element.

```
autolink_html(html, *args, **kw)
```

```
word_break(el, max_width=40,
            avoid_elements=_avoid_word_break_elements,
            avoid_classes=_avoid_word_break_classes, break_character=u'\u200b')
```

Breaks any long words found in the body of the text (not attributes).

Doesn't effect any of the tags in `avoid_elements`, by default `<textarea>` and `<pre>`

Breaks words by inserting `&#8203;`, which is a unicode character for Zero Width Space character. This generally takes up no space in rendering, but does copy as a space, and in monospace contexts usually takes up space.

See <http://www.cs.tut.fi/~jkorpela/html/nobr.html> for a discussion

```
word_break_html(html, *args, **kw)
```

### B.10.2 Variables

Name	Description
clean	<b>Value:</b> Cleaner()
clean_html	<b>Value:</b> clean.clean_html

### B.10.3 Class Cleaner



Instances cleans the document of each of the possible offending elements. The cleaning is controlled by attributes; you can override attributes in a subclass, or set them in the constructor.

**scripts:** Removes any `<script>` tags.

**javascript:** Removes any Javascript, like an `onclick` attribute.

**comments:** Removes any comments.

**style:** Removes any style tags or attributes.

**links:** Removes any `<link>` tags

**meta:** Removes any `<meta>` tags

**page\_structure:** Structural parts of a page: `<head>`, `<html>`, `<title>`.

**processing\_instructions:** Removes any processing instructions.

**embedded:** Removes any embedded objects (flash, iframes)

**frames:** Removes any frame-related tags

**forms:** Removes any form tags

**annoying\_tags:** Tags that aren't *wrong*, but are annoying. `<blink>` and `<marque>`

**remove\_tags:** A list of tags to remove.

**allow\_tags:** A list of tags to include (default include all).

**remove\_unknown\_tags:** Remove any tags that aren't standard parts of HTML.

**safe\_attrs\_only:** If true, only include 'safe' attributes (specifically the list from [feed-parser](#)).

**add\_nofollow:** If true, then any `<a>` tags will have `rel="nofollow"` added to them.

**host\_whitelist:** A list or set of hosts that you can use for embedded content (for content

like `<object>`, `<link rel="stylesheet">`, etc). You can also implement/override the method `allow_embedded_url(el, url)` or `allow_element(el)` to implement more complex rules for what can be embedded. Anything that passes this test will be shown, regardless of the value of (for instance) `embedded`.

Note that this parameter might not work as intended if you do not make the links absolute before doing the cleaning.

**whitelist\_tags:** A set of tags that can be included with `host_whitelist`. The default is `iframe` and `embed`; you may wish to include other tags like `script`, or you may want to implement `allow_embedded_url` for more control. Set to `None` to include all tags.

This modifies the document *in place*.

## Methods

<b><code>__init__(self, **kw)</code></b> <code>x.__init__(...)</code> initializes <code>x</code> ; see <code>x.__class__.__doc__</code> for signature. Overrides: <code>object.__init__</code> extit(inherited documentation)
--

<b><code>__call__(self, doc)</code></b> <hr/> Cleans the document.
---

<b><code>allow_follow(self, anchor)</code></b> <hr/> Override to suppress <code>rel="nofollow"</code> on some anchors.
---

<b><code>allow_element(self, el)</code></b>
---

<b><code>allow_embedded_url(self, el, url)</code></b>
---

<b><code>kill_conditional_comments(self, doc)</code></b> <hr/> IE conditional comments basically embed HTML that the parser doesn't normally see. We can't allow anything like that, so we'll kill any comments that could be conditional.
---

<b><code>clean_html(self, html)</code></b>
--

*Inherited from `object`*

`__delattr__()`, `__getattr__()`, `__hash__()`, `__new__()`, `__reduce__()`, `__reduce_ex__()`,  
`__repr__()`, `__setattr__()`, `__str__()`

### Properties

Name	Description
<i>Inherited from object</i>	
<code>__class__</code>	

### Class Variables

Name	Description
<code>scripts</code>	<b>Value:</b> True
<code>javascript</code>	<b>Value:</b> True
<code>comments</code>	<b>Value:</b> True
<code>style</code>	<b>Value:</b> False
<code>links</code>	<b>Value:</b> True
<code>meta</code>	<b>Value:</b> True
<code>page_structure</code>	<b>Value:</b> True
<code>processing_instructions</code>	<b>Value:</b> True
<code>embedded</code>	<b>Value:</b> True
<code>frames</code>	<b>Value:</b> True
<code>forms</code>	<b>Value:</b> True
<code>annoying_tags</code>	<b>Value:</b> True
<code>remove_tags</code>	<b>Value:</b> None
<code>allow_tags</code>	<b>Value:</b> None
<code>remove_unknown_tags</code>	<b>Value:</b> True
<code>safe_attrs_only</code>	<b>Value:</b> True
<code>add_nofollow</code>	<b>Value:</b> False
<code>host_whitelist</code>	<b>Value:</b> ()
<code>whitelist_tags</code>	<b>Value:</b> set(['iframe', 'embed'])

## B.11 Module lxml.html.defs

### B.11.1 Variables

Name	Description
empty_tags	<b>Value:</b> ['area', 'base', 'basefont', 'br', 'col', 'frame', 'hr', ...]
deprecated_tags	<b>Value:</b> ['applet', 'basefont', 'center', 'dir', 'font', 'isindex'...]
link_attrs	<b>Value:</b> ['action', 'archive', 'background', 'cite', 'classid', 'c...]
event_attrs	<b>Value:</b> ['onblur', 'onchange', 'onclick', 'ondblclick', 'onerror'...]
safe_attrs	<b>Value:</b> ['abbr', 'accept', 'accept-charset', 'accesskey', 'action...]
top_level_tags	<b>Value:</b> ['html', 'head', 'body', 'frameset']
head_tags	<b>Value:</b> ['base', 'isindex', 'link', 'meta', 'script', 'style', 't...]
general_block_tags	<b>Value:</b> ['address', 'blockquote', 'center', 'del', 'div', 'h1', '...]
list_tags	<b>Value:</b> ['dir', 'dl', 'dt', 'dd', 'li', 'menu', 'ol', 'ul']
table_tags	<b>Value:</b> ['table', 'caption', 'colgroup', 'col', 'thead', 'tfoot',...]
block_tags	<b>Value:</b> ['address', 'blockquote', 'center', 'del', 'div', 'h1', '...]
form_tags	<b>Value:</b> ['form', 'button', 'fieldset', 'legend', 'input', 'label'...]
special_inline_tags	<b>Value:</b> ['a', 'applet', 'basefont', 'bdo', 'br', 'embed', 'font',...]
phrase_tags	<b>Value:</b> ['abbr', 'acronym', 'cite', 'code', 'del', 'dfn', 'em', '...]
font_style_tags	<b>Value:</b> ['b', 'big', 'i', 's', 'small', 'strike', 'tt', 'u']
frame_tags	<b>Value:</b> ['frameset', 'frame', 'noframes']
nonstandard_tags	<b>Value:</b> ['blink', 'marque']
tags	<b>Value:</b> ['html', 'head', 'body', 'frameset', 'base', 'isindex', '...]

## B.12 Module lxml.html.diff

### B.12.1 Functions

```
html_annotate(doclist, markup=<function default_markup at 0x893d95c>)
```

*doclist* should be ordered from oldest to newest, like:

```
>>> version1 = 'Hello World'
>>> version2 = 'Goodbye World'
>>> html_annotate([(version1, 'version 1'),
...                 (version2, 'version 2')])
u'<span title="version 2">Goodbye</span> <span ti-
tle="version 1">World</span>'
```

The documents must be *fragments* (str/UTF8 or unicode), not complete documents

The *markup* argument is a function to markup the spans of words. This function is called like `markup('Hello', 'version 2')`, and returns HTML. The first argument is text and never includes any markup. The default uses a span with a title:

```
>>> default_markup('Some Text', 'by Joe')
u'<span title="by Joe">Some Text</span>'
```

```
htmldiff(old_html, new_html)
```

Do a diff of the old and new document. The documents are HTML *fragments* (str/UTF8 or unicode), they are not complete documents (i.e., no `<html>` tag).

Returns HTML with `<ins>` and `<del>` tags added around the appropriate text.

Markup is generally ignored, with the markup from *new\_html* preserved, and possibly some markup from *old\_html* (though it is considered acceptable to lose some of the old markup). Only the words in the HTML are diffed. The exception is `<img>` tags, which are treated like words, and the href attribute of `<a>` tags, which are noted inside the tag itself when there are changes.

