lxml

2010-03-02

Contents

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.

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.

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 (see EuroPython 2008 talk)
  - 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.

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 2.2.6, released 2010-03-02 (changes for 2.2.6). 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.

Mailing list

Questions? Suggestions? Code to contribute? We have a mailing list.

You can search the archive with Gmane or Google.

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)!

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.

Old Versions

See the web sites of lxml 1.3, 2.0, 2.1 and the current in-development version.

- lxml 2.2.5, released 2010-02-28 (changes for 2.2.5)
- lxml 2.2.4, released 2009-11-11 (changes for 2.2.4)
- lxml 2.2.3, released 2009-10-30 (changes for 2.2.3)
- lxml 2.2.2, released 2009-06-21 (changes for 2.2.2)
- lxml 2.2.1, released 2009-06-02 (changes for 2.2.1)
- lxml 2.2, released 2009-03-21 (changes for 2.2)
- lxml 2.2beta4, released 2009-02-27 (changes for 2.2beta4)
- lxml 2.2beta3, released 2009-02-17 (changes for 2.2beta3)
- lxml 2.2beta2, released 2009-01-25 (changes for 2.2beta2)
- lxml 2.2beta1, released 2008-12-12 (changes for 2.2beta1)
- lxml 2.2alpha1, released 2008-11-23 (changes for 2.2alpha1)
- lxml 2.1.5, released 2009-01-06 (changes for 2.1.5)
- lxml 2.1.4, released 2008-12-12 (changes for 2.1.4)
- lxml 2.1.3, released 2008-11-17 (changes for 2.1.3)

- lxml 2.1.2, released 2008-09-05 (changes for 2.1.2)
- lxml 2.1.1, released 2008-07-24 (changes for 2.1.1)
- lxml 2.1, released 2008-07-09 (changes for 2.1)
- lxml 2.0.11, released 2008-12-12 (changes for 2.0.11)
- lxml 2.0.10, released 2008-11-17 (changes for 2.0.10)
- lxml 2.0.9, released 2008-09-05 (changes for 2.0.9)
- lxml 2.0.8, released 2008-07-24 (changes for 2.0.8)
- lxml 2.0.7, released 2008-06-20 (changes for 2.0.7)
- lxml 2.0.6, released 2008-05-31 (changes for 2.0.6)
- lxml 2.0.5, released 2008-05-01 (changes for 2.0.5)
- lxml 2.0.4, released 2008-04-14 (changes for 2.0.4)
- lxml 2.0.3, released 2008-03-26 (changes for 2.0.3)
- lxml 2.0.2, released 2008-02-22 (changes for 2.0.2)
- lxml 2.0.1, released 2008-02-13 (changes for 2.0.1)
- lxml 2.0, released 2008-02-01 (changes for 2.0)
- lxml 1.3.6, released 2007-10-29 (changes for 1.3.6)
- lxml 1.3.5, released 2007-10-22 (changes for 1.3.5)
- lxml 1.3.4, released 2007-08-30 (changes for 1.3.4)
- lxml 1.3.3, released 2007-07-26 (changes for 1.3.3)
- lxml 1.3.2, released 2007-07-03 (changes for 1.3.2)
- lxml 1.3.1, released 2007-07-02 (changes for 1.3.1)
- lxml 1.3, released 2007-06-24 (changes for 1.3)
- lxml 1.2.1, released 2007-02-27 (changes for 1.2.1)
- lxml 1.2, released 2007-02-20 (changes for 1.2)
- lxml 1.1.2, released 2006-10-30 (changes for 1.1.2)
- lxml 1.1.1, released 2006-09-21 (changes for 1.1.1)
- lxml 1.1, released 2006-09-13 (changes for 1.1)
- lxml 1.0.4, released 2006-09-09 (changes for 1.0.4)
- lxml 1.0.3, released 2006-08-08 (changes for 1.0.3)
- lxml 1.0.2, released 2006-06-27 (changes for 1.0.2)
- lxml 1.0.1, released 2006-06-09 (changes for 1.0.1)
- lxml 1.0, released 2006-06-01 (changes for 1.0)
- lxml 0.9.2, released 2006-05-10 (changes for 0.9.2)

- lxml 0.9.1, released 2006-03-30 (changes for 0.9.1)
- lxml 0.9, released 2006-03-20 (changes for 0.9)
- lxml 0.8, released 2005-11-03 (changes for 0.8)
- lxml 0.7, released 2005-06-15 (changes for 0.7)
- lxml 0.6, released 2005-05-14 (changes for 0.6)
- lxml 0.5.1, released 2005-04-09 (changes for 0.5.1)
- lxml 0.5, released 2005-04-08

Chapter 2

Why lxml?

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.

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

For special installation instructions regarding MS Windows and MacOS-X, see below.

Requirements

You need Python 2.3 or later.

Unless you are using a static binary distribution (e.g. a Windows binary egg from PyPI), you need to install libxml2 and libxslt, in particular:

- libxml 2.6.21 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. We recommend libxml2 2.7.2 or later.

- 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. XML Schema support is also still worked on in libxml2, so newer versions will give you better complience with the W3C spec.

Installation

Get the easy_install tool and run the following as super-user (or administrator):

```
easy_install lxml
```

- On **MS Windows**, the above will install the binary builds that we provide. If there is no binary build of the latest release yet, please search PyPI for the last release that has them and pass that version to `easy_install` like this:

  ```
  easy_install lxml==2.2.2
  ```

- On **Linux** (and most other well-behaved operating systems), `easy_install` will manage to build the source distribution as long as libxml2 and libxslt are properly installed, including development packages, i.e. header files, etc. Use your package management tool to look for packages like `libxml2-dev` or `libxslt-devel` if the build fails, and make sure they are installed.

- On **MacOS-X**, use the following to build the source distribution, and make sure you have a working Internet connection, as this will download libxml2 and libxslt in order to build them:

```
STATIC_DEPS=true easy_install lxml
```

Installation in ActivePython

ActiveState provides ready-made lxml builds for different platforms in its package repository for the PyPM package manager. PyPM is similar to apt-get in that there is a repository of automaticaly prebuilt packages for Windows, Mac and Linux.

To install lxml in ActivePython, type the following on one of these operating systems:

```
$ pypm install lxml
```

To test the installation, try:

```
$ python -c "import lxml; print lxml.__file__"
```

This should show you the directory where the package was installed.

Building lxml from sources

If you want to build lxml from SVN you should read how to build lxml from source (or the file `doc/build.txt` in 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.

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 PyPI.

Unless you know what you are doing, this means: *do not install libxml2 or libxslt if you use a binary build of lxml.* Just use `easy_install` by following the installation instructions above.

Only 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.

MacOS-X

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

If you want to use a more recent lxml release, you may have to build it yourself. Apple doesn't help here, as the system libraries of libxml2 and libxslt installed under MacOS-X are horribly outdated, and updating them is everything but easy. In any case, you cannot run lxml 2.x with the system provided libraries, so you have to use newer libraries.

Luckily, lxml's `setup.py` script has built-in support for building and integrating these libraries statically during the build. Please read the MacOS-X build instructions.

A number of users also reported success with updated libraries (e.g. using fink or macports), but needed to set the runtime environment variable `DYLD_LIBRARY_PATH` to the directory where fink keeps the libraries. In any case, this method is easy to get wrong and everything but safe. Unless you know what you are doing, follow the static build instructions above.

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.

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.

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.

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.

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.clearErrorLog()`, use `etree.clear_error_log()`
  - `etree.useGlobalPythonLog()`, use `etree.use_global_python_log()`
  - `etree.ElementClassLookup.setFallback()`, use `etree.ElementClassLookup.set_fallback()`
  - `etree.getDefaultParser()`, use `etree.get_default_parser()`
  - `etree.setDefaultParser()`, use `etree.set_default_parser()`
  - `etree.setElementClassLookup()`, use `etree.set_element_class_lookup()`
  - `XMLParser.setElementClassLookup()`, use `.set_element_class_lookup()`
  - `HTMLParser.setElementClassLookup()`, use `.set_element_class_lookup()`

    Note that `parser.setElementClassLookup()` has not been removed yet, although `parser.set_element_clas` should be used instead.
  - `xpath_evaluator.registerNamespace()`, use `xpath_evaluator.register_namespace()`
  - `xpath_evaluator.registerNamespaces()`, use `xpath_evaluator.register_namespaces()`
  - `objectify.setPytypeAttributeTag`, use `objectify.set_pytype_attribute_tag`
  - `objectify.setDefaultParser()`, use `objectify.set_default_parser()`
- The `.getiterator()` method on Elements and ElementTrees was renamed to `.iter()` to follow ElementTree 1.3.

New modules

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

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
```

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.

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

lxml.etree is a very fast XML library. Most of this is due to the speed of libxml2, e.g. the parser and serialiser, or the XPath engine. Other areas of lxml were specifically written for high performance in high-level operations, such as the tree iterators.

On the other hand, the simplicity of lxml sometimes hides internal operations that are more costly than the API suggests. If you are not aware of these cases, lxml may not always perform as you expect. A common example in the Python world is the Python list type. New users often expect it to be a linked list, while it actually is implemented as an array, which results in a completely different complexity for common operations.

Similarly, the tree model of libxml2 is more complex than what lxml's ElementTree API projects into Python space, so some operations may show unexpected performance. Rest assured that most lxml users will not notice this in real life, as lxml is very fast in absolute numbers. It is definitely 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.

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.

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 2.2 (with libxml2 2.7.3) to the February 2009 SVN versions of ElementTree (1.3alpha2) and cElementTree (1.0.6). They were run single-threaded on a 1.8GHz Intel Core Duo machine under Ubuntu Linux 8.10 (Intrepid). The C libraries were compiled with the same platform specific optimisation flags. The Python interpreter (2.6.1) was manually compiled for the platform. Note that many of the following ElementTree timings are therefore better then what a normal Python installation with the standard library (c)ElementTree modules would yield.

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.

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 (which is part of the standard library since Python 2.5), lxml is still more than 7 times as fast as the much improved ElementTree 1.3:

```
lxe: tostring_utf16  (SATR T1)   22.4042 msec/pass
cET: tostring_utf16  (SATR T1)  184.5090 msec/pass
ET : tostring_utf16  (SATR T1)  182.4350 msec/pass

lxe: tostring_utf16  (UATR T1)   23.1769 msec/pass
cET: tostring_utf16  (UATR T1)  188.6780 msec/pass
ET : tostring_utf16  (UATR T1)  186.7781 msec/pass

lxe: tostring_utf16  (S-TR T2)   21.8501 msec/pass
cET: tostring_utf16  (S-TR T2)  200.0139 msec/pass
ET : tostring_utf16  (S-TR T2)  190.8720 msec/pass
```

```
lxe: tostring_utf8   (S-TR T2)   17.1690 msec/pass
cET: tostring_utf8   (S-TR T2)  192.3709 msec/pass
ET : tostring_utf8   (S-TR T2)  189.7140 msec/pass

lxe: tostring_utf8   (U-TR T3)    4.9832 msec/pass
cET: tostring_utf8   (U-TR T3)   60.2911 msec/pass
ET : tostring_utf8   (U-TR T3)   57.8101 msec/pass
```

The same applies to plain text serialisation. Note that cElementTree does not currently support this, as it is a new feature in ET 1.3 and lxml.etree 2.0:

```
lxe: tostring_text_ascii     (S-TR T1)    4.3709 msec/pass
ET : tostring_text_ascii     (S-TR T1)   83.9939 msec/pass

lxe: tostring_text_ascii     (S-TR T3)    1.3590 msec/pass
ET : tostring_text_ascii     (S-TR T3)   26.6340 msec/pass

lxe: tostring_text_utf16     (S-TR T1)    6.2978 msec/pass
ET : tostring_text_utf16     (S-TR T1)   84.7399 msec/pass

lxe: tostring_text_utf16     (U-TR T1)    7.7510 msec/pass
ET : tostring_text_utf16     (U-TR T1)   79.9279 msec/pass
```

Unlike ElementTree, the `tostring()` function in lxml also supports serialisation to a Python unicode string object:

```
lxe: tostring_text_unicode   (S-TR T1)    4.6940 msec/pass
lxe: tostring_text_unicode   (U-TR T1)    6.3069 msec/pass
lxe: tostring_text_unicode   (S-TR T3)    1.3652 msec/pass
lxe: tostring_text_unicode   (U-TR T3)    2.0702 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)   50.0100 msec/pass
cET: parse_stringIO  (SAXR T1)   19.3238 msec/pass
ET : parse_stringIO  (SAXR T1)  318.2330 msec/pass

lxe: parse_stringIO  (S-XR T3)    6.1851 msec/pass
cET: parse_stringIO  (S-XR T3)    5.7080 msec/pass
ET : parse_stringIO  (S-XR T3)   83.5931 msec/pass

lxe: parse_stringIO  (UAXR T3)   34.4319 msec/pass
cET: parse_stringIO  (UAXR T3)   28.8520 msec/pass
ET : parse_stringIO  (UAXR T3)  164.5968 msec/pass
```

While about as fast for smaller documents, the expat parser allows cET to be up to 2 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)   57.8308 msec/pass
cET: iterparse_stringIO  (SAXR T1)   23.8140 msec/pass
ET : iterparse_stringIO  (SAXR T1)  349.5209 msec/pass

lxe: iterparse_stringIO  (UAXR T3)   37.2162 msec/pass
cET: iterparse_stringIO  (UAXR T3)   30.2329 msec/pass
ET : iterparse_stringIO  (UAXR T3)  171.4060 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)   60.2388 msec/pass
cET: write_utf8_parse_stringIO  (S-TR T1)  314.9750 msec/pass
ET : write_utf8_parse_stringIO  (S-TR T1)  616.4260 msec/pass

lxe: write_utf8_parse_stringIO  (UATR T2)   71.7540 msec/pass
cET: write_utf8_parse_stringIO  (UATR T2)  364.4099 msec/pass
ET : write_utf8_parse_stringIO  (UATR T2)  684.5109 msec/pass

lxe: write_utf8_parse_stringIO  (S-TR T3)   10.7441 msec/pass
cET: write_utf8_parse_stringIO  (S-TR T3)  103.3869 msec/pass
ET : write_utf8_parse_stringIO  (S-TR T3)  179.5921 msec/pass

lxe: write_utf8_parse_stringIO  (SATR T4)    1.1981 msec/pass
cET: write_utf8_parse_stringIO  (SATR T4)    7.0901 msec/pass
ET : write_utf8_parse_stringIO  (SATR T4)   10.4899 msec/pass
```

For applications that require a high parser throughput of large files, and that do little to no serialization, cET is the best choice. Also for iterparse applications that extract small amounts of data or aggregate information from large XML data sets that do not fit into memory. If it comes to round-trip performance, however, lxml tends to be multiple times faster in total. So, whenever the input documents are not considerably larger than the output, lxml is the clear winner.

Regarding HTML parsing, Ian Bicking has done some benchmarking on lxml's HTML parser, comparing it to a number of other famous HTML parser tools for Python. lxml wins this contest by quite a length. To give an idea, the numbers suggest that lxml.html can run a couple of parse-serialise cycles in the time that other tools need for parsing alone. The comparison even shows some very favourable results regarding memory consumption.

Liza Daly has written an article that presents a couple of tweaks to get the most out of lxml's parser for very large XML documents. She quite favourably positions `lxml.etree` as a tool for high-performance XML parsing.

Finally, xml.com has a couple of publications about XML parser performance. Farwick and Hafner have written two interesting articles that compare the parser of libxml2 to some major Java based XML parsers. One deals with event-driven parser performance, the other one presents benchmark results comparing DOM parsers. Both comparisons suggest that libxml2's parser performance is largely superiour to all commonly used Java parsers in almost all cases. Note that the C parser benchmark results are based on xmlbench, which uses a simpler setup for libxml2 than lxml does.

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 and full XPath support, 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: 0.0502 0.0572 0.0613 0.0494 0.0575 0.0615
     T2: 0.0602 0.0691 0.0747 0.0651 0.0745 0.0796
     T3: 0.0145 0.0157 0.0176 0.0392 0.0411 0.0415
     T4: 0.0003 0.0003 0.0003 0.0008 0.0008 0.0008
```

```
cET:        --      S-      U-      -A      SA      UA
     T1: 0.0092 0.0094 0.0094 0.0094 0.0096 0.0093
     T2: 0.0152 0.0151 0.0152 0.0156 0.0154 0.0154
     T3: 0.0079 0.0080 0.0079 0.0106 0.0107 0.0134
     T4: 0.0001 0.0001 0.0001 0.0001 0.0001 0.0001
ET :        --      S-      U-      -A      SA      UA
     T1: 0.1017 0.1715 0.1962 0.1080 0.2470 0.1049
     T2: 0.3130 0.3324 0.1130 0.3897 0.1158 0.4246
     T3: 0.0341 0.0323 0.0338 0.0358 0.3965 0.0359
     T4: 0.0006 0.0005 0.0006 0.0006 0.0007 0.0006
```

While lxml is still a lot faster than ET in most cases, cET can be up to five times faster than lxml here. One of the reasons is that lxml must encode incoming string data and tag names into UTF-8, and 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.

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)    0.0148 msec/pass
cET: root_list_children        (--TR T1)    0.0050 msec/pass
ET : root_list_children        (--TR T1)    0.0219 msec/pass

lxe: root_list_children        (--TR T2)    0.1719 msec/pass
cET: root_list_children        (--TR T2)    0.0260 msec/pass
ET : root_list_children        (--TR T2)    0.3390 msec/pass
```

This handicap is also visible when accessing single children:

```
lxe: first_child               (--TR T2)    0.1879 msec/pass
cET: first_child               (--TR T2)    0.1760 msec/pass
ET : first_child               (--TR T2)    0.8099 msec/pass

lxe: last_child                (--TR T1)    0.1910 msec/pass
cET: last_child                (--TR T1)    0.1872 msec/pass
ET : last_child                (--TR T1)    0.8099 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)    0.2189 msec/pass
cET: middle_child              (--TR T1)    0.1779 msec/pass
ET : middle_child              (--TR T1)    0.8030 msec/pass

lxe: middle_child              (--TR T2)    2.4071 msec/pass
cET: middle_child              (--TR T2)    0.1781 msec/pass
ET : middle_child              (--TR T2)    0.8039 msec/pass
```

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)    2.1949 msec/pass
cET: create_elements            (--TC T2)    0.1941 msec/pass
ET : create_elements            (--TC T2)    1.2760 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)    1.8370 msec/pass
cET: makeelement                (--TC T2)    0.3200 msec/pass
ET : makeelement                (--TC T2)    1.5380 msec/pass

lxe: create_subelements         (--TC T2)    1.6761 msec/pass
cET: create_subelements         (--TC T2)    0.2329 msec/pass
ET : create_subelements         (--TC T2)    3.0999 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.

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)    3.4299 msec/pass
cET: append_from_document       (--TR T1,T2)    0.2639 msec/pass
ET : append_from_document       (--TR T1,T2)    1.1489 msec/pass

lxe: append_from_document       (--TR T3,T4)    0.0429 msec/pass
cET: append_from_document       (--TR T3,T4)    0.0169 msec/pass
ET : append_from_document       (--TR T3,T4)    0.0780 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)    6.1119 msec/pass
cET: insert_from_document       (--TR T1,T2)    0.4129 msec/pass
ET : insert_from_document       (--TR T1,T2)    1.4160 msec/pass
```

or replacing the child slice by a newly created element:

```
lxe: replace_children_element  (--TC T1)    0.1769 msec/pass
cET: replace_children_element  (--TC T1)    0.0250 msec/pass
ET : replace_children_element  (--TC T1)    0.1538 msec/pass
```

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

```
lxe: replace_children          (--TC T1)    0.0169 msec/pass
cET: replace_children          (--TC T1)    0.0119 msec/pass
ET : replace_children          (--TC T1)    0.0758 msec/pass
```

While these numbers are too small to provide a major performance impact in practice, you should keep this difference in mind when you merge very large trees.

deepcopy

Deep copying a tree is fast in lxml:

```
lxe: deepcopy_all              (--TR T1)   10.0670 msec/pass
cET: deepcopy_all              (--TR T1)  115.8700 msec/pass
ET : deepcopy_all              (--TR T1)  866.8201 msec/pass

lxe: deepcopy_all              (-ATR T2)   12.4321 msec/pass
cET: deepcopy_all              (-ATR T2)  130.1000 msec/pass
ET : deepcopy_all              (-ATR T2)  901.1638 msec/pass

lxe: deepcopy_all              (S-TR T3)    2.6951 msec/pass
cET: deepcopy_all              (S-TR T3)   28.9950 msec/pass
ET : deepcopy_all              (S-TR T3)  218.7109 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.

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)    4.7209 msec/pass
cET: getiterator_all      (--TR T1)   45.8400 msec/pass
ET : getiterator_all      (--TR T1)   22.9480 msec/pass

lxe: getiterator_islice   (--TR T2)    0.0398 msec/pass
cET: getiterator_islice   (--TR T2)    0.3798 msec/pass
ET : getiterator_islice   (--TR T2)    0.1900 msec/pass

lxe: getiterator_tag      (--TR T2)    0.0160 msec/pass
cET: getiterator_tag      (--TR T2)    0.8149 msec/pass
ET : getiterator_tag      (--TR T2)    0.3560 msec/pass

lxe: getiterator_tag_all  (--TR T2)    0.6580 msec/pass
cET: getiterator_tag_all  (--TR T2)   46.3769 msec/pass
ET : getiterator_tag_all  (--TR T2)   20.3989 msec/pass
```

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

```
lxe: findall              (--TR T2)    6.7198 msec/pass
cET: findall              (--TR T2)   51.2750 msec/pass
ET : findall              (--TR T2)   26.9110 msec/pass
```

```
lxe: findall            (--TR T3)    1.4520 msec/pass
cET: findall            (--TR T3)   14.2760 msec/pass
ET : findall            (--TR T3)    8.4310 msec/pass

lxe: findall_tag        (--TR T2)    0.7401 msec/pass
cET: findall_tag        (--TR T2)   46.5961 msec/pass
ET : findall_tag        (--TR T2)   20.3760 msec/pass

lxe: findall_tag        (--TR T3)    0.3331 msec/pass
cET: findall_tag        (--TR T3)   11.5960 msec/pass
ET : findall_tag        (--TR T3)    5.4510 msec/pass
```

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

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)    1.5750 msec/pass
lxe: xpath_method       (--TC T2)   20.9570 msec/pass
lxe: xpath_method       (--TC T3)    0.1199 msec/pass
lxe: xpath_method       (--TC T4)    1.0121 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)    0.6301 msec/pass
lxe: xpath_class        (--TC T2)    2.6128 msec/pass
lxe: xpath_class        (--TC T3)    0.0498 msec/pass
lxe: xpath_class        (--TC T4)    0.1400 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)    0.2739 msec/pass
lxe: xpath_element      (--TR T2)   10.8800 msec/pass
lxe: xpath_element      (--TR T3)    0.0660 msec/pass
lxe: xpath_element      (--TR T4)    0.2739 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)    1.5399 msec/pass
lxe: xpath_class_repeat   (--TC T2)   20.5159 msec/pass
lxe: xpath_class_repeat   (--TC T3)    0.1178 msec/pass
lxe: xpath_class_repeat   (--TC T4)    0.9880 msec/pass
```

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 increadibly 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 developers to optimise iterparse in one of the next releases...

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.

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)    6.9990 msec/pass
lxe: attribute                  (--TR T2)   29.2060 msec/pass
lxe: attribute                  (--TR T4)    6.9048 msec/pass

lxe: objectpath                 (--TR T1)    3.5410 msec/pass
lxe: objectpath                 (--TR T2)   24.9801 msec/pass
lxe: objectpath                 (--TR T4)    3.5069 msec/pass

lxe: attributes_deep            (--TR T1)   16.9580 msec/pass
lxe: attributes_deep            (--TR T2)   39.8140 msec/pass
lxe: attributes_deep            (--TR T4)   16.9699 msec/pass

lxe: objectpath_deep            (--TR T1)    9.4180 msec/pass
lxe: objectpath_deep            (--TR T2)   31.7512 msec/pass
lxe: objectpath_deep            (--TR T4)    9.4421 msec/pass
```

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

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)    5.1420 msec/pass
lxe: attribute_cached           (--TR T2)   27.0739 msec/pass
lxe: attribute_cached           (--TR T4)    5.1429 msec/pass

lxe: attributes_deep_cached     (--TR T1)    7.0908 msec/pass
lxe: attributes_deep_cached     (--TR T2)   29.5591 msec/pass
```

```
lxe: attributes_deep_cached     (--TR T4)    7.1721 msec/pass

lxe: objectpath_deep_cached     (--TR T1)    2.2731 msec/pass
lxe: objectpath_deep_cached     (--TR T2)   23.1631 msec/pass
lxe: objectpath_deep_cached     (--TR T4)    2.3179 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.

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` keyword 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 correspondents to `pi.target + " " + pi.text` in lxml.etree.
- Because etree is built on top of libxml2, which is namespace prefix aware, etree preserves namespaces 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.

General Questions

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 documentation of the extended etree API also contains many examples for `lxml.etree`. Fredrik Lundh's element library contains a lot of nice recipes that show how to solve common tasks in ElementTree and 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. Liza Daly wrote a nice article about high-performance aspects when parsing large files with lxml.

Where can I find more documentation about lxml?

There is a lot of documentation on the web and also in the Python standard library documentation, as lxml implements the well-known ElementTree API and tries to follow its documentation as closely as possible. The recipes in Fredrik Lundh's element library are generally worth taking a look at. 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 source distribution and on the web page.

The generated API documentation is a comprehensive API reference for the lxml package.

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. The important ones are:

- XML 1.0
- HTML 4
- XML namespaces
- XML Schema 1.0
- XPath 1.0
- XInclude 1.0
- XSLT 1.0
- EXSLT
- XML catalogs
- canonical XML
- RelaxNG
- xml:id
- xml:base

Support for XML Schema is currently not 100% complete in libxml2, but is definitely very close to compliance. Schematron is supported, although not necessarily complete. libxml2 also supports loading documents through HTTP and FTP.

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. Many people who deploy Zope or Plone use it together with lxml. 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
- PyQuery, a query framework for XML/HTML, similar to jQuery for JavaScript
- Rambler, a meta search engine that aggregates different data sources
- rdfadict, 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
- zif.sedna, an XQuery based interface to the Sedna OpenSource XML database

And don't miss the quotes by our generally happy users, and other sites that link to lxml. As Liza Daly puts it: "Many software products come with the pick-two caveat, meaning that you must choose only two: speed, flexibility, or readability. When used carefully, lxml can provide all three."

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.

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.

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:

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

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

>>> root.tail = None
>>> print(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.

How can I find out if an Element is a comment or PI?

```
>>> root = etree.XML("<?my PI?><root><!-- empty --></root>")

>>> root.tag
'root'
>>> root.getprevious().tag is etree.PI
True
>>> root[0].tag is etree.Comment
True
```

How can I map an XML tree into a dict of dicts?

I'm glad you asked.

```
def recursive_dict(element):
     return element.tag, \
            dict(map(recursive_dict, element)) or element.text
```

Installation

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.

Where are the binary builds?

Sidnei da Silva regularly contributes Windows binaries for new releases. This is because two of the major problems of Microsoft Windows make it non-trivial for users to build lxml on this platform: the lack of a pre-installed standard compiler and the missing package management.

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

```
http://pypi.python.org/pypi/lxml/2.1.5
```

Apart from that, we generally do not provide binary builds of lxml, as most of the other operating systems out there can build lxml without problems (with the exception of MacOS-X), and the sheer mass of variations between platforms makes it futile to provide builds for everyone.

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.

Contributing

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.

How can I contribute?

If you find something that you would like lxml to do (or do better), then please tell us about it on the mailing list. Patches are always appreciated, especially when accompanied by unit tests and documentation (doctests would be great). See the `tests` subdirectories in the lxml source tree (below the `src` directory) and the ReST text files in the `doc` directory.

We also have a list of missing features that we would like to implement but didn't due to lack if time. If *you* find the time, patches are very welcome.

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 heared 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.
- enhance the web site. We put some work into making the web site usable, understandable and also easy to find, but there's always things that can be done better. You may notice that we are not top-ranked when searching the web for "Python and XML", so maybe you have an idea how to improve that.
- help with the tutorial. A tutorial is the most important stating 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.

Bugs

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 section on bug reporting to learn how to do that.

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.

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.

Remember: even if you see lxml appear in a crash stack trace, it is not necessarily lxml that *caused* the crash.

My application crashes on MacOS-X!

This was a common problem up to lxml 2.1.x. Since lxml 2.2, the only officially supported way to use it on this platform is through a static build against freshly downloaded versions of libxml2 and libxslt. See the build instructions for MacOS-X.

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:

```
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
```

If you can figure that the problem is not in lxml but in the underlying libxml2 or libxslt, you can ask right on the respective mailing lists, which may considerably reduce the time to find a fix or work-around. See the next question for some hints on how to do that.

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.

It is important that you always report the version of lxml, libxml2 and libxslt that you get from the code snippet above. If we do not know the library versions you are using, we will ask back, so it will take longer for you to get a helpful answer.

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

How do I know a bug is really in lxml and not in libxml2?

A large part of lxml's functionality is implemented by libxml2 and libxslt, so problems that you encounter may be in one or the other. Knowing the right place to ask will reduce the time it takes to fix the problem, or to find a work-around.

Both libxml2 and libxslt come with their own command line frontends, namely `xmllint` and `xsltproc`. If you encounter problems with XSLT processing for specific stylesheets or with validation for specific schemas, try to run the XSLT with `xsltproc` or the validation with `xmllint` respectively to find out if it fails there as well. If it does, please report directly to the mailing lists of the respective project, namely:

- libxml2 mailing list
- libxslt mailing list

On the other hand, everything that seems to be related to Python code, including custom resolvers, custom XPath functions, etc. is likely outside of the scope of libxml2/libxslt. If you encounter problems here or you are not sure where there the problem may come from, please ask on the lxml mailing list first.

In any case, a good explanation of the problem including some simple test code and some input data will help us (or the libxml2 developers) see and understand the problem, which largely increases your chance of getting help. See the question above for a few hints on what is helpful here.

Threading

Can I use threads to concurrently access the lxml API?

Short answer: yes, if you use lxml 2.2 and later.

Since version 1.1, 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 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 if you do not need any special configuration. The same applies to the XPath evaluators, which use an internal lock to protect their prepared evaluation contexts. It is therefore best to use separate evaluator instances in threads.

Warning: Before lxml 2.2, and especially before 2.1, there were various issues when moving subtrees between different threads, or when applying XSLT objects from one thread to trees parsed or modified in another. If you need code to run with older versions, 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

- do not modify these trees and do not move their elements to other trees, or
- 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)

Since lxml 2.2, even multi-thread pipelines are supported. However, note that it is more efficient to do all tree work inside one thread, than to let multiple threads work on a tree one after the other. This is because trees inherit state from the thread that created them, which must be maintained when the tree is modified inside another thread.

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 zero. 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 very selective XPath expressions and 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.

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

Possibly, yes. 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. You can also build libxml2 without pthread support (`--without-pthreads` option), which may add another bit of performance. Note that this will leave internal data structures entirely without thread protection, so make sure you really do not use lxml outside of the main application thread in this case.

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

Since later lxml 2.0 versions, you can do this. There is some overhead involved as the result document needs an additional cleanup traversal when the input document and/or the stylesheet were created in other threads. However, on a multi-processor machine, the gain of freeing the GIL easily covers this drawback.

If you need even the last bit of performance, consider keeping (a copy of) the stylesheet in thread-local storage, and try creating the input document(s) in the same thread. And do not forget to benchmark your code to see if the increased code complexity is really worth it.

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. Version 2.2 of lxml contains many improvements.

- 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. Please read the bugs section regarding MacOS-X in this FAQ.

- 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.

- look out for fancy XSLT stuff like foreign document access or passing in subtrees trough XSLT variables. This might or might not work, depending on your specific usage. Again, later versions of lxml and libxslt provide safer support here.

- try copying trees at suspicious places in your code and working with those instead of a tree shared between threads. Note that the copying must happen inside the target thread to be effective, not in the thread that created the tree. Serialising in one thread and parsing in another is also a simple (and fast) way of separating thread contexts.

- 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).

Note that most of these options will degrade performance and/or your code quality. If you are unsure what to do, please ask on the mailing list.

Parsing and Serialisation

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(filename, 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.

Note that the `remove_blank_text` option also uses a heuristic if it has no definite knowledge about the document's ignorable whitespace. It will keep blank text nodes that appear after non-blank text nodes at the same level. This is to prevent document-style XML from breaking.

If you want to be sure all blank text is removed, you have to use either a DTD to tell the parser which whitespace it can safely ignore, or remove the ignorable whitespace manually after parsing, e.g. by setting all tail text to None:

> `.. sourcecode:: python`
>
> **for element in root.iter():** element.tail = None

Fredrik Lundh also has a Python-level function for indenting XML by appending whitespace to tags. It can be found on his element library recipe page.

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.

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.

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.

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.

XPath and Document Traversal

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.

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.

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

You can traverse the document (`root.iter()`) 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.

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.

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>
```

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
```

Elements carry attributes

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

```
>>> root = etree.Element("root", interesting="totally")
>>> etree.tostring(root)
b'<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
```

Elements contain text

Elements can contain text:

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

>>> print(root.text)
TEXT

>>> etree.tostring(root)
b'<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"

>>> etree.tostring(html)
b'<html><body>TEXT</body></html>'

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

>>> br.tail = "TAIL"
>>> etree.tostring(html)
b'<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:

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

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

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
```

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("%s - %s" % (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("%s - %s" % (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("%s - %s" % (element.tag, element.text))
...     else:
...         print("SPECIAL: %s - %s" % (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("%s - %s" % (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.