## B.13 Module `lxml.html.formfill`

### B.13.1 Functions

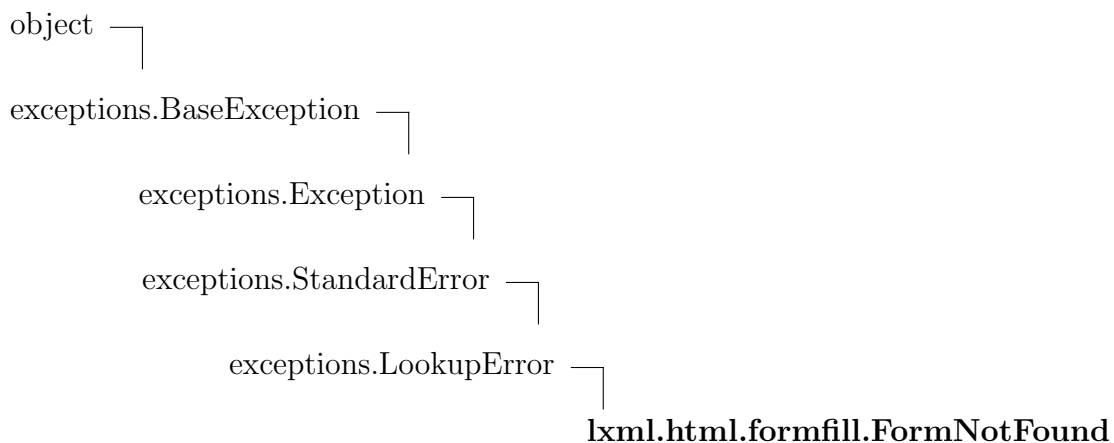
```
fill_form(el, values, form_id=None, form_index=None)
```

```
fill_form_html(html, values, form_id=None, form_index=None)
```

```
insert_errors(el, errors, form_id=None, form_index=None,
              error_class='error', error_creator=default_error_creator)
```

```
insert_errors_html(html, values, **kw)
```

### B.13.2 Class `FormNotFound`



Raised when no form can be found

#### Methods

#### *Inherited from `exceptions.LookupError`*

```
__init__(), __new__()
```

#### *Inherited from `exceptions.BaseException`*

```
__delattr__(), __getattr__(), __getitem__(), __getslice__(), __reduce__(), __repr__(),
__setattr__(), __setstate__(), __str__()
```

#### *Inherited from `object`*

```
__hash__(), __reduce_ex__()
```

**Properties**

Name	Description
	<i>Inherited from <code>exceptions.BaseException</code></i> args, message
	<i>Inherited from <code>object</code></i> <code>__class__</code>

**B.13.3 Class `DefaultErrorCreator`****Methods**

<code>__init__(self, **kw)</code>  x. <code>__init__</code> (...) initializes x; see x. <code>__class__.__doc__</code> for signature Overrides: object. <code>__init__</code> <code>__exit__</code> (inherited documentation)
<code>__call__(self, el, is_block, message)</code>

***Inherited from `object`***

`__delattr__`(), `__getattr__`(), `__hash__`(), `__new__`(), `__reduce__`(), `__reduce_ex__`(),  
`__repr__`(), `__setattr__`(), `__str__`()

**Properties**

Name	Description
	<i>Inherited from <code>object</code></i> <code>__class__</code>

**Class Variables**

Name	Description
<code>insert_before</code>	<b>Value:</b> True
<code>block_inside</code>	<b>Value:</b> True
<code>error_container_tag</code>	<b>Value:</b> 'div'
<code>error_message_class</code>	<b>Value:</b> 'error-message'

*continued on next page*

<b>Name</b>	<b>Description</b>
error_block_class	<b>Value:</b> 'error-block'
default_message	<b>Value:</b> 'Invalid'

## B.14 Module `lxml.html.soupparser`

External interface to the BeautifulSoup HTML parser.

### B.14.1 Functions

**fromstring**(*data*, *beautifulsoup*=None, *makeelement*=None, **\*\*bsargs**)

Parse a string of HTML data into an Element tree using the BeautifulSoup parser.

Returns the root `<html>` Element of the tree.

You can pass a different BeautifulSoup parser through the `beautifulsoup` keyword, and a different Element factory function through the `makeelement` keyword. By default, the standard `BeautifulSoup` class and the default factory of `lxml.html` are used.

**parse**(*file*, *beautifulsoup*=None, *makeelement*=None, **\*\*bsargs**)

Parse a file into an ElementTree using the BeautifulSoup parser.

You can pass a different BeautifulSoup parser through the `beautifulsoup` keyword, and a different Element factory function through the `makeelement` keyword. By default, the standard `BeautifulSoup` class and the default factory of `lxml.html` are used.

**convert\_tree**(*beautiful\_soup\_tree*, *makeelement*=None)

Convert a BeautifulSoup tree to a list of Element trees.

Returns a list instead of a single root Element to support HTML-like soup with more than one root element.

You can pass a different Element factory through the `makeelement` keyword.

## **B.15 Module lxml.html.usedoctest**

Doctest module for HTML comparison.

Usage:

```
>>> import lxml.html.usedoctest
>>> # now do your HTML doctests ...
```

See `lxml.doctestcompare`.

## B.16 Module `lxml.objectify`

The `lxml.objectify` module implements a Python object API for XML. It is based on `lxml.etree`. **Version:** 2.0.10-60473

### B.16.1 Functions

**DataElement**(*\_value*, *attrib=None*, *nsmmap=None*, *\_pytype=None*, *\_xsi=None*, **\*\*\_attributes**)

Create a new element from a Python value and XML attributes taken from keyword arguments or a dictionary passed as second argument.

Automatically adds a 'pytype' attribute for the Python type of the value, if the type can be identified. If '\_pytype' or '\_xsi' are among the keyword arguments, they will be used instead.

If the *\_value* argument is an `ObjectifiedDataElement` instance, its `py:pytype`, `xsi:type` and other attributes and `nsmmap` are reused unless they are redefined in `attrib` and/or keyword arguments.

**Element**(*\_tag*, *attrib=None*, *nsmmap=None*, *\_pytype=None*, **\*\*\_attributes**)

Objectify specific version of the `lxml.etree Element()` factory that always creates a structural (tree) element.

NOTE: requires parser based element class lookup activated in `lxml.etree`!

**XML**(*xml*, *parser=None*)

Objectify specific version of the `lxml.etree fromstring()` function that uses the objectify parser.

You can pass a different parser as second argument.

```
annotate(element_or_tree, ignore_old=True, ignore_xsi=False,  
empty_pytype=None, empty_type=None, annotate_xsi=0,  
annotate_pytype=1)
```

Recursively annotates the elements of an XML tree with 'xsi:type' and/or 'py:pytype' attributes.

If the 'ignore\_old' keyword argument is True (the default), current 'py:pytype' attributes will be ignored for the type annotation. Set to False if you want reuse existing 'py:pytype' information (iff appropriate for the element text value).

If the 'ignore\_xsi' keyword argument is False (the default), existing 'xsi:type' attributes will be used for the type annotation, if they fit the element text values.

Note that the mapping from Python types to XSI types is usually ambiguous. Currently, only the first XSI type name in the corresponding PyType definition will be used for annotation. Thus, you should consider naming the widest type first if you define additional types.

The default 'py:pytype' annotation of empty elements can be set with the `empty_pytype` keyword argument. Pass 'str', for example, to make string values the default.

The default 'xsi:type' annotation of empty elements can be set with the `empty_type` keyword argument. The default is not to annotate empty elements. Pass 'string', for example, to make string values the default.

The keyword arguments 'annotate\_xsi' (default: 0) and 'annotate\_pytype' (default: 1) control which kind(s) of annotation to use.

```
deannotate(element_or_tree, pytype=True, xsi=True)
```

Recursively de-annotate the elements of an XML tree by removing 'pytype' and/or 'type' attributes.

If the 'pytype' keyword argument is True (the default), 'pytype' attributes will be removed. If the 'xsi' keyword argument is True (the default), 'xsi:type' attributes will be removed.

**dump(...)**

`dump(_Element element not None)`

Return a recursively generated string representation of an element.

**enableRecursiveStr(*on=True*)**

Enable a recursively generated tree representation for `str(element)`, based on `objectify.dump(element)`.

**fromstring(*xml, parser=None*)**

Objectify specific version of the `lxml.etree fromstring()` function that uses the `objectify` parser.

You can pass a different parser as second argument.

**getRegisteredTypes()**

Returns a list of the currently registered `PyType` objects.

To add a new type, retrieve this list and call `unregister()` for all entries. Then add the new type at a suitable position (possibly replacing an existing one) and call `register()` for all entries.

This is necessary if the new type interferes with the type check functions of existing ones (normally only `int/float/bool`) and must be tried before other types. To add a type that is not yet parsable by the current type check functions, you can simply `register()` it, which will append it to the end of the type list.

**makeparser(*remove\_blank\_text=True, \*\*kw*)**

Create a new XML parser for `objectify` trees.

You can pass all keyword arguments that are supported by `etree.XMLParser()`. Note that this parser defaults to removing blank text. You can disable this by passing the `remove_blank_text` boolean keyword option yourself.

---

**parse**(*f*, *parser=None*)

Parse a file or file-like object with the objectify parser.

You can pass a different parser as second argument.

---

**pyannotate**(*element\_or\_tree*, *ignore\_old=False*, *ignore\_xsi=False*, *empty\_pytype=None*)

Recursively annotates the elements of an XML tree with 'pytype' attributes.

If the 'ignore\_old' keyword argument is True (the default), current 'pytype' attributes will be ignored and replaced. Otherwise, they will be checked and only replaced if they no longer fit the current text value.

Setting the keyword argument `ignore_xsi` to True makes the function additionally ignore existing `xsi:type` annotations. The default is to use them as a type hint.

The default annotation of empty elements can be set with the `empty_pytype` keyword argument. The default is not to annotate empty elements. Pass 'str', for example, to make string values the default.

---

**pytypename**(*obj*)

Find the name of the corresponding PyType for a Python object.

---

**setDefaultParser**(...)

**Deprecated:** use `set_default_parser()` instead.

---

**setPytypeAttributeTag**(...)

**Deprecated:** use `set_pytype_attribute_tag()` instead.

**set\_default\_parser**(*new\_parser=None*)

Replace the default parser used by `objectify`'s `Element()` and `fromstring()` functions.

The new parser must be an `etree.XMLParser`.

Call without arguments to reset to the original parser.

**set\_pytype\_attribute\_tag**(*attribute\_tag=None*)

Change name and namespace of the XML attribute that holds Python type information.

Do not use this unless you know what you are doing.

Reset by calling without argument.

Default: “`{http://codespeak.net/lxml/objectify/pytype}pytype`”

**xsiannotate**(*element\_or\_tree*, *ignore\_old=False*, *ignore\_pytype=False*, *empty\_type=None*)

Recursively annotates the elements of an XML tree with `'xsi:type'` attributes.

If the `'ignore_old'` keyword argument is `True` (the default), current `'xsi:type'` attributes will be ignored and replaced. Otherwise, they will be checked and only replaced if they no longer fit the current text value.

Note that the mapping from Python types to XSI types is usually ambiguous. Currently, only the first XSI type name in the corresponding `PyType` definition will be used for annotation. Thus, you should consider naming the widest type first if you define additional types.

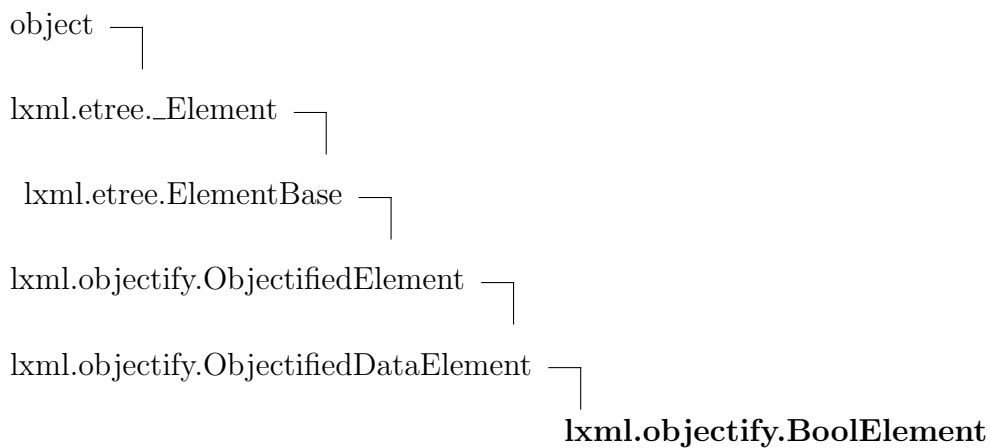
Setting the keyword argument `ignore_pytype` to `True` makes the function additionally ignore existing `pytype` annotations. The default is to use them as a type hint.

The default annotation of empty elements can be set with the `empty_type` keyword argument. The default is not to annotate empty elements. Pass `'string'`, for example, to make string values the default.

## B.16.2 Variables

Name	Description
E	<b>Value:</b> <lxml.objectify.ElementMaker object at 0x87dcb1c>
PYTYPE_ATTRIBUTE	<b>Value:</b> '{http://codespeak.net/lxml/objectify/pytype}pytype'

### B.16.3 Class BoolElement



Boolean type base on string values: 'true' or 'false'.

#### Methods

<code>__eq__(x, y)</code>
<code>x==y</code>

<code>__ge__(x, y)</code>
<code>x&gt;=y</code>

<code>__gt__(x, y)</code>
<code>x&gt;y</code>

<code>__le__(x, y)</code>
<hr/>
<code>x&lt;=y</code>

<code>__lt__(x, y)</code>
<hr/>
<code>x&lt;y</code>

<code>__ne__(x, y)</code>
<hr/>
<code>x!=y</code>

<code>__new__(T, S, ...)</code>
<hr/>
<b>Return Value</b> a new object with type S, a subtype of T
Overrides: object.__new__

<code>__nonzero__(x)</code>
<hr/>
<code>x != 0</code> Overrides: lxml.etree._Element.__nonzero__

<code>__repr__(self)</code>
<hr/>
<code>repr(x)</code> Overrides: object.__repr__ exitit(inherited documentation)

<code>__str__(...)</code>
<hr/>
<code>str(x)</code> Overrides: object.__str__ exitit(inherited documentation)

**Inherited from *lxml.objectify.ObjectifiedElement*(Section B.16.12)**

`__delattr__()`, `__delitem__()`, `__getattr__()`, `__getattribute__()`, `__getitem__()`, `__iter__()`, `__len__()`, `__setattr__()`, `__setitem__()`, `addattr()`, `countchildren()`, `descendant-paths()`, `find()`, `findall()`, `findtext()`, `getchildren()`, `iterfind()`

**Inherited from *lxml.etree.\_Element***

`__contains__()`, `__copy__()`, `__deepcopy__()`, `__reversed__()`, `addnext()`, `addprevious()`, `append()`, `clear()`, `extend()`, `get()`, `getiterator()`, `getnext()`, `getparent()`, `getprevious()`, `getroottree()`, `index()`, `insert()`, `items()`, `iter()`, `iterancestors()`, `iterchildren()`, `iterdescendants()`, `itersiblings()`, `itertext()`, `keys()`, `makeelement()`, `remove()`, `replace()`, `set()`, `values()`, `xpath()`

**Inherited from object**

`__hash__()`, `__init__()`, `__reduce__()`, `__reduce_ex__()`

**Properties**

Name	Description
pyval	
<i>Inherited from lxml.objectify.ObjectifiedElement (Section B.16.12)</i>	
text	
<i>Inherited from lxml.etree._Element</i>	
attrib, nsmap, prefix, sourceline, tag, tail	
<i>Inherited from object</i>	
__class__	

**B.16.4 Class ElementMaker**

`ElementMaker(self, namespace=None, nsmap=None, annotate=True, makeelement=None)`

**Methods**

<code>__getattr__(...)</code>
<code>__getattribute__(...)</code>
<code>x.__getattribute__('name') &lt;==&gt; x.name</code> Overrides: <code>object.__getattribute__</code>
<code>__init__(self, namespace=None, nsmap=None, annotate=True, makeelement=None)</code>
<code>x.__init__(...)</code> initializes x; see <code>x.__class__.__doc__</code> for signature Overrides: <code>object.__init__</code>

`__new__(T, S, ...)`

**Return Value**

a new object with type `S`, a subtype of `T`

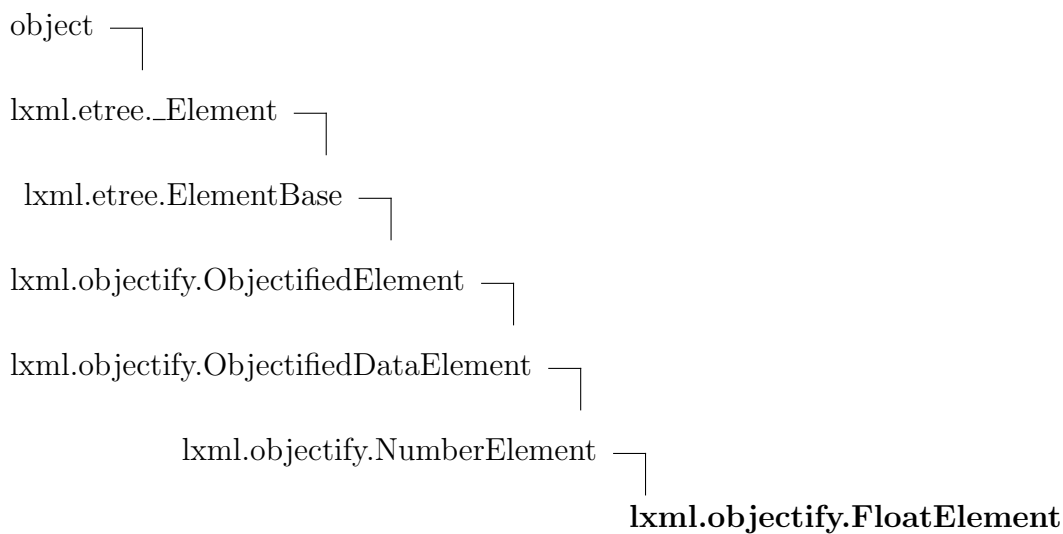
Overrides: `object.__new__`

**Inherited from `object`**

`__delattr__()`, `__hash__()`, `__reduce__()`, `__reduce_ex__()`, `__repr__()`, `__setattr__()`,  
`__str__()`

**Properties**

Name	Description
<i>Inherited from <code>object</code></i>	
<code>__class__</code>	

**B.16.5 Class `FloatElement`****Methods**

`__new__(T, S, ...)`

**Return Value**

a new object with type `S`, a subtype of `T`

Overrides: `object.__new__`

**Inherited from `lxml.objectify.NumberElement` (Section [B.16.9](#))**

`__abs__()`, `__add__()`, `__and__()`, `__div__()`, `__eq__()`, `__float__()`, `__ge__()`, `__gt__()`,

`__int__()`, `__invert__()`, `__le__()`, `__long__()`, `__lshift__()`, `__lt__()`, `__mod__()`, `__mul__()`, `__ne__()`, `__neg__()`, `__nonzero__()`, `__or__()`, `__pos__()`, `__pow__()`, `__radd__()`, `__rand__()`, `__rdiv__()`, `__repr__()`, `__rlshift__()`, `__rmod__()`, `__rmul__()`, `__ror__()`, `__rpow__()`, `__rrshift__()`, `__rshift__()`, `__rsub__()`, `__rtruediv__()`, `__rxor__()`, `__str__()`, `__sub__()`, `__truediv__()`, `__xor__()`

***Inherited from `lxml.objectify.ObjectifiedElement` (Section [B.16.12](#))***

`__delattr__()`, `__delitem__()`, `__getattr__()`, `__getattribute__()`, `__getitem__()`, `__iter__()`, `__len__()`, `__setattr__()`, `__setitem__()`, `addattr()`, `countchildren()`, `descendant-paths()`, `find()`, `findall()`, `findtext()`, `getchildren()`, `iterfind()`