Serialisation

Serialisation commonly uses the `tostring()` function that returns a string, or the `ElementTree.write()` method that writes to a file, a file-like object, or a URL (via FTP PUT or HTTP POST). Both calls 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>')

>>> etree.tostring(root)
b'<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 that pretty printing appends a newline at the end.

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>')

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

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

>>> etree.tostring(root, method='html')
b'<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>

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

As for XML serialisation, the default encoding for plain text serialisation is ASCII:

```
>>> br = root.find('.//br')
>>> br.tail = u'W\xf6rld'

>>> etree.tostring(root, method='text')  # doctest: +ELLIPSIS
Traceback (most recent call last):
  ...
UnicodeEncodeError: 'ascii' codec can't encode character u'\xf6' ...

>>> etree.tostring(root, method='text', encoding="UTF-8")
b'HelloW\xc3\xb6rld'
```

Here, serialising to a Python unicode string instead of a byte string might become handy. Just pass the `unicode` type as encoding:

```
>>> etree.tostring(root, encoding=unicode, method='text')
u'HelloW\xf6rld'
```

The W3C has a good article about the Unicode character set and character encodings.

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:

```
>>> 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 lxml 2.0. Before, the examples were serialised without DTD content, which made lxml loose DTD information in an input-output cycle.

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.

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
>>> etree.tostring(root)
b'<root>data</root>'
```

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
>>> etree.tostring(root)
b'<root>data</root>'
```

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)

>>> etree.tostring(tree)
b'<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
>>> etree.tostring(root)
b'<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.

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)

>>> etree.tostring(root)
b'<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

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

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

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 = [ b"<roo", b"t><", b"a/", b"><", b"/root>" ]
...     def read(self, requested_size):
...         try:
...             return self.data.pop(0)
...         except IndexError:
...             return b''

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

>>> etree.tostring(tree)
b'<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()

>>> etree.tostring(root)
b'<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()
>>> etree.tostring(root)
b'<root/>'
```

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 = []
...     close_count = 0
...     def start(self, tag, attrib):
...         self.events.append(("start", tag, attrib))
...     def close(self):
...         events, self.events = self.events, []
...         self.close_count += 1
...         return events

>>> parser_target = ParserTarget()

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

>>> print(parser_target.close_count)
1

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

You can reuse the parser and its target as often as you like, so you should take care that the `.close()` methods really resets the target to a usable state (also in the case of an error!).

```
>>> events = etree.fromstring('<root test="true"/>', parser)
>>> print(parser_target.close_count)
2
>>> events = etree.fromstring('<root test="true"/>', parser)
>>> print(parser_target.close_count)
3
>>> events = etree.fromstring('<root test="true"/>', parser)
>>> print(parser_target.close_count)
4

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

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

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

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 reservered 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 reservered 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.

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.

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
```

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.

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
>>> etree.tostring(tree)
b'<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.

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']
```

The most common way to traverse an XML tree is depth-first, which traverses the tree in document order. This is implemented by the `.iter()` method. While there is no dedicated method for breadth-first traversal, it is almost as simple if you use the `collections.deque` type from Python 2.4.

```
>>> root = etree.XML('<root><a><b/><c/></a><d><e/></d></root>')
>>> print(etree.tostring(root, pretty_print=True, encoding=unicode))
<root>
  <a>
    <b/>
    <c/>
  </a>
  <d>
    <e/>
  </d>
</root>

>>> queue = deque([root])
>>> while queue:
...     el = queue.popleft()  # pop next element
...     queue.extend(el)      # append its children
...     print(el.tag)
root
a
d
b
c
e
```

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

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 occured and "might have" lead to the problem from the error log copy attached to the exception:

```
>>> etree.clear_error_log()
>>> 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 mismatch: 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)
```

```
PARSER
>>> print(entry.type_name)
ERR_TAG_NAME_MISMATCH
>>> print(entry.filename)
<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)
PARSER
>>> print(entry.type_name)
ERR_TAG_NOT_FINISHED
>>> print(entry.filename)
<string>
```

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.

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>")
>>> etree.tostring(root)
b'<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")
b'<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)
b'<\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):
  ...
lxml.etree.XMLSyntaxError: ...
```

CDATA

By default, lxml's parser will strip CDATA sections from the tree and replace them by their plain text content. As real applications for CDATA are rare, this is the best way to deal with this issue.

However, in some cases, keeping CDATA sections or creating them in a document is required to adhere to existing XML language definitions. For these special cases, you can instruct the parser to leave CDATA sections in the document:

```
>>> parser = etree.XMLParser(strip_cdata=False)
>>> root = etree.XML('<root><![CDATA[test]]></root>', parser)
>>> root.text
'test'

>>> etree.tostring(root)
b'<root><![CDATA[test]]></root>'
```

Note how the `.text` property does not give any indication that the text content is wrapped by a CDATA section. If you want to make sure your data is wrapped by a CDATA block, you can use the `CDATA()` text wrapper:

```
>>> root.text = 'test'

>>> root.text
'test'
>>> etree.tostring(root)
b'<root>test</root>'

>>> root.text = etree.CDATA(root.text)

>>> root.text
'test'
>>> etree.tostring(root)
b'<root><![CDATA[test]]></root>'
```

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.

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/></a>')
>>> tree = etree.parse(f)
>>> f2 = StringIO()
>>> tree.write_c14n(f2)
>>> print(f2.getvalue().decode("utf-8"))
<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
```

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)
>>> etree.tostring(root)
b'<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))
>>> etree.tostring(tree.getroot())
b'<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")
```

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)
>>> etree.tostring(tree.getroot())
b'<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 (on by default)
- 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
- remove_pis - discard processing instructions
- strip_cdata - replace CDATA sections by normal text content (on by default)
- resolve_entities - replace entities by their text value (on by default)
- huge_tree - disable security restrictions and support very deep trees and very long text content (only affects libxml2 2.7+)
- compact - use compact storage for short text content (on by default)

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):
  ...
lxml.etree.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)
1
>>> print(error.column)
11
```

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 a valid HTML tree with all content it can manage to parse. It will not raise an exception on parser errors. 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)

>>> result = etree.tostring(tree.getroot(),
...                         pretty_print=True, method="html")
>>> print(result)
<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)
>>> result = etree.tostring(html, pretty_print=True, method="html")
>>> print(result)
<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.

Note that the result is a valid HTML tree, but it may not be a well-formed XML tree. For example, XML forbids double hyphens in comments, which the HTML parser will happily accept in recovery mode. Therefore, if your goal is to serialise an HTML document as an XML/XHTML document after parsing, you may have to apply some manual preprocessing first.

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
```

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 %s %s" % (tag, attrib))
...     def end(self, tag):
...         print("end %s" % tag)
...     def data(self, data):
...         print("data %r" % data)
...     def comment(self, text):
...         print("comment %s" % text)
...     def close(self):
...         print("close")
...         return "closed!"

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

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

```
>>> print(result)
closed!
```

It is important for the .close() method to reset the parser target to a usable state, so that you can reuse the parser as often as you like:

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

>>> print(result)
closed!
```

Note that the parser does *not* build a tree when using a parser target. The result of the parser run is whatever 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. An example for a parser target that builds a tree is the TreeBuilder.

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

>>> result = etree.XML("<element>some<!--comment-->text</element>",
...                    parser)

>>> print(result.tag)
element
>>> print(result[0].text)
comment
```

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.

In lxml.etree, you can use both interfaces to a parser at the same time: the parse() or XML() functions, and the feed parser interface. Both are independent and will not conflict (except if used in conjunction with a parser target object as described above).

To start parsing with a feed parser, just call its feed() method to feed it some data.

```
>>> 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 feeds will keep appending data, usually resulting in a non well-formed document and an unexpected parser error. So make sure you always close the parser after use, also 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 creation 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.

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 supported by ElementTree and lxml.etree are the strings 'start', 'end', 'start-ns' and 'end-ns'.

The 'start' and 'end' events represent opening and closing elements. They are accompanied by the respective Element instance. By default, only 'end' events are generated:

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

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

```
end: element
end: {http://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("%s: %s" % (action, elem.tag))
start: root
start: element
end: element
start: element
end: element
start: {http://testns/}empty-element
end: {http://testns/}empty-element
end: root
```

The 'start-ns' and 'end-ns' events notify about namespace declarations. They do not come with Elements. Instead, the value of the 'start-ns' event is a tuple `(prefix, namespaceURI)` that designates the beginning of a prefix-namespace mapping. The corresponding `end-ns` event does not have a value (None). It is common practice to use a list as namespace stack and pop the last entry on the 'end-ns' event.

```
>>> print(xml[:-1])
<root>
  <element key='value'>text</element>
  <element>text</element>tail
  <empty-element xmlns="http://testns/" />
</root>

>>> events = ("start", "end", "start-ns", "end-ns")
>>> context = etree.iterparse(StringIO(xml), events=events)
>>> for action, elem in context:
...     if action in ('start', 'end'):
...         print("%s: %s" % (action, elem.tag))
...     elif action == 'start-ns':
...         print("%s: %s" % (action, elem))
...     else:
...         print(action)
start: root
start: element
end: element
start: element
end: element
start-ns: ('', 'http://testns/')
start: {http://testns/}empty-element
end: {http://testns/}empty-element
end-ns
end: root
```

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("%s: %s" % (action, elem.tag))
end: element
end: element

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

Comments and PIs

As an extension over ElementTree, the `iterparse()` function in lxml.etree also supports the event types 'comment' and 'pi' for the respective XML structures.

```
>>> commented_xml = '''\
... <?some pi ?>
... <!-- a comment -->
... <root>
...   <element key='value'>text</element>
...   <!-- another comment -->
...   <element>text</element>tail
...   <empty-element xmlns="http://testns/" />
... </root>
... '''

>>> events = ("start", "end", "comment", "pi")
>>> context = etree.iterparse(StringIO(commented_xml), events=events)
>>> for action, elem in context:
...     if action in ('start', 'end'):
...         print("%s: %s" % (action, elem.tag))
...     elif action == 'pi':
...         print("%s: -%s=%s-" % (action, elem.target, elem.text))
...     else: # 'comment'
...         print("%s: -%s-" % (action, elem.text))
pi: -some=pi -
comment: - a comment -
start: root
start: element
end: element
comment: - another comment -
start: element
end: element
start: {http://testns/}empty-element
end: {http://testns/}empty-element
end: root
```

```
>>> print(context.root.tag)
root
```

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.

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("%s: %s" % (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("%s: %s" % (action, elem.tag))
start: element
end: element
start: element
end: element
```

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.

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)
b'<test> &#63697; + &#63698; </test>'

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

As an extension, lxml.etree recognises the unicode type as an argument to the encoding parameter 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
```

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):
lxml.etree.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.

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")
```

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):
  ...
lxml.etree.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.

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):
  ...
lxml.etree.DocumentInvalid: Element 'c': This element is not expected. Expected 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 expected. Expected i
```

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
```

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 XPath 1.0, XSLT 1.0 and the EXSLT extensions through libxml2 and libxslt in a standards compliant way.

The usual setup procedure:

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

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.

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!
```

Namespaces and prefixes

If your XPath expression uses namespace prefixes, you must define them in a prefix mapping. To this end, pass a dictionary to the **namespaces** keyword argument that maps 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'
```

The prefixes you choose here are not linked to the prefixes used inside the XML document. The document may define whatever prefixes it likes, including the empty prefix, without breaking the above code.

Note that XPath does not have a notion of a default namespace. The empty prefix is therefore undefined for XPath and cannot be used in namespace prefix mappings.

There is also an optional `extensions` argument which is used to define custom extension functions in Python that are local to this evaluation. The namespace prefixes that they use in the XPath expression must also be defined in the namespace prefix mapping.

XPath return values

The return value types 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.

There are certain cases where the smart string behaviour is undesirable. For example, it means that the tree will be kept alive by the string, which may have a considerable memory impact in the case that the string value is the only thing in the tree that is actually of interest. For these cases, you can deactivate the parental relationship using the keyword argument `smart_strings`.

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

>>> find_text = etree.XPath("//text()")
>>> text = find_text(root)[0]
>>> print(text)
TEXT
>>> print(text.getparent().text)
TEXT

>>> find_text = etree.XPath("//text()", smart_strings=False)
>>> text = find_text(root)[0]
>>> print(text)
TEXT
>>> hasattr(text, 'getparent')
```

```
        False
```

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
```

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.

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.

`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
```

Error handling

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

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

During evaluation, lxml will emit an XPathEvalError on errors:

```
>>> find = etree.XPath("//ns:a")
>>> find(root)
Traceback (most recent call last):
  ...
lxml.etree.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):
  ...
lxml.etree.XPathEvalError: Invalid predicate

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

>>> find = root.xpath("\\")
Traceback (most recent call last):
  ...
lxml.etree.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`.

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, XSLT extension elements and document resolvers. There is a separate section on controlling access to external documents and resources.

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
```

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:param name="a" />
...     <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 integer expression:

```
>>> result = transform(doc, a="5")
>>> str(result)
'<?xml version="1.0"?>\n<foo>5</foo>\n'
```

You can use any valid XPath expression as parameter value:

```
>>> result = transform(doc, a="/a/b/text()")
>>> str(result)
'<?xml version="1.0"?>\n<foo>Text</foo>\n'
```

Passing a string expression looks like this:

```
>>> result = transform(doc, a="'A'")
>>> str(result)
'<?xml version="1.0"?>\n<foo>A</foo>\n'
```

To pass a string that (potentially) contains quotes, you can use the `.strparam()` class method. Note that it does not escape the string. Instead, it returns an opaque object that keeps the string value.

```
>>> plain_string_value = etree.XSLT.strparam(
...                          """ It's "Monty Python" """)
>>> result = transform(doc, a=plain_string_value)
>>> str(result)
'<?xml version="1.0"?>\n<foo> It\'s "Monty Python" </foo>\n'
```

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'
```

Dealing with stylesheet complexity

Some applications require a larger set of rather diverse stylesheets. lxml.etree allows you to deal with this in a number of ways. Here are some ideas to try.

The most simple way to reduce the diversity is by using XSLT parameters that you pass at call time to configure the stylesheets. The `partial()` function in the `functools` module of Python 2.5 may come in handy here. It allows you to bind a set of keyword arguments (i.e. stylesheet parameters) to a reference of a callable stylesheet. The same works for instances of the `XPath()` evaluator, obviously.

You may also consider creating stylesheets programmatically. Just create an XSL tree, e.g. from a parsed template, and then add or replace parts as you see fit. Passing an XSL tree into the `XSLT()` constructor multiple times will create independent stylesheets, so later modifications of the tree will not be reflected in the already created stylesheets. This makes stylesheet generation very straight forward.

A third thing to remember is the support for custom extension functions and XSLT extension elements. Some things are much easier to express in XSLT than in Python, while for others it is the complete opposite. Finding the right mixture of Python code and XSL code can help a great deal in keeping applications well designed and maintainable.

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" average="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.

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.

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:

```
>>> 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
```

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
```

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(5),
...   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="int">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>
```

Namespace handling

During tag lookups, 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("{http://ns/}root")
>>> b = etree.SubElement(root, "{http://ns/}b")
>>> c = etree.SubElement(root, "{http://other/}c")

>>> print(root.b.tag)
{http://ns/}b
```

Note that the `SubElement()` factory of `lxml.etree` does not inherit any namespaces when creating a new subelement. Element creation must be explicit about the namespace, and is simplified through the E-factory as described above.

Lookups, however, inherit namespaces implicitly:

```
>>> print(root.b.tag)
{http://ns/}b

>>> print(root.c)
Traceback (most recent call last):
    ...
```

```
AttributeError: no such child: {http://ns/}c
```

To access an element in a different namespace than its parent, you can use getattr():

```
>>> c = getattr(root, "{http://other/}c")
>>> print(c.tag)
{http://other/}c
```

For convenience, there is also a quick way through item access:

```
>>> c = root["{http://other/}c"]
>>> print(c.tag)
{http://other/}c
```

The same approach must be used to access children with tag names that are not valid Python identifiers:

```
>>> el = etree.SubElement(root, "{http://ns/}tag-name")
>>> print(root["tag-name"].tag)
{http://ns/}tag-name

>>> new_el = objectify.Element("{http://ns/}new-element")
>>> el = etree.SubElement(new_el, "{http://ns/}child")
>>> el = etree.SubElement(new_el, "{http://ns/}child")
>>> el = etree.SubElement(new_el, "{http://ns/}child")

>>> root["tag-name"] = [ new_el, new_el ]
>>> print(len(root["tag-name"]))
2
>>> print(root["tag-name"].tag)
{http://ns/}tag-name

>>> print(len(root["tag-name"].child))
3
>>> print(root["tag-name"].child.tag)
{http://ns/}child
>>> print(root["tag-name"][1].child.tag)
{http://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
```

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):

```
>>> 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):
lxml.etree.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.

ObjectPath

For both convenience and speed, objectify supports its own path language, represented by the `ObjectPath` class:

```
>>> root = objectify.Element("{http://ns/}root")
>>> b1 = etree.SubElement(root, "{http://ns/}b")
>>> c  = etree.SubElement(b1,   "{http://ns/}c")
>>> b2 = etree.SubElement(root, "{http://ns/}b")
>>> d  = etree.SubElement(root, "{http://other/}d")

>>> path = objectify.ObjectPath("root.b.c")
>>> print(path)
root.b.c
>>> path.hasattr(root)
True
>>> print(path.find(root).tag)
{http://ns/}c
```

```
>>> find = objectify.ObjectPath("root.b.c")
>>> print(find(root).tag)
{http://ns/}c

>>> find = objectify.ObjectPath("root.{http://other/}d")
>>> print(find(root).tag)
{http://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 {http://ns/}root

>>> find = objectify.ObjectPath("root.b[1]")
>>> print(find(root).tag)
{http://ns/}b

>>> find = objectify.ObjectPath("root.{http://ns/}b[1]")
>>> print(find(root).tag)
{http://ns/}b
```

Apart from strings, ObjectPath also accepts lists of path segments:

```
>>> find = objectify.ObjectPath(['root', 'b', 'c'])
>>> print(find(root).tag)
{http://ns/}c

>>> find = objectify.ObjectPath(['root', '{http://ns/}b[1]'])
>>> print(find(root).tag)
{http://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)
{http://ns/}b

>>> find = objectify.ObjectPath(['', 'b[1]'])
>>> print(find(root).tag)
{http://ns/}b

>>> find = objectify.ObjectPath(".unknown[1]")
>>> print(find(root).tag)
Traceback (most recent call last):
  ...
AttributeError: no such child: {http://ns/}unknown

>>> find = objectify.ObjectPath(".{http://other/}unknown[1]")
>>> print(find(root).tag)
Traceback (most recent call last):
  ...
AttributeError: no such child: {http://other/}unknown
```

For convenience, a single dot represents the empty ObjectPath (identity):

```
>>> find = objectify.ObjectPath(".")
>>> print(find(root).tag)
{http://ns/}root
```

ObjectPath objects can be used to manipulate trees:

```
>>> root = objectify.Element("{http://ns/}root")

>>> path = objectify.ObjectPath(".some.child.{http://other/}unknown")
>>> path.hasattr(root)
False
>>> path.find(root)
Traceback (most recent call last):
  ...
AttributeError: no such child: {http://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["{http://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.

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.

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’
```

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.enable_recursive_str()

>>> 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.enable_recursive_str(False)
```

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 leafs. 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`.

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>
... """ % ns)

>>> 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'
```

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>
...      <i xsi:type="xsd:int"   >5</i>
...      <s xsi:type="xsd:string">5</s>
...    </root>
...    ''')
>>> print(objectify.dump(root))
root = None [ObjectifiedElement]
    d = 5.0 [FloatElement]
      * xsi:type = 'xsd:double'
    i = 5 [IntElement]
      * xsi:type = 'xsd:int'
    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:integer'
    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>
...   <i xsi:type="xsd:int"   >5</i>
...   <s xsi:type="xsd:string">5</s>
... </root>''')
>>> objectify.annotate(root)
>>> print(objectify.dump(root))
root = None [ObjectifiedElement]
    d = 5.0 [FloatElement]
      * xsi:type = 'xsd:double'
      * py:pytype = 'float'
    i = 5 [IntElement]
      * xsi:type = 'xsd:int'
      * py:pytype = 'int'
    s = '5' [StringElement]
      * xsi:type = 'xsd:string'
      * py:pytype = 'str'
```

```
>>> objectify.deannotate(root)
>>> print(objectify.dump(root))
root = None [ObjectifiedElement]
    d = 5 [IntElement]
    i = 5 [IntElement]
    s = 5 [IntElement]
```

You can control which type attributes should be de-annotated with the keyword arguments 'pytype' (default: True) and 'xsi' (default: True). `deannotate()` can also remove 'xsi:nil' attributes by setting 'xsi_nil=True' (default: False):

```
>>> 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>
...   <i xsi:type="xsd:int"   >5</i>
...   <s xsi:type="xsd:string">5</s>
...   <n xsi:nil="true"/>
... </root>''')
>>> objectify.annotate(root)
>>> print(objectify.dump(root))
root = None [ObjectifiedElement]
    d = 5.0 [FloatElement]
      * xsi:type = 'xsd:double'
      * py:pytype = 'float'
    i = 5 [IntElement]
      * xsi:type = 'xsd:int'
      * py:pytype = 'int'
    s = '5' [StringElement]
      * xsi:type = 'xsd:string'
      * py:pytype = 'str'
    n = None [NoneElement]
      * xsi:nil = 'true'
      * py:pytype = 'NoneType'
>>> objectify.deannotate(root, xsi_nil=True)
>>> print(objectify.dump(root))
root = None [ObjectifiedElement]
    d = 5 [IntElement]
    i = 5 [IntElement]
    s = 5 [IntElement]
    n = u'' [StringElement]
```

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="int")
>>> print(objectify.dump(root))
root = None [ObjectifiedElement]
    x = 5 [IntElement]
      * py:pytype = 'int'

>>> 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 = 5 [IntElement]
      * py:pytype = ’int’
      * 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/XMLSche
  <s xsi:type="xsd:string">5</s>
</root>
```

`DataElement()` uses a default nsmap to set these prefixes:

```
>>> el = objectify.DataElement(’5’, _xsi=’string’)
>>> namespaces = list(el.nsmap.items())
>>> namespaces.sort()
>>> for prefix, namespace in namespaces:
...     print("%s - %s" % (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’,
...           nsmap={’foo’: ’http://www.w3.org/2001/XMLSchema’})
>>> namespaces = list(el.nsmap.items())
>>> namespaces.sort()
>>> for prefix, namespace in namespaces:
...     print("%s - %s" % (prefix, namespace))
foo - http://www.w3.org/2001/XMLSchema
py - http://codespeak.net/lxml/objectify/pytype
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',
...          nsmap={'foo': 'http://www.w3.org/2001/XMLSchema',
...                 'myxsi': 'http://www.w3.org/2001/XMLSchema-instance'})
>>> namespaces = list(el.nsmap.items())
>>> namespaces.sort()
>>> for prefix, namespace in namespaces:
...     print("%s - %s" % (prefix, namespace))
foo - http://www.w3.org/2001/XMLSchema
myxsi - http://www.w3.org/2001/XMLSchema-instance
py - http://codespeak.net/lxml/objectify/pytype

>>> 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 = objectify.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/XMLSc

>>> 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/XMLSche
</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/XMLSche
</root>
```

Of course, it is discouraged to use different prefixes for one and the same namespace when building up an objectify tree.

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.

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.

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

Author: Ian Bicking

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.

Parsing HTML

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(strin 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.

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.

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.

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.

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.

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).

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.

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.

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.

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>
```

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']
```

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>
      <img src="evil!">
    </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>
    <img src="evil!">
  </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>
    <img src="evil!">
  </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.

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.

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.

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)
```

Examples

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.

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']
```

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`.

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.

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, just like lxml supports it based on the parser of libxml2. BeautifulSoup uses a different parsing approach. It is not a real HTML parser but uses regular expressions to dive through tag soup. It is therefore more forgiving in some cases and less good in others. It is not uncommon that lxml/libxml2 parses and fixes broken HTML better, but BeautifulSoup has superiour support for encoding detection. It very much depends on the input which parser works better.

To prevent users from having to choose their parser library in advance, lxml can interface to 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 into an `lxml.html` document, and `convert_tree()` to convert an existing BeautifulSoup tree into a list of top-level Elements.

Parsing with the soupparser

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`.

For a quick comparison, libxml2 2.6.32 parses the same tag soup as follows. The main difference is that libxml2 tries harder to adhere to the structure of an HTML document and moves misplaced tags where they (likely) belong. Note, however, that the result can vary between parser versions.

```
<html>
  <head>
    <meta/>
    <title>Hello</title>
  </head>
  <body>
    <p>Hi all</p>
    <p/>
  </body>
</html>
```

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 just serialise with the default encoding, which is 'US-ASCII'.

```
>>> tostring(body)
'<body>&#169;&#8364;-&#245;&#445;<p/></body>'

>>> tostring(body, method="html")
'<body>&#169;&#8364;-&#245;&#445;<p></p></body>'
```

Any other encoding will output the respective byte sequences.

```
>>> tostring(body, encoding="utf-8")
'<body>\xc2\xa9\xe2\x82\xac-\xc3\xb5\xc6\xbd<p/></body>'

>>> tostring(body, method="html", encoding="utf-8")
'<body>\xc2\xa9\xe2\x82\xac-\xc3\xb5\xc6\xbd<p></p></body>'

>>> tostring(body, encoding=unicode)
u'<body>\xa9\u20ac-\xf5\u01bd<p/></body>'

>>> tostring(body, method="html", encoding=unicode)
u'<body>\xa9\u20ac-\xf5\u01bd<p></p></body>'
```

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)
```

Using only the encoding detection

If you prefer a 'real' (and fast) HTML parser instead of the regular expression based one in BeautifulSoup, you can still benefit from BeautifulSoup's support for encoding detection in the `UnicodeDammit` class.

```
>>> from BeautifulSoup import UnicodeDammit

>>> def decode_html(html_string):
...     converted = UnicodeDammit(html_string, isHTML=True)
...     if not converted.unicode:
...         raise UnicodeDecodeError(
...             "Failed to detect encoding, tried [%s]",
...             ', '.join(converted.triedEncodings))
...     # print converted.originalEncoding
...     return converted.unicode

>>> root = lxml.html.fromstring(decode_html(tag_soup))
```

Chapter 17

html5lib Parser

html5lib is a Python package that implements the HTML5 parsing algorithm which is heavily influenced by current browsers and based on the WHATWG HTML5 specification.

lxml can benefit from the parsing capabilities of *html5lib* through the `lxml.html.html5parser` module. It provides a similar interface to the `lxml.html` module by providing `fromstring()`, `parse()`, `document_fromstring()`, `fragment_fromstring()` and `fragments_fromstring()` that work like the regular html parsing functions.

Differences to regular HTML parsing

There are a few differences in the returned tree to the regular HTML parsing functions from `lxml.html`. html5lib normalizes some elements and element structures to a common format. For example even if a tables does not have a *tbody* html5lib will inject one automatically:

```
>>> from lxml.html import tostring, html5parser
>>> tostring(html5parser.fromstring("<table><td>foo"))
'<table><tbody><tr><td>foo</td></tr></tbody></table>'
```

Also the parameters the functions accept are different.

Function Reference

`parse(filename_url_or_file)`: Parses the named file or url, or if the object has a `.read()` method, parses from that.

`document_fromstring(html, guess_charset=True)`: 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.

If a bytestring is passed and `guess_charset` is true the chardet library (if installed) will guess the charset if ambiguities exist.

`fragment_fromstring(string, create_parent=False, guess_charset=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>`. If `create_parent` is true the default parent tag (div) is used.

If a bytestring is passed and `guess_charset` is true the chardet library (if installed) will guess the charset if ambiguities exist.

`fragments_fromstring(string, no_leading_text=False, parser=None):` Returns a list of the elements found in the fragment. The first item in the list may be a string. 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.

If a bytestring is passed and `guess_charset` is true the chardet library (if installed) will guess the charset if ambiguities exist.

`fromstring(string):` Returns `document_fromstring` or `fragment_fromstring`, based on whether the string looks like a full document, or just a fragment.

Additionally all parsing functions accept an `parser` keyword argument that can be set to a custom parser instance. To create custom parsers you can subclass the `HTMLParser` and `XHTMLParser` from the same module. Note that these are the parser classes provided by html5lib.

Part III

Extending lxml

Chapter 18

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.

URI 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 = '<!DOCTYPE doc SYSTEM "MissingDTD.dtd"><doc>&myentity;</doc>'
>>> 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.

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):
  ...
lxml.etree.XSLTParseError: Cannot resolve URI honk:test

>>> transform = etree.XSLT(hoi_doc)
Traceback (most recent call last):
  ...
lxml.etree.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):
  ...
lxml.etree.XSLTApplyError: Cannot resolve URI hoi:test

>>> result = transform(hoi_doc)
Traceback (most recent call last):
  ...
lxml.etree.XSLTApplyError: Cannot resolve URI hoi:test

>>> result = transform(honk_doc)
Traceback (most recent call last):
  ...
lxml.etree.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)[:-1])
<?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)[:-1])
<?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.

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):
  ...
lxml.etree.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 19

Python extensions for XPath and XSLT

This document describes how to use Python extension functions in XPath and XSLT like this:

```
<xsl:value-of select="f:myPythonFunction(.//sometag)" />
```

and extension elements in XSLT as in the following example:

```
<xsl:template match="*">
    <my:python-extension>
        <some-content />
    </my:python-extension>
</xsl:template>
```

XPath Extension functions

Here is how an extension 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)
```

The FunctionNamespace

In order to use a function in XPath or 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
```

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.

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("%s: %s" % (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 = list(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')]
```

Evaluators and XSLT

Extension functions work for all ways of evaluating XPath expressions and for XSL transformations:

```
>>> e = etree.XPathEvaluator(doc)
>>> print(e('es:hello(local-name(/a))'))
Ola a

>>> namespaces = {'f' : 'http://mydomain.org/myfunctions'}
>>> e = etree.XPathEvaluator(doc, namespaces=namespaces)
>>> print(e('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:

```
>>> f = StringIO('<a xmlns="http://mydomain.org/myfunctions" />')
>>> ns_doc = etree.parse(f)
>>> e = etree.XPathEvaluator(ns_doc)
>>> e('/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.register_namespace(’foo’, ’http://mydomain.org/myfunctions’)
>>> e(’/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(’/foo:a’)
...
lxml.etree.XPathEvalError: Undefined namespace prefix
```

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(’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(’l:function1(string(b))’))
1Haegar
```

The optional second argument to `Extension` can either be 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('function1(function2(function3(string(b))))'))
123Haegar

>>> extensions = etree.Extension( ext_module, functions, ns=None )
>>> e = etree.XPathEvaluator(doc, extensions=extensions)
>>> print(e('function1(function2(function3(string(b))))'))
123Haegar

>>> extensions = etree.Extension(ext_module)
>>> e = etree.XPathEvaluator(doc, extensions=extensions)
>>> print(e('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('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('function1(l:function2(function3(string(b))))'))
123Haegar
```

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("float()")
1.7
>>> e("int()")
1.0
>>> int( e("int()") )
1
>>> e("bool()")
True
>>> e("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("new-node-set()/result")
>>> print([ t.text for t in r ])
['Alpha', 'Beta', 'Gamma', 'Delta']

>>> r = e("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'

>>> etree.tostring(r[0])
b'<results1><result>Alpha</result><result>Beta</result></results1>'

>>> etree.tostring(r[1])
b'<results2><result>Gamma</result><result>Delta</result><subresult/></results2>'

>>> etree.tostring(r[2])
b'<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("%s - %s" % (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.

XSLT extension elements

Just like the XPath extension functions described above, lxml supports custom extension *elements* in XSLT. This means, you can write XSLT code like this:

```
<xsl:template match="*">
    <my:python-extension>
        <some-content />
    </my:python-extension>
</xsl:template>
```

And then you can implement the element in Python like this:

```
>>> class MyExtElement(etree.XSLTExtension):
...     def execute(self, context, self_node, input_node, output_parent):
...         print("Hello from XSLT!")
...         output_parent.text = "I did it!"
...         # just copy own content input to output
...         output_parent.extend( list(self_node) )
```

The arguments passed to the `.execute()` method are

context The opaque evaluation context. You need this when calling back into the XSLT processor.

self_node A read-only Element object that represents the extension element in the stylesheet.

input_node The current context Element in the input document (also read-only).

output_parent The current insertion point in the output document. You can append elements or set the text value (not the tail). Apart from that, the Element is read-only.

Declaring extension elements

In XSLT, extension elements can be used like any other XSLT element, except that they must be declared as extensions using the standard XSLT `extension-element-prefixes` option:

```
>>> xslt_ext_tree = etree.XML('''
... <xsl:stylesheet version="1.0"
...     xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
...     xmlns:my="testns"
...     extension-element-prefixes="my">
...     <xsl:template match="/">
...         <foo><my:ext><child>XYZ</child></my:ext></foo>
...     </xsl:template>
...     <xsl:template match="child">
...         <CHILD>--xyz--</CHILD>
...     </xsl:template>
... </xsl:stylesheet>''')
```

To register the extension, add its namespace and name to the extension mapping of the XSLT object:

```
>>> my_extension = MyExtElement()
>>> extensions = { ('testns', 'ext') : my_extension }
>>> transform = etree.XSLT(xslt_ext_tree, extensions = extensions)
```

Note how we pass an instance here, not the class of the extension. Now we can run the transformation and see how our extension is called:

```
>>> root = etree.XML('<dummy/>')
>>> result = transform(root)
Hello from XSLT!
>>> str(result)
'<?xml version="1.0"?>\n<foo>I did it!<child>XYZ</child></foo>\n'
```

Applying XSL templates

XSLT extensions are a very powerful feature that allows you to interact directly with the XSLT processor. You have full read-only access to the input document and the stylesheet, and you can even call back into the XSLT processor to process templates. Here is an example that passes an Element into the `.apply_templates()` method of the `XSLTExtension` instance:

```
>>> class MyExtElement(etree.XSLTExtension):
...     def execute(self, context, self_node, input_node, output_parent):
...         child = self_node[0]
...         results = self.apply_templates(context, child)
...         output_parent.append(results[0])

>>> my_extension = MyExtElement()
>>> extensions = { ('testns', 'ext') : my_extension }
>>> transform = etree.XSLT(xslt_ext_tree, extensions = extensions)

>>> root = etree.XML('<dummy/>')
>>> result = transform(root)
>>> str(result)
'<?xml version="1.0"?>\n<foo><CHILD>--xyz--</CHILD></foo>\n'
```

Note how we applied the templates to a child of the extension element itself, i.e. to an element inside the stylesheet instead of an element of the input document.

Working with read-only elements

There is one important thing to keep in mind: all Elements that the `execute()` method gets to deal with are read-only Elements, so you cannot modify them. They also will not easily work in the API. For example, you cannot pass them to the `tostring()` function or wrap them in an `ElementTree`.

What you can do, however, is to deepcopy them to make them normal Elements, and then modify them using the normal etree API. So this will work:

```
>>> from copy import deepcopy
>>> class MyExtElement(etree.XSLTExtension):
...     def execute(self, context, self_node, input_node, output_parent):
...         child = deepcopy(self_node[0])
...         child.text = "NEW TEXT"
...         output_parent.append(child)

>>> my_extension = MyExtElement()
>>> extensions = { ('testns', 'ext') : my_extension }
>>> transform = etree.XSLT(xslt_ext_tree, extensions = extensions)

>>> root = etree.XML('<dummy/>')
>>> result = transform(root)
>>> str(result)
'<?xml version="1.0"?>\n<foo><child>NEW TEXT</child></foo>\n'
```

Chapter 20

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, only for a specific tag name in a specific namespace or even 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:

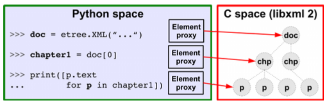
```
>>> from lxml import etree

>>> class honk(etree.ElementBase):
...    def honking(self):
...       return self.get('honking') == 'true'
...    honking = property(honking)
```

This defines a new Element class `honk` with a property `honking`.

The following document describes how you can make lxml.etree use these custom Element classes.

Background on Element proxies

Being based on libxml2, lxml.etree holds the entire XML tree in a C structure. To communicate with Python code, it creates Python proxy objects for the XML elements on demand.



```
>>> doc = etree.XML("...")

>>> chapter1 = doc[0]

>>> print([p.text
...          for p in chapter1])
```

The mapping between C elements and Python Element classes is completely configurable. When you ask lxml.etree for an Element by using its API, it will instantiate your classes for you. All you have to do is tell lxml which class to use for which kind of Element. This is done through a class lookup scheme, as described in the sections below.

Element initialization

There is one thing to know up front. Element classes *must not* have an `__init___` or `__new__` method. There should not be any internal state either, 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 proxy is created for them. Even worse, when the `__init__` method is called, the object is not even 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 once on Element class 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 C 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 throught 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.

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 the `honk` Element as a default element class for a parser works as follows:

```
>>> parser_lookup = etree.ElementDefaultClassLookup(element=honk)
>>> 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:

```
>>> el = etree.Element("root")
>>> print(isinstance(el, honk))
False
```

You should therefore avoid using this factory function in code that uses custom classes. The `makeelement()` method of parsers provides a simple replacement:

```
>>> el = parser.makeelement("root")
>>> print(isinstance(el, honk))
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_level_parser = etree.XMLParser()
>>> Element = module_level_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, honk))
False

>>> element = etree.XML("<test/>", parser)
>>> print(isinstance(element, honk))
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, honk))
True
>>> el = etree.SubElement(element, "subel")
>>> print(isinstance(el, honk))
True
```

For testing code in the Python interpreter and 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.

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 nice 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, honk))
False

>>> lookup = etree.ElementDefaultClassLookup(element=honk)
>>> parser.set_element_class_lookup(lookup)

>>> el = parser.makeelement("myelement")
>>> print(isinstance(el, honk))
```

```
True
>>> el.honking
False
>>> el = parser.makeelement("myelement", honking='true')
>>> etree.tostring(el)
b'<myelement honking="true"/>'
>>> el.honking
True
```

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=honk)
>>> lookup = etree.ElementNamespaceClassLookup(fallback)
>>> parser.set_element_class_lookup(lookup)
```

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 in a dictionary. It is set up as follows:

```
>>> id_class_mapping = {'1234' : honk} # maps attribute values to classes

>>> lookup = etree.AttributeBasedElementClassLookup(
...                                       'id', id_class_mapping)
>>> parser = etree.XMLParser()
>>> parser.set_element_class_lookup(lookup)
```

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)
```

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 honk # be a bit more selective here ...

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

The `.lookup()` method must return either 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", "entity"), the XML document of the element, or its namespace or tag name.

Tree based element class lookup in Python

Taking more elaborate decisions than allowed by the custom scheme is difficult to achieve in pure Python, as it results in a chicken-and-egg problem. It would require access to the tree - before the elements in the tree have been instantiated as Python Element proxies.

Luckily, there is a way to do this. The `PythonElementClassLookup` works similar to the custom lookup scheme:

```
>>> class MyLookup(etree.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.

Sidenote: this lookup scheme originally lived in a separate module called `lxml.pyclasslookup`.

Generating XML with custom classes

Up to lxml 2.1, you could not instantiate proxy classes yourself. Only lxml.etree could do that when creating an object representation of an existing XML element. Since lxml 2.2, however, instantiating this class will simply create a new Element:

```
>>> el = honk(honking = 'true')
>>> el.tag
'honk'
>>> el.honking
True
```

Note, however, that the proxy you create here will be garbage collected just like any other proxy. You can therefore not count on lxml.etree using the same class that you instantiated when you access this Element a second time after letting its reference go. You should therefore always use a corresponding class lookup scheme that returns your Element proxy classes for the elements that they create. The `ElementNamespaceClassLookup` is generally a good match.

You can use custom Element classes to quickly create XML fragments:

```
>>> class hale(etree.ElementBase): pass
>>> class bopp(etree.ElementBase): pass

>>> el = hale( "some ", honk(honking = 'true'), bopp, " text" )

>>> print(etree.tostring(el, encoding=unicode))
<hale>some <honk honking="true"/><bopp/> text</hale>
```

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'] = honk
```

If you have many Element classes declared in one module, and they are all named like the elements they create, you can simply use `namespace.update(vars())` at the end of your module to declare them automatically. The implementation is smart enough to ignore everything that is not an Element class.

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 `honk` Element 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)
...
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 # default Element for namespace

>>> class HonkElement(HonkNSElement):
...    def honking(self):
...       return self.get('honking') == 'true'
...    honking = property(honking)
>>> namespace['honk'] = HonkElement # Element for specific tag
```

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))
<class 'HonkElement'>
>>> print(type(honk_element[0]))
<class 'HonkNSElement'>

>>> print(honk_element.honking)
True
>>> print(honk_element.honk())
HONK

>>> print(honk_element[0].honk())
HONK
>>> print(honk_element[0].honking)
...
AttributeError: 'HonkNSElement' object has no attribute 'honking'
```

Chapter 21

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.

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())
b'<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.

Producing SAX events from an ElementTree or Element

Let's make a tree we can generate SAX events for:

```
>>> 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'
```

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 connot 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 22

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.

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(
     etree.DefaultElementClassLookup(element=NewElementClass))
```

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 23

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 must install before trying to build lxml. 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.

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 you want to be an lxml developer, then you do need a working Cython installation. You can use EasyInstall to install it:

```
easy_install Cython==0.11
```

lxml currently requires Cython 0.11, later release versions should work as well.

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 Subversion repository through the web, use your favourite SVN client to access it, or browse the Subversion history.

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`.

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, ...).

Building an egg

This is the procedure to make an lxml egg for your platform (assuming that you have setuptools installed):

- 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`

This will put the egg into the `dist` directory.

Building lxml on MacOS-X

Apple regularly ships new system releases with horribly outdated system libraries. This is specifically the case for libxml2 and libxslt, where the system provided versions are too old to build lxml.

While the Unix environment in MacOS-X makes it relatively easy to install Unix/Linux style package management tools and new software, it actually seems to be hard to get libraries set up for exclusive usage that MacOS-X ships in an older version. Alternative distributions (like macports) install their libraries in addition to the system libraries, but the compiler and the runtime loader on MacOS still sees the system libraries before the new libraries. This can lead to undebuggable crashes where the newer library seems to be loaded but the older system library is used.

Apple discourages static building against libraries, which would help working around this problem. Apple does not ship static library binaries with its system and several package management systems follow this decision. Therefore, building static binaries requires building the dependencies first. The `setup.py` script does this automatically when you call it like this:

```
python setup.py build --static-deps
```

This will download and build the latest versions of libxml2 and libxslt from the official FTP download site. If you want to use specific versions, or want to prevent any online access, you can download both `tar.gz` release files yourself, place them into a subdirectory `libs` in the lxml distribution, and call `setup.py` with the desired target versions like this:

```
python setup.py build --static-deps \
       --libxml2-version=2.7.3 \
       --libxslt-version=1.1.24 \

sudo python setup.py install
```

Instead of `build`, you can use any target, like `bdist_egg` if you want to use setuptools to build an installable egg.

Note that this also works with EasyInstall. Since you can't pass command line options in this case, you have to use an environment variable instead:

```
STATIC_DEPS=true easy_install lxml
```

Some machines may require an additional run with "sudo" to install the package into the Python package directory:

```
STATIC_DEPS=true sudo easy_install lxml
```

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. 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.

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 dependecies 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 24

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.

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.

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 descendence 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.

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()`.

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.

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 soure 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 safe 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.

readonlytree.pxi A separate read-only implementation of the Element API. This is used in places where non-intrusive access to a tree is required, such as the `PythonElementClassLookup` or XSLT extension elements.

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.

xmlid.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

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.

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. The implementation resides in the file lxml.objectify.pyx.

lxml.html

A specialised toolkit for HTML handling, based on lxml.etree. This is implemented in pure Python.

Chapter 25

Credits

Main contributors

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

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

David Sankel building statically on Windows

Marcin Kasperski PDF documentation generation

... 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!

Special thanks goes to:

- Daniel Veillard and the libxml2 project for a great XML library.
- Fredrik Lundh for ElementTree, its API, and the competition through cElementTree.
- 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

Appendix A

Changes

2.2.6 (2010-03-02)

Bugs fixed

- Fixed several Python 3 regressions by building with Cython 0.11.3.

2.2.5 (2010-02-28)

Features added

- Support for running XSLT extension elements on the input root node (e.g. in a template matching on "/").

Bugs fixed

- Crash in XPath evaluation when reading smart strings from a document other than the original context document.
- Support recent versions of html5lib by not requiring its `XHTMLParser` in `htmlparser.py` anymore.
- Manually instantiating the custom element classes in `lxml.objectify` could crash.
- Invalid XML text characters were not rejected by the API when they appeared in unicode strings directly after non-ASCII characters.
- lxml.html.open_http_urllib() did not work in Python 3.
- The functions `strip_tags()` and `strip_elements()` in `lxml.etree` did not remove all occurrences of a tag in all cases.
- Crash in XSLT extension elements when the XSLT context node is not an element.

2.2.4 (2009-11-11)

Bugs fixed

- Static build of libxml2/libxslt was broken.

2.2.3 (2009-10-30)

Features added

Bugs fixed

- The `resolve_entities` option did not work in the incremental feed parser.
- Looking up and deleting attributes without a namespace could hit a namespaced attribute of the same name instead.
- Late errors during calls to `SubElement()` (e.g. attribute related ones) could leave a partially initialised element in the tree.
- Modifying trees that contain parsed entity references could result in an infinite loop.
- `ObjectifiedElement.__setattr__` created an empty-string child element when the attribute value was rejected as a non-unicode/non-ascii string
- Syntax errors in `lxml.cssselect` could result in misleading error messages.
- Invalid syntax in CSS expressions could lead to an infinite loop in the parser of `lxml.cssselect`.
- CSS special character escapes were not properly handled in `lxml.cssselect`.
- CSS Unicode escapes were not properly decoded in `lxml.cssselect`.
- Select options in HTML forms that had no explicit `value` attribute were not handled correctly. The HTML standard dictates that their value is defined by their text content. This is now supported by lxml.html.
- XPath raised a TypeError when finding CDATA sections. This is now fully supported.
- Calling `help(lxml.objectify)` didn't work at the prompt.
- The `ElementMaker` in lxml.objectify no longer defines the default namespaces when annotation is disabled.
- Feed parser failed to honour the 'recover' option on parse errors.
- Diverting the error logging to Python's logging system was broken.

Other changes

2.2.2 (2009-06-21)

Features added

- New helper functions `strip_attributes()`, `strip_elements()`, `strip_tags()` in lxml.etree to remove attributes/subtrees/tags from a subtree.

Bugs fixed

- Namespace cleanup on subtree insertions could result in missing namespace declarations (and potentially crashes) if the element defining a namespace was deleted and the namespace was not used by the top element of the inserted subtree but only in deeper subtrees.
- Raising an exception from a parser target callback didn't always terminate the parser.
- Only {true, false, 1, 0} are accepted as the lexical representation for BoolElement ({True, False, T, F, t, f} not any more), restoring lxml <= 2.0 behaviour.

Other changes

2.2.1 (2009-06-02)

Features added

- Injecting default attributes into a document during XML Schema validation (also at parse time).
- Pass `huge_tree` parser option to disable parser security restrictions imposed by libxml2 2.7.

Bugs fixed

- The script for statically building libxml2 and libxslt didn't work in Py3.
- `XMLSchema()` also passes invalid schema documents on to libxml2 for parsing (which could lead to a crash before release 2.6.24).

Other changes

2.2 (2009-03-21)

Features added

- Support for `standalone` flag in XML declaration through `tree.docinfo.standalone` and by passing `standalone=True/False` on serialisation.

Bugs fixed

- Crash when parsing an XML Schema with external imports from a filename.

2.2beta4 (2009-02-27)

Features added

- Support strings and instantiable Element classes as child arguments to the constructor of custom Element classes.
- GZip compression support for serialisation to files and file-like objects.

Bugs fixed

- Deep-copying an ElementTree copied neither its sibling PIs and comments nor its internal/external DTD subsets.
- Soupparser failed on broken attributes without values.
- Crash in XSLT when overwriting an already defined attribute using `xsl:attribute`.
- Crash bug in exception handling code under Python 3. This was due to a problem in Cython, not lxml itself.
- `lxml.html.FormElement._name()` failed for non top-level forms.
- `TAG` special attribute in constructor of custom Element classes was evaluated incorrectly.

Other changes

- Official support for Python 3.0.1.
- `Element.findtext()` now returns an empty string instead of None for Elements without text content.

2.2beta3 (2009-02-17)

Features added

- `XSLT.strparam()` class method to wrap quoted string parameters that require escaping.

Bugs fixed

- Memory leak in XPath evaluators.
- Crash when parsing indented XML in one thread and merging it with other documents parsed in another thread.

- Setting the `base` attribute in `lxml.objectify` from a unicode string failed.
- Fixes following changes in Python 3.0.1.
- Minor fixes for Python 3.

Other changes

- The global error log (which is copied into the exception log) is now local to a thread, which fixes some race conditions.
- More robust error handling on serialisation.

2.2beta2 (2009-01-25)

Bugs fixed

- Potential memory leak on exception handling. This was due to a problem in Cython, not lxml itself.
- `iter_links` (and related link-rewriting functions) in `lxml.html` would interpret CSS like `url("link")` incorrectly (treating the quotation marks as part of the link).
- Failing import on systems that have an `io` module.

2.1.5 (2009-01-06)

Bugs fixed

- Potential memory leak on exception handling. This was due to a problem in Cython, not lxml itself.
- Failing import on systems that have an `io` module.

2.2beta1 (2008-12-12)

Features added

- Allow `lxml.html.diff.htmldiff` to accept Element objects, not just HTML strings.

Bugs fixed

- Crash when using an XPath evaluator in multiple threads.
- Fixed missing whitespace before `Link:...` in `lxml.html.diff`.

Other changes

- Export `lxml.html.parse`.

2.1.4 (2008-12-12)

Bugs fixed

- Crash when using an XPath evaluator in multiple threads.

2.0.11 (2008-12-12)

Bugs fixed

- Crash when using an XPath evaluator in multiple threads.

2.2alpha1 (2008-11-23)

Features added

- Support for XSLT result tree fragments in XPath/XSLT extension functions.
- QName objects have new properties `namespace` and `localname`.
- New options for exclusive C14N and C14N without comments.
- Instantiating a custom Element classes creates a new Element.

Bugs fixed

- XSLT didn't inherit the parse options of the input document.
- 0-bytes could slip through the API when used inside of Unicode strings.
- With `lxml.html.clean.autolink`, links with balanced parenthesis, that end in a parenthesis, will be linked in their entirety (typical with Wikipedia links).

Other changes

2.1.3 (2008-11-17)

Features added

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.
- Parser Unicode decoding errors could get swallowed by other exceptions.
- Name/import errors in some Python modules.
- Internal DTD subsets that did not specify a system or public ID were not serialised and did not appear in the docinfo property of ElementTrees.
- Fix a pre-Py3k warning when parsing from a gzip file in Py2.6.
- Test suite fixes for libxml2 2.7.
- Resolver.resolve_string() did not work for non-ASCII byte strings.
- Resolver.resolve_file() was broken.
- Overriding the parser encoding didn't work for many encodings.

Other changes

2.0.10 (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.

2.1.2 (2008-09-05)

Features added

- lxml.etree now tries to find the absolute path name of files when parsing from a file-like object. This helps custom resolvers when resolving relative URLs, as lixbml2 can prepend them with the path of the source document.

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.

Other changes

2.0.9 (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.

2.1.1 (2008-07-24)

Features added

Bugs fixed

- Crash when parsing XSLT stylesheets in a thread and using them in another.
- Encoding problem when including text with ElementInclude under Python 3.

Other changes

2.0.8 (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

2.1 (2008-07-09)

Features added

- Smart strings can be switched off in XPath (`smart_strings` keyword option).
- `lxml.html.rewrite_links()` strips links to work around documents with whitespace in URL attributes.

Bugs fixed

- Custom resolvers were not used for XMLSchema includes/imports and XInclude processing.
- CSS selector parser dropped remaining expression after a function with parameters.

Other changes

- `objectify.enableRecursiveStr()` was removed, use `objectify.enable_recursive_str()` instead
- Speed-up when running XSLTs on documents from other threads

2.0.7 (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.

2.1beta3 (2008-06-19)

Features added

- Major overhaul of `tools/xpathgrep.py` script.
- Pickling `ElementTree` objects in lxml.objectify.
- Support for parsing from file-like objects that return unicode strings.
- New function `etree.cleanup_namespaces(el)` that removes unused namespace declarations from a (sub)tree (experimental).
- XSLT results support the buffer protocol in Python 3.
- Polymorphic functions in `lxml.html` that accept either a tree or a parsable string will return either a UTF-8 encoded byte string, a unicode string or a tree, based on the type of the input. Previously, the result was always a byte string or a tree.
- Support for Python 2.6 and 3.0 beta.
- File name handling now uses a heuristic to convert between byte strings (usually filenames) and unicode strings (usually URLs).
- Parsing from a plain file object frees the GIL under Python 2.x.
- Running `iterparse()` on a plain file (or filename) frees the GIL on reading under Python 2.x.
- Conversion functions `html_to_xhtml()` and `xhtml_to_html()` in lxml.html (experimental).
- Most features in lxml.html work for XHTML namespaced tag names (experimental).

Bugs fixed

- `ElementTree.parse()` didn't handle target parser result.
- Crash in Element class lookup classes when the __init__() method of the super class is not called from Python subclasses.
- A number of problems related to unicode/byte string conversion of filenames and error messages were fixed.
- Building on MacOS-X now passes the "flat_namespace" option to the C compiler, which reportedly prevents build quirks and crashes on this platform.
- Windows build was broken.
- Rare crash when serialising to a file object with certain encodings.

Other changes

- Non-ASCII characters in attribute values are no longer escaped on serialisation.
- Passing non-ASCII byte strings or invalid unicode strings as .tag, namespaces, etc. will result in a ValueError instead of an AssertionError (just like the tag well-formedness check).
- Up to several times faster attribute access (i.e. tree traversal) in lxml.objectify.

2.0.6 (2008-05-31)

Features added

Bugs fixed

- Incorrect evaluation of `el.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.

2.1beta2 (2008-05-02)

Features added

- All parse functions in lxml.html take a `parser` keyword argument.
- lxml.html has a new parser class `XHTMLParser` and a module attribute `xhtml_parser` that provide XML parsers that are pre-configured for the lxml.html package.

Bugs fixed

- 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.
- Passing an nsmap when creating an Element will no longer strip redundantly defined namespace URIs. This prevented the definition of more than one prefix for a namespace on the same Element.

Other changes

- If the default namespace is redundantly defined with a prefix on the same Element, the prefix will now be preferred for subelements and attributes. This allows users to work around a problem in libxml2 where attributes from the default namespace could serialise without a prefix even when they appear on an Element with a different namespace (i.e. they would end up in the wrong namespace).

2.0.5 (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 >= 2.6.31.

Other changes

2.1beta1 (2008-04-15)

Features added

- Error logging in Schematron (requires libxml2 2.6.32 or later).
- Parser option `strip_cdata` for normalising or keeping CDATA sections. Defaults to `True` as before, thus replacing CDATA sections by their text content.
- `CDATA()` factory to wrap string content as CDATA section.

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 >= 2.6.31.
- lxml.etree accepted non well-formed namespace prefix names.

Other changes

- Major cleanup in internal `moveNodeToDocument()` function, which takes care of namespace cleanup when moving elements between different namespace contexts.
- New Elements created through the `makeelement()` method of an HTML parser or through lxml.html now end up in a new HTML document (doctype HTML 4.01 Transitional) instead of a generic XML document. This mostly impacts the serialisation and the availability of a DTD context.

2.0.4 (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

2.1alpha1 (2008-03-27)

Features added

- New event types 'comment' and 'pi' in `iterparse()`.
- `XSLTAccessControl` instances have a property `options` that returns a dict of access configuration options.
- Constant instances `DENY_ALL` and `DENY_WRITE` on `XSLTAccessControl` class.
- Extension elements for XSLT (experimental!)
- `Element.base` property returns the xml:base or HTML base URL of an Element.
- `docinfo.URL` property is writable.

Bugs fixed

- Default encoding for plain text serialisation was different from that of XML serialisation (UTF-8 instead of ASCII).

Other changes

- Minor API speed-ups.
- The benchmark suite now uses tail text in the trees, which makes the absolute numbers incomparable to previous results.
- Generating the HTML documentation now requires Pygments, which is used to enable syntax highlighting for the doctest examples.

Most long-time deprecated functions and methods were removed:

- `etree.clearErrorLog()`, use `etree.clear_error_log()`
- `etree.useGlobalPythonLog()`, use `etree.use_global_python_log()`

- `etree.ElementClassLookup.setFallback()`, use `etree.ElementClassLookup.set_fallback()`
- `etree.getDefaultParser()`, use `etree.get_default_parser()`
- `etree.setDefaultParser()`, use `etree.set_default_parser()`
- `etree.setElementClassLookup()`, use `etree.set_element_class_lookup()`

  Note that `parser.setElementClassLookup()` has not been removed yet, although `parser.set_element_class_lo` should be used instead.
- `xpath_evaluator.registerNamespace()`, use `xpath_evaluator.register_namespace()`
- `xpath_evaluator.registerNamespaces()`, use `xpath_evaluator.register_namespaces()`
- `objectify.setPytypeAttributeTag`, use `objectify.set_pytype_attribute_tag`
- `objectify.setDefaultParser()`, use `objectify.set_default_parser()`

2.0.3 (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.

2.0.2 (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.

2.0.1 (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.

2.0 (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.
- 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.
- 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;`.
- New properties `position` and `code` on ParseError exception (as in ET 1.3)
- 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(el)`.
- 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.
- 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()`.
- `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)
- 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

- Missing import in `lxml.html.clean`.
- Some Python 2.4-isms prevented lxml from building/running under Python 2.3.
- XPath on ElementTrees could crash when selecting the virtual root node of the ElementTree.
- Compilation `--without-threading` was buggy in alpha5/6.
- Memory leak in the `parse()` function.
- Minor bugs in XSLT error message formatting.
- Result document memory leak in target parser.
- 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).
- AttributeError in feed parser on parse errors
- XML feed parser setup problem
- Type annotation for unicode strings in `DataElement()`
- 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
- 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

- 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)
- Minor performance tweaks for Element instantiation and subelement creation
- Various places in the XPath, XSLT and iteration APIs now require keyword-only arguments.
- The argument order in `element.itersiblings()` 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.
- 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.
- 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 `<>&/"'`.
- 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 loosing type information that is already there.
- `element.getiterator()` returns a list, use `element.iter()` to retrieve an iterator (ElementTree 1.3 compatible behaviour)
- 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

1.3.6 (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.

1.3.5 (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

1.3.4 (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).

1.3.3 (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.

1.3.2 (2007-07-03)

Features added

Bugs fixed

- "deallocating None" crash bug

1.3.1 (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()`

1.3 (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(el)` and `Element.addprevious(el)` 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

1.2.1 (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

1.2 (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 <= 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 compatibiliy of the public C-API

1.1.2 (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

1.1.1 (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 `nsmap` 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)

1.1 (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.itersiblings() 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.

1.0.4 (2006-09-09)

Features added

- List-like `Element.extend()` method

Bugs fixed

- Crash in tail handling in `Element.replace()`

1.0.3 (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

1.0.2 (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

1.0.1 (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)

1.0 (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)
- PyErrorLog 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, out-

put)

- 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

0.9.2 (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

0.9.1 (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)

0.9 (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)

0.8 (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 libxslt.
- 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.

0.7 (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.

0.6 (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!

0.5.1 (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.