***Inherited from `lxml.etree._Element`***

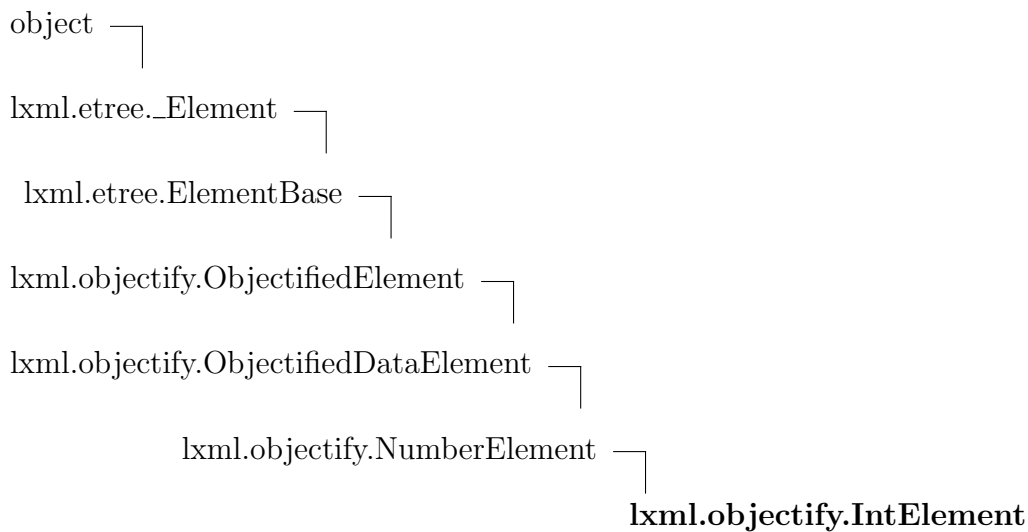
`__contains__()`, `__copy__()`, `__deepcopy__()`, `__reversed__()`, `addnext()`, `addprevious()`, `append()`, `clear()`, `extend()`, `get()`, `getiterator()`, `getnext()`, `getparent()`, `getprevious()`, `getroottree()`, `index()`, `insert()`, `items()`, `iter()`, `iterancestors()`, `iterchildren()`, `iterdescendants()`, `itersiblings()`, `itertext()`, `keys()`, `makeelement()`, `remove()`, `replace()`, `set()`, `values()`, `xpath()`

***Inherited from `object`***

`__hash__()`, `__init__()`, `__reduce__()`, `__reduce_ex__()`

**Properties**

Name	Description
<i>Inherited from <code>lxml.objectify.NumberElement</code> (Section <a href="#">B.16.9</a>)</i> pyval	
<i>Inherited from <code>lxml.objectify.ObjectifiedElement</code> (Section <a href="#">B.16.12</a>)</i> text	
<i>Inherited from <code>lxml.etree._Element</code></i> attrib, nsmap, prefix, sourceline, tag, tail	
<i>Inherited from <code>object</code></i> __class__	

**B.16.6 Class `IntElement`****Methods**

<b><code>__new__(T, S, ...)</code></b> <b>Return Value</b> a new object with type S, a subtype of T Overrides: <code>object.__new__</code>
---

***Inherited from `lxml.objectify.NumberElement`(Section [B.16.9](#))***

`__abs__()`, `__add__()`, `__and__()`, `__div__()`, `__eq__()`, `__float__()`, `__ge__()`, `__gt__()`,  
`__int__()`, `__invert__()`, `__le__()`, `__long__()`, `__lshift__()`, `__lt__()`, `__mod__()`, `__mul__()`,  
`__ne__()`, `__neg__()`, `__nonzero__()`, `__or__()`, `__pos__()`, `__pow__()`, `__radd__()`,  
`__rand__()`, `__rdiv__()`, `__repr__()`, `__rlshift__()`, `__rmod__()`, `__rmul__()`, `__ror__()`,  
`__rpow__()`, `__rrshift__()`, `__rshift__()`, `__rsub__()`, `__rtruediv__()`, `__rxor__()`, `__str__()`,  
`__sub__()`, `__truediv__()`, `__xor__()`

***Inherited from `lxml.objectify.ObjectifiedElement`(Section [B.16.12](#))***

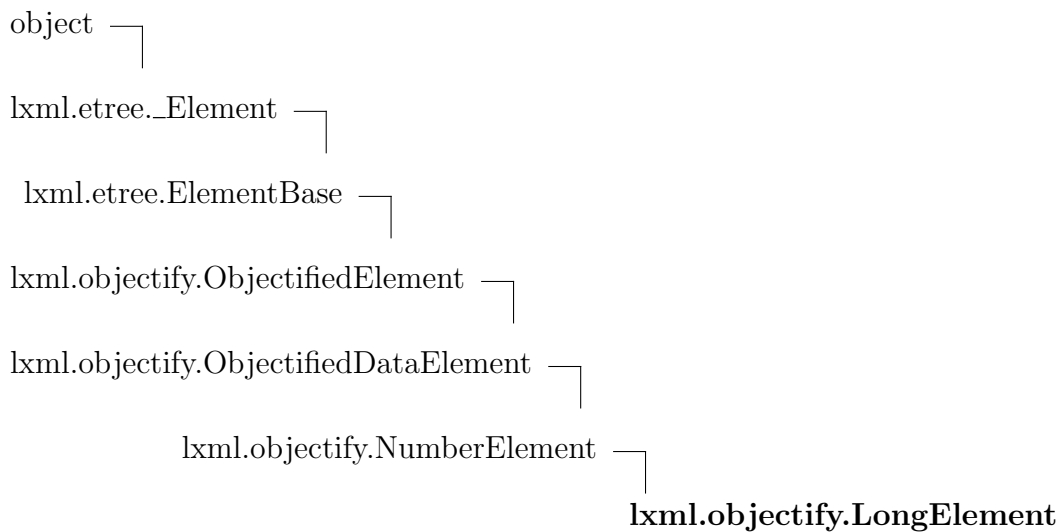
`__delattr__()`, `__delitem__()`, `__getattr__()`, `__getattribute__()`, `__getitem__()`, `__iter__()`,  
`__len__()`, `__setattr__()`, `__setitem__()`, `addattr()`, `countchildren()`, `descendant-`  
`paths()`, `find()`, `findall()`, `findtext()`, `getchildren()`, `iterfind()`

***Inherited from `lxml.etree._Element`***

`__contains__()`, `__copy__()`, `__deepcopy__()`, `__reversed__()`, `addnext()`, `addprevi-`  
`ous()`, `append()`, `clear()`, `extend()`, `get()`, `getiterator()`, `getnext()`, `getparent()`,  
`getprevious()`, `getroottree()`, `index()`, `insert()`, `items()`, `iter()`, `iterancestors()`,  
`iterchildren()`, `iterdescendants()`, `itersiblings()`, `itertext()`, `keys()`, `makeelement()`,  
`remove()`, `replace()`, `set()`, `values()`, `xpath()`

**Inherited from object**`__hash__()`, `__init__()`, `__reduce__()`, `__reduce_ex__()`**Properties**

Name	Description
<code>pyval</code>	<i>Inherited from <code>lxml.objectify.NumberElement</code> (Section <a href="#">B.16.9</a>)</i>
<code>text</code>	<i>Inherited from <code>lxml.objectify.ObjectifiedElement</code> (Section <a href="#">B.16.12</a>)</i>
<code>attrib</code> , <code>nsmap</code> , <code>prefix</code> , <code>sourceline</code> , <code>tag</code> , <code>tail</code>	<i>Inherited from <code>lxml.etree._Element</code></i>
<code>__class__</code>	<i>Inherited from <code>object</code></i>

**B.16.7 Class LongElement****Methods**`__new__(T, S, ...)`**Return Value**

a new object with type S, a subtype of T

Overrides: `object.__new__`*Inherited from `lxml.objectify.NumberElement`(Section [B.16.9](#))*

`__abs__()`, `__add__()`, `__and__()`, `__div__()`, `__eq__()`, `__float__()`, `__ge__()`, `__gt__()`,  
`__int__()`, `__invert__()`, `__le__()`, `__long__()`, `__lshift__()`, `__lt__()`, `__mod__()`, `__mul__()`,  
`__ne__()`, `__neg__()`, `__nonzero__()`, `__or__()`, `__pos__()`, `__pow__()`, `__radd__()`,  
`__rand__()`, `__rdiv__()`, `__repr__()`, `__rlshift__()`, `__rmod__()`, `__rmul__()`, `__ror__()`,  
`__rpow__()`, `__rrshift__()`, `__rshift__()`, `__rsub__()`, `__rtruediv__()`, `__rxor__()`, `__str__()`,  
`__sub__()`, `__truediv__()`, `__xor__()`

***Inherited from lxml.objectify.ObjectifiedElement (Section B.16.12)***

`__delattr__()`, `__delitem__()`, `__getattr__()`, `__getattribute__()`, `__getitem__()`, `__iter__()`,  
`__len__()`, `__setattr__()`, `__setitem__()`, `addattr()`, `countchildren()`, `descendant-`  
`paths()`, `find()`, `findall()`, `findtext()`, `getchildren()`, `iterfind()`

***Inherited from lxml.etree.\_Element***

`__contains__()`, `__copy__()`, `__deepcopy__()`, `__reversed__()`, `addnext()`, `addprevi-`  
`ous()`, `append()`, `clear()`, `extend()`, `get()`, `getiterator()`, `getnext()`, `getparent()`,  
`getprevious()`, `getroottree()`, `index()`, `insert()`, `items()`, `iter()`, `iterancestors()`,  
`iterchildren()`, `iterdescendants()`, `itersiblings()`, `itertext()`, `keys()`, `makeelement()`,  
`remove()`, `replace()`, `set()`, `values()`, `xpath()`

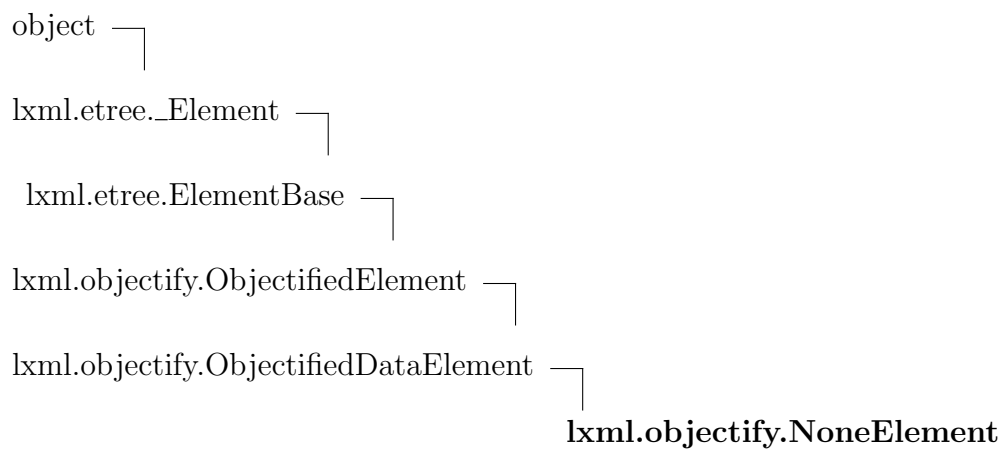
***Inherited from object***

`__hash__()`, `__init__()`, `__reduce__()`, `__reduce_ex__()`

**Properties**

Name	Description
<i>Inherited from lxml.objectify.NumberElement (Section B.16.9)</i> pyval	
<i>Inherited from lxml.objectify.ObjectifiedElement (Section B.16.12)</i> text	
<i>Inherited from lxml.etree._Element</i> attrib, nsmap, prefix, sourceline, tag, tail	
<i>Inherited from object</i> __class__	