0.5 (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. 221)
- **builder**: The `E` Element factory for generating XML documents.
  (Section B.3, p. 223)
- **cssselect**: CSS Selectors based on XPath.
  (Section B.4, p. 226)
- **doctestcompare**: lxml-based doctest output comparison.
  (Section B.5, p. 230)
- **etree**: The `lxml.etree` module implements the extended ElementTree API for XML.
  (Section B.6, p. 234)
- **html**: The `lxml.html` tool set for HTML handling.
  (Section B.7, p. 365)
  - **ElementSoup**: Legacy interface to the BeautifulSoup HTML parser.
    (Section B.8, p. 370)
  - **_dictmixin** *(Section* **??**, *p.* **??***)*
  - **_setmixin** *(Section* **??**, *p.* **??***)*
  - **builder**: A set of HTML generator tags for building HTML documents.
    (Section B.9, p. 371)
  - **clean**: A cleanup tool for HTML.
    (Section B.10, p. 374)
  - **defs** *(Section B.11, p. 378)*
  - **diff** *(Section B.12, p. 379)*
  - **formfill** *(Section B.13, p. 380)*
  - **html5parser**: An interface to html5lib.
    (Section B.14, p. 382)
  - **soupparser**: External interface to the BeautifulSoup HTML parser.
    (Section B.15, p. 386)
  - **usedoctest**: Doctest module for HTML comparison.
    (Section B.16, p. 387)
- **objectify**: The `lxml.objectify` module implements a Python object API for XML. It is based on `lxml.etree`.
  (Section B.17, p. 388)
- **pyclasslookup** *(Section B.18, p. 421)*
- **sax**: SAX-based adapter to copy trees from/to the Python standard library.
  (Section B.19, p. 422)
- **usedoctest**: Doctest module for XML comparison.
  (Section B.20, p. 429)

B.1.2 Variables

Name	Description
__package__	**Value:** `None`

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

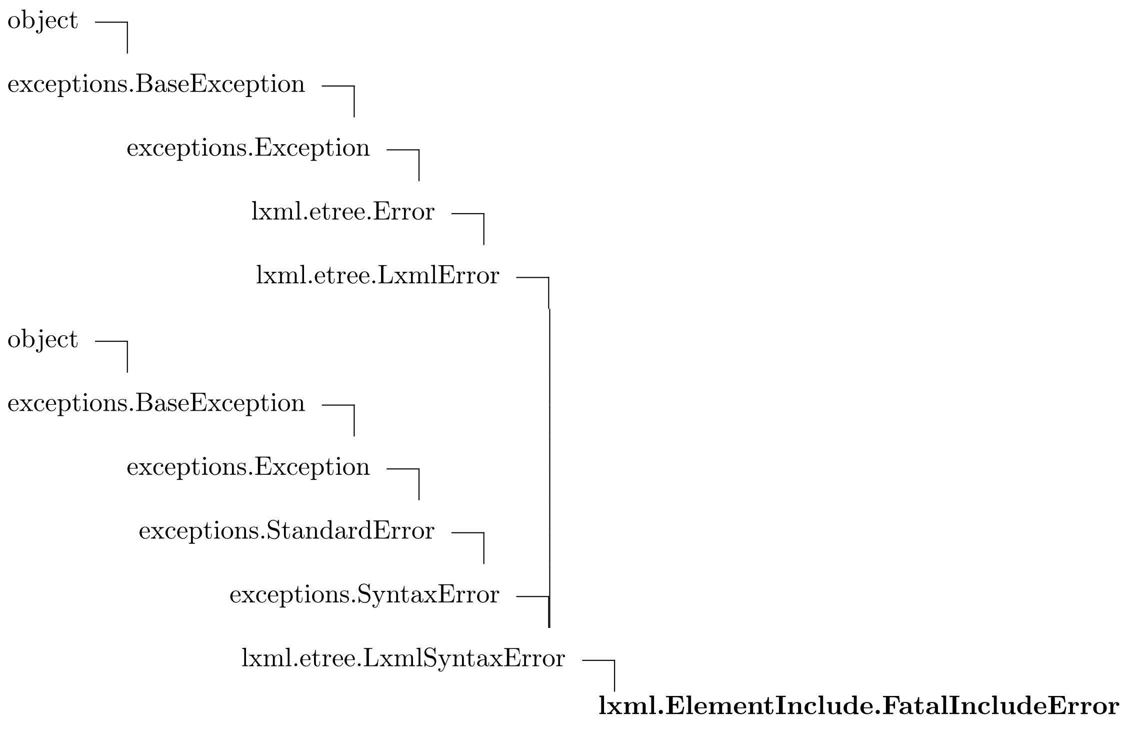
default_loader(*href*, *parse*, *encoding*=`None`)

include(*elem*, *loader*=`None`, *base_url*=`None`)

B.2.2 Variables

Name	Description
XINCLUDE	**Value:** `'{http://www.w3.org/2001/XInclude}'`
XINCLUDE_INCLUDE	**Value:** `'{http://www.w3.org/2001/XInclude}include'`
XINCLUDE_FALLBACK	**Value:** `'{http://www.w3.org/2001/XInclude}fallback'`
__package__	**Value:** `'lxml'`

B.2.3 Class FatalIncludeError



Methods

Inherited from lxml.etree.LxmlError(Section B.6.31)

__init__()

Inherited from exceptions.SyntaxError

__new__(), __str__()

Inherited from exceptions.BaseException

__delattr__(), __getattribute__(), __getitem__(), __getslice__(), __reduce__(), __repr__(), __setattr__(), __setstate__(), __unicode__()

Inherited from object

__format__(), __hash__(), __reduce_ex__(), __sizeof__(), __subclasshook__()

Properties

Name	Description
Inherited from exceptions.SyntaxError filename, lineno, msg, offset, print_file_and_line, text	
Inherited from exceptions.BaseException args, message	
Inherited from object __class__	

B.3 Module lxml.builder

The `E` Element factory for generating XML documents.

B.3.1 Functions

callable(*f*)

B.3.2 Variables

Name	Description
E	**Value:** `ElementMaker()`
__package__	**Value:** `'lxml'`

B.3.3 Class ElementMaker

object

lxml.builder.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 paragraph 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 reservered characters: <spam&egg>."),
                ET.XML("<p>And finally, here is an embedded 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 paragraph, with <a href="http://www.python.org">link</a>.<
        <p>Here are some reservered characters: &lt;spam&amp;egg&gt;.</p>
        <p>And finally, here is an embedded XHTML fragment.</p>
      </body>
    </html>

For namespace support, you can pass a namespace map (``nsmap``)
and/or a specific target ``namespace`` to the ElementMaker class::

>>> E = ElementMaker(namespace="http://my.ns/")
>>> print(ET.tostring( E.test ))
<test xmlns="http://my.ns/"/>

>>> E = ElementMaker(namespace="http://my.ns/", nsmap={'p':'http://my.ns/'})
>>> print(ET.tostring( E.test ))
<p:test xmlns:p="http://my.ns/"/>
```

Methods

__init__(*self*, *typemap*=None, *namespace*=None, *nsmap*=None, *makeelement*=None)

x.__init__(...) initializes x; see x.__class__.__doc__ for signature
Overrides: object.__init__ extit(inherited documentation)

__call__(*self*, *tag*, **children*, ***attrib*)

__getattr__(*self*, *tag*)

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

Properties

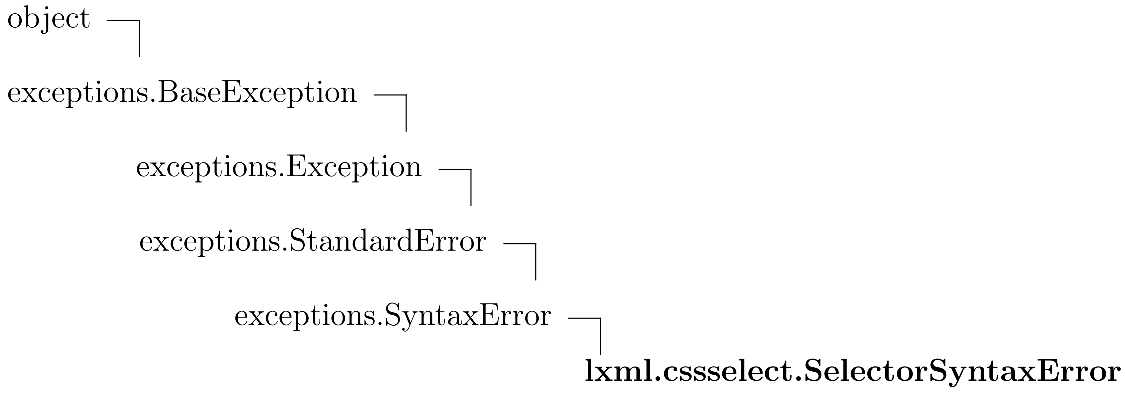
Name	Description
Inherited from object __class__	

B.4 Module lxml.cssselect

CSS Selectors based on XPath.

This module supports selecting XML/HTML tags based on CSS selectors. See the `CSSSelector` class for details.

B.4.1 Class SelectorSyntaxError



Methods

Inherited from exceptions.SyntaxError

__init__(), __new__(), __str__()

Inherited from exceptions.BaseException

__delattr__(), __getattribute__(), __getitem__(), __getslice__(), __reduce__(), __repr__(), __setattr__(), __setstate__(), __unicode__()

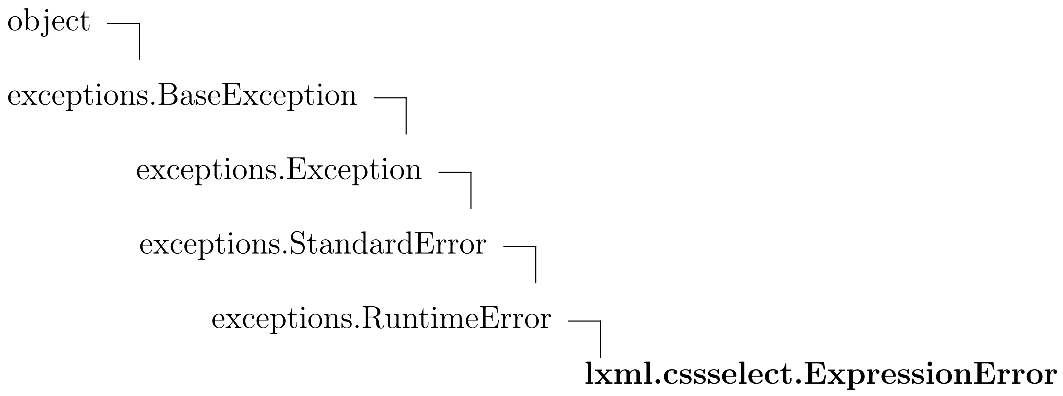
Inherited from object

__format__(), __hash__(), __reduce_ex__(), __sizeof__(), __subclasshook__()

Properties

Name	Description
Inherited from exceptions.SyntaxError filename, lineno, msg, offset, print_file_and_line, text	
Inherited from exceptions.BaseException args, message	
Inherited from object __class__	

B.4.2 Class ExpressionError



Methods

Inherited from exceptions.RuntimeError

__init__(), __new__()

Inherited from exceptions.BaseException

__delattr__(), __getattribute__(), __getitem__(), __getslice__(), __reduce__(), __repr__(), __setattr__(), __setstate__(), __str__(), __unicode__()

Inherited from object

__format__(), __hash__(), __reduce_ex__(), __sizeof__(), __subclasshook__()

Properties

Name	Description
Inherited from exceptions.BaseException args, message	
Inherited from object __class__	

B.4.3 Class CSSSelector

object
lxml.etree._XPathEvaluatorBase
lxml.etree.XPath
lxml.cssselect.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

__init__(*self*, *css*)

x.__init__(...) initializes x; see x.__class__.__doc__ for signature
Overrides: object.__init__ extit(inherited documentation)

__repr__(*self*)

repr(x) Overrides: object.__repr__ extit(inherited documentation)

Inherited from lxml.etree.XPath(Section B.6.63)

__call__(), __new__()

Inherited from lxml.etree._ XPathEvaluatorBase

evaluate()

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __reduce__(), __reduce_ex__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

Properties

Name	Description
Inherited from lxml.etree.XPath (Section B.6.63)	

continued on next page

Name	**Description**
path	
Inherited from lxml.etree._XPathEvaluatorBase error_log	
Inherited from object __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 `lxmldoctest.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

install(*html*=`False`)

Install doctestcompare for all future doctests.

If html 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
PARSE_HTML	**Value:** `1024`
PARSE_XML	**Value:** `2048`
NOPARSE_MARKUP	**Value:** `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 extit(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 extit(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
empty_tags	**Value:** `('param', 'img', 'area', 'br', 'basefont', 'input', 'base...`

B.5.4 Class LHTMLOutputChecker

doctest.OutputChecker

lxml.doctestcompare.LXMLOutputChecker

lxml.doctestcompare.LHTMLOutputChecker

Methods

get_default_parser(*self*)
Overrides: lxml.doctestcompare.LXMLOutputChecker.get_default_parser

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
Inherited from lxml.doctestcompare.LXMLOutputChecker (Section B.5.3) empty_tags	

B.6 Module lxml.etree

The `lxml.etree` module implements the extended ElementTree API for XML. **Version:** 2.2.6

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`, *nsmap*=`None`, **_ *extra*)

Element factory. This function returns an object implementing the Element interface.

Also look at the `_Element.makeelement()` and `_BaseParser.makeelement()` methods, which provide a faster way to create an Element within a specific document or parser context.

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. Returns the root node (or the result returned by a parser target). 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, *nsmap*=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 or fragment from a string constant. Returns the root node (or the result returned by a parser target). 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`, *smart_ strings*=`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). Smart strings will be returned for string results unless you pass `smart_strings=False`.

cleanup_ namespaces(*tree_ or_ element*)

Remove all namespace declarations from a subtree that are not used by any of the elements or attributes in that tree.

clear_ error_ log()

Clear the global error log. Note that this log is already bound to a fixed size.

Note: since lxml 2.2, the global error log is local to a thread and this function will only clear the global error log of the current thread.

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 or fragment from a string. Returns the root node (or the result returned by a parser target).

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. Returns the root node (or the result returned by a parser target).

To override the default parser with a different parser you can pass it to the `parser` keyword argument.

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 `source` can be any of the following:

- a file name/path
- a file object
- a file-like object
- a URL using the HTTP or FTP protocol

To parse from a string, use the `fromstring()` function instead.

Note that it is generally faster to parse from a file path or URL than from an open file object or file-like object. Transparent decompression from gzip compressed sources is supported (unless explicitly disabled in libxml2).

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.

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.

strip_attributes(*tree_or_element*, **attribute_names*)

Delete all attributes with the provided attribute names from an Element (or ElementTree) and its descendants.

Example usage:

```
    strip_attributes(root_element,
                     'simpleattr',
                     '{http://some/ns}attrname')
```

strip_elements(*tree_or_element*, *with_tail*=`True`, **tag_names*)

Delete all elements with the provided tag names from a tree or subtree. This will remove the elements and their entire subtree, including all their attributes, text content and descendants. It will also remove the tail text of the element unless you explicitly set the `with_tail` option to False.

Note that this will not delete the element (or ElementTree root element) that you passed even if it matches. It will only treat its descendants. If you want to include the root element, check its tag name directly before even calling this function.

Example usage:

```
    strip_elements(some_element,
        'simpletagname',             # non-namespaced tag
        '{http://some/ns}tagname',   # namespaced tag
        '{http://some/other/ns}*'    # any tag from a namespace
        Comment                      # comments
        )
```

strip_tags(*tree_or_element*, **tag_names*)

Delete all elements with the provided tag names from a tree or subtree. This will remove the elements and their attributes, but *not* their text/tail content or descendants. Instead, it will merge the text content and children of the element into its parent.

Note that this will not delete the element (or ElementTree root element) that you passed even if it matches. It will only treat its descendants.

Example usage:

```
    strip_tags(some_element,
        'simpletagname',             # non-namespaced tag
        '{http://some/ns}tagname',   # namespaced tag
        '{http://some/other/ns}*'    # any tag from a namespace
        Comment                      # comments (including their text!)
        )
```

tostring(*element_or_tree*, *encoding*=`None`, *method*=`"xml"`, *xml_declaration*=`None`, *pretty_print*=`False`, *with_tail*=`True`, *standalone*=`None`)

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 (or `str` in Py3).

The keyword argument 'pretty_print' (bool) enables formatted XML.

The keyword argument 'method' selects the output method: 'xml', 'html' or plain 'text'.

Passing a boolean value to the `standalone` option will output an XML declaration with the corresponding `standalone` flag.

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.

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.

Note: prior to lxml 2.2, this changed the error log globally. Since lxml 2.2, the global error log is local to a thread and this function will only set the global error log of the current thread.

B.6.2 Variables

Name	Description
DEBUG	**Value:** `1`
LIBXML_COMPILED_-VERSION	**Value:** `(2, 7, 5)`
LIBXML_VERSION	**Value:** `(2, 7, 5)`
LIBXSLT_COMPILED-_VERSION	**Value:** `(1, 1, 24)`
LIBXSLT_VERSION	**Value:** `(1, 1, 24)`
LXML_VERSION	**Value:** `(2, 2, 6, 0)`
__package__	**Value:** `'lxml'`
__pyx_capi__	**Value:** `{'appendChild': <PyCObject object at 0x2acd9e4cabe8>, 'at...`

B.6.3 Class AncestorsIterator

object

lxml.etree._ElementTagMatcher

lxml.etree._ElementIterator

lxml.etree.AncestorsIterator

AncestorsIterator(self, node, tag=None) Iterates over the ancestors of an element (from parent to parent).

Methods

__init__(*self*, *node*, *tag*=`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._ ElementIterator

__iter__(), __next__(), next()

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

Properties

Name	Description
Inherited from object __class__	

B.6.4 Class AttributeBasedElementClassLookup

object

lxml.etree.ElementClassLookup

lxml.etree.FallbackElementClassLookup

lxml.etree.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

__init__(*self*, *attribute_name*, *class_mapping*, *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__

Inherited from lxml.etree.FallbackElementClassLookup(Section B.6.29)

set_fallback()

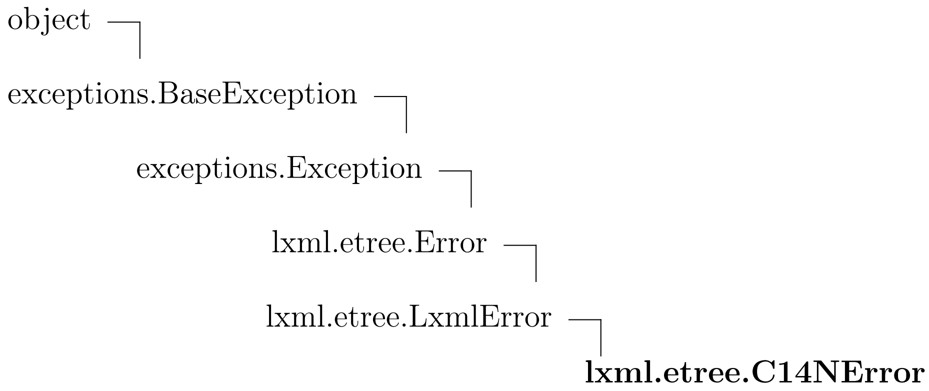
Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

Properties

Name	Description
Inherited from lxml.etree.FallbackElementClassLookup (Section B.6.29) fallback	
Inherited from object __class__	

B.6.5 Class C14NError



Error during C14N serialisation.

Methods

Inherited from lxml.etree.LxmlError(Section B.6.31)

__init__()

Inherited from exceptions.Exception

__new__()

Inherited from exceptions.BaseException

__delattr__(), __getattribute__(), __getitem__(), __getslice__(), __reduce__(), __repr__(), __setattr__(), __setstate__(), __str__(), __unicode__()

Inherited from object

__format__(), __hash__(), __reduce_ex__(), __sizeof__(), __subclasshook__()

Properties

Name	Description
Inherited from exceptions.BaseException args, message	

continued on next page

Name	Description
Inherited from object	
__class__	

B.6.6 Class CDATA

object ─┐
lxml.etree.CDATA

CDATA(data)

CDATA factory. This factory creates an opaque data object that can be used to set Element text. The usual way to use it is:

```
>>> from lxml import etree
>>> el = etree.Element('content')
>>> el.text = etree.CDATA('a string')
```

Methods

__init__(*data*)

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__(), __format__(), __getattribute__(), __hash__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

Properties

Name	Description
Inherited from object	
__class__	

B.6.7 Class CommentBase

object

lxml.etree._Element

lxml.etree.__ContentOnlyElement

lxml.etree._Comment

lxml.etree.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

__new__(*T*, *S*, *...*)

Return Value

a new object with type S, a subtype of T

Overrides: object.__new__

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__(), __format__(), __getattribute__(), __hash__(), __init__(), __reduce__(), __reduce_ex__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

Properties

Name	Description
Inherited from lxml.etree._Comment tag	
Inherited from lxml.etree.__ContentOnlyElement attrib, text	
Inherited from lxml.etree._Element base, nsmap, prefix, sourceline, tail	
Inherited from object __class__	

B.6.8 Class CustomElementClassLookup

object

lxml.etree.ElementClassLookup

lxml.etree.FallbackElementClassLookup

lxml.etree.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.6.29)

__init__(), set_fallback()

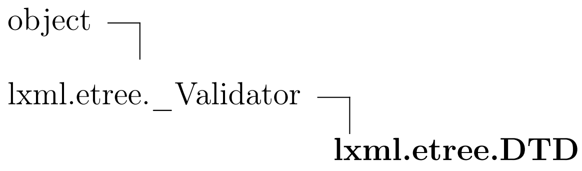
Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

Properties

Name	Description
Inherited from lxml.etree.FallbackElementClassLookup (Section B.6.29) fallback	
Inherited from object __class__	

B.6.9 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

__call__(*self*, *etree*)

Validate doc using the DTD.

Returns true if the document is valid, false if not.

__init__(*self*, *file*=`None`, *external_id*=`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._Validator

assertValid(), assert_(), validate()

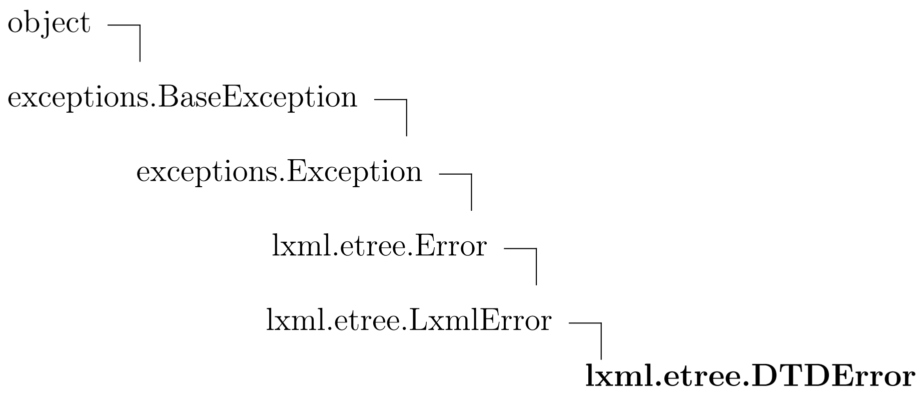
Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

Properties

Name	Description
Inherited from lxml.etree._Validator error_log	
Inherited from object __class__	

B.6.10 Class DTDError



Known Subclasses: lxml.etree.DTDParseError, lxml.etree.DTDValidateError

Base class for DTD errors.

Methods

Inherited from lxml.etree.LxmlError(Section B.6.31)

__init__()

Inherited from exceptions.Exception

__new__()

Inherited from exceptions.BaseException

__delattr__(), __getattribute__(), __getitem__(), __getslice__(), __reduce__(), __repr__(), __setattr__(), __setstate__(), __str__(), __unicode__()

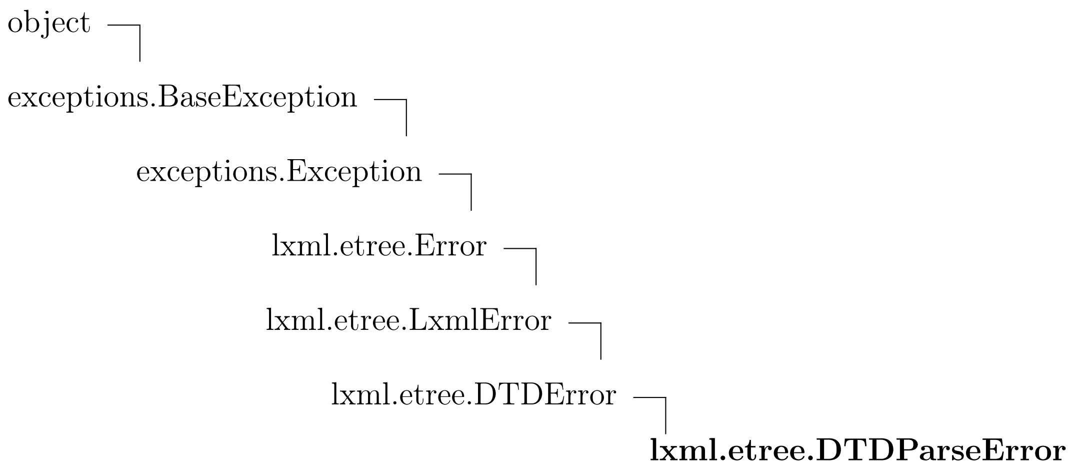
Inherited from object

__format__(), __hash__(), __reduce_ex__(), __sizeof__(), __subclasshook__()

Properties

Name	Description
Inherited from exceptions.BaseException args, message	
Inherited from object __class__	

B.6.11 Class DTDParseError



Error while parsing a DTD.

Methods

Inherited from lxml.etree.LxmlError(Section B.6.31)

__init__()

Inherited from exceptions.Exception

__new__()

Inherited from exceptions.BaseException

__delattr__(), __getattribute__(), __getitem__(), __getslice__(), __reduce__(), __repr__(), __setattr__(), __setstate__(), __str__(), __unicode__()

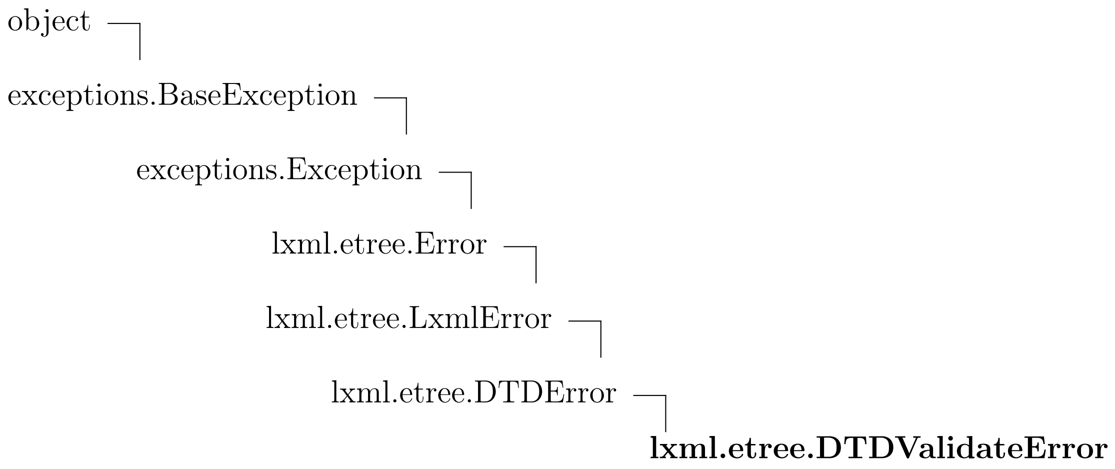
Inherited from object

__format__(), __hash__(), __reduce_ex__(), __sizeof__(), __subclasshook__()

Properties

Name	Description
Inherited from exceptions.BaseException args, message	
Inherited from object __class__	

B.6.12 Class DTDValidateError



Error while validating an XML document with a DTD.

Methods

Inherited from lxml.etree.LxmlError(Section B.6.31)

__init__()

Inherited from exceptions.Exception

__new__()

Inherited from exceptions.BaseException

__delattr__(), __getattribute__(), __getitem__(), __getslice__(), __reduce__(), __repr__(), __setattr__(), __setstate__(), __str__(), __unicode__()

Inherited from object

__format__(), __hash__(), __reduce_ex__(), __sizeof__(), __subclasshook__()

Properties

Name	**Description**
Inherited from exceptions.BaseException args, message	
Inherited from object __class__	

B.6.13 Class DocInfo

object —┐
lxml.etree.DocInfo

Document information provided by parser and DTD.

Methods

__init__(...)

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__(), __format__(), __getattribute__(), __hash__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

Properties

Name	Description
URL	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.
standalone	Returns the standalone flag as declared by the document. The possible values are True (`standalone='yes'`), False (`standalone='no'` or flag not provided in the declaration), and None (unknown or no declaration found). Note that a normal truth test on this value will always tell if the `standalone` flag was set to `'yes'` or not.
system_url	Returns the system ID of the DOCTYPE.
xml_version	Returns the XML version as declared by the document.
Inherited from object __class__	

B.6.14 Class DocumentInvalid

object

exceptions.BaseException

exceptions.Exception

lxml.etree.Error

lxml.etree.LxmlError

lxml.etree.DocumentInvalid

Validation error.

Raised by all document validators when their `assertValid(tree)` method fails.

Methods

Inherited from lxml.etree.LxmlError(Section B.6.31)

__init__()

Inherited from exceptions.Exception

__new__()

Inherited from exceptions.BaseException

__delattr__(), __getattribute__(), __getitem__(), __getslice__(), __reduce__(), __repr__(), __setattr__(), __setstate__(), __str__(), __unicode__()

Inherited from object

__format__(), __hash__(), __reduce_ex__(), __sizeof__(), __subclasshook__()

Properties

Name	Description
Inherited from exceptions.BaseException args, message	
Inherited from object __class__	

B.6.15 Class ETCompatXMLParser

object

lxml.etree._BaseParser

lxml.etree._FeedParser

lxml.etree.XMLParser

lxml.etree.ETCompatXMLParser

ETCompatXMLParser(self, encoding=None, attribute_defaults=False, dtd_validation=False, load_dtd=False, no_network=True, ns_clean=False, recover=False, schema=None, remove_blank_text=False, resolve_entities=True, remove_comments=True, remove_pis=True, strip_cdata=True, target=None, compact=True)

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*, *encoding*=`None`, *attribute_defaults*=`False`, *dtd_validation*=`False`, *load_dtd*=`False`, *no_network*=`True`, *ns_clean*=`False`, *recover*=`False`, *schema*=`None`, *remove_blank_text*=`False`, *resolve_entities*=`True`, *remove_comments*=`True`, *remove_pis*=`True`, *strip_cdata*=`True`, *target*=`None`, *compact*=`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._FeedParser

close(), feed()

Inherited from lxml.etree._BaseParser

copy(), makeelement(), setElementClassLookup(), set_element_class_lookup()

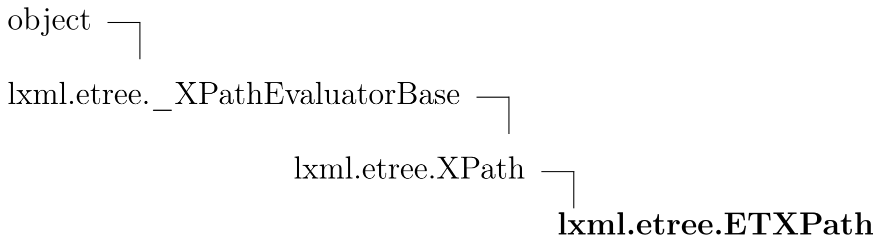
Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

Properties

Name	Description
Inherited from lxml.etree._FeedParser feed_error_log	
Inherited from lxml.etree._BaseParser error_log, resolvers, version	
Inherited from object __class__	

B.6.16 Class 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. Smart strings will be returned for string results unless you pass `smart_strings=False`.

Methods

__init__(*self*, *path*, *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.XPath(Section B.6.63)

__call__(), __repr__()

Inherited from lxml.etree._XPathEvaluatorBase

evaluate()

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __reduce__(), __reduce_ex__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

Properties

Name	Description
Inherited from lxml.etree.XPath (Section B.6.63)	

continued on next page

Name	Description
path	
Inherited from lxml.etree._XPathEvaluatorBase error_log	
Inherited from object __class__	

B.6.17 Class ElementBase

object

lxml.etree._Element

lxml.etree.ElementBase

Known Subclasses: lxml.objectify.ObjectifiedElement, lxml.html.HtmlElement

ElementBase(*children, attrib=None, nsmap=None, **_extra)

The public Element class. All custom Element classes must inherit from this one. To create an Element, use the `Element()` factory.

BIG FAT WARNING: 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 directly after object creation.

Subclasses of this class can be instantiated to create a new Element. By default, the tag name will be the class name and the namespace will be empty. You can modify this with the following class attributes:

- TAG - the tag name, possibly containing a namespace in Clark notation
- NAMESPACE - the default namespace URI, unless provided as part of the TAG attribute.
- HTML - flag if the class is an HTML tag, as opposed to an XML tag. This only applies to un-namespaced tags and defaults to false (i.e. XML).
- PARSER - the parser that provides the configuration for the newly created document. Providing an HTML parser here will default to creating an HTML element.

In user code, the latter three are commonly inherited in class hierarchies that implement a common namespace.

Methods

__init__(*attrib*=`None`, *nsmap*=`None`, **children*, **_*extra*)

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._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__(), __format__(), __getattribute__(), __hash__(), __reduce__(), __reduce_ex__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

Properties

Name	Description
Inherited from lxml.etree._Element attrib, base, nsmap, prefix, sourceline, tag, tail, text	
Inherited from object __class__	

B.6.18 Class ElementChildIterator

object

lxml.etree._ElementTagMatcher

lxml.etree._ElementIterator

lxml.etree.ElementChildIterator

ElementChildIterator(self, node, tag=None, reversed=False) Iterates over the children of an element.

Methods

__init__(*self*, *node*, *tag*=`None`, *reversed*=`False`)

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._ElementIterator

__iter__(), __next__(), next()

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

Properties

Name	Description
Inherited from object __class__	

B.6.19 Class ElementClassLookup

object ┐
lxml.etree.ElementClassLookup

Known Subclasses: lxml.etree.FallbackElementClassLookup, lxml.etree.ElementDefaultClassLookup, lxml.objectify.ObjectifyElementClassLookup

ElementClassLookup(self) Superclass of Element class lookups.

Methods

__new__(*T*, *S*, *...*)

Return Value

a new object with type S, a subtype of T

Overrides: object.__new__

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __init__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

Properties

Name	Description
Inherited from object	
__class__	

B.6.20 Class ElementDefaultClassLookup

object

lxml.etree.ElementClassLookup

lxml.etree.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

__init__(*self*, *element*=`None`, *comment*=`None`, *pi*=`None`, *entity*=`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__(), __format__(), __getattribute__(), __hash__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

Properties

Name	Description
comment_class	
element_class	
entity_class	
pi_class	
Inherited from object __class__	

B.6.21 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

__init__(*self*, *node*, *tag*=`None`, *inclusive*=`True`)

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__(...)

next(*x*)

Return Value

the next value, or raise StopIteration

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

Properties

Name	Description
Inherited from object __class__	

B.6.22 Class ElementNamespaceClassLookup

object

lxml.etree.ElementClassLookup

lxml.etree.FallbackElementClassLookup

lxml.etree.ElementNamespaceClassLookup

ElementNamespaceClassLookup(self, fallback=None)

Element class lookup scheme that searches the Element class in the Namespace registry.

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__

get_namespace(*self*, *ns_uri*)

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.6.29)

set_fallback()

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

Properties

Name	Description
Inherited from lxml.etree.FallbackElementClassLookup (Section B.6.29) fallback	
Inherited from object __class__	

B.6.23 Class ElementTextIterator

object ─┐
lxml.etree.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

__init__(*self*, *element*, *tag*=`None`, *with_tail*=`True`)

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__(...)

next(x)

Return Value

the next value, or raise StopIteration

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

Properties

Name	Description
Inherited from object __class__	

B.6.24 Class EntityBase

object

lxml.etree._Element

lxml.etree.__ContentOnlyElement

lxml.etree._Entity

lxml.etree.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__(), __format__(), __getattribute__(), __hash__(), __init__(), __reduce__(), __reduce_ex__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

Properties

Name	Description
Inherited from lxml.etree._Entity name, tag, text	
Inherited from lxml.etree.__ContentOnlyElement attrib	
Inherited from lxml.etree._Element base, nsmap, prefix, sourceline, tail	
Inherited from object __class__	

B.6.25 Class Error

object

exceptions.BaseException

exceptions.Exception

lxml.etree.Error

Known Subclasses: lxml.etree.LxmlError

Methods

Inherited from exceptions.Exception

__init__(), __new__()

Inherited from exceptions.BaseException

__delattr__(), __getattribute__(), __getitem__(), __getslice__(), __reduce__(), __repr__(), __setattr__(), __setstate__(), __str__(), __unicode__()

Inherited from object

__format__(), __hash__(), __reduce_ex__(), __sizeof__(), __subclasshook__()

Properties

Name	Description
Inherited from exceptions.BaseException args, message	
Inherited from object __class__	

B.6.26 Class ErrorDomains

Libxml2 error domains

Class Variables

Name	Description
C14N	**Value:** 21
CATALOG	**Value:** 20
CHECK	**Value:** 24
DATATYPE	**Value:** 15
DTD	**Value:** 4
FTP	**Value:** 9
HTML	**Value:** 5
HTTP	**Value:** 10
I18N	**Value:** 27
IO	**Value:** 8
MEMORY	**Value:** 6
MODULE	**Value:** 26
NAMESPACE	**Value:** 3
NONE	**Value:** 0
OUTPUT	**Value:** 7
PARSER	**Value:** 1
REGEXP	**Value:** 14
RELAXNGP	**Value:** 18
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.6.27 Class ErrorLevels

Libxml2 error levels

Class Variables

Name	Description
ERROR	**Value:** 2
FATAL	**Value:** 3
NONE	**Value:** 0
WARNING	**Value:** 1

B.6.28 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
CATALOG_ENTRY_BROKEN	**Value:** 1651
CATALOG_MISSING_ATTR	**Value:** 1650
CATALOG_NOT_CATALOG	**Value:** 1653
CATALOG_PREFER_VALUE	**Value:** 1652
CATALOG_RECURSION	**Value:** 1654
CHECK_	**Value:** 6005
CHECK_ENTITY_TYPE	**Value:** 5012
CHECK_FOUND_ATTRIBUTE	**Value:** 5001

continued on next page

Name	Description
CHECK_FOUND_CDATA	**Value:** 5003
CHECK_FOUND_COMMENT	**Value:** 5007
CHECK_FOUND_DOCTYPE	**Value:** 5008
CHECK_FOUND_ELEMENT	**Value:** 5000
CHECK_FOUND_ENTITY	**Value:** 5005
CHECK_FOUND_ENTITYREF	**Value:** 5004
CHECK_FOUND_FRAGMENT	**Value:** 5009
CHECK_FOUND_NOTATION	**Value:** 5010
CHECK_FOUND_PI	**Value:** 5006
CHECK_FOUND_TEXT	**Value:** 5002
CHECK_NAME_NOT_NULL	**Value:** 5037
CHECK_NOT_ATTR	**Value:** 5023
CHECK_NOT_ATTR_DECL	**Value:** 5024
CHECK_NOT_DTD	**Value:** 5022
CHECK_NOT_ELEM_DECL	**Value:** 5025
CHECK_NOT_ENTITY_DECL	**Value:** 5026
CHECK_NOT_NCNAME	**Value:** 5034
CHECK_NOT_NS_DECL	**Value:** 5027
CHECK_NOT_UTF8	**Value:** 5032
CHECK_NO_DICT	**Value:** 5033
CHECK_NO_DOC	**Value:** 5014
CHECK_NO_ELEM	**Value:** 5016
CHECK_NO_HREF	**Value:** 5028
CHECK_NO_NAME	**Value:** 5015
CHECK_NO_NEXT	**Value:** 5020
CHECK_NO_PARENT	**Value:** 5013
CHECK_NO_PREV	**Value:** 5018
CHECK_NS_ANCESTOR	**Value:** 5031
CHECK_NS_SCOPE	**Value:** 5030

continued on next page

Name	Description
CHECK_OUTSIDE_DICT	**Value:** 5035
CHECK_UNKNOWN_NODE	**Value:** 5011
CHECK_WRONG_DOC	**Value:** 5017
CHECK_WRONG_NAME	**Value:** 5036
CHECK_WRONG_NEXT	**Value:** 5021
CHECK_WRONG_PARENT	**Value:** 5029
CHECK_WRONG_PREV	**Value:** 5019
CHECK_X	**Value:** 6006
DTD_ATTRIBUTE_DEFAULT	**Value:** 500
DTD_ATTRIBUTE_REDEFINED	**Value:** 501
DTD_ATTRIBUTE_VALUE	**Value:** 502
DTD_CONTENT_ERROR	**Value:** 503
DTD_CONTENT_MODEL	**Value:** 504
DTD_CONTENT_NOT_DETERMINIST	**Value:** 505
DTD_DIFFERENT_PREFIX	**Value:** 506
DTD_DUP_TOKEN	**Value:** 541
DTD_ELEM_DEFAULT_NAMESPACE	**Value:** 507
DTD_ELEM_NAMESPACE	**Value:** 508
DTD_ELEM_REDEFINED	**Value:** 509
DTD_EMPTY_NOTATION	**Value:** 510
DTD_ENTITY_TYPE	**Value:** 511
DTD_ID_FIXED	**Value:** 512
DTD_ID_REDEFINED	**Value:** 513
DTD_ID_SUBSET	**Value:** 514
DTD_INVALID_CHILD	**Value:** 515
DTD_INVALID_DEFAULT	**Value:** 516

continued on next page

Name	Description
DTD_LOAD_ERROR	**Value:** 517
DTD_MISSING_ATTRIBUTE	**Value:** 518
DTD_MIXED_CORRUPT	**Value:** 519
DTD_MULTIPLE_ID	**Value:** 520
DTD_NOTATION_REDEFINED	**Value:** 526
DTD_NOTATION_VALUE	**Value:** 527
DTD_NOT_EMPTY	**Value:** 528
DTD_NOT_PCDATA	**Value:** 529
DTD_NOT_STANDALONE	**Value:** 530
DTD_NO_DOC	**Value:** 521
DTD_NO_DTD	**Value:** 522
DTD_NO_ELEM_NAME	**Value:** 523
DTD_NO_PREFIX	**Value:** 524
DTD_NO_ROOT	**Value:** 525
DTD_ROOT_NAME	**Value:** 531
DTD_STANDALONE_DEFAULTED	**Value:** 538
DTD_STANDALONE_WHITE_SPACE	**Value:** 532
DTD_UNKNOWN_ATTRIBUTE	**Value:** 533
DTD_UNKNOWN_ELEM	**Value:** 534
DTD_UNKNOWN_ENTITY	**Value:** 535
DTD_UNKNOWN_ID	**Value:** 536
DTD_UNKNOWN_NOTATION	**Value:** 537
DTD_XMLID_TYPE	**Value:** 540
DTD_XMLID_VALUE	**Value:** 539
ERR_ATTLIST_NOT_FINISHED	**Value:** 51
ERR_ATTLIST_NOT_STARTED	**Value:** 50
ERR_ATTRIBUTE_NOT_FINISHED	**Value:** 40
ERR_ATTRIBUTE_NOT_STARTED	**Value:** 39
ERR_ATTRIBUTE_REDEFINED	**Value:** 42

continued on next page

Name	Description
ERR_ATTRIBUTE_WITHOUT_VALUE	**Value:** 41
ERR_CDATA_NOT_FINISHED	**Value:** 63
ERR_CHARREF_AT_EOF	**Value:** 10
ERR_CHARREF_IN_DTD	**Value:** 13
ERR_CHARREF_IN_EPILOG	**Value:** 12
ERR_CHARREF_IN_PROLOG	**Value:** 11
ERR_COMMENT_NOT_FINISHED	**Value:** 45
ERR_CONDSEC_INVALID	**Value:** 83
ERR_CONDSEC_INVALID_KEYWORD	**Value:** 95
ERR_CONDSEC_NOT_FINISHED	**Value:** 59
ERR_CONDSEC_NOT_STARTED	**Value:** 58
ERR_DOCTYPE_NOT_FINISHED	**Value:** 61
ERR_DOCUMENT_EMPTY	**Value:** 4
ERR_DOCUMENT_END	**Value:** 5
ERR_DOCUMENT_START	**Value:** 3
ERR_ELEMCONTENT_NOT_FINISHED	**Value:** 55
ERR_ELEMCONTENT_NOT_STARTED	**Value:** 54
ERR_ENCODING_NAME	**Value:** 79
ERR_ENTITYREF_AT_EOF	**Value:** 14
ERR_ENTITYREF_IN_DTD	**Value:** 17
ERR_ENTITYREF_IN_EPILOG	**Value:** 16
ERR_ENTITYREF_IN_PROLOG	**Value:** 15
ERR_ENTITYREF_NO_NAME	**Value:** 22

continued on next page

Name	Description
ERR_ENTITYREF_SEMICOL_MISSING	**Value:** 23
ERR_ENTITY_BOUNDARY	**Value:** 90
ERR_ENTITY_CHAR_ERROR	**Value:** 87
ERR_ENTITY_IS_EXTERNAL	**Value:** 29
ERR_ENTITY_IS_PARAMETER	**Value:** 30
ERR_ENTITY_LOOP	**Value:** 89
ERR_ENTITY_NOT_FINISHED	**Value:** 37
ERR_ENTITY_NOT_STARTED	**Value:** 36
ERR_ENTITY_PE_INTERNAL	**Value:** 88
ERR_ENTITY_PROCESSING	**Value:** 104
ERR_EQUAL_REQUIRED	**Value:** 75
ERR_EXTRA_CONTENT	**Value:** 86
ERR_EXT_ENTITY_STANDALONE	**Value:** 82
ERR_EXT_SUBSET_NOT_FINISHED	**Value:** 60
ERR_GT_REQUIRED	**Value:** 73
ERR_HYPHEN_IN_COMMENT	**Value:** 80
ERR_INTERNAL_ERROR	**Value:** 1
ERR_INVALID_CHAR	**Value:** 9
ERR_INVALID_CHARREF	**Value:** 8
ERR_INVALID_DEC_CHARREF	**Value:** 7
ERR_INVALID_ENCODING	**Value:** 81
ERR_INVALID_HEX_CHARREF	**Value:** 6
ERR_INVALID_URI	**Value:** 91
ERR_LITERAL_NOT_FINISHED	**Value:** 44
ERR_LITERAL_NOT_STARTED	**Value:** 43

continued on next page

Name	Description
ERR_LTSLASH_REQUIRED	**Value:** 74
ERR_LT_IN_ATTRIBUTE	**Value:** 38
ERR_LT_REQUIRED	**Value:** 72
ERR_MISPLACED_CDATA_END	**Value:** 62
ERR_MISSING_ENCODING	**Value:** 101
ERR_MIXED_NOT_FINISHED	**Value:** 53
ERR_MIXED_NOT_STARTED	**Value:** 52
ERR_NAME_REQUIRED	**Value:** 68
ERR_NMTOKEN_REQUIRED	**Value:** 67
ERR_NOTATION_NOT_FINISHED	**Value:** 49
ERR_NOTATION_NOT_STARTED	**Value:** 48
ERR_NOTATION_PROCESSING	**Value:** 105
ERR_NOT_STANDALONE	**Value:** 103
ERR_NOT_WELL_BALANCED	**Value:** 85
ERR_NO_DTD	**Value:** 94
ERR_NO_MEMORY	**Value:** 2
ERR_NS_DECL_ERROR	**Value:** 35
ERR_OK	**Value:** 0
ERR_PCDATA_REQUIRED	**Value:** 69
ERR_PEREF_AT_EOF	**Value:** 18
ERR_PEREF_IN_EPILOG	**Value:** 20
ERR_PEREF_IN_INT_SUBSET	**Value:** 21
ERR_PEREF_IN_PROLOG	**Value:** 19
ERR_PEREF_NO_NAME	**Value:** 24
ERR_PEREF_SEMICOL_MISSING	**Value:** 25

continued on next page

Name	Description
ERR_PI_NOT_FINISHED	**Value:** 47
ERR_PI_NOT_STARTED	**Value:** 46
ERR_PUBID_REQUIRED	**Value:** 71
ERR_RESERVED_XML_NAME	**Value:** 64
ERR_SEPARATOR_REQUIRED	**Value:** 66
ERR_SPACE_REQUIRED	**Value:** 65
ERR_STANDALONE_VALUE	**Value:** 78
ERR_STRING_NOT_CLOSED	**Value:** 34
ERR_STRING_NOT_STARTED	**Value:** 33
ERR_TAG_NAME_MISMATCH	**Value:** 76
ERR_TAG_NOT_FINISHED	**Value:** 77
ERR_UNDECLARED_ENTITY	**Value:** 26
ERR_UNKNOWN_ENCODING	**Value:** 31
ERR_UNKNOWN_VERSION	**Value:** 108
ERR_UNPARSED_ENTITY	**Value:** 28
ERR_UNSUPPORTED_ENCODING	**Value:** 32
ERR_URI_FRAGMENT	**Value:** 92
ERR_URI_REQUIRED	**Value:** 70
ERR_VALUE_REQUIRED	**Value:** 84
ERR_VERSION_MISMATCH	**Value:** 109
ERR_VERSION_MISSING	**Value:** 96
ERR_XMLDECL_NOT_FINISHED	**Value:** 57
ERR_XMLDECL_NOT_STARTED	**Value:** 56
FTP_ACCNT	**Value:** 2002

continued on next page

Name	Description
FTP_EPSV_ANSWER	**Value:** 2001
FTP_PASV_ANSWER	**Value:** 2000
FTP_URL_SYNTAX	**Value:** 2003
HTML_STRUCURE_ERROR	**Value:** 800
HTML_UNKNOWN_TAG	**Value:** 801
HTTP_UNKNOWN_HOST	**Value:** 2022
HTTP_URL_SYNTAX	**Value:** 2020
HTTP_USE_IP	**Value:** 2021
I18N_CONV_FAILED	**Value:** 6003
I18N_EXCESS_HANDLER	**Value:** 6002
I18N_NO_HANDLER	**Value:** 6001
I18N_NO_NAME	**Value:** 6000
I18N_NO_OUTPUT	**Value:** 6004
IO_BUFFER_FULL	**Value:** 1548
IO_EACCES	**Value:** 1501
IO_EADDRINUSE	**Value:** 1554
IO_EAFNOSUPPORT	**Value:** 1556
IO_EAGAIN	**Value:** 1502
IO_EALREADY	**Value:** 1555
IO_EBADF	**Value:** 1503
IO_EBADMSG	**Value:** 1504
IO_EBUSY	**Value:** 1505
IO_ECANCELED	**Value:** 1506
IO_ECHILD	**Value:** 1507
IO_ECONNREFUSED	**Value:** 1552
IO_EDEADLK	**Value:** 1508
IO_EDOM	**Value:** 1509
IO_EEXIST	**Value:** 1510
IO_EFAULT	**Value:** 1511
IO_EFBIG	**Value:** 1512
IO_EINPROGRESS	**Value:** 1513
IO_EINTR	**Value:** 1514
IO_EINVAL	**Value:** 1515
IO_EIO	**Value:** 1516
IO_EISCONN	**Value:** 1551
IO_EISDIR	**Value:** 1517
IO_EMFILE	**Value:** 1518
IO_EMLINK	**Value:** 1519
IO_EMSGSIZE	**Value:** 1520
IO_ENAMETOOLONG	**Value:** 1521
IO_ENCODER	**Value:** 1544
IO_ENETUNREACH	**Value:** 1553

continued on next page

Name	Description
IO_ENFILE	**Value:** 1522
IO_ENODEV	**Value:** 1523
IO_ENOENT	**Value:** 1524
IO_ENOEXEC	**Value:** 1525
IO_ENOLCK	**Value:** 1526
IO_ENOMEM	**Value:** 1527
IO_ENOSPC	**Value:** 1528
IO_ENOSYS	**Value:** 1529
IO_ENOTDIR	**Value:** 1530
IO_ENOTEMPTY	**Value:** 1531
IO_ENOTSOCK	**Value:** 1550
IO_ENOTSUP	**Value:** 1532
IO_ENOTTY	**Value:** 1533
IO_ENXIO	**Value:** 1534
IO_EPERM	**Value:** 1535
IO_EPIPE	**Value:** 1536
IO_ERANGE	**Value:** 1537
IO_EROFS	**Value:** 1538
IO_ESPIPE	**Value:** 1539
IO_ESRCH	**Value:** 1540
IO_ETIMEDOUT	**Value:** 1541
IO_EXDEV	**Value:** 1542
IO_FLUSH	**Value:** 1545
IO_LOAD_ERROR	**Value:** 1549
IO_NETWORK_ATTEMPT	**Value:** 1543
IO_NO_INPUT	**Value:** 1547
IO_UNKNOWN	**Value:** 1500
IO_WRITE	**Value:** 1546
MODULE_CLOSE	**Value:** 4901
MODULE_OPEN	**Value:** 4900
NS_ERR_ATTRIBUTE_REDEFINED	**Value:** 203
NS_ERR_COLON	**Value:** 205
NS_ERR_EMPTY	**Value:** 204
NS_ERR_QNAME	**Value:** 202
NS_ERR_UNDEFINED_NAMESPACE	**Value:** 201
NS_ERR_XML_NAMESPACE	**Value:** 200
REGEXP_COMPILE_ERROR	**Value:** 1450
RNGP_ANYNAME_ATTR_ANCESTOR	**Value:** 1000
RNGP_ATTRIBUTE_CHILDREN	**Value:** 1002

continued on next page

Name	Description
RNGP_ATTRIBUTE_-CONTENT	**Value:** 1003
RNGP_ATTRIBUTE_-EMPTY	**Value:** 1004
RNGP_ATTRIBUTE_-NOOP	**Value:** 1005
RNGP_ATTR_CONFLICT	**Value:** 1001
RNGP_CHOICE_CONTENT	**Value:** 1006
RNGP_CHOICE_EMPTY	**Value:** 1007
RNGP_CREATE_FAILURE	**Value:** 1008
RNGP_DATA_CONTENT	**Value:** 1009
RNGP_DEFINE_CREATE_FAILED	**Value:** 1011
RNGP_DEFINE_EMPTY	**Value:** 1012
RNGP_DEFINE_MISSING	**Value:** 1013
RNGP_DEFINE_NAME_MISSING	**Value:** 1014
RNGP_DEF_CHOICE_AND_INTERLEAVE	**Value:** 1010
RNGP_ELEMENT_CONTENT	**Value:** 1018
RNGP_ELEMENT_EMPTY	**Value:** 1017
RNGP_ELEMENT_NAME	**Value:** 1019
RNGP_ELEMENT_NO_CONTENT	**Value:** 1020
RNGP_ELEM_CONTENT_EMPTY	**Value:** 1015
RNGP_ELEM_CONTENT_ERROR	**Value:** 1016
RNGP_ELEM_TEXT_-CONFLICT	**Value:** 1021
RNGP_EMPTY	**Value:** 1022
RNGP_EMPTY_CONSTRUCT	**Value:** 1023
RNGP_EMPTY_CONTENT	**Value:** 1024

continued on next page

Name	Description
RNGP_EMPTY_NOT_EMPTY	**Value:** 1025
RNGP_ERROR_TYPE_LIB	**Value:** 1026
RNGP_EXCEPT_EMPTY	**Value:** 1027
RNGP_EXCEPT_MISSING	**Value:** 1028
RNGP_EXCEPT_MULTIPLE	**Value:** 1029
RNGP_EXCEPT_NO_CONTENT	**Value:** 1030
RNGP_EXTERNALREF_EMTPY	**Value:** 1031
RNGP_EXTERNALREF_RECURSE	**Value:** 1033
RNGP_EXTERNAL_REF_FAILURE	**Value:** 1032
RNGP_FORBIDDEN_ATTRIBUTE	**Value:** 1034
RNGP_FOREIGN_ELEMENT	**Value:** 1035
RNGP_GRAMMAR_CONTENT	**Value:** 1036
RNGP_GRAMMAR_EMPTY	**Value:** 1037
RNGP_GRAMMAR_MISSING	**Value:** 1038
RNGP_GRAMMAR_NO_START	**Value:** 1039
RNGP_GROUP_ATTR_CONFLICT	**Value:** 1040
RNGP_HREF_ERROR	**Value:** 1041
RNGP_INCLUDE_EMPTY	**Value:** 1042
RNGP_INCLUDE_FAILURE	**Value:** 1043
RNGP_INCLUDE_RECURSE	**Value:** 1044
RNGP_INTERLEAVE_ADD	**Value:** 1045
RNGP_INTERLEAVE_CREATE_FAILED	**Value:** 1046
RNGP_INTERLEAVE_EMPTY	**Value:** 1047

continued on next page

Name	Description
RNGP_INTERLEAVE_NO_CONTENT	**Value:** 1048
RNGP_INVALID_DEFINE_NAME	**Value:** 1049
RNGP_INVALID_URI	**Value:** 1050
RNGP_INVALID_VALUE	**Value:** 1051
RNGP_MISSING_HREF	**Value:** 1052
RNGP_NAME_MISSING	**Value:** 1053
RNGP_NEED_COMBINE	**Value:** 1054
RNGP_NOTALLOWED_NOT_EMPTY	**Value:** 1055
RNGP_NSNAME_ATTR_ANCESTOR	**Value:** 1056
RNGP_NSNAME_NO_NS	**Value:** 1057
RNGP_PARAM_FORBIDDEN	**Value:** 1058