## B.16.8 Class NoneElement



## Methods

<code>__eq__(x, y)</code>
<hr/>
<code>x==y</code>

<code>__ge__(x, y)</code>
<hr/>
<code>x&gt;=y</code>

<code>__gt__(x, y)</code>
<hr/>
<code>x&gt;y</code>

<code>__le__(x, y)</code>
<hr/>
<code>x&lt;=y</code>

<code>__lt__(x, y)</code>
<hr/>
<code>x&lt;y</code>

<code>__ne__(x, y)</code>
<code>x!=y</code>

<code>__new__(T, S, ...)</code>
<b>Return Value</b> a new object with type S, a subtype of T
Overrides: <code>object.__new__</code>

<code>__nonzero__(x)</code>
<code>x != 0</code> Overrides: <code>lxml.etree._Element.__nonzero__</code>

<code>__repr__(self)</code>
<code>repr(x)</code> Overrides: <code>object.__repr__</code> <code>exitit(inherited documentation)</code>

<code>__str__(...)</code>
<code>str(x)</code> Overrides: <code>object.__str__</code> <code>exitit(inherited documentation)</code>

**Inherited from `lxml.objectify.ObjectifiedElement` (Section [B.16.12](#))**

`__delattr__()`, `__delitem__()`, `__getattr__()`, `__getattribute__()`, `__getitem__()`, `__iter__()`, `__len__()`, `__setattr__()`, `__setitem__()`, `addattr()`, `countchildren()`, `descendant-paths()`, `find()`, `findall()`, `findtext()`, `getchildren()`, `iterfind()`

**Inherited from `lxml.etree._Element`**

`__contains__()`, `__copy__()`, `__deepcopy__()`, `__reversed__()`, `addnext()`, `addprevious()`, `append()`, `clear()`, `extend()`, `get()`, `getiterator()`, `getnext()`, `getparent()`, `getprevious()`, `getroottree()`, `index()`, `insert()`, `items()`, `iter()`, `iterancestors()`, `iterchildren()`, `iterdescendants()`, `itersiblings()`, `itertext()`, `keys()`, `makeelement()`, `remove()`, `replace()`, `set()`, `values()`, `xpath()`

**Inherited from `object`**

`__hash__()`, `__init__()`, `__reduce__()`, `__reduce_ex__()`

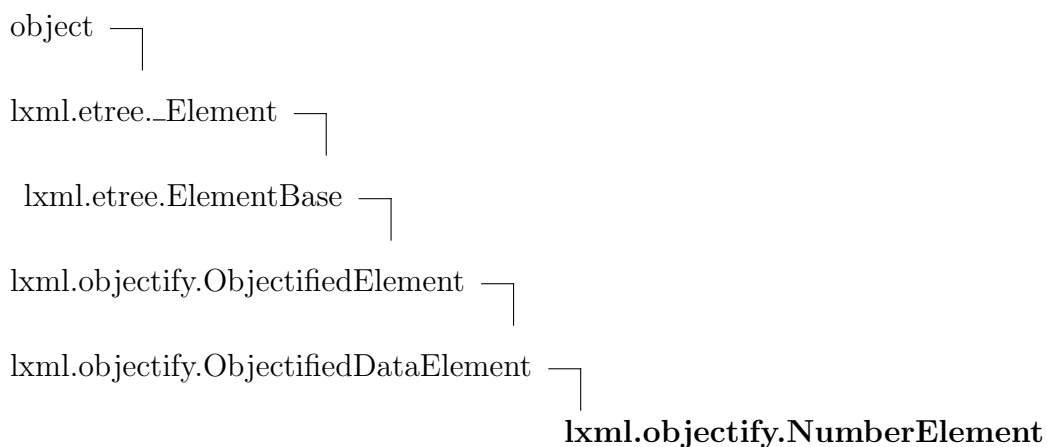
**Properties**

Name	Description
<code>pyval</code>	
<i>Inherited from <code>lxml.objectify.ObjectifiedElement</code> (Section <a href="#">B.16.12</a>)</i>	

*continued on next page*

Name	Description
<code>text</code>	
	<i>Inherited from <code>lxml.etree._Element</code></i>
	<code>attrib</code> , <code>nsmap</code> , <code>prefix</code> , <code>sourceline</code> , <code>tag</code> , <code>tail</code>
	<i>Inherited from <code>object</code></i>
<code>__class__</code>	

### B.16.9 Class `NumberElement`



**Known Subclasses:** `lxml.objectify.FloatElement`, `lxml.objectify.IntElement`, `lxml.objectify.LongElement`

#### Methods

<code>__abs__(...)</code>
<code>__add__(...)</code>
<code>__and__(...)</code>
<code>__div__(...)</code>
<code>__eq__(<i>x</i>, <i>y</i>)</code>
<code>x==y</code>
<code>__float__(...)</code>

`__ge__(x, y)`

`x >= y`

`__gt__(x, y)`

`x > y`

`__int__(...)`

`__invert__(...)`

`__le__(x, y)`

`x <= y`

`__long__(...)`

`__lshift__(...)`

`__lt__(x, y)`

`x < y`

`__mod__(...)`

`__mul__(...)`

`__ne__(x, y)`

`x != y`

`__neg__(...)`

**\_\_new\_\_**(*T*, *S*, ...)

**Return Value**

a new object with type *S*, a subtype of *T*

Overrides: object.\_\_new\_\_

**\_\_nonzero\_\_**(*x*)

$x \neq 0$  Overrides: lxml.etree.Element.\_\_nonzero\_\_

**\_\_or\_\_**(...)

**\_\_pos\_\_**(...)

**\_\_pow\_\_**(*x*, *y*, *z*=...)

$\text{pow}(x, y[, z])$

**\_\_radd\_\_**(*x*, *y*)

$y+x$

**\_\_rand\_\_**(*x*, *y*)

$y\&x$

**\_\_rdiv\_\_**(*x*, *y*)

$y/x$

**\_\_repr\_\_**(*self*)

$\text{repr}(x)$  Overrides: object.\_\_repr\_\_ extit(herited documentation)

**\_\_rlshift\_\_**(*x*, *y*)

$y \ll x$

`__rmod__(x, y)`  
-----  
`y%x`

`__rmul__(x, y)`  
-----  
`y*x`

`__ror__(x, y)`  
-----  
`y|x`

`__rpow__(y, x, z=...)`  
-----  
`pow(x, y[, z])`

`__rrshift__(x, y)`  
-----  
`y>>x`

`__rshift__(...)`

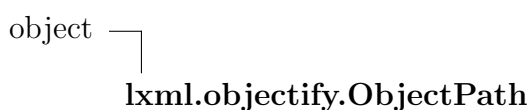
`__rsub__(x, y)`  
-----  
`y-x`

`__rtruediv__(x, y)`  
-----  
`y/x`

`__rxor__(x, y)`  
-----  
`y^x`

`__str__(...)``str(x)` Overrides: `object.__str__` `exitit`(inherited documentation)`__sub__(...)``__truediv__(...)``__xor__(...)`***Inherited from `lxml.objectify.ObjectifiedElement` (Section [B.16.12](#))***`__delattr__()`, `__delitem__()`, `__getattr__()`, `__getattribute__()`, `__getitem__()`, `__iter__()`, `__len__()`, `__setattr__()`, `__setitem__()`, `addattr()`, `countchildren()`, `descendant-paths()`, `find()`, `findall()`, `findtext()`, `getchildren()`, `iterfind()`***Inherited from `lxml.etree._Element`***`__contains__()`, `__copy__()`, `__deepcopy__()`, `__reversed__()`, `addnext()`, `addprevious()`, `append()`, `clear()`, `extend()`, `get()`, `getiterator()`, `getnext()`, `getparent()`, `getprevious()`, `getroottree()`, `index()`, `insert()`, `items()`, `iter()`, `iterancestors()`, `iterchildren()`, `iterdescendants()`, `itersiblings()`, `itertext()`, `keys()`, `makeelement()`, `remove()`, `replace()`, `set()`, `values()`, `xpath()`***Inherited from `object`***`__hash__()`, `__init__()`, `__reduce__()`, `__reduce_ex__()`**Properties**

Name	Description
<code>pyval</code>	
<code>text</code>	<i>Inherited from <code>lxml.objectify.ObjectifiedElement</code> (Section <a href="#">B.16.12</a>)</i>
<code>attrib</code> , <code>nsmap</code> , <code>prefix</code> , <code>sourceline</code> , <code>tag</code> , <code>tail</code>	<i>Inherited from <code>lxml.etree._Element</code></i>
<code>__class__</code>	<i>Inherited from <code>object</code></i>

**B.16.10 Class *ObjectPath***

`ObjectPath(path)` Immutable object that represents a compiled object path.

Example for a path: `'root.child[1].{other}child[25]'`

## Methods

**`__call__`**(...)

Follow the attribute path in the object structure and return the target attribute value.

If it is not found, either returns a default value (if one was passed as second argument) or raises `AttributeError`.

**`__init__`**(*path*)

`x.__init__(...)` initializes `x`; see `x.__class__.__doc__` for signature. Overrides: `object.__init__`

**`__new__`**(*T*, *S*, ...)

### Return Value

a new object with type `S`, a subtype of `T`

Overrides: `object.__new__`

**`__str__`**(...)

`str(x)` Overrides: `object.__str__` `exitit`(inherited documentation)

**`addattr`**(*self*, *root*, *value*)

Append a value to the target element in a subtree.

If any of the children on the path does not exist, it is created.

**`hasattr`**(*self*, *root*)

<b>setattr</b> ( <i>self</i> , <i>root</i> , <i>value</i> )
Set the value of the target element in a subtree. If any of the children on the path does not exist, it is created.

**Inherited from object**

`__delattr__()`, `__getattr__()`, `__hash__()`, `__reduce__()`, `__reduce_ex__()`, `__repr__()`,  
`__setattr__()`

**Properties**

Name	Description
find	
<i>Inherited from object</i>	
<code>__class__</code>	

**B.16.11 Class ObjectifiedDataElement**

object └

lxml.etree.\_Element └

lxml.etree.ElementBase └

lxml.objectify.ObjectifiedElement └

**lxml.objectify.ObjectifiedDataElement**

**Known Subclasses:** `lxml.objectify.BoolElement`, `lxml.objectify.NumberElement`, `lxml.objectify.NoneElement`, `lxml.objectify.StringElement`

This is the base class for all data type Elements. Subclasses should override the 'pyval' property and possibly the `__str__` method.

**Methods**

<code>__new__(<i>T</i>, <i>S</i>, ...)</code>
<b>Return Value</b> a new object with type <i>S</i> , a subtype of <i>T</i>
Overrides: <code>object.__new__</code>

<code>__repr__(self)</code>
-----------------------------

<code>repr(x)</code> Overrides: <code>object.__repr__</code> extit(inherited documentation)
---

<code>__str__(...)</code>
---------------------------

<code>str(x)</code> Overrides: <code>object.__str__</code> extit(inherited documentation)
---

***Inherited from `lxml.objectify.ObjectifiedElement` (Section [B.16.12](#))***

`__delattr__()`, `__delitem__()`, `__getattr__()`, `__getattribute__()`, `__getitem__()`, `__iter__()`, `__len__()`, `__setattr__()`, `__setitem__()`, `addattr()`, `countchildren()`, `descendant-paths()`, `find()`, `findall()`, `findtext()`, `getchildren()`, `iterfind()`

***Inherited from `lxml.etree._Element`***

`__contains__()`, `__copy__()`, `__deepcopy__()`, `__nonzero__()`, `__reversed__()`, `addnext()`, `addprevious()`, `append()`, `clear()`, `extend()`, `get()`, `getiterator()`, `getnext()`, `get-parent()`, `getprevious()`, `getroottree()`, `index()`, `insert()`, `items()`, `iter()`, `iter-ancestors()`, `iterchildren()`, `iterdescendants()`, `itersiblings()`, `itertext()`, `keys()`, `makeelement()`, `remove()`, `replace()`, `set()`, `values()`, `xpath()`

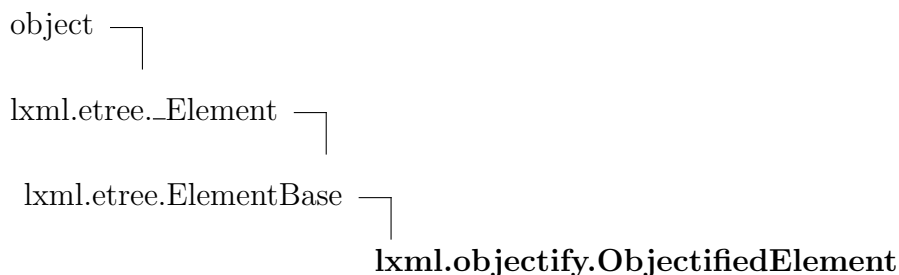
***Inherited from `object`***

`__hash__()`, `__init__()`, `__reduce__()`, `__reduce_ex__()`

**Properties**

Name	Description
<code>pyval</code>	
<i>Inherited from <code>lxml.objectify.ObjectifiedElement</code> (Section <a href="#">B.16.12</a>)</i>	
<code>text</code>	
<i>Inherited from <code>lxml.etree._Element</code></i>	
<code>attrib</code> , <code>nsmmap</code> , <code>prefix</code> , <code>sourceline</code> , <code>tag</code> , <code>tail</code>	
<i>Inherited from <code>object</code></i>	
<code>__class__</code>	

### B.16.12 Class ObjectifiedElement



**Known Subclasses:** lxml.objectify.ObjectifiedDataElement

Main XML Element class.

Element children are accessed as object attributes. Multiple children with the same name are available through a list index. Example:

```

>>> root = etree.XML("<root><c1><c2>0</c2><c2>1</c2></c1></root>")
>>> second_c2 = root.c1.c2[1]
  
```

Note that you cannot (and must not) instantiate this class or its subclasses.

#### Methods

<b>__delattr__</b> (...) <hr/> x.__delattr__('name') <==> del x.name Overrides: object.__delattr__
---

<b>__delitem__</b> (x, y) <hr/> del x[y] Overrides: lxml.etree._Element.__delitem__
--

<b>__getattr__</b> (...) <hr/> Return the (first) child with the given tag name. If no namespace is provided, the child will be looked up in the same one as self.
---

<b>__getattribute__</b> (...) <hr/> x.__getattribute__('name') <==> x.name Overrides: object.__getattribute__
--

---

**\_\_getitem\_\_**(...)
 

---

Return a sibling, counting from the first child of the parent. The method behaves like both a dict and a sequence.

- If argument is an integer, returns the sibling at that position.
- If argument is a string, does the same as `getattr()`. This can be used to provide namespaces for element lookup, or to look up children with special names (`text` etc.).
- If argument is a slice object, returns the matching slice.

Overrides: `lxml.etree._Element.__getitem__`

---

**\_\_iter\_\_**(*self*)
 

---

Iterate over self and all siblings with the same tag. Overrides: `lxml.etree._Element.__iter__`

---

**\_\_len\_\_**(*x*)
 

---

`len(x)` Overrides: `lxml.etree._Element.__len__`

---

**\_\_new\_\_**(*T*, *S*, ...)
 

---

**Return Value**

a new object with type *S*, a subtype of *T*

Overrides: `object.__new__`

---

**\_\_setattr\_\_**(...)
 

---

`x.__setattr__('name', value) <==> x.name = value` Overrides: `object.__setattr__`

---

**\_\_setitem\_\_**(*x*, *i*, *y*)
 

---

`x[i]=y` Overrides: `lxml.etree._Element.__setitem__`

**\_\_str\_\_**(...)

str(x) Overrides: object.\_\_str\_\_ exitit(inherited documentation)

**addattr**(*self*, *tag*, *value*)

Add a child value to the element.

As opposed to append(), it sets a data value, not an element.

**countchildren**(*self*)

Return the number of children of this element, regardless of their name.

**descendantpaths**(*self*, *prefix=None*)

Returns a list of object path expressions for all descendants.

**find**(*self*, *path*)

Finds the first matching subelement, by tag name or path. Overrides: lxml.etree.\_Element.find

**findall**(*self*, *path*)

Finds all matching subelements, by tag name or path. Overrides: lxml.etree.\_Element.findall

**findtext**(*self*, *path*, *default=None*)

Finds text for the first matching subelement, by tag name or path. Overrides: lxml.etree.\_Element.findtext

**getchildren**(*self*)

Returns a sequence of all direct children. The elements are returned in document order. Overrides: lxml.etree.\_Element.getchildren

**iterfind**(*self*, *path*)

Iterates over all matching subelements, by tag name or path. Overrides: lxml.etree.\_Element.iterfind

**Inherited from *lxml.etree.\_Element***

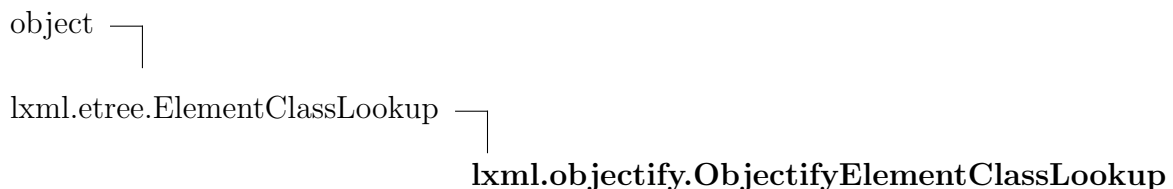
`__contains__()`, `__copy__()`, `__deepcopy__()`, `__nonzero__()`, `__repr__()`, `__reversed__()`, `addnext()`, `addprevious()`, `append()`, `clear()`, `extend()`, `get()`, `getiterator()`, `getnext()`, `getparent()`, `getprevious()`, `getroottree()`, `index()`, `insert()`, `items()`, `iter()`, `iterancestors()`, `iterchildren()`, `iterdescendants()`, `itersiblings()`, `itertext()`, `keys()`, `makeelement()`, `remove()`, `replace()`, `set()`, `values()`, `xpath()`

**Inherited from *object***

`__hash__()`, `__init__()`, `__reduce__()`, `__reduce_ex__()`

**Properties**

Name	Description
text	Text before the first subelement. This is either a string or the value None, if there was no text.
<i>Inherited from <i>lxml.etree._Element</i></i>	
attrib, nsmmap, prefix, sourceline, tag, tail	
<i>Inherited from <i>object</i></i>	
__class__	

**B.16.13 Class ObjectifyElementClassLookup**

`ObjectifyElementClassLookup(self, tree_class=None, empty_data_class=None)` Element class lookup method that uses the objectify classes.

**Methods**

`__init__(self, tree_class=None, empty_data_class=None)`

`x.__init__(...)` initializes x; see `x.__class__.__doc__` for signature Overrides: `object.__init__`

```
__new__(T, S, ...)
```

**Return Value**

a new object with type S, a subtype of T

Overrides: object.\_\_new\_\_

**Inherited from object**

```
__delattr__(), __getattr__(), __hash__(), __reduce__(), __reduce_ex__(), __repr__(),
__setattr__(), __str__()
```

**Properties**

Name	Description
<i>Inherited from object</i>	
<code>__class__</code>	

**B.16.14 Class PyType**

```
object ┌
      │
      └─ lxml.objectify.PyType
```

`PyType(self, name, type_check, type_class, stringify=None)` User defined type.

Named type that contains a type check function and a type class that inherits from `ObjectifiedDataElement`. The type check must take a string as argument and raise `ValueError` or `TypeError` if it cannot handle the string value. It may be `None` in which case it is not considered for type guessing.

Example:

```
PyType('int', int, MyIntClass).register()
```

Note that the order in which types are registered matters. The first matching type will be used.

**Methods**

```
__init__(self, name, type_check, type_class, stringify=None)
```

`x.__init__(...)` initializes x; see `x.__class__.__doc__` for signature Overrides: `object.__init__`

`__new__(T, S, ...)`

**Return Value**

a new object with type S, a subtype of T

Overrides: `object.__new__`

`__repr__(...)`

`repr(x)` Overrides: `object.__repr__` `exitit`(inherited documentation)

`register(self, before=None, after=None)`

Register the type.

The additional keyword arguments 'before' and 'after' accept a sequence of type names that must appear before/after the new type in the type list. If any of them is not currently known, it is simply ignored. Raises `ValueError` if the dependencies cannot be fulfilled.

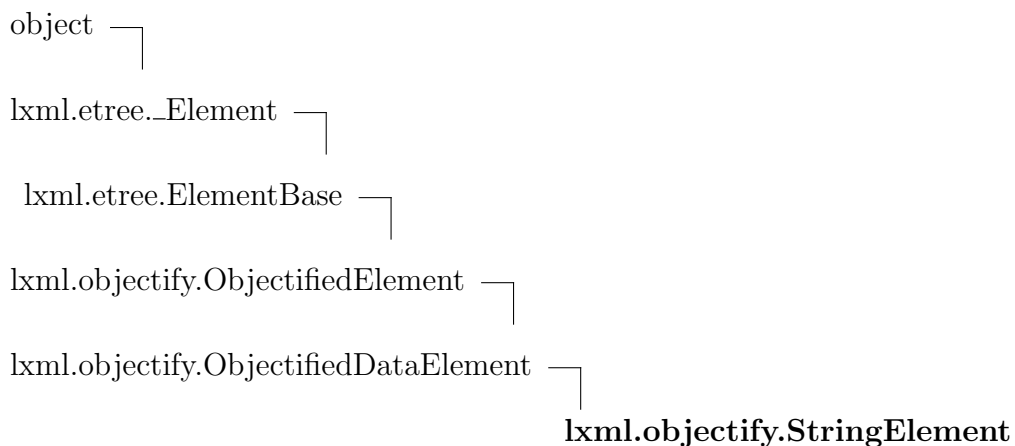
`unregister(self)`

***Inherited from object***

`__delattr__()`, `__getattr__()`, `__hash__()`, `__reduce__()`, `__reduce_ex__()`, `__setattr__()`, `__str__()`

**Properties**

Name	Description
<code>name</code>	
<code>stringify</code>	
<code>type_check</code>	
<code>xmlSchemaTypes</code>	The list of XML Schema datatypes this Python type maps to. Note that this must be set before registering the type!
<i>Inherited from object</i>	
<code>__class__</code>	

**B.16.15 Class `StringElement`**

String data class.

Note that this class does *not* support the sequence protocol of strings: `len()`, `iter()`, `str_attr[0]`, `str_attr[0:1]`, etc. are *not* supported. Instead, use the `.text` attribute to get a 'real' string.

**Methods**

<code>__add__(...)</code>
<code>__eq__(x, y)</code> <hr/> <code>x==y</code>
<code>__ge__(x, y)</code> <hr/> <code>x&gt;=y</code>
<code>__gt__(x, y)</code> <hr/> <code>x&gt;y</code>
<code>__le__(x, y)</code> <hr/> <code>x&lt;=y</code>

`__lt__(x, y)`

---

`x < y`

`__mod__(...)`

`__mul__(...)`

`__ne__(x, y)`

---

`x != y`

`__new__(T, S, ...)`

**Return Value**  
a new object with type `S`, a subtype of `T`

Overrides: `object.__new__`

`__nonzero__(x)`

---

`x != 0` Overrides: `lxml.etree.Element.__nonzero__`

`__radd__(x, y)`

---

`y + x`

`__repr__(self)`

`repr(x)` Overrides: `object.__repr__` `exitit`(inherited documentation)

`__rmod__(x, y)`

---

`y % x`

`__rmul__(x, y)`

---

`y * x`

<code>strlen(...)</code>
--------------------------

*Inherited from `lxml.objectify.ObjectifiedDataElement` (Section [B.16.11](#))*

`__str__()`

*Inherited from `lxml.objectify.ObjectifiedElement` (Section [B.16.12](#))*

`__delattr__()`, `__delitem__()`, `__getattr__()`, `__getattribute__()`, `__getitem__()`, `__iter__()`, `__len__()`, `__setattr__()`, `__setitem__()`, `addattr()`, `countchildren()`, `descendant-paths()`, `find()`, `findall()`, `findtext()`, `getchildren()`, `iterfind()`

*Inherited from `lxml.etree._Element`*

`__contains__()`, `__copy__()`, `__deepcopy__()`, `__reversed__()`, `addnext()`, `addprevious()`, `append()`, `clear()`, `extend()`, `get()`, `getiterator()`, `getnext()`, `getparent()`, `getprevious()`, `getroottree()`, `index()`, `insert()`, `items()`, `iter()`, `iterancestors()`, `iterchildren()`, `iterdescendants()`, `itersiblings()`, `itertext()`, `keys()`, `makeelement()`, `remove()`, `replace()`, `set()`, `values()`, `xpath()`

*Inherited from object*

`__hash__()`, `__init__()`, `__reduce__()`, `__reduce_ex__()`

## Properties

Name	Description
<code>pyval</code>	
<code>text</code>	<i>Inherited from <code>lxml.objectify.ObjectifiedElement</code> (Section <a href="#">B.16.12</a>)</i>
<code>attrib</code> , <code>nsmap</code> , <code>prefix</code> , <code>sourceline</code> , <code>tag</code> , <code>tail</code>	<i>Inherited from <code>lxml.etree._Element</code></i>
<code>__class__</code>	<i>Inherited from object</i>

## B.17 Module `lxml.pyclasslookup`

A whole-tree Element class lookup scheme for `lxml.etree`.

This class lookup scheme allows access to the entire XML tree in read-only mode. To use it, let a class inherit from `PythonElementClassLookup` and re-implement the `lookup(self, doc, root)` method:

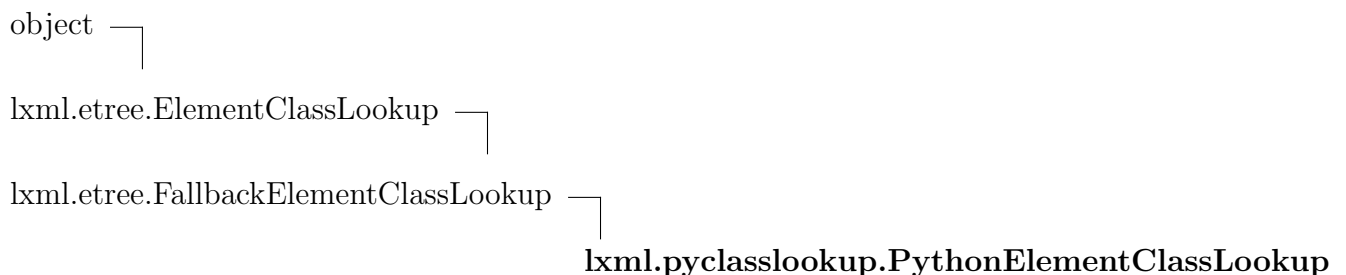
```
>>> from lxml import etree, pyclasslookup
>>>
>>> class MyElementClass(etree.ElementBase):
...     honkey = True
...
>>> class MyLookup(pyclasslookup.PythonElementClassLookup):
...     def lookup(self, doc, root):
...         if root.tag == "sometag":
...             return MyElementClass
...         else:
...             for child in root:
...                 if child.tag == "someothertag":
...                     return MyElementClass
...         # delegate to default
...         return None
```

Note that the API of the Element objects is not complete. It is purely read-only and does not support all features of the normal `lxml.etree` API (such as XPath, extended slicing or some iteration methods).

Also, you cannot wrap such a read-only Element in an `ElementTree`, and you must take care not to keep a reference to them outside of the `lookup()` method.

See [http://codespeak.net/lxml/element\\_classes.html](http://codespeak.net/lxml/element_classes.html) **Version:** 2.0.10-60473

### B.17.1 Class `PythonElementClassLookup`



`PythonElementClassLookup(self, fallback=None)` Element class lookup based on a subclass method.

To use it, inherit from this class and override the `lookup` method to lookup the element

class for a node:

```
lookup(self, document, node_proxy)
```

The first argument is the opaque document instance that contains the Element. The second arguments is a lightweight Element proxy implementation that is only valid during the lookup. Do not try to keep a reference to it. Once the lookup is done, the proxy will be invalid.

If you return None from this method, the fallback will be called.

## Methods

```
__new__(T, S, ...)
```

### Return Value

a new object with type S, a subtype of T

Overrides: object.\_\_new\_\_

```
lookup(self, doc, element)
```

Override this method to implement your own lookup scheme.

*Inherited from [lxml.etree.FallbackElementClassLookup](#) (Section [B.6.28](#))*

```
__init__(), setFallback(), set_fallback()
```

*Inherited from [object](#)*

```
__delattr__(), __getattr__(), __hash__(), __reduce__(), __reduce_ex__(), __repr__(),
__setattr__(), __str__()
```

## Properties

Name	Description
<i>Inherited from <a href="#">lxml.etree.FallbackElementClassLookup</a> (Section <a href="#">B.6.28</a>)</i> fallback	
<i>Inherited from <a href="#">object</a></i> __class__	

## B.18 Module `lxml.sax`

SAX-based adapter to copy trees from/to the Python standard library.

Use the `ElementTreeContentHandler` class to build an `ElementTree` from SAX events.

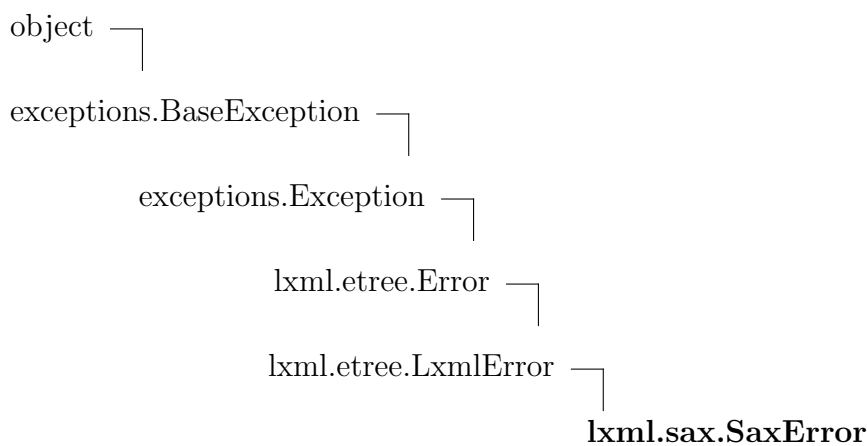
Use the `ElementTreeProducer` class or the `saxify()` function to fire the SAX events of an `ElementTree` against a SAX `ContentHandler`.

See <http://codespeak.net/lxml/sax.html>

### B.18.1 Functions

<p><code>saxify(<i>element_or_tree</i>, <i>content_handler</i>)</code></p> <hr/> <p>One-shot helper to generate SAX events from an XML tree and fire them against a SAX <code>ContentHandler</code>.</p>
--

### B.18.2 Class `SaxError`



General SAX error.

#### Methods

*Inherited from `lxml.etree.LxmlError` (Section [B.6.30](#))*

`__init__()`

*Inherited from `exceptions.Exception`*

`__new__()`

**Inherited from `exceptions.BaseException`**

`__delattr__()`, `__getattr__()`, `__getitem__()`, `__getslice__()`, `__reduce__()`, `__repr__()`,  
`__setattr__()`, `__setstate__()`, `__str__()`

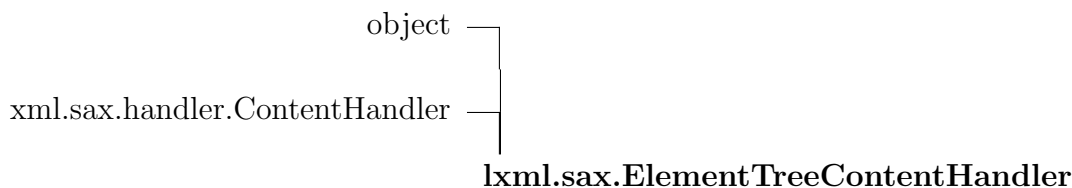
**Inherited from `object`**

`__hash__()`, `__reduce_ex__()`

**Properties**

Name	Description
<i>Inherited from <code>exceptions.BaseException</code></i> args, message	
<i>Inherited from <code>object</code></i> __class__	

**B.18.3 Class `ElementTreeContentHandler`**



Build an `lxml ElementTree` from SAX events.

**Methods**

<p><code>__init__(self, makeelement=None)</code></p> <p><code>x.__init__(...)</code> initializes x; see <code>x.__class__.__doc__</code> for signature Overrides:  <code>xml.sax.handler.ContentHandler.__init__</code></p>
---

**setDocumentLocator**(*self*, *locator*)

Called by the parser to give the application a locator for locating the origin of document events.

SAX parsers are strongly encouraged (though not absolutely required) to supply a locator: if it does so, it must supply the locator to the application by invoking this method before invoking any of the other methods in the `DocumentHandler` interface.

The locator allows the application to determine the end position of any document-related event, even if the parser is not reporting an error. Typically, the application will use this information for reporting its own errors (such as character content that does not match an application's business rules). The information returned by the locator is probably not sufficient for use with a search engine.

Note that the locator will return correct information only during the invocation of the events in this interface. The application should not attempt to use it at any other time. Overrides:  
`xml.sax.handler.ContentHandler.setDocumentLocator` extit(inherited documentation)

**startDocument**(*self*)

Receive notification of the beginning of a document.

The SAX parser will invoke this method only once, before any other methods in this interface or in `DTDHandler` (except for `setDocumentLocator`). Overrides:  
`xml.sax.handler.ContentHandler.startDocument` extit(inherited documentation)

**endDocument**(*self*)

Receive notification of the end of a document.

The SAX parser will invoke this method only once, and it will be the last method invoked during the parse. The parser shall not invoke this method until it has either abandoned parsing (because of an unrecoverable error) or reached the end of input. Overrides:  
`xml.sax.handler.ContentHandler.endDocument` extit(inherited documentation)

**startPrefixMapping**(*self*, *prefix*, *uri*)

Begin the scope of a prefix-URI Namespace mapping.

The information from this event is not necessary for normal Namespace processing: the SAX XML reader will automatically replace prefixes for element and attribute names when the

<http://xml.org/sax/features/namespaces> feature is true (the default).

There are cases, however, when applications need to use prefixes in character data or in attribute values, where they cannot safely be expanded automatically; the start/endPrefixMapping event supplies the information to the application to expand prefixes in those contexts itself, if necessary.

Note that start/endPrefixMapping events are not guaranteed to be properly nested relative to each-other: all startPrefixMapping events will occur before the corresponding startElement event, and all endPrefixMapping events will occur after the corresponding endElement event, but their order is not guaranteed. Overrides: `xml.sax.handler.ContentHandler.startPrefixMapping` `exitit`(inherited documentation)

**endPrefixMapping**(*self*, *prefix*)

End the scope of a prefix-URI mapping.

See startPrefixMapping for details. This event will always occur after the corresponding endElement event, but the order of endPrefixMapping events is not otherwise guaranteed. Overrides:

`xml.sax.handler.ContentHandler.endPrefixMapping` `exitit`(inherited documentation)

**startElementNS**(*self*, *ns\_name*, *qname*, *attributes=*None)

Signals the start of an element in namespace mode.

The name parameter contains the name of the element type as a (uri, localname) tuple, the qname parameter the raw XML 1.0 name used in the source document, and the attrs parameter holds an instance of the Attributes class containing the attributes of the element.

The uri part of the name tuple is None for elements which have no namespace. Overrides: `xml.sax.handler.ContentHandler.startElementNS` `exitit`(inherited documentation)

**processingInstruction**(*self*, *target*, *data*)

Receive notification of a processing instruction.

The Parser will invoke this method once for each processing instruction found: note that processing instructions may occur before or after the main document element.

A SAX parser should never report an XML declaration (XML 1.0, section 2.8) or a text declaration (XML 1.0, section 4.3.1) using this method.

Overrides: `xml.sax.handler.ContentHandler.processingInstruction`  
`exitit`(inherited documentation)

**endElementNS**(*self*, *ns\_name*, *qname*)

Signals the end of an element in namespace mode.

The name parameter contains the name of the element type, just as with the `startElementNS` event. Overrides:

`xml.sax.handler.ContentHandler.endElementNS` `exitit`(inherited documentation)

**startElement**(*self*, *name*, *attributes=*`None`)

Signals the start of an element in non-namespace mode.

The name parameter contains the raw XML 1.0 name of the element type as a string and the `attrs` parameter holds an instance of the `Attributes` class containing the attributes of the element. Overrides:

`xml.sax.handler.ContentHandler.startElement` `exitit`(inherited documentation)

**endElement**(*self*, *name*)

Signals the end of an element in non-namespace mode.

The name parameter contains the name of the element type, just as with the `startElement` event. Overrides:

`xml.sax.handler.ContentHandler.endElement` `exitit`(inherited documentation)

**characters**(*self*, *data*)

Receive notification of character data.

The Parser will call this method to report each chunk of character data. SAX parsers may return all contiguous character data in a single chunk, or they may split it into several chunks; however, all of the characters in any single event must come from the same external entity so that the Locator provides useful information. Overrides:

`xml.sax.handler.ContentHandler.characters` extit(inherited documentation)

**ignorableWhitespace**(*self*, *data*)

Receive notification of character data.

The Parser will call this method to report each chunk of character data. SAX parsers may return all contiguous character data in a single chunk, or they may split it into several chunks; however, all of the characters in any single event must come from the same external entity so that the Locator provides useful information. Overrides:

`xml.sax.handler.ContentHandler.ignorableWhitespace` extit(inherited documentation)

***Inherited from object***

`__delattr__()`, `__getattr__()`, `__hash__()`, `__new__()`, `__reduce__()`, `__reduce_ex__()`, `__repr__()`, `__setattr__()`, `__str__()`

***Inherited from xml.sax.handler.ContentHandler***

`skippedEntity()`

**Properties**

Name	Description
<code>etree</code>	Contains the generated <code>ElementTree</code> after parsing is finished.
<i>Inherited from object</i>	
<code>__class__</code>	

**B.18.4 Class `ElementTreeProducer`**

Produces SAX events for an element and children.

**Methods**

<code>__init__(self, element_or_tree, content_handler)</code>
---

<code>x.__init__(...)</code> initializes x; see <code>x.__class__.__doc__</code> for signature Overrides: <code>object.__init__</code> extit(inherited documentation)
---

<code>saxify(self)</code>
---------------------------

***Inherited from object***

`__delattr__()`, `__getattr__()`, `__hash__()`, `__new__()`, `__reduce__()`, `__reduce_ex__()`,  
`__repr__()`, `__setattr__()`, `__str__()`

**Properties**

Name	Description
<i>Inherited from object</i>	
<code>__class__</code>	

## **B.19 Module lxml.usedoctest**

Doctest module for XML comparison.

Usage:

```
>>> import lxml.usedoctest
>>> # now do your XML doctests ...
```

See `lxml.doctestcompare`