RNGP_PARAM_NAME_MISSING	**Value:** 1059
RNGP_PARENTREF_CREATE_FAILED	**Value:** 1060
RNGP_PARENTREF_NAME_INVALID	**Value:** 1061
RNGP_PARENTREF_NOT_EMPTY	**Value:** 1064
RNGP_PARENTREF_NO_NAME	**Value:** 1062
RNGP_PARENTREF_NO_PARENT	**Value:** 1063
RNGP_PARSE_ERROR	**Value:** 1065
RNGP_PAT_ANYNAME_EXCEPT_ANYNAME	**Value:** 1066
RNGP_PAT_ATTR_ATTR	**Value:** 1067
RNGP_PAT_ATTR_ELEM	**Value:** 1068
RNGP_PAT_DATA_EXCEPT_ATTR	**Value:** 1069
RNGP_PAT_DATA_EXCEPT_ELEM	**Value:** 1070

continued on next page

Name	Description
RNGP_PAT_DATA_EXCEPT_EMPTY	**Value:** 1071
RNGP_PAT_DATA_EXCEPT_GROUP	**Value:** 1072
RNGP_PAT_DATA_EXCEPT_INTERLEAVE	**Value:** 1073
RNGP_PAT_DATA_EXCEPT_LIST	**Value:** 1074
RNGP_PAT_DATA_EXCEPT_ONEMORE	**Value:** 1075
RNGP_PAT_DATA_EXCEPT_REF	**Value:** 1076
RNGP_PAT_DATA_EXCEPT_TEXT	**Value:** 1077
RNGP_PAT_LIST_ATTR	**Value:** 1078
RNGP_PAT_LIST_ELEM	**Value:** 1079
RNGP_PAT_LIST_INTERLEAVE	**Value:** 1080
RNGP_PAT_LIST_LIST	**Value:** 1081
RNGP_PAT_LIST_REF	**Value:** 1082
RNGP_PAT_LIST_TEXT	**Value:** 1083
RNGP_PAT_NSNAME_EXCEPT_ANYNAME	**Value:** 1084
RNGP_PAT_NSNAME_EXCEPT_NSNAME	**Value:** 1085
RNGP_PAT_ONEMORE_GROUP_ATTR	**Value:** 1086
RNGP_PAT_ONEMORE_INTERLEAVE_ATTR	**Value:** 1087
RNGP_PAT_START_ATTR	**Value:** 1088
RNGP_PAT_START_DATA	**Value:** 1089
RNGP_PAT_START_EMPTY	**Value:** 1090
RNGP_PAT_START_GROUP	**Value:** 1091
RNGP_PAT_START_INTERLEAVE	**Value:** 1092

continued on next page

Name	Description
RNGP_PAT_START_LIST	**Value:** 1093
RNGP_PAT_START_ONEMORE	**Value:** 1094
RNGP_PAT_START_TEXT	**Value:** 1095
RNGP_PAT_START_VALUE	**Value:** 1096
RNGP_PREFIX_UNDEFINED	**Value:** 1097
RNGP_REF_CREATE_FAILED	**Value:** 1098
RNGP_REF_CYCLE	**Value:** 1099
RNGP_REF_NAME_INVALID	**Value:** 1100
RNGP_REF_NOT_EMPTY	**Value:** 1103
RNGP_REF_NO_DEF	**Value:** 1101
RNGP_REF_NO_NAME	**Value:** 1102
RNGP_START_CHOICE_AND_INTERLEAVE	**Value:** 1104
RNGP_START_CONTENT	**Value:** 1105
RNGP_START_EMPTY	**Value:** 1106
RNGP_START_MISSING	**Value:** 1107
RNGP_TEXT_EXPECTED	**Value:** 1108
RNGP_TEXT_HAS_CHILD	**Value:** 1109
RNGP_TYPE_MISSING	**Value:** 1110
RNGP_TYPE_NOT_FOUND	**Value:** 1111
RNGP_TYPE_VALUE	**Value:** 1112
RNGP_UNKNOWN_ATTRIBUTE	**Value:** 1113
RNGP_UNKNOWN_COMBINE	**Value:** 1114
RNGP_UNKNOWN_CONSTRUCT	**Value:** 1115
RNGP_UNKNOWN_TYPE_LIB	**Value:** 1116

continued on next page

Name	Description
RNGP_URI_FRAGMENT	**Value:** 1117
RNGP_URI_NOT_ABSOLUTE	**Value:** 1118
RNGP_VALUE_EMPTY	**Value:** 1119
RNGP_VALUE_NO_CONTENT	**Value:** 1120
RNGP_XMLNS_NAME	**Value:** 1121
RNGP_XML_NS	**Value:** 1122
SAVE_CHAR_INVALID	**Value:** 1401
SAVE_NOT_UTF8	**Value:** 1400
SAVE_NO_DOCTYPE	**Value:** 1402
SAVE_UNKNOWN_ENCODING	**Value:** 1403
SCHEMAP_AG_PROPS_CORRECT	**Value:** 3087
SCHEMAP_ATTRFORMDEFAULT_VALUE	**Value:** 1701
SCHEMAP_ATTRGRP_NONAME_NOREF	**Value:** 1702
SCHEMAP_ATTR_NONAME_NOREF	**Value:** 1703
SCHEMAP_AU_PROPS_CORRECT	**Value:** 3089
SCHEMAP_AU_PROPS_CORRECT_2	**Value:** 3078
SCHEMAP_A_PROPS_CORRECT_2	**Value:** 3079
SCHEMAP_A_PROPS_CORRECT_3	**Value:** 3090
SCHEMAP_COMPLEXTYPE_NONAME_NOREF	**Value:** 1704
SCHEMAP_COS_ALL_LIMITED	**Value:** 3091
SCHEMAP_COS_CT_EXTENDS_1_1	**Value:** 3063
SCHEMAP_COS_CT_EXTENDS_1_2	**Value:** 3088
SCHEMAP_COS_CT_EXTENDS_1_3	**Value:** 1800
SCHEMAP_COS_ST_DERIVED_OK_2_1	**Value:** 3031

continued on next page

Name	Description
SCHEMAP_COS_ST_-DERIVED_OK_2_2	**Value:** 3032
SCHEMAP_COS_ST_-RESTRICTS_1_1	**Value:** 3011
SCHEMAP_COS_ST_-RESTRICTS_1_2	**Value:** 3012
SCHEMAP_COS_ST_-RESTRICTS_1_3_1	**Value:** 3013
SCHEMAP_COS_ST_-RESTRICTS_1_3_2	**Value:** 3014
SCHEMAP_COS_ST_-RESTRICTS_2_1	**Value:** 3015
SCHEMAP_COS_ST_-RESTRICTS_2_3_1_1	**Value:** 3016
SCHEMAP_COS_ST_-RESTRICTS_2_3_1_2	**Value:** 3017
SCHEMAP_COS_ST_-RESTRICTS_2_3_2_1	**Value:** 3018
SCHEMAP_COS_ST_-RESTRICTS_2_3_2_2	**Value:** 3019
SCHEMAP_COS_ST_-RESTRICTS_2_3_2_3	**Value:** 3020
SCHEMAP_COS_ST_-RESTRICTS_2_3_2_4	**Value:** 3021
SCHEMAP_COS_ST_-RESTRICTS_2_3_2_5	**Value:** 3022
SCHEMAP_COS_ST_-RESTRICTS_3_1	**Value:** 3023
SCHEMAP_COS_ST_-RESTRICTS_3_3_1	**Value:** 3024
SCHEMAP_COS_ST_-RESTRICTS_3_3_1_2	**Value:** 3025
SCHEMAP_COS_ST_-RESTRICTS_3_3_2_1	**Value:** 3027
SCHEMAP_COS_ST_-RESTRICTS_3_3_2_2	**Value:** 3026
SCHEMAP_COS_ST_-RESTRICTS_3_3_2_3	**Value:** 3028
SCHEMAP_COS_ST_-RESTRICTS_3_3_2_4	**Value:** 3029
SCHEMAP_COS_ST_-RESTRICTS_3_3_2_5	**Value:** 3030
SCHEMAP_COS_VALI-D_DEFAULT_1	**Value:** 3058
SCHEMAP_COS_VALI-D_DEFAULT_2_1	**Value:** 3059

continued on next page

Name	Description
SCHEMAP_COS_VALID_DEFAULT_2_2_1	**Value:** 3060
SCHEMAP_COS_VALID_DEFAULT_2_2_2	**Value:** 3061
SCHEMAP_CT_PROPS_CORRECT_1	**Value:** 1782
SCHEMAP_CT_PROPS_CORRECT_2	**Value:** 1783
SCHEMAP_CT_PROPS_CORRECT_3	**Value:** 1784
SCHEMAP_CT_PROPS_CORRECT_4	**Value:** 1785
SCHEMAP_CT_PROPS_CORRECT_5	**Value:** 1786
SCHEMAP_CVC_SIMPLE_TYPE	**Value:** 3062
SCHEMAP_C_PROPS_CORRECT	**Value:** 3080
SCHEMAP_DEF_AND_PREFIX	**Value:** 1768
SCHEMAP_DERIVATION_OK_RESTRICTION_1	**Value:** 1787
SCHEMAP_DERIVATION_OK_RESTRICTION_2_1_1	**Value:** 1788
SCHEMAP_DERIVATION_OK_RESTRICTION_2_1_2	**Value:** 1789
SCHEMAP_DERIVATION_OK_RESTRICTION_2_1_3	**Value:** 3077
SCHEMAP_DERIVATION_OK_RESTRICTION_2_2	**Value:** 1790
SCHEMAP_DERIVATION_OK_RESTRICTION_3	**Value:** 1791
SCHEMAP_DERIVATION_OK_RESTRICTION_4_1	**Value:** 1797
SCHEMAP_DERIVATION_OK_RESTRICTION_4_2	**Value:** 1798

continued on next page

Name	Description
SCHEMAP_DERIVATION_OK_RESTRICTION_4_3	**Value:** 1799
SCHEMAP_ELEMFORMDEFAULT_VALUE	**Value:** 1705
SCHEMAP_ELEM_DEFAULT_FIXED	**Value:** 1755
SCHEMAP_ELEM_NONAME_NOREF	**Value:** 1706
SCHEMAP_EXTENSION_NO_BASE	**Value:** 1707
SCHEMAP_E_PROPS_CORRECT_2	**Value:** 3045
SCHEMAP_E_PROPS_CORRECT_3	**Value:** 3046
SCHEMAP_E_PROPS_CORRECT_4	**Value:** 3047
SCHEMAP_E_PROPS_CORRECT_5	**Value:** 3048
SCHEMAP_E_PROPS_CORRECT_6	**Value:** 3049
SCHEMAP_FACET_NO_VALUE	**Value:** 1708
SCHEMAP_FAILED_BUILD_IMPORT	**Value:** 1709
SCHEMAP_FAILED_LOAD	**Value:** 1757
SCHEMAP_FAILED_PARSE	**Value:** 1766
SCHEMAP_GROUP_NONAME_NOREF	**Value:** 1710
SCHEMAP_IMPORT_NAMESPACE_NOT_URI	**Value:** 1711
SCHEMAP_IMPORT_REDEFINE_NSNAME	**Value:** 1712
SCHEMAP_IMPORT_SCHEMA_NOT_URI	**Value:** 1713
SCHEMAP_INCLUDE_SCHEMA_NOT_URI	**Value:** 1770
SCHEMAP_INCLUDE_SCHEMA_NO_URI	**Value:** 1771
SCHEMAP_INTERNAL	**Value:** 3069
SCHEMAP_INTERSECTION_NOT_EXPRESSIBLE	**Value:** 1793

continued on next page

Name	Description
SCHEMAP_INVALID_ATTR_COMBINATION	**Value:** 1777
SCHEMAP_INVALID_ATTR_INLINE_COMBINATION	**Value:** 1778
SCHEMAP_INVALID_ATTR_NAME	**Value:** 1780
SCHEMAP_INVALID_ATTR_USE	**Value:** 1774
SCHEMAP_INVALID_BOOLEAN	**Value:** 1714
SCHEMAP_INVALID_ENUM	**Value:** 1715
SCHEMAP_INVALID_FACET	**Value:** 1716
SCHEMAP_INVALID_FACET_VALUE	**Value:** 1717
SCHEMAP_INVALID_MAXOCCURS	**Value:** 1718
SCHEMAP_INVALID_MINOCCURS	**Value:** 1719
SCHEMAP_INVALID_REF_AND_SUBTYPE	**Value:** 1720
SCHEMAP_INVALID_WHITE_SPACE	**Value:** 1721
SCHEMAP_MG_PROPS_CORRECT_1	**Value:** 3074
SCHEMAP_MG_PROPS_CORRECT_2	**Value:** 3075
SCHEMAP_MISSING_SIMPLETYPE_CHILD	**Value:** 1779
SCHEMAP_NOATTR_NOREF	**Value:** 1722
SCHEMAP_NOROOT	**Value:** 1759
SCHEMAP_NOTATION_NO_NAME	**Value:** 1723
SCHEMAP_NOTHING_TO_PARSE	**Value:** 1758
SCHEMAP_NOTYPE_NOREF	**Value:** 1724
SCHEMAP_NOT_DETERMINISTIC	**Value:** 3070
SCHEMAP_NOT_SCHEMA	**Value:** 1772

continued on next page

Name	Description
SCHEMAP_NO_XMLNS	**Value:** 3056
SCHEMAP_NO_XSI	**Value:** 3057
SCHEMAP_PREFIX_UNDEFINED	**Value:** 1700
SCHEMAP_P_PROPS_CORRECT_1	**Value:** 3042
SCHEMAP_P_PROPS_CORRECT_2_1	**Value:** 3043
SCHEMAP_P_PROPS_CORRECT_2_2	**Value:** 3044
SCHEMAP_RECURSIVE	**Value:** 1775
SCHEMAP_REDEFINED_ATTR	**Value:** 1764
SCHEMAP_REDEFINED_ATTRGROUP	**Value:** 1763
SCHEMAP_REDEFINED_ELEMENT	**Value:** 1762
SCHEMAP_REDEFINED_GROUP	**Value:** 1760
SCHEMAP_REDEFINED_NOTATION	**Value:** 1765
SCHEMAP_REDEFINED_TYPE	**Value:** 1761
SCHEMAP_REF_AND_CONTENT	**Value:** 1781
SCHEMAP_REF_AND_SUBTYPE	**Value:** 1725
SCHEMAP_REGEXP_INVALID	**Value:** 1756
SCHEMAP_RESTRICTION_NONAME_NOREF	**Value:** 1726
SCHEMAP_S4S_ATTR_INVALID_VALUE	**Value:** 3037
SCHEMAP_S4S_ATTR_MISSING	**Value:** 3036
SCHEMAP_S4S_ATTR_NOT_ALLOWED	**Value:** 3035
SCHEMAP_S4S_ELEM_MISSING	**Value:** 3034
SCHEMAP_S4S_ELEM_NOT_ALLOWED	**Value:** 3033
SCHEMAP_SIMPLETYPE_NONAME	**Value:** 1727

continued on next page

Name	Description
SCHEMAP_SRC_ATTRIBUTE_1	**Value:** 3051
SCHEMAP_SRC_ATTRIBUTE_2	**Value:** 3052
SCHEMAP_SRC_ATTRIBUTE_3_1	**Value:** 3053
SCHEMAP_SRC_ATTRIBUTE_3_2	**Value:** 3054
SCHEMAP_SRC_ATTRIBUTE_4	**Value:** 3055
SCHEMAP_SRC_ATTRIBUTE_GROUP_1	**Value:** 3071
SCHEMAP_SRC_ATTRIBUTE_GROUP_2	**Value:** 3072
SCHEMAP_SRC_ATTRIBUTE_GROUP_3	**Value:** 3073
SCHEMAP_SRC_CT_1	**Value:** 3076
SCHEMAP_SRC_ELEMENT_1	**Value:** 3038
SCHEMAP_SRC_ELEMENT_2_1	**Value:** 3039
SCHEMAP_SRC_ELEMENT_2_2	**Value:** 3040
SCHEMAP_SRC_ELEMENT_3	**Value:** 3041
SCHEMAP_SRC_IMPORT	**Value:** 3082
SCHEMAP_SRC_IMPORT_1_1	**Value:** 3064
SCHEMAP_SRC_IMPORT_1_2	**Value:** 3065
SCHEMAP_SRC_IMPORT_2	**Value:** 3066
SCHEMAP_SRC_IMPORT_2_1	**Value:** 3067
SCHEMAP_SRC_IMPORT_2_2	**Value:** 3068
SCHEMAP_SRC_IMPORT_3_1	**Value:** 1795
SCHEMAP_SRC_IMPORT_3_2	**Value:** 1796
SCHEMAP_SRC_INCLUDE	**Value:** 3050

continued on next page

Name	Description
SCHEMAP_SRC_LIST_ITEMTYPE_OR_SIMPLETYPE	**Value:** 3006
SCHEMAP_SRC_REDEFINE	**Value:** 3081
SCHEMAP_SRC_RESOLVE	**Value:** 3004
SCHEMAP_SRC_RESTRICTION_BASE_OR_SIMPLETYPE	**Value:** 3005
SCHEMAP_SRC_SIMPLE_TYPE_1	**Value:** 3000
SCHEMAP_SRC_SIMPLE_TYPE_2	**Value:** 3001
SCHEMAP_SRC_SIMPLE_TYPE_3	**Value:** 3002
SCHEMAP_SRC_SIMPLE_TYPE_4	**Value:** 3003
SCHEMAP_SRC_UNION_MEMBERTYPES_OR_SIMPLETYPES	**Value:** 3007
SCHEMAP_ST_PROPS_CORRECT_1	**Value:** 3008
SCHEMAP_ST_PROPS_CORRECT_2	**Value:** 3009
SCHEMAP_ST_PROPS_CORRECT_3	**Value:** 3010
SCHEMAP_SUPERNUMEROUS_LIST_ITEM_TYPE	**Value:** 1776
SCHEMAP_TYPE_AND_SUBTYPE	**Value:** 1728
SCHEMAP_UNION_NOT_EXPRESSIBLE	**Value:** 1794
SCHEMAP_UNKNOWN_ALL_CHILD	**Value:** 1729
SCHEMAP_UNKNOWN_ANYATTRIBUTE_CHILD	**Value:** 1730
SCHEMAP_UNKNOWN_ATTRGRP_CHILD	**Value:** 1732
SCHEMAP_UNKNOWN_ATTRIBUTE_GROUP	**Value:** 1733
SCHEMAP_UNKNOWN_ATTR_CHILD	**Value:** 1731

continued on next page

Name	Description
SCHEMAP_UNKNOWN_BASE_TYPE	**Value:** 1734
SCHEMAP_UNKNOWN_CHOICE_CHILD	**Value:** 1735
SCHEMAP_UNKNOWN_COMPLEXCONTENT_CHILD	**Value:** 1736
SCHEMAP_UNKNOWN_COMPLEXTYPE_CHILD	**Value:** 1737
SCHEMAP_UNKNOWN_ELEM_CHILD	**Value:** 1738
SCHEMAP_UNKNOWN_EXTENSION_CHILD	**Value:** 1739
SCHEMAP_UNKNOWN_FACET_CHILD	**Value:** 1740
SCHEMAP_UNKNOWN_FACET_TYPE	**Value:** 1741
SCHEMAP_UNKNOWN_GROUP_CHILD	**Value:** 1742
SCHEMAP_UNKNOWN_IMPORT_CHILD	**Value:** 1743
SCHEMAP_UNKNOWN_INCLUDE_CHILD	**Value:** 1769
SCHEMAP_UNKNOWN_LIST_CHILD	**Value:** 1744
SCHEMAP_UNKNOWN_MEMBER_TYPE	**Value:** 1773
SCHEMAP_UNKNOWN_NOTATION_CHILD	**Value:** 1745
SCHEMAP_UNKNOWN_PREFIX	**Value:** 1767
SCHEMAP_UNKNOWN_PROCESSCONTENT_CHILD	**Value:** 1746
SCHEMAP_UNKNOWN_REF	**Value:** 1747
SCHEMAP_UNKNOWN_RESTRICTION_CHILD	**Value:** 1748
SCHEMAP_UNKNOWN_SCHEMAS_CHILD	**Value:** 1749
SCHEMAP_UNKNOWN_SEQUENCE_CHILD	**Value:** 1750

continued on next page

Name	Description
SCHEMAP_UNKNOWN_SIMPLECONTENT_CHILD	**Value:** 1751
SCHEMAP_UNKNOWN_SIMPLETYPE_CHILD	**Value:** 1752
SCHEMAP_UNKNOWN_TYPE	**Value:** 1753
SCHEMAP_UNKNOWN_UNION_CHILD	**Value:** 1754
SCHEMAP_WARN_ATTR_POINTLESS_PROH	**Value:** 3086
SCHEMAP_WARN_ATTR_REDECL_PROH	**Value:** 3085
SCHEMAP_WARN_SKIP_SCHEMA	**Value:** 3083
SCHEMAP_WARN_UNLOCATED_SCHEMA	**Value:** 3084
SCHEMAP_WILDCARD_INVALID_NS_MEMBER	**Value:** 1792
SCHEMATRONV_ASSERT	**Value:** 4000
SCHEMATRONV_REPORT	**Value:** 4001
SCHEMAV_ATTRINVALID	**Value:** 1821
SCHEMAV_ATTRUNKNOWN	**Value:** 1820
SCHEMAV_CONSTRUCT	**Value:** 1817
SCHEMAV_CVC_ATTRIBUTE_1	**Value:** 1861
SCHEMAV_CVC_ATTRIBUTE_2	**Value:** 1862
SCHEMAV_CVC_ATTRIBUTE_3	**Value:** 1863
SCHEMAV_CVC_ATTRIBUTE_4	**Value:** 1864
SCHEMAV_CVC_AU	**Value:** 1874
SCHEMAV_CVC_COMPLEX_TYPE_1	**Value:** 1873
SCHEMAV_CVC_COMPLEX_TYPE_2_1	**Value:** 1841

continued on next page

Name	Description
SCHEMAV_CVC_COMPLEX_TYPE_2_2	**Value:** 1842
SCHEMAV_CVC_COMPLEX_TYPE_2_3	**Value:** 1843
SCHEMAV_CVC_COMPLEX_TYPE_2_4	**Value:** 1844
SCHEMAV_CVC_COMPLEX_TYPE_3_1	**Value:** 1865
SCHEMAV_CVC_COMPLEX_TYPE_3_2_1	**Value:** 1866
SCHEMAV_CVC_COMPLEX_TYPE_3_2_2	**Value:** 1867
SCHEMAV_CVC_COMPLEX_TYPE_4	**Value:** 1868
SCHEMAV_CVC_COMPLEX_TYPE_5_1	**Value:** 1869
SCHEMAV_CVC_COMPLEX_TYPE_5_2	**Value:** 1870
SCHEMAV_CVC_DATATYPE_VALID_1_2_1	**Value:** 1824
SCHEMAV_CVC_DATATYPE_VALID_1_2_2	**Value:** 1825
SCHEMAV_CVC_DATATYPE_VALID_1_2_3	**Value:** 1826
SCHEMAV_CVC_ELT_1	**Value:** 1845
SCHEMAV_CVC_ELT_2	**Value:** 1846
SCHEMAV_CVC_ELT_3_1	**Value:** 1847
SCHEMAV_CVC_ELT_3_2_1	**Value:** 1848
SCHEMAV_CVC_ELT_3_2_2	**Value:** 1849
SCHEMAV_CVC_ELT_4_1	**Value:** 1850
SCHEMAV_CVC_ELT_4_2	**Value:** 1851
SCHEMAV_CVC_ELT_4_3	**Value:** 1852
SCHEMAV_CVC_ELT_5_1_1	**Value:** 1853

continued on next page

Name	Description
SCHEMAV_CVC_ELT_5_1_2	**Value:** 1854
SCHEMAV_CVC_ELT_5_2_1	**Value:** 1855
SCHEMAV_CVC_ELT_5_2_2_1	**Value:** 1856
SCHEMAV_CVC_ELT_5_2_2_2_1	**Value:** 1857
SCHEMAV_CVC_ELT_5_2_2_2_2	**Value:** 1858
SCHEMAV_CVC_ELT_6	**Value:** 1859
SCHEMAV_CVC_ELT_7	**Value:** 1860
SCHEMAV_CVC_ENUMERATION_VALID	**Value:** 1840
SCHEMAV_CVC_FACET_VALID	**Value:** 1829
SCHEMAV_CVC_FRACTIONDIGITS_VALID	**Value:** 1838
SCHEMAV_CVC_IDC	**Value:** 1877
SCHEMAV_CVC_LENGTH_VALID	**Value:** 1830
SCHEMAV_CVC_MAXEXCLUSIVE_VALID	**Value:** 1836
SCHEMAV_CVC_MAXINCLUSIVE_VALID	**Value:** 1834
SCHEMAV_CVC_MAXLENGTH_VALID	**Value:** 1832
SCHEMAV_CVC_MINEXCLUSIVE_VALID	**Value:** 1835
SCHEMAV_CVC_MININCLUSIVE_VALID	**Value:** 1833
SCHEMAV_CVC_MINLENGTH_VALID	**Value:** 1831
SCHEMAV_CVC_PATTERN_VALID	**Value:** 1839
SCHEMAV_CVC_TOTALDIGITS_VALID	**Value:** 1837
SCHEMAV_CVC_TYPE_1	**Value:** 1875
SCHEMAV_CVC_TYPE_2	**Value:** 1876
SCHEMAV_CVC_TYPE_3_1_1	**Value:** 1827

continued on next page

Name	Description
SCHEMAV_CVC_TYPE_3_1_2	**Value:** 1828
SCHEMAV_CVC_WILDCARD	**Value:** 1878
SCHEMAV_DOCUMENT_ELEMENT_MISSING	**Value:** 1872
SCHEMAV_ELEMCONT	**Value:** 1810
SCHEMAV_ELEMENT_CONTENT	**Value:** 1871
SCHEMAV_EXTRACONTENT	**Value:** 1813
SCHEMAV_FACET	**Value:** 1823
SCHEMAV_HAVEDEFAULT	**Value:** 1811
SCHEMAV_INTERNAL	**Value:** 1818
SCHEMAV_INVALIDATTR	**Value:** 1814
SCHEMAV_INVALIDELEM	**Value:** 1815
SCHEMAV_ISABSTRACT	**Value:** 1808
SCHEMAV_MISC	**Value:** 1879
SCHEMAV_MISSING	**Value:** 1804
SCHEMAV_NOROLLBACK	**Value:** 1807
SCHEMAV_NOROOT	**Value:** 1801
SCHEMAV_NOTDETERMINIST	**Value:** 1816
SCHEMAV_NOTEMPTY	**Value:** 1809
SCHEMAV_NOTNILLABLE	**Value:** 1812
SCHEMAV_NOTSIMPLE	**Value:** 1819
SCHEMAV_NOTTOPLEVEL	**Value:** 1803
SCHEMAV_NOTYPE	**Value:** 1806
SCHEMAV_UNDECLAREDELEM	**Value:** 1802
SCHEMAV_VALUE	**Value:** 1822
SCHEMAV_WRONGELEM	**Value:** 1805
TREE_INVALID_DEC	**Value:** 1301

continued on next page

Name	Description
TREE_INVALID_HEX	**Value:** 1300
TREE_NOT_UTF8	**Value:** 1303
TREE_UNTERMINATED_ENTITY	**Value:** 1302
WAR_CATALOG_PI	**Value:** 93
WAR_ENTITY_REDEFINED	**Value:** 107
WAR_LANG_VALUE	**Value:** 98
WAR_NS_COLUMN	**Value:** 106
WAR_NS_URI	**Value:** 99
WAR_NS_URI_RELATIVE	**Value:** 100
WAR_SPACE_VALUE	**Value:** 102
WAR_UNDECLARED_ENTITY	**Value:** 27
WAR_UNKNOWN_VERSION	**Value:** 97
XINCLUDE_BUILD_FAILED	**Value:** 1609
XINCLUDE_DEPRECATED_NS	**Value:** 1617
XINCLUDE_ENTITY_DEF_MISMATCH	**Value:** 1602
XINCLUDE_FALLBACKS_IN_INCLUDE	**Value:** 1615
XINCLUDE_FALLBACK_NOT_IN_INCLUDE	**Value:** 1616
XINCLUDE_FRAGMENT_ID	**Value:** 1618
XINCLUDE_HREF_URI	**Value:** 1605
XINCLUDE_INCLUDE_IN_INCLUDE	**Value:** 1614
XINCLUDE_INVALID_CHAR	**Value:** 1608
XINCLUDE_MULTIPLE_ROOT	**Value:** 1611
XINCLUDE_NO_FALLBACK	**Value:** 1604
XINCLUDE_NO_HREF	**Value:** 1603
XINCLUDE_PARSE_VALUE	**Value:** 1601
XINCLUDE_RECURSION	**Value:** 1600

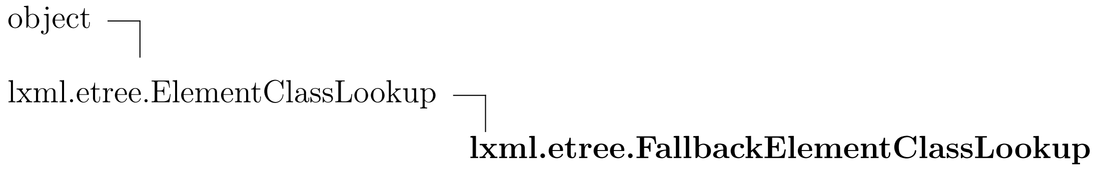
continued on next page

Name	Description
XINCLUDE_TEXT_DOCUMENT	**Value:** 1607
XINCLUDE_TEXT_FRAGMENT	**Value:** 1606
XINCLUDE_UNKNOWN_ENCODING	**Value:** 1610
XINCLUDE_XPTR_FAILED	**Value:** 1612
XINCLUDE_XPTR_RESULT	**Value:** 1613
XPATH_ENCODING_ERROR	**Value:** 1220
XPATH_EXPRESSION_OK	**Value:** 1200
XPATH_EXPR_ERROR	**Value:** 1207
XPATH_INVALID_ARITY	**Value:** 1212
XPATH_INVALID_CHAR_ERROR	**Value:** 1221
XPATH_INVALID_CTXT_POSITION	**Value:** 1214
XPATH_INVALID_CTXT_SIZE	**Value:** 1213
XPATH_INVALID_OPERAND	**Value:** 1210
XPATH_INVALID_PREDICATE_ERROR	**Value:** 1206
XPATH_INVALID_TYPE	**Value:** 1211
XPATH_MEMORY_ERROR	**Value:** 1215
XPATH_NUMBER_ERROR	**Value:** 1201
XPATH_START_LITERAL_ERROR	**Value:** 1203
XPATH_UNCLOSED_ERROR	**Value:** 1208
XPATH_UNDEF_PREFIX_ERROR	**Value:** 1219
XPATH_UNDEF_VARIABLE_ERROR	**Value:** 1205
XPATH_UNFINISHED_LITERAL_ERROR	**Value:** 1202
XPATH_UNKNOWN_FUNC_ERROR	**Value:** 1209

continued on next page

Name	Description
XPATH_VARIABLE_REF_ERROR	**Value:** `1204`
XPTR_CHILDSEQ_START	**Value:** `1901`
XPTR_EVAL_FAILED	**Value:** `1902`
XPTR_EXTRA_OBJECTS	**Value:** `1903`
XPTR_RESOURCE_ERROR	**Value:** `1217`
XPTR_SUB_RESOURCE_ERROR	**Value:** `1218`
XPTR_SYNTAX_ERROR	**Value:** `1216`
XPTR_UNKNOWN_SCHEME	**Value:** `1900`

B.6.29 Class FallbackElementClassLookup


object
lxml.etree.ElementClassLookup
lxml.etree.FallbackElementClassLookup

Known Subclasses: lxml.etree.AttributeBasedElementClassLookup, lxml.etree.CustomElementClass lxml.etree.ElementNamespaceClassLookup, lxml.etree.ParserBasedElementClassLookup, lxml.etree.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__

set_fallback(*self*, *lookup*)

Sets the fallback scheme for this lookup method.

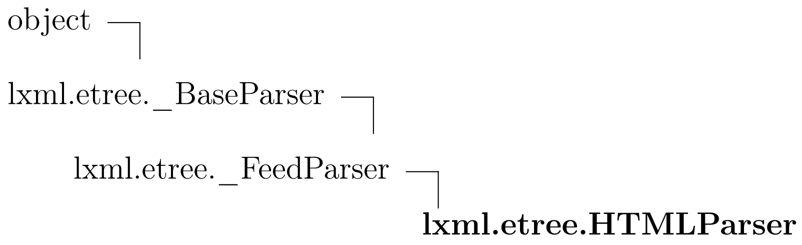
Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

Properties

Name	Description
fallback	
Inherited from object __class__	

B.6.30 Class HTMLParser



Known Subclasses: lxml.html.HTMLParser

HTMLParser(self, encoding=None, remove_blank_text=False, remove_comments=False, remove_pis=False, strip_cdata=True, no_network=True, target=None, XMLSchema schema=None, recover=True, compact=True)

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
- strip_cdata - replace CDATA sections by normal text content (default: True)
- 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

__init__(*self*, *encoding*=`None`, *remove_blank_text*=`False`, *remove_comments*=`False`, *remove_pis*=`False`, *strip_cdata*=`True`, *no_network*=`True`, *target*=`None`, *XMLSchema schema*=`None`, *recover*=`True`, *compact*=`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._FeedParser

close(), feed()

Inherited from lxml.etree._BaseParser

copy(), makeelement(), setElementClassLookup(), set_element_class_lookup()

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

Properties

Name	Description
Inherited from lxml.etree._FeedParser	

continued on next page

Name	Description
feed_error_log	
Inherited from lxml.etree._BaseParser error_log, resolvers, version	
Inherited from object __class__	

B.6.31 Class LxmlError

object

exceptions.BaseException

exceptions.Exception

lxml.etree.Error

lxml.etree.LxmlError

Known Subclasses: lxml.etree.LxmlSyntaxError, lxml.sax.SaxError, lxml.etree.C14NError, lxml.etree.DTDError, lxml.etree.DocumentInvalid, lxml.etree.LxmlRegistryError, lxml.etree.ParserErro lxml.etree.RelaxNGError, lxml.etree.SchematronError, lxml.etree.SerialisationError, lxml.etree.XIncluc lxml.etree.XMLSchemaError, lxml.etree.XPathError, lxml.etree.XSLTError

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__ extit(inherited documentation)

Inherited from exceptions.Exception

__new__()

Inherited from exceptions.BaseException

__delattr__(), __getattribute__(), __getitem__(), __getslice__(), __reduce__(), __repr__(), __setattr__(), __setstate__(), __str__(), __unicode__()

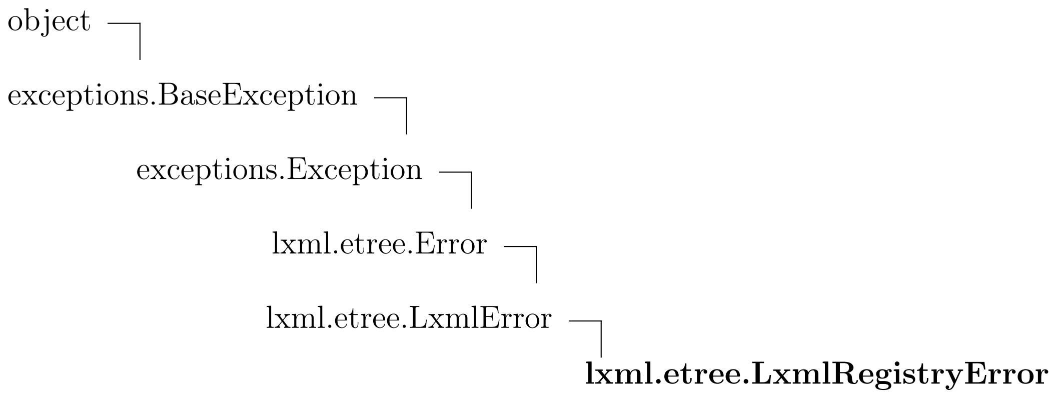
Inherited from object

__format__(), __hash__(), __reduce_ex__(), __sizeof__(), __subclasshook__()

Properties

Name	Description
Inherited from exceptions.BaseException args, message	
Inherited from object __class__	

B.6.32 Class LxmlRegistryError



Known Subclasses: lxml.etree.NamespaceRegistryError

Base class of lxml registry errors.

Methods

Inherited from lxml.etree.LxmlError(Section B.6.31)

__init__()

Inherited from exceptions.Exception

__new__()

Inherited from exceptions.BaseException

__delattr__(), __getattribute__(), __getitem__(), __getslice__(), __reduce__(), __repr__(), __setattr__(), __setstate__(), __str__(), __unicode__()

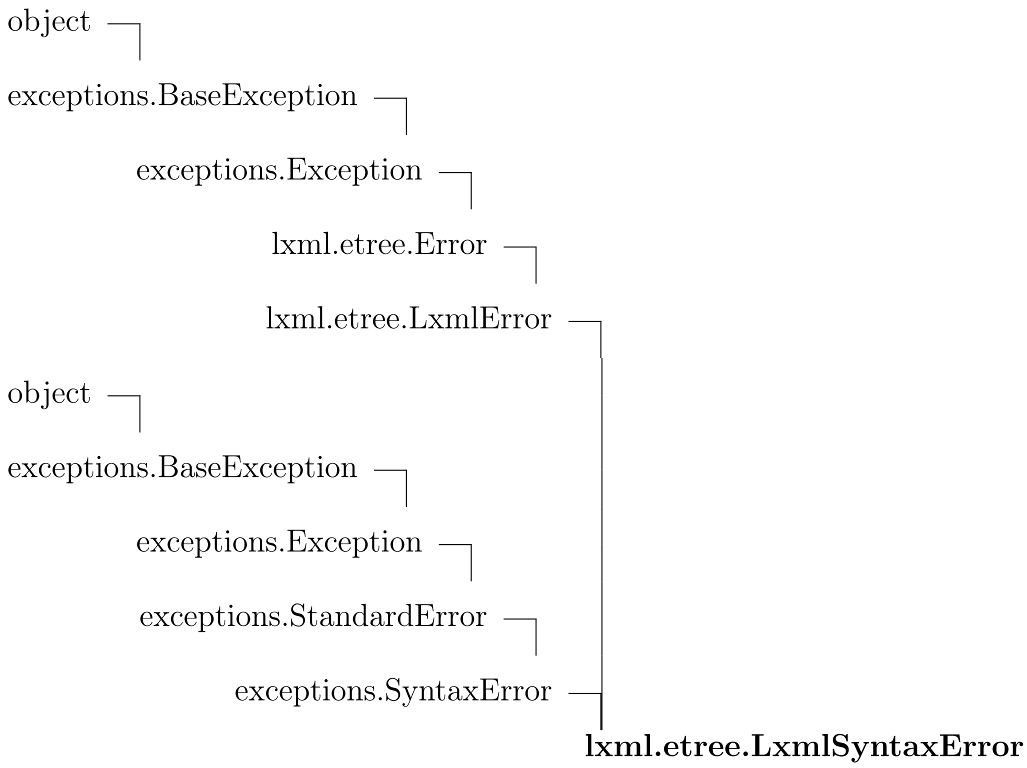
Inherited from object

__format__(), __hash__(), __reduce_ex__(), __sizeof__(), __subclasshook__()

Properties

Name	Description
Inherited from exceptions.BaseException args, message	
Inherited from object __class__	

B.6.33 Class LxmlSyntaxError



Known Subclasses: lxml.ElementInclude.FatalIncludeError, lxml.etree.ParseError, lxml.etree.XPathS

Base class for all syntax errors.

Methods

Inherited from lxml.etree.LxmlError(Section B.6.31)

__init__()

Inherited from exceptions.SyntaxError

__new__(), __str__()

Inherited from exceptions.BaseException

__delattr__(), __getattribute__(), __getitem__(), __getslice__(), __reduce__(), __repr__(), __setattr__(), __setstate__(), __unicode__()

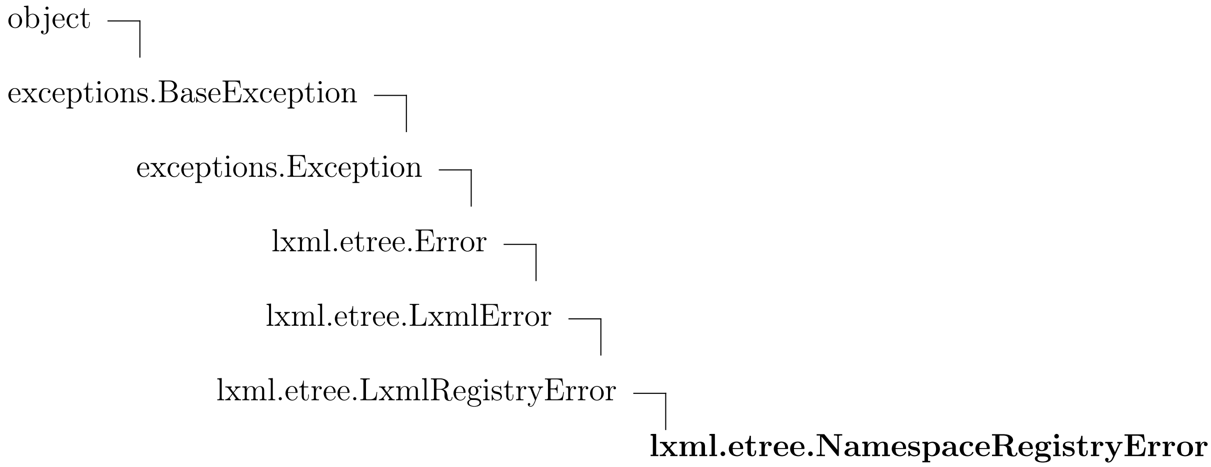
Inherited from object

__format__(), __hash__(), __reduce_ex__(), __sizeof__(), __subclasshook__()

Properties

Name	Description
Inherited from exceptions.SyntaxError filename, lineno, msg, offset, print_file_and_line, text	
Inherited from exceptions.BaseException args, message	
Inherited from object __class__	

B.6.34 Class NamespaceRegistryError



Error registering a namespace extension.

Methods

Inherited from lxml.etree.LxmlError(Section B.6.31)

__init__()

Inherited from exceptions.Exception

__new__()

Inherited from exceptions.BaseException

__delattr__(), __getattribute__(), __getitem__(), __getslice__(), __reduce__(), __repr__(), __setattr__(), __setstate__(), __str__(), __unicode__()

Inherited from object

__format__(), __hash__(), __reduce_ex__(), __sizeof__(), __subclasshook__()

Properties

Name	Description
Inherited from exceptions.BaseException args, message	
Inherited from object __class__	

B.6.35 Class PIBase

object
lxml.etree._Element
lxml.etree.__ContentOnlyElement
lxml.etree._ProcessingInstruction
lxml.etree.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()

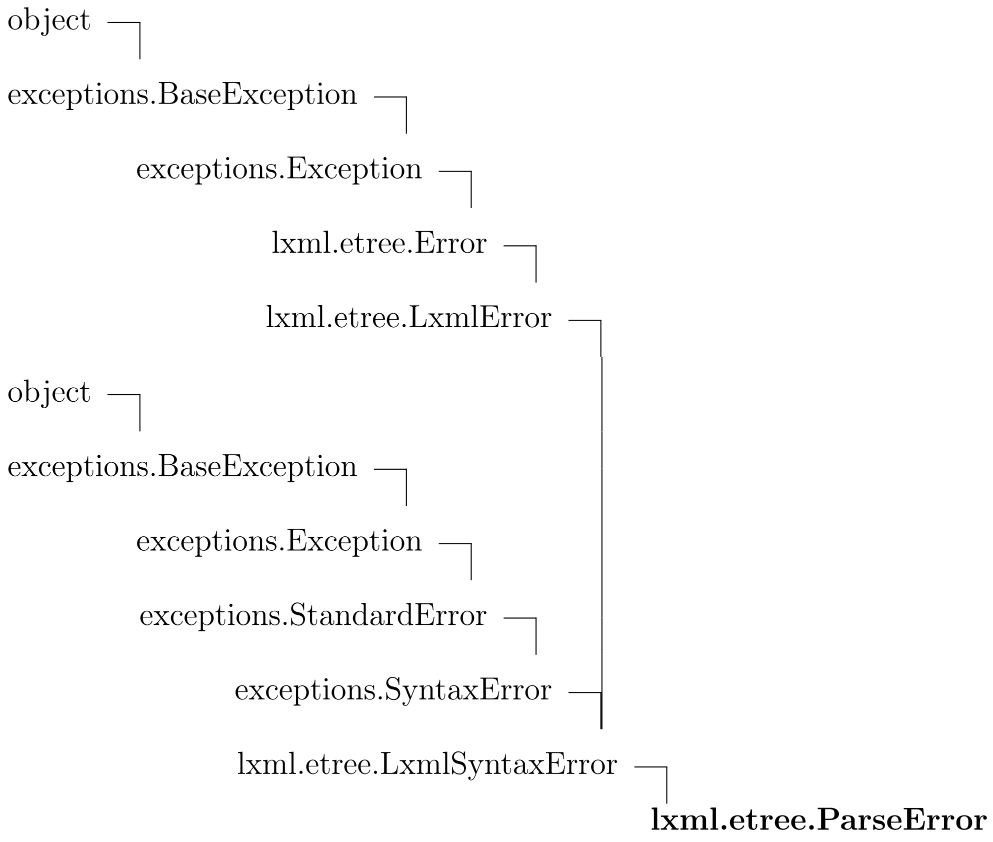
Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __init__(), __reduce__(), __reduce_ex__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

Properties

Name	Description
Inherited from lxml.etree._ProcessingInstruction tag, target	
Inherited from lxml.etree.__ContentOnlyElement attrib, text	
Inherited from lxml.etree._Element base, nsmap, prefix, sourceline, tail	
Inherited from object __class__	

B.6.36 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.6.31)

__init__()

Inherited from exceptions.SyntaxError

__new__(), __str__()

Inherited from exceptions.BaseException

__delattr__(), __getattribute__(), __getitem__(), __getslice__(), __reduce__(), __repr__(), __setattr__(), __setstate__(), __unicode__()

Inherited from object

__format__(), __hash__(), __reduce_ex__(), __sizeof__(), __subclasshook__()

Properties

Name	Description
Inherited from exceptions.SyntaxError filename, lineno, msg, offset, print_file_and_line, text	
Inherited from exceptions.BaseException args, message	
Inherited from object __class__	

B.6.37 Class ParserBasedElementClassLookup

object
lxml.etree.ElementClassLookup
lxml.etree.FallbackElementClassLookup
lxml.etree.ParserBasedElementClassLookup

ParserBasedElementClassLookup(self, fallback=None) Element class lookup based on the XML parser.

Methods

__new__(*T*, *S*, *...*)

Return Value

a new object with type S, a subtype of T

Overrides: object.__new__

Inherited from lxml.etree.FallbackElementClassLookup(Section B.6.29)

__init__(), set_fallback()

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

Properties

Name	Description
Inherited from lxml.etree.FallbackElementClassLookup (Section B.6.29) fallback	
Inherited from object	

continued on next page

Name	Description
__class__	

B.6.38 Class ParserError

object

exceptions.BaseException

exceptions.Exception

lxml.etree.Error

lxml.etree.LxmlError

lxml.etree.ParserError

Internal lxml parser error.

Methods

Inherited from lxml.etree.LxmlError(Section B.6.31)

__init__()

Inherited from exceptions.Exception

__new__()

Inherited from exceptions.BaseException

__delattr__(), __getattribute__(), __getitem__(), __getslice__(), __reduce__(), __repr__(), __setattr__(), __setstate__(), __str__(), __unicode__()

Inherited from object

__format__(), __hash__(), __reduce_ex__(), __sizeof__(), __subclasshook__()

Properties

Name	Description
Inherited from exceptions.BaseException args, message	
Inherited from object __class__	

B.6.39 Class PyErrorLog

object

lxml.etree._BaseErrorLog

lxml.etree.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

__init__(*self*, *logger_name*=`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__

copy(*...*)

Dummy method that returns an empty error log. Overrides: lxml.etree._BaseErrorLog.copy

log(*self*, *log_ entry*, *message*, **args*)

Called by the .receive() method to log a _LogEntry instance to the Python logging system. This handles the error level mapping.

In the default implementation, the `message` argument receives a complete log line, and there are no further `args`. To change the message format, it is best to override the .receive() method instead of this one.

receive(*self*, *log_ entry*)

Receive a _LogEntry instance from the logging system. Calls the .log() method with appropriate parameters:

```
    self.log(log_entry, repr(log_entry))
```

You can override this method to provide your own log output format.

Inherited from lxml.etree._ BaseErrorLog

__repr__()

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __reduce__(), __reduce_ex__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

Properties

Name	Description
level_map	
Inherited from lxml.etree._ BaseErrorLog last_error	
Inherited from object __class__	

B.6.40 Class PythonElementClassLookup

object

lxml.etree.ElementClassLookup

lxml.etree.FallbackElementClassLookup

lxml.etree.PythonElementClassLookup

PythonElementClassLookup(self, fallback=None) Element class lookup based on a subclass method.

This class lookup scheme allows access to the entire XML tree in read-only mode. To use it, re-implement the `lookup(self, doc, root)` method in a subclass:

```
>>> 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
```

If you return None from this method, the fallback will be called.

The first argument is the opaque document instance that contains the Element. The second argument 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.

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.

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).

See http://codespeak.net/lxml/element_classes.html

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.6.29)

__init__(), set_fallback()

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

Properties

Name	Description
Inherited from lxml.etree.FallbackElementClassLookup (Section B.6.29) fallback	
Inherited from object __class__	

B.6.41 Class QName

object ─┐
lxml.etree.QName

QName(text_or_uri, tag=None)

QName wrapper for qualified XML names.

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. The `namespace` and `localname` properties hold the respective parts of the tag name.

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__ extit(inherited documentation)

Inherited from object

__delattr__(), __format__(), __getattribute__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __subclasshook__()

Properties

Name	Description
localname	
namespace	
text	
Inherited from object __class__	

B.6.42 Class RelaxNG

object
lxml.etree._Validator
lxml.etree.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.

__init__(*self*, *etree*=`None`, *file*=`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._Validator

assertValid(), assert_(), validate()

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

Properties

Name	Description
Inherited from lxml.etree._Validator error_log	
Inherited from object __class__	

B.6.43 Class RelaxNGError

object
exceptions.BaseException
exceptions.Exception
lxml.etree.Error
lxml.etree.LxmlError
lxml.etree.RelaxNGError

Known Subclasses: lxml.etree.RelaxNGParseError, lxml.etree.RelaxNGValidateError

Base class for RelaxNG errors.

Methods

Inherited from lxml.etree.LxmlError(Section B.6.31)

__init__()

Inherited from exceptions.Exception

__new__()

Inherited from exceptions.BaseException

__delattr__(), __getattribute__(), __getitem__(), __getslice__(), __reduce__(), __repr__(), __setattr__(), __setstate__(), __str__(), __unicode__()

Inherited from object

__format__(), __hash__(), __reduce_ex__(), __sizeof__(), __subclasshook__()

Properties

Name	Description
Inherited from exceptions.BaseException args, message	
Inherited from object __class__	

B.6.44 Class RelaxNGErrorTypes

Libxml2 RelaxNG error types

Class Variables

Name	Description
RELAXNG_ERR_ATTREXTRANS	**Value:** 20
RELAXNG_ERR_ATTRNAME	**Value:** 14
RELAXNG_ERR_ATTRNONS	**Value:** 16
RELAXNG_ERR_ATTRVALID	**Value:** 24
RELAXNG_ERR_ATTRWRONGNS	**Value:** 18
RELAXNG_ERR_CONTENTVALID	**Value:** 25

continued on next page

Name	Description
RELAXNG_ERR_DATAELEM	**Value:** 28
RELAXNG_ERR_DATATYPE	**Value:** 31
RELAXNG_ERR_DUPID	**Value:** 4
RELAXNG_ERR_ELEMEXTRANS	**Value:** 19
RELAXNG_ERR_ELEMNAME	**Value:** 13
RELAXNG_ERR_ELEMNONS	**Value:** 15
RELAXNG_ERR_ELEMNOTEMPTY	**Value:** 21
RELAXNG_ERR_ELEMWRONG	**Value:** 38
RELAXNG_ERR_ELEMWRONGNS	**Value:** 17
RELAXNG_ERR_EXTRACONTENT	**Value:** 26
RELAXNG_ERR_EXTRADATA	**Value:** 35
RELAXNG_ERR_INTEREXTRA	**Value:** 12
RELAXNG_ERR_INTERNAL	**Value:** 37
RELAXNG_ERR_INTERNODATA	**Value:** 10
RELAXNG_ERR_INTERSEQ	**Value:** 11
RELAXNG_ERR_INVALIDATTR	**Value:** 27
RELAXNG_ERR_LACKDATA	**Value:** 36
RELAXNG_ERR_LIST	**Value:** 33
RELAXNG_ERR_LISTELEM	**Value:** 30
RELAXNG_ERR_LISTEMPTY	**Value:** 9
RELAXNG_ERR_LISTEXTRA	**Value:** 8
RELAXNG_ERR_MEMORY	**Value:** 1
RELAXNG_ERR_NODEFINE	**Value:** 7

continued on next page

Name	Description
RELAXNG_ERR_NOELEM	**Value:** 22
RELAXNG_ERR_NOGRAMMAR	**Value:** 34
RELAXNG_ERR_NOSTATE	**Value:** 6
RELAXNG_ERR_NOTELEM	**Value:** 23
RELAXNG_ERR_TEXTWRONG	**Value:** 39
RELAXNG_ERR_TYPE	**Value:** 2
RELAXNG_ERR_TYPECMP	**Value:** 5
RELAXNG_ERR_TYPEVAL	**Value:** 3
RELAXNG_ERR_VALELEM	**Value:** 29
RELAXNG_ERR_VALUE	**Value:** 32
RELAXNG_OK	**Value:** 0

B.6.45 Class RelaxNGParseError

object

exceptions.BaseException

exceptions.Exception

lxml.etree.Error

lxml.etree.LxmlError

lxml.etree.RelaxNGError

lxml.etree.RelaxNGParseError

Error while parsing an XML document as RelaxNG.

Methods

Inherited from lxml.etree.LxmlError(Section B.6.31)

__init__()

Inherited from exceptions.Exception

__new__()

Inherited from exceptions.BaseException

__delattr__(), __getattribute__(), __getitem__(), __getslice__(), __reduce__(), __repr__(), __setattr__(), __setstate__(), __str__(), __unicode__()

Inherited from object

__format__(), __hash__(), __reduce_ex__(), __sizeof__(), __subclasshook__()

Properties

Name	Description
Inherited from exceptions.BaseException args, message	
Inherited from object __class__	

B.6.46 Class RelaxNGValidateError

object
exceptions.BaseException
exceptions.Exception
lxml.etree.Error
lxml.etree.LxmlError
lxml.etree.RelaxNGError
lxml.etree.RelaxNGValidateError

Error while validating an XML document with a RelaxNG schema.

Methods

Inherited from lxml.etree.LxmlError(Section B.6.31)

__init__()

Inherited from exceptions.Exception

__new__()

Inherited from exceptions.BaseException

__delattr__(), __getattribute__(), __getitem__(), __getslice__(), __reduce__(), __repr__(), __setattr__(), __setstate__(), __str__(), __unicode__()

Inherited from object

__format__(), __hash__(), __reduce_ex__(), __sizeof__(), __subclasshook__()

Properties

Name	Description
Inherited from exceptions.BaseException args, message	
Inherited from object __class__	

B.6.47 Class Resolver

object

lxml.etree.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. You can pass the base URL or filename of the file through the `base_url` keyword argument.

Note that using `.resolve_filename()` is more efficient, especially in threaded environments.

resolve_filename(*self*, *filename*, *context*)

Return the name of a parsable file as input document.

Pass filename and context as parameters. You can also pass a URL with an HTTP, FTP or file target.

resolve_string(*self*, *string*, *context*, *base_url*=`None`)

Return a parsable string as input document.

Pass data string and context as parameters. You can pass the source URL or filename through the `base_url` keyword argument.

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __init__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

Properties

Name	Description
Inherited from object __class__	

B.6.48 Class Schematron

object

lxml.etree._Validator

lxml.etree.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.

Schematron is a less well known, but very powerful schema language. The main idea is to use the capabilities of XPath to put restrictions on the structure and the content of XML documents. Here is a simple example:

```
>>> schematron = etree.Schematron(etree.XML(’’’
... <schema xmlns="http://www.ascc.net/xml/schematron" >
...   <pattern name="id is the only permited attribute name">
...     <rule context="*">
...       <report test="@*[not(name()=’id’)]">Attribute
...         <name path="@*[not(name()=’id’)]"/> is forbidden<name/>
...       </report>
...     </rule>
...   </pattern>
... </schema>
... ’’’))

>>> xml = etree.XML(’’’
... <AAA name="aaa">
...   <BBB id="bbb"/>
...   <CCC color="ccc"/>
... </AAA>
... ’’’)

>>> schematron.validate(xml)
0

>>> xml = etree.XML(’’’
... <AAA id="aaa">
...   <BBB id="bbb"/>
...   <CCC/>
... </AAA>
... ’’’)

>>> schematron.validate(xml)
1
```

Schematron was added to libxml2 in version 2.6.21. Before version 2.6.32, however, Schematron lacked support for error reporting other than to stderr. This version is therefore required to retrieve validation warnings and errors in lxml.

Methods

__call__(*self*, *etree*)

Validate doc using Schematron.

Returns true if document is valid, false if not.

__init__(*self*, *etree*=`None`, *file*=`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._Validator

assertValid(), assert_(), validate()

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

Properties

Name	Description
Inherited from lxml.etree._Validator error_log	
Inherited from object __class__	

B.6.49 Class SchematronError

object

exceptions.BaseException

exceptions.Exception

lxml.etree.Error

lxml.etree.LxmlError

lxml.etree.SchematronError

Known Subclasses: lxml.etree.SchematronParseError, lxml.etree.SchematronValidateError

Base class of all Schematron errors.

Methods

Inherited from lxml.etree.LxmlError(Section B.6.31)

__init__()

Inherited from exceptions.Exception

__new__()

Inherited from exceptions.BaseException

__delattr__(), __getattribute__(), __getitem__(), __getslice__(), __reduce__(), __repr__(), __setattr__(), __setstate__(), __str__(), __unicode__()

Inherited from object

__format__(), __hash__(), __reduce_ex__(), __sizeof__(), __subclasshook__()

Properties

Name	Description
Inherited from exceptions.BaseException args, message	
Inherited from object __class__	

B.6.50 Class SchematronParseError

object

exceptions.BaseException

exceptions.Exception

lxml.etree.Error

lxml.etree.LxmlError

lxml.etree.SchematronError

lxml.etree.SchematronParseError

Error while parsing an XML document as Schematron schema.

Methods

Inherited from lxml.etree.LxmlError(Section B.6.31)

__init__()

Inherited from exceptions.Exception

__new__()

Inherited from exceptions.BaseException

__delattr__(), __getattribute__(), __getitem__(), __getslice__(), __reduce__(), __repr__(), __setattr__(), __setstate__(), __str__(), __unicode__()

Inherited from object

__format__(), __hash__(), __reduce_ex__(), __sizeof__(), __subclasshook__()

Properties

Name	Description
Inherited from exceptions.BaseException args, message	
Inherited from object __class__	

B.6.51 Class SchematronValidateError

object
exceptions.BaseException
exceptions.Exception
lxml.etree.Error
lxml.etree.LxmlError
lxml.etree.SchematronError
lxml.etree.SchematronValidateError

Error while validating an XML document with a Schematron schema.

Methods

Inherited from lxml.etree.LxmlError(Section B.6.31)

__init__()

Inherited from exceptions.Exception

__new__()

Inherited from exceptions.BaseException

__delattr__(), __getattribute__(), __getitem__(), __getslice__(), __reduce__(), __repr__(), __setattr__(), __setstate__(), __str__(), __unicode__()

Inherited from object

__format__(), __hash__(), __reduce_ex__(), __sizeof__(), __subclasshook__()

Properties

Name	Description
Inherited from exceptions.BaseException args, message	
Inherited from object __class__	

B.6.52 Class SerialisationError

object

exceptions.BaseException

exceptions.Exception

lxml.etree.Error

lxml.etree.LxmlError

lxml.etree.SerialisationError

A libxml2 error that occurred during serialisation.

Methods

Inherited from lxml.etree.LxmlError(Section B.6.31)

__init__()

Inherited from exceptions.Exception

__new__()

Inherited from exceptions.BaseException

__delattr__(), __getattribute__(), __getitem__(), __getslice__(), __reduce__(), __repr__(), __setattr__(), __setstate__(), __str__(), __unicode__()

Inherited from object

__format__(), __hash__(), __reduce_ex__(), __sizeof__(), __subclasshook__()

Properties

Name	Description
Inherited from exceptions.BaseException args, message	
Inherited from object __class__	

B.6.53 Class SiblingsIterator

object

lxml.etree._ElementTagMatcher

lxml.etree._ElementIterator

lxml.etree.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

__init__(*self*, *node*, *tag*=`None`, *preceding*=`False`)

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._ElementIterator

__iter__(), __next__(), next()

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

Properties

Name	Description
Inherited from object __class__	

B.6.54 Class TreeBuilder

object ┐

lxml.etree._SaxParserTarget ┐

lxml.etree.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*, *nsmap*=`None`)

Opens a new element.

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

Properties

Name	Description
Inherited from object	
__class__	

B.6.55 Class XInclude

object
   lxml.etree.XInclude

XInclude(self) XInclude processor.

Create an instance and call it on an Element to run XInclude processing.

Methods

__call__(*self*, *node*)

__init__(*self*)

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__(), __format__(), __getattribute__(), __hash__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

Properties

Name	Description
error_log	
Inherited from object __class__	

B.6.56 Class XIncludeError

object
exceptions.BaseException
exceptions.Exception
lxml.etree.Error
lxml.etree.LxmlError
lxml.etree.XIncludeError

Error during XInclude processing.

Methods

Inherited from lxml.etree.LxmlError(Section B.6.31)

__init__()

Inherited from exceptions.Exception

__new__()

Inherited from exceptions.BaseException

__delattr__(), __getattribute__(), __getitem__(), __getslice__(), __reduce__(), __repr__(), __setattr__(), __setstate__(), __str__(), __unicode__()

Inherited from object

__format__(), __hash__(), __reduce_ex__(), __sizeof__(), __subclasshook__()

Properties

Name	Description
Inherited from exceptions.BaseException args, message	
Inherited from object __class__	

B.6.57 Class XMLParser

object
lxml.etree._BaseParser
lxml.etree._FeedParser
lxml.etree.XMLParser

Known Subclasses: lxml.etree.ETCompatXMLParser, lxml.html.XHTMLParser

XMLParser(self, encoding=None, attribute_defaults=False, dtd_validation=False, load_dtd=False, no_network=True, ns_clean=False, recover=False, XMLSchema schema=None, remove_blank_text= resolve_entities=True, remove_comments=False, remove_pis=False, strip_cdata=True, target=None, compact=True)

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 DTD validation or attribute default values are requested (unless you additionally provide an XMLSchema from which the default attributes can be read).

Available boolean keyword arguments:

- attribute_defaults - inject default attributes from DTD or XMLSchema
- dtd_validation - validate against a DTD referenced by the document
- 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
- strip_cdata - replace CDATA sections by normal text content (default: True)
- compact - safe memory for short text content (default: True)
- resolve_entities - replace entities by their text value (default: True)
- **huge_tree - disable security restrictions and support very deep trees** and very long text content (only affects libxml2 2.7+)

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*, *encoding*=`None`, *attribute_defaults*=`False`, *dtd_validation*=`False`, *load_dtd*=`False`, *no_network*=`True`, *ns_clean*=`False`, *recover*=`False`, *XMLSchema schema*=`None`, *remove_blank_text*=`False`, *resolve_entities*=`True`, *remove_comments*=`False`, *remove_pis*=`False`, *strip_cdata*=`True`, *target*=`None`, *compact*=`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._FeedParser

close(), feed()

Inherited from lxml.etree._BaseParser

copy(), makeelement(), setElementClassLookup(), set_element_class_lookup()

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

Properties

Name	Description
Inherited from lxml.etree._FeedParser feed_error_log	
Inherited from lxml.etree._BaseParser error_log, resolvers, version	
Inherited from object __class__	

B.6.58 Class XMLSchema

object

lxml.etree._Validator

lxml.etree.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.

Passing the `attribute_defaults` boolean option will make the schema insert default/fixed attributes into validated documents.

Methods

__call__(*self*, *etree*)

Validate doc using XML Schema.

Returns true if document is valid, false if not.

__init__(*self*, *etree*=`None`, *file*=`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._Validator

assertValid(), assert_(), validate()

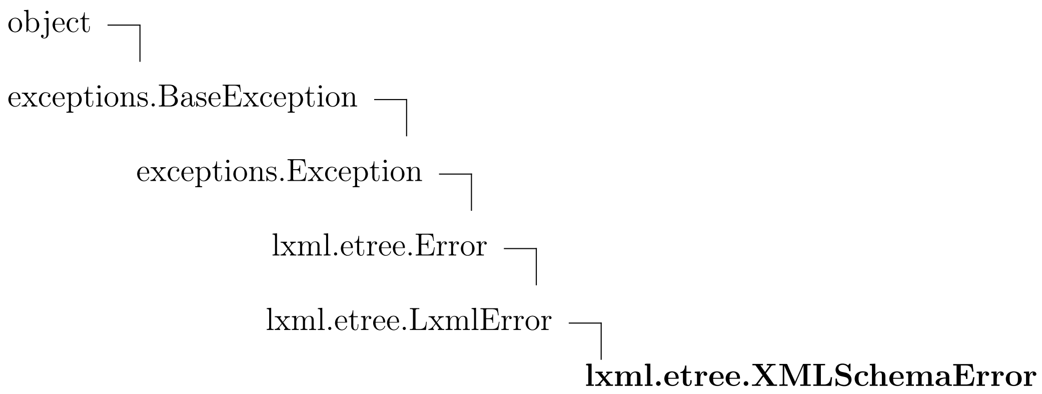
Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

Properties

Name	Description
Inherited from lxml.etree._Validator error_log	
Inherited from object __class__	

B.6.59 Class XMLSchemaError



object

exceptions.BaseException

exceptions.Exception

lxml.etree.Error

lxml.etree.LxmlError

lxml.etree.XMLSchemaError

Known Subclasses: lxml.etree.XMLSchemaParseError, lxml.etree.XMLSchemaValidateError

Base class of all XML Schema errors

Methods

Inherited from lxml.etree.LxmlError(Section B.6.31)

__init__()

Inherited from exceptions.Exception

__new__()

Inherited from exceptions.BaseException

__delattr__(), __getattribute__(), __getitem__(), __getslice__(), __reduce__(), __repr__(), __setattr__(), __setstate__(), __str__(), __unicode__()

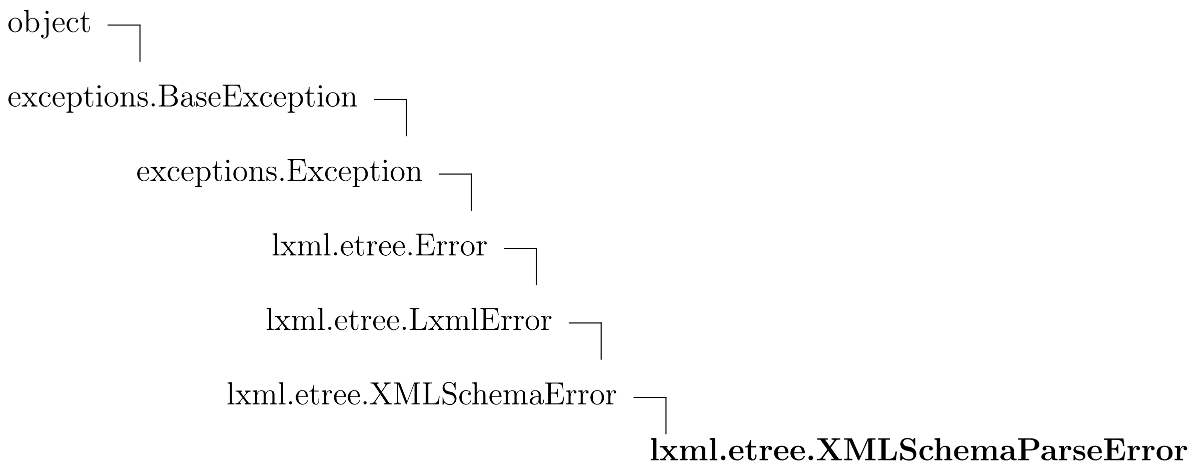
Inherited from object

__format__(), __hash__(), __reduce_ex__(), __sizeof__(), __subclasshook__()

Properties

Name	Description
Inherited from exceptions.BaseException args, message	
Inherited from object __class__	

B.6.60 Class XMLSchemaParseError



Error while parsing an XML document as XML Schema.

Methods

Inherited from lxml.etree.LxmlError(Section B.6.31)

__init__()

Inherited from exceptions.Exception

__new__()

Inherited from exceptions.BaseException

__delattr__(), __getattribute__(), __getitem__(), __getslice__(), __reduce__(), __repr__(), __setattr__(), __setstate__(), __str__(), __unicode__()

Inherited from object

__format__(), __hash__(), __reduce_ex__(), __sizeof__(), __subclasshook__()

Properties

Name	Description
Inherited from exceptions.BaseException args, message	
Inherited from object __class__	

B.6.61 Class XMLSchemaValidateError

object
exceptions.BaseException
exceptions.Exception
lxml.etree.Error
lxml.etree.LxmlError
lxml.etree.XMLSchemaError
lxml.etree.XMLSchemaValidateError

Error while validating an XML document with an XML Schema.

Methods

Inherited from lxml.etree.LxmlError(Section B.6.31)

__init__()

Inherited from exceptions.Exception

__new__()

Inherited from exceptions.BaseException

__delattr__(), __getattribute__(), __getitem__(), __getslice__(), __reduce__(), __repr__(), __setattr__(), __setstate__(), __str__(), __unicode__()

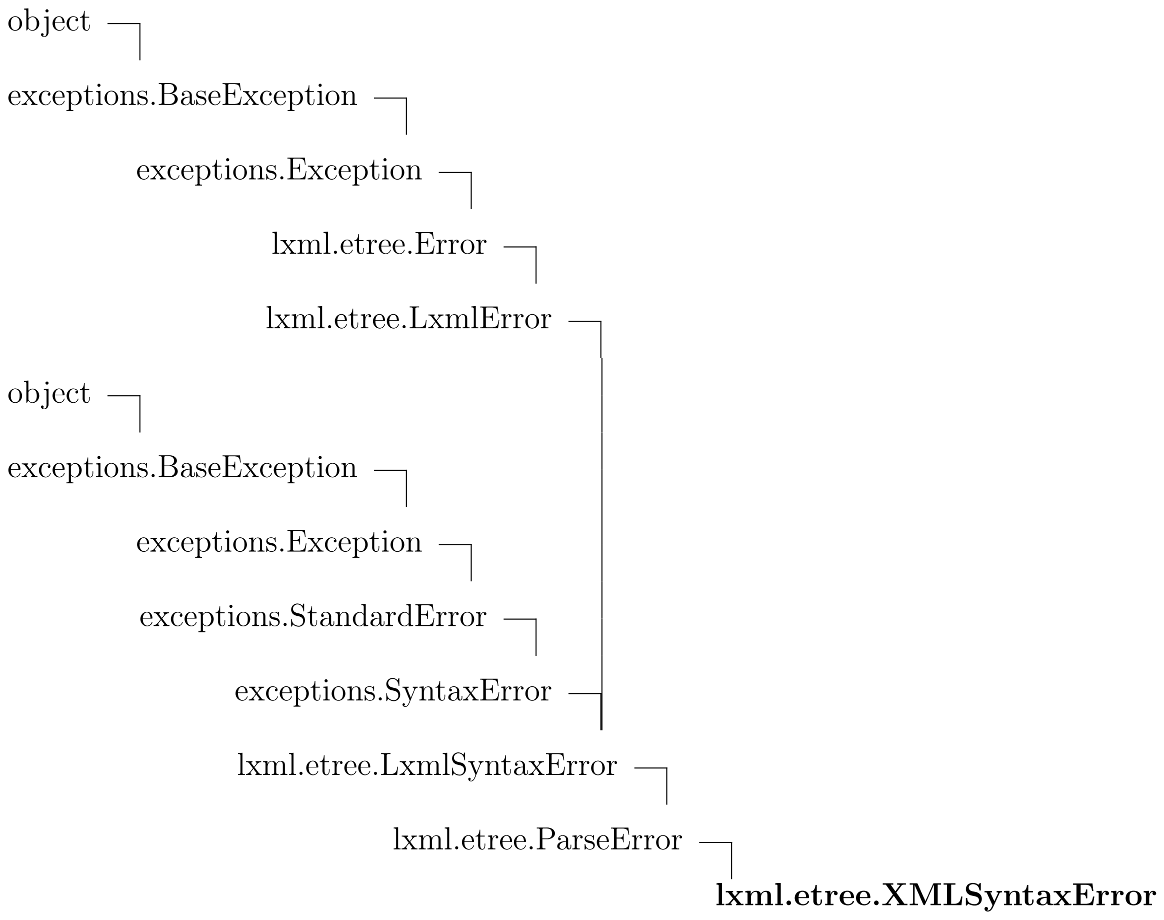
Inherited from object

__format__(), __hash__(), __reduce_ex__(), __sizeof__(), __subclasshook__()

Properties

Name	Description
Inherited from exceptions.BaseException args, message	
Inherited from object __class__	

B.6.62 Class XMLSyntaxError



Syntax error while parsing an XML document.

Methods

__init__(...)

x.__init__(...) initializes x; see x.__class__.__doc__ for signature
Overrides: object.__init__ extit(inherited documentation)

Inherited from exceptions.SyntaxError

__new__(), __str__()

Inherited from exceptions.BaseException

__delattr__(), __getattribute__(), __getitem__(), __getslice__(), __reduce__(), __repr__(), __setattr__(), __setstate__(), __unicode__()

Inherited from object

__format__(), __hash__(), __reduce_ex__(), __sizeof__(), __subclasshook__()

Properties

Name	Description
Inherited from exceptions.SyntaxError filename, lineno, msg, offset, print_file_and_line, text	
Inherited from exceptions.BaseException args, message	
Inherited from object __class__	

B.6.63 Class XPath

object
lxml.etree._XPathEvaluatorBase
lxml.etree.XPath

Known Subclasses: lxml.cssselect.CSSSelector, lxml.etree.ETXPath

XPath(self, path, namespaces=None, extensions=None, regexp=True, smart_strings=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). Smart strings will be returned for string results unless you pass `smart_strings=False`.

Methods

__call__(*self*, *_etree_or_element*, ***_variables*)

__init__(*self*, *path*, *namespaces*=`None`, *extensions*=`None`, *regexp*=`True`, *smart_strings*=`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__

__repr__(*...*)

repr(x) Overrides: object.__repr__ extit(inherited documentation)

Inherited from lxml.etree._XPathEvaluatorBase

evaluate()

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __reduce__(), __reduce_ex__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

Properties

Name	Description
path	
Inherited from lxml.etree._XPathEvaluatorBase error_log	
Inherited from object __class__	

B.6.64 Class XPathDocumentEvaluator

object
lxml.etree._XPathEvaluatorBase
lxml.etree.XPathElementEvaluator
lxml.etree.XPathDocumentEvaluator

XPathDocumentEvaluator(self, etree, namespaces=None, extensions=None, regexp=True, smart_strings=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). Smart strings will be returned for string results unless you pass `smart_strings=False`.

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`, *smart_strings*=`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.6.65)

register_namespace(), register_namespaces()

Inherited from lxml.etree._XPathEvaluatorBase

evaluate()

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

Properties

Name	Description
Inherited from lxml.etree._XPathEvaluatorBase error_log	
Inherited from object __class__	

B.6.65 Class XPathElementEvaluator

object

lxml.etree._XPathEvaluatorBase

lxml.etree.XPathElementEvaluator

Known Subclasses: lxml.etree.XPathDocumentEvaluator

XPathElementEvaluator(self, element, namespaces=None, extensions=None, regexp=True, smart_strings=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). Smart strings will be returned for string results unless you pass `smart_strings=False`.

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.

Absolute XPath expressions (starting with '/') will be evaluated against the ElementTree as returned by getroottree().

__init__(*self*, *element*, *namespaces*=`None`, *extensions*=`None`, *regexp*=`True`, *smart_strings*=`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__

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__(), __format__(), __getattribute__(), __hash__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

Properties

Name	Description
Inherited from lxml.etree._XPathEvaluatorBase error_log	
Inherited from object __class__	

B.6.66 Class XPathError

object

exceptions.BaseException

exceptions.Exception

lxml.etree.Error

lxml.etree.LxmlError

lxml.etree.XPathError

Known Subclasses: lxml.etree.XPathEvalError, lxml.etree.XPathSyntaxError

Base class of all XPath errors.

Methods

Inherited from lxml.etree.LxmlError(Section B.6.31)

__init__()

Inherited from exceptions.Exception

__new__()

Inherited from exceptions.BaseException

__delattr__(), __getattribute__(), __getitem__(), __getslice__(), __reduce__(), __repr__(), __setattr__(), __setstate__(), __str__(), __unicode__()

Inherited from object

__format__(), __hash__(), __reduce_ex__(), __sizeof__(), __subclasshook__()

Properties

Name	Description
Inherited from exceptions.BaseException args, message	
Inherited from object __class__	

B.6.67 Class XPathEvalError

object

exceptions.BaseException

exceptions.Exception

lxml.etree.Error

lxml.etree.LxmlError

lxml.etree.XPathError

lxml.etree.XPathEvalError

Known Subclasses: lxml.etree.XPathFunctionError, lxml.etree.XPathResultError

Error during XPath evaluation.

Methods

Inherited from lxml.etree.LxmlError(Section B.6.31)

__init__()

Inherited from exceptions.Exception

__new__()

Inherited from exceptions.BaseException

__delattr__(), __getattribute__(), __getitem__(), __getslice__(), __reduce__(), __repr__(), __setattr__(), __setstate__(), __str__(), __unicode__()

Inherited from object

__format__(), __hash__(), __reduce_ex__(), __sizeof__(), __subclasshook__()

Properties

Name	Description
Inherited from exceptions.BaseException args, message	
Inherited from object __class__	

B.6.68 Class XPathFunctionError

object

exceptions.BaseException

exceptions.Exception

lxml.etree.Error

lxml.etree.LxmlError

lxml.etree.XPathError

lxml.etree.XPathEvalError

lxml.etree.XPathFunctionError

Internal error looking up an XPath extension function.

Methods

Inherited from lxml.etree.LxmlError(Section B.6.31)

__init__()

Inherited from exceptions.Exception

__new__()

Inherited from exceptions.BaseException

__delattr__(), __getattribute__(), __getitem__(), __getslice__(), __reduce__(), __repr__(), __setattr__(), __setstate__(), __str__(), __unicode__()

Inherited from object

__format__(), __hash__(), __reduce_ex__(), __sizeof__(), __subclasshook__()

Properties

Name	Description
Inherited from exceptions.BaseException args, message	
Inherited from object __class__	

B.6.69 Class XPathResultError

object

exceptions.BaseException

exceptions.Exception

lxml.etree.Error

lxml.etree.LxmlError

lxml.etree.XPathError

lxml.etree.XPathEvalError

lxml.etree.XPathResultError

Error handling an XPath result.

Methods

Inherited from lxml.etree.LxmlError(Section B.6.31)

__init__()

Inherited from exceptions.Exception

__new__()

Inherited from exceptions.BaseException

__delattr__(), __getattribute__(), __getitem__(), __getslice__(), __reduce__(), __repr__(), __setattr__(), __setstate__(), __str__(), __unicode__()

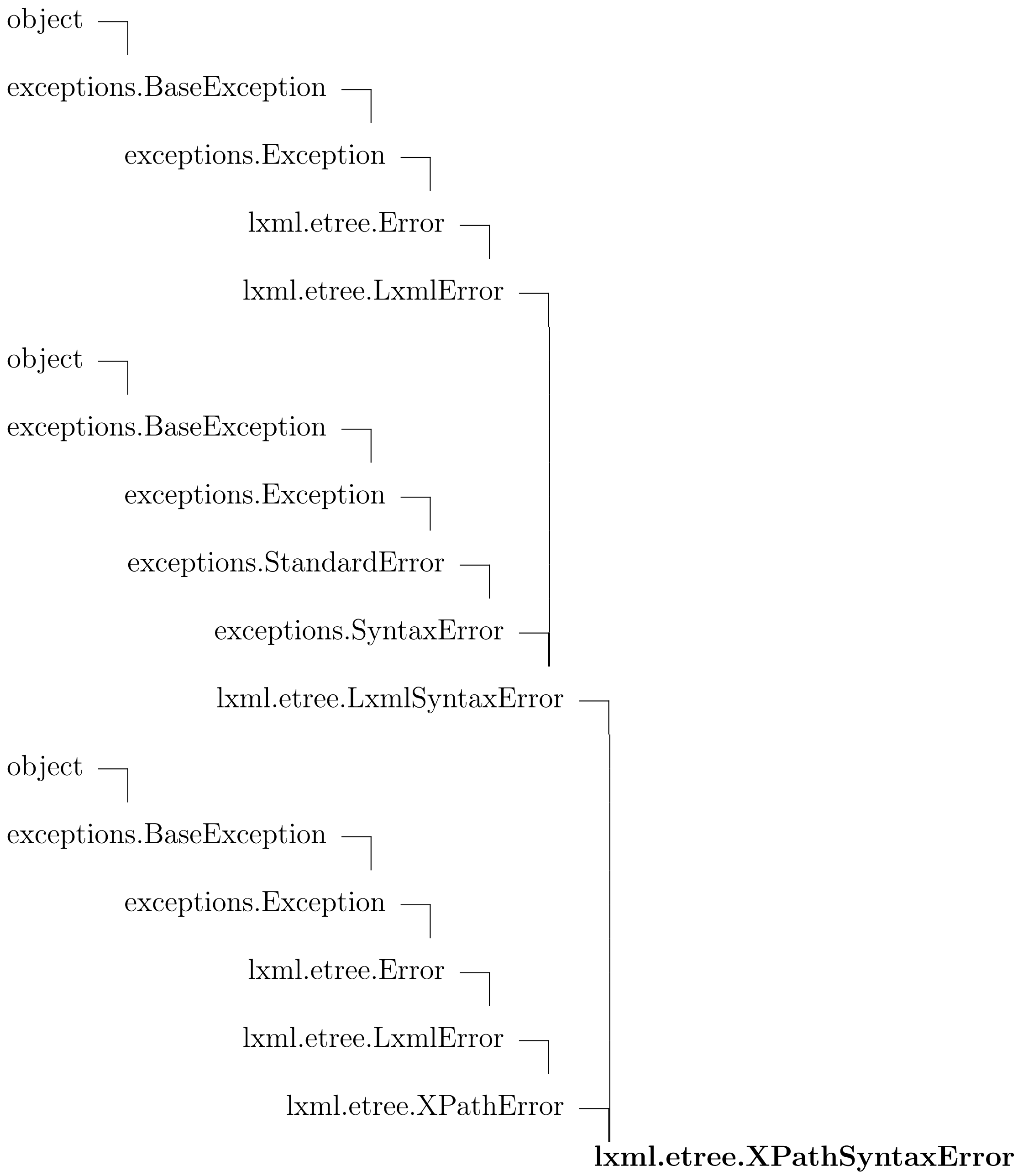
Inherited from object

__format__(), __hash__(), __reduce_ex__(), __sizeof__(), __subclasshook__()

Properties

Name	Description
Inherited from exceptions.BaseException args, message	
Inherited from object __class__	

B.6.70 Class XPathSyntaxError



Methods

Inherited from lxml.etree.LxmlError(Section B.6.31)

__init__()

Inherited from exceptions.SyntaxError

__new__(), __str__()

Inherited from exceptions.BaseException

__delattr__(), __getattribute__(), __getitem__(), __getslice__(), __reduce__(), __repr__(), __setattr__(), __setstate__(), __unicode__()

Inherited from object

__format__(), __hash__(), __reduce_ex__(), __sizeof__(), __subclasshook__()

Properties

Name	Description
Inherited from exceptions.SyntaxError filename, lineno, msg, offset, print_file_and_line, text	
Inherited from exceptions.BaseException args, message	
Inherited from object __class__	

B.6.71 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:

- extensions: a dict mapping `(namespace, name)` pairs to extension functions or extension elements
- 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.

strparam(*strval*)

Mark an XSLT string parameter that requires quote escaping before passing it into the transformation. Use it like this:

```
    result = transform(doc, some_strval = XSLT.strparam(
        '''it's "Monty Python's" ...'''))
```

Escaped string parameters can be reused without restriction.

tostring(*self*, *result_tree*)

Save result doc to string based on stylesheet output method.
Deprecated: use str(result_tree) instead.

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

Properties

Name	Description
error_log	The log of errors and warnings of an XSLT execution.
Inherited from object __class__	

B.6.72 Class XSLTAccessControl

object
lxml.etree.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

For convenience, there is also a class member `DENY_ALL` that provides an XSLTAccessControl instance that is readily configured to deny everything, and a `DENY_WRITE` member that denies all write access but allows read access.

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__

__repr__(...)

repr(x) Overrides: object.__repr__ extit(inherited documentation)

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __reduce__(), __reduce_ex__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

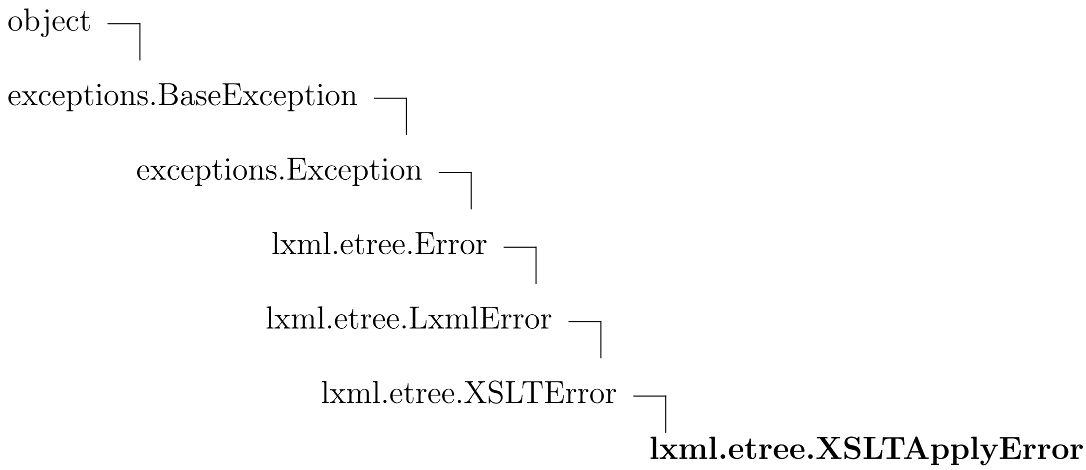
Properties

Name	Description
options	The access control configuration as a map of options.
Inherited from object __class__	

Class Variables

Name	Description
DENY_ALL	**Value:** `XSLTAccessControl(create_dir=False, read_file=False, read...`
DENY_WRITE	**Value:** `XSLTAccessControl(create_dir=False, read_file=True, read_...`

B.6.73 Class XSLTApplyError



Error running an XSL transformation.

Methods

Inherited from lxml.etree.LxmlError(Section B.6.31)

__init__()

Inherited from exceptions.Exception

__new__()

Inherited from exceptions.BaseException

__delattr__(), __getattribute__(), __getitem__(), __getslice__(), __reduce__(), __repr__(), __setattr__(), __setstate__(), __str__(), __unicode__()

Inherited from object

__format__(), __hash__(), __reduce_ex__(), __sizeof__(), __subclasshook__()

Properties

Name	Description
Inherited from exceptions.BaseException args, message	
Inherited from object __class__	

B.6.74 Class XSLTError

object

exceptions.BaseException

exceptions.Exception

lxml.etree.Error

lxml.etree.LxmlError

lxml.etree.XSLTError

Known Subclasses: lxml.etree.XSLTApplyError, lxml.etree.XSLTExtensionError, lxml.etree.XSLTPa lxml.etree.XSLTSaveError

Base class of all XSLT errors.

Methods

Inherited from lxml.etree.LxmlError(Section B.6.31)

__init__()

Inherited from exceptions.Exception

__new__()

Inherited from exceptions.BaseException

__delattr__(), __getattribute__(), __getitem__(), __getslice__(), __reduce__(), __repr__(), __setattr__(), __setstate__(), __str__(), __unicode__()

Inherited from object

__format__(), __hash__(), __reduce_ex__(), __sizeof__(), __subclasshook__()

Properties

Name	Description
Inherited from exceptions.BaseException args, message	
Inherited from object __class__	

B.6.75 Class XSLTExtension

object
lxml.etree.XSLTExtension

Base class of an XSLT extension element.

Methods

__new__(*T*, *S*, *...*)

Return Value

a new object with type S, a subtype of T

Overrides: object.__new__

apply_templates(*self*, *context*, *node*)

Call this method to retrieve the result of applying templates to an element.

The return value is a list of elements or text strings that were generated by the XSLT processor.

execute(*self*, *context*, *self_node*, *input_node*, *output_parent*)

Execute this extension element.

Subclasses must override this method. They may append elements to the `output_parent` element here, or set its text content. To this end, the `input_node` provides read-only access to the current node in the input document, and the `self_node` points to the extension element in the stylesheet.

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __init__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

Properties

Name	Description
Inherited from object __class__	

B.6.76 Class XSLTExtensionError

object
exceptions.BaseException
exceptions.Exception
lxml.etree.Error
lxml.etree.LxmlError
lxml.etree.XSLTError
lxml.etree.XSLTExtensionError

Error registering an XSLT extension.

Methods

Inherited from lxml.etree.LxmlError(Section B.6.31)

__init__()

Inherited from exceptions.Exception

__new__()

Inherited from exceptions.BaseException

__delattr__(), __getattribute__(), __getitem__(), __getslice__(), __reduce__(), __repr__(), __setattr__(), __setstate__(), __str__(), __unicode__()

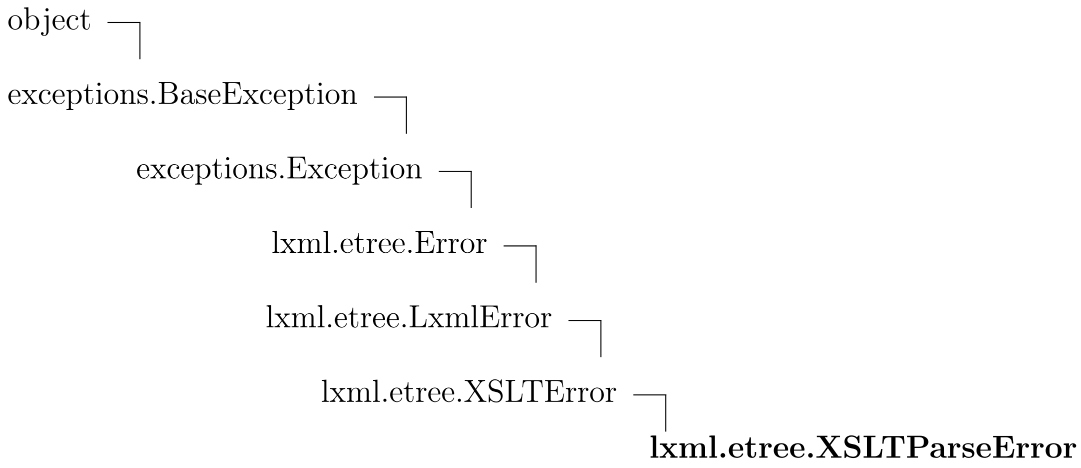
Inherited from object

__format__(), __hash__(), __reduce_ex__(), __sizeof__(), __subclasshook__()

Properties

Name	Description
Inherited from exceptions.BaseException args, message	
Inherited from object __class__	

B.6.77 Class XSLTParseError



Error parsing a stylesheet document.

Methods

Inherited from lxml.etree.LxmlError(Section B.6.31)

__init__()

Inherited from exceptions.Exception

__new__()

Inherited from exceptions.BaseException

__delattr__(), __getattribute__(), __getitem__(), __getslice__(), __reduce__(), __repr__(), __setattr__(), __setstate__(), __str__(), __unicode__()

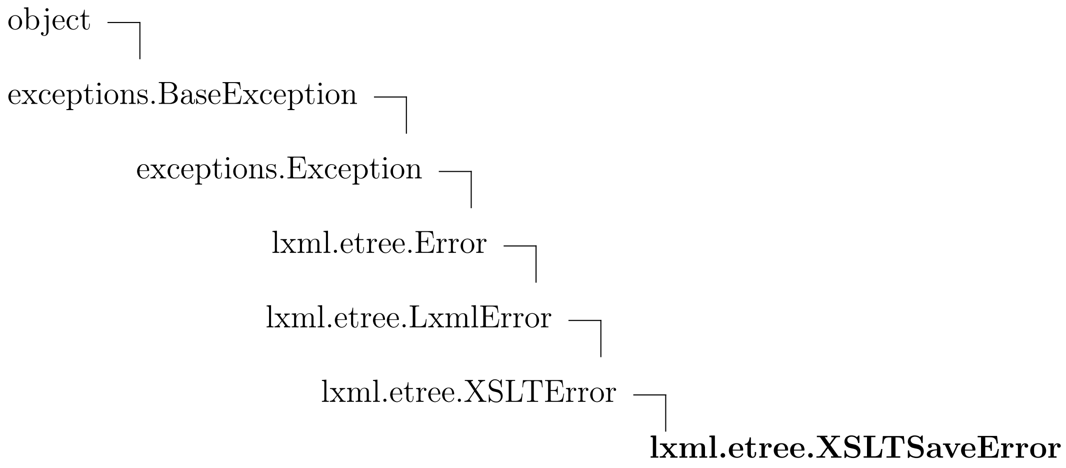
Inherited from object

__format__(), __hash__(), __reduce_ex__(), __sizeof__(), __subclasshook__()

Properties

Name	Description
Inherited from exceptions.BaseException args, message	
Inherited from object __class__	

B.6.78 Class XSLTSaveError



Error serialising an XSLT result.

Methods

Inherited from lxml.etree.LxmlError(Section B.6.31)

__init__()

Inherited from exceptions.Exception

__new__()

Inherited from exceptions.BaseException

__delattr__(), __getattribute__(), __getitem__(), __getslice__(), __reduce__(), __repr__(), __setattr__(), __setstate__(), __str__(), __unicode__()

Inherited from object

__format__(), __hash__(), __reduce_ex__(), __sizeof__(), __subclasshook__()

Properties

Name	Description
Inherited from exceptions.BaseException args, message	
Inherited from object __class__	

B.6.79 Class iterparse

object

lxml.etree._BaseParser

lxml.etree.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, huge_tree=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
- strip_cdata: replace CDATA sections by normal text content (default: True)
- compact: safe memory for short text content (default: True)
- resolve_entities: replace entities by their text value (default: True)
- **huge_tree: disable security restrictions and support very deep trees**

and very long text content (only affects libxml2 2.7+)

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`, *huge_tree*=`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 extit(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__(), __format__(), __getattribute__(), __hash__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

Properties

Name	Description
error_log	The error log of the last (or current) parser run.
root	
Inherited from lxml.etree._BaseParser resolvers, version	
Inherited from object __class__	

B.6.80 Class iterwalk

object
  lxml.etree.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

__init__(*self*, *element_or_tree*, *events*=`("end", )`, *tag*=`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__(...)

next(*x*)

Return Value

the next value, or raise StopIteration

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

Properties

Name	Description
Inherited from object __class__	

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. 370)
- **_dictmixin** *(Section ??, p. ??)*
- **_setmixin** *(Section ??, p. ??)*
- **builder**: A set of HTML generator tags for building HTML documents.
  (Section B.9, p. 371)
- **clean**: A cleanup tool for HTML.
  (Section B.10, p. 374)
- **defs** *(Section B.11, p. 378)*
- **diff** *(Section B.12, p. 379)*
- **formfill** *(Section B.13, p. 380)*
- **html5parser**: An interface to html5lib.
  (Section B.14, p. 382)
- **soupparser**: External interface to the BeautifulSoup HTML parser.
  (Section B.15, p. 386)
- **usedoctest**: Doctest module for HTML comparison.
  (Section B.16, p. 387)

B.7.2 Functions

document_fromstring(*html*, *parser*=`None`, ***kw*)

fragments_fromstring(*html*, *no_leading_text*=`False`, *base_url*=`None`, *parser*=`None`, ***kw*)

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`, *parser*=`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`, *parser*=`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)

parse(*filename_or_url*, *parser*=`None`, *base_url*=`None`, ***kw*)

Parse a filename, URL, or file-like object into an HTML document tree. Note: this returns a tree, not an element. Use `parse(...).getroot()` to get the document root.

You can override the base URL with the `base_url` keyword. This is most useful when parsing from a file-like object.

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 (defauts 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')
'Helloworld!'

>>> html.tostring(root, method='text', encoding=unicode)
u'Helloworld!'
```

open_in_browser(*doc*)

Open the HTML document in a web browser (saving it to a temporary file to open it).

Element(**args*, ***kw*)

Create a new HTML Element.

This can also be used for XHTML documents.

B.7.3 Variables

Name	Description
find_rel_links	**Value:** `_MethodFunc('find_rel_links', copy= False)`
find_class	**Value:** `_MethodFunc('find_class', copy= False)`
make_links_absolute	**Value:** `_MethodFunc('make_links_absolute', copy= True)`
resolve_base_href	**Value:** `_MethodFunc('resolve_base_href', copy= True)`
iterlinks	**Value:** `_MethodFunc('iterlinks', copy= False)`
rewrite_links	**Value:** `_MethodFunc('rewrite_links', copy= True)`

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

CLASS(*v*)

FOR(*v*)

B.9.2 Variables

Name	Description
E	**Value:** `ElementMaker(makeelement= html_parser.makeelement)`
A	**Value:** `E.a`
ABBR	**Value:** `E.abbr`
ACRONYM	**Value:** `E.acronym`
ADDRESS	**Value:** `E.address`
APPLET	**Value:** `E.applet`
AREA	**Value:** `E.area`
B	**Value:** `E.b`
BASE	**Value:** `E.base`
BASEFONT	**Value:** `E.basefont`

continued on next page

Name	Description
BDO	**Value:** `E.bdo`
BIG	**Value:** `E.big`
BLOCKQUOTE	**Value:** `E.blockquote`
BODY	**Value:** `E.body`
BR	**Value:** `E.br`
BUTTON	**Value:** `E.button`
CAPTION	**Value:** `E.caption`
CENTER	**Value:** `E.center`
CITE	**Value:** `E.cite`
CODE	**Value:** `E.code`
COL	**Value:** `E.col`
COLGROUP	**Value:** `E.colgroup`
DD	**Value:** `E.dd`
DEL	**Value:** `getattr(E, 'del')`
DFN	**Value:** `E.dfn`
DIR	**Value:** `E.dir`
DIV	**Value:** `E.div`
DL	**Value:** `E.dl`
DT	**Value:** `E.dt`
EM	**Value:** `E.em`
FIELDSET	**Value:** `E.fieldset`
FONT	**Value:** `E.font`
FORM	**Value:** `E.form`
FRAME	**Value:** `E.frame`
FRAMESET	**Value:** `E.frameset`
H1	**Value:** `E.h1`
H2	**Value:** `E.h2`
H3	**Value:** `E.h3`
H4	**Value:** `E.h4`
H5	**Value:** `E.h5`
H6	**Value:** `E.h6`
HEAD	**Value:** `E.head`
HR	**Value:** `E.hr`
HTML	**Value:** `E.html`
I	**Value:** `E.i`
IFRAME	**Value:** `E.iframe`
IMG	**Value:** `E.img`
INPUT	**Value:** `E.input`
INS	**Value:** `E.ins`
ISINDEX	**Value:** `E.isindex`
KBD	**Value:** `E.kbd`
LABEL	**Value:** `E.label`
LEGEND	**Value:** `E.legend`
LI	**Value:** `E.li`
LINK	**Value:** `E.link`
MAP	**Value:** `E.map`

continued on next page

Name	Description
MENU	**Value:** `E.menu`
META	**Value:** `E.meta`
NOFRAMES	**Value:** `E.noframes`
NOSCRIPT	**Value:** `E.noscript`
OBJECT	**Value:** `E.object`
OL	**Value:** `E.ol`
OPTGROUP	**Value:** `E.optgroup`
OPTION	**Value:** `E.option`
P	**Value:** `E.p`
PARAM	**Value:** `E.param`
PRE	**Value:** `E.pre`
Q	**Value:** `E.q`
S	**Value:** `E.s`
SAMP	**Value:** `E.samp`
SCRIPT	**Value:** `E.script`
SELECT	**Value:** `E.select`
SMALL	**Value:** `E.small`
SPAN	**Value:** `E.span`
STRIKE	**Value:** `E.strike`
STRONG	**Value:** `E.strong`
STYLE	**Value:** `E.style`
SUB	**Value:** `E.sub`
SUP	**Value:** `E.sup`
TABLE	**Value:** `E.table`
TBODY	**Value:** `E.tbody`
TD	**Value:** `E.td`
TEXTAREA	**Value:** `E.textarea`
TFOOT	**Value:** `E.tfoot`
TH	**Value:** `E.th`
THEAD	**Value:** `E.thead`
TITLE	**Value:** `E.title`
TR	**Value:** `E.tr`
TT	**Value:** `E.tt`
U	**Value:** `E.u`
UL	**Value:** `E.ul`
VAR	**Value:** `E.var`
__package__	**Value:** `'lxml.html'`

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 element's 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*=`unichr(0x200b)`)

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	**Value:** `Cleaner()`
clean_html	**Value:** `clean.clean_html`

B.10.3 Class Cleaner

object

lxml.html.clean.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 feedparser).

`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

__init__(*self*, ***kw*)

x.__init__(...) initializes x; see x.__class__.__doc__ for signature
Overrides: object.__init__ extit(inherited documentation)

__call__(*self*, *doc*)

Cleans the document.

allow_follow(*self*, *anchor*)

Override to suppress rel="nofollow" on some anchors.

allow_element(*self*, *el*)

allow_embedded_url(*self*, *el*, *url*)

kill_conditional_comments(*self*, *doc*)

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.

clean_html(*self*, *html*)

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

Properties

Name	Description
Inherited from object __class__	

Class Variables

Name	Description
scripts	**Value:** `True`
javascript	**Value:** `True`
comments	**Value:** `True`
style	**Value:** `False`
links	**Value:** `True`
meta	**Value:** `True`
page_structure	**Value:** `True`
processing_instructions	**Value:** `True`
embedded	**Value:** `True`
frames	**Value:** `True`
forms	**Value:** `True`
annoying_tags	**Value:** `True`
remove_tags	**Value:** `None`
allow_tags	**Value:** `None`
remove_unknown_tags	**Value:** `True`
safe_attrs_only	**Value:** `True`
add_nofollow	**Value:** `False`
host_whitelist	**Value:** `()`
whitelist_tags	**Value:** `set(['iframe', 'embed'])`

B.11 Module lxml.html.defs

B.11.1 Variables

Name	Description
empty_tags	**Value:** `frozenset(['area', 'base', 'basefont', 'br', 'col', 'fram...`
deprecated_tags	**Value:** `frozenset(['applet', 'basefont', 'center', 'dir', 'font',...`
link_attrs	**Value:** `frozenset(['action', 'archive', 'background', 'cite', 'cl...`
event_attrs	**Value:** `frozenset(['onblur', 'onchange', 'onclick', 'ondblclick',...`
safe_attrs	**Value:** `frozenset(['abbr', 'accept', 'accept-charset', 'accesskey...`
top_level_tags	**Value:** `frozenset(['body', 'frameset', 'head', 'html'])`
head_tags	**Value:** `frozenset(['base', 'isindex', 'link', 'meta', 'script', '...`
general_block_tags	**Value:** `frozenset(['address', 'blockquote', 'center', 'del', 'div...`
list_tags	**Value:** `frozenset(['dd', 'dir', 'dl', 'dt', 'li', 'menu', 'ol', '...`
table_tags	**Value:** `frozenset(['caption', 'col', 'colgroup', 'table', 'tbody'...`
block_tags	**Value:** `frozenset(['address', 'blockquote', 'caption', 'center', ...`
form_tags	**Value:** `frozenset(['button', 'fieldset', 'form', 'input', 'label'...`
special_inline_tags	**Value:** `frozenset(['a', 'applet', 'area', 'basefont', 'bdo', 'br'...`
phrase_tags	**Value:** `frozenset(['abbr', 'acronym', 'cite', 'code', 'del', 'dfn...`
font_style_tags	**Value:** `frozenset(['b', 'big', 'i', 's', 'small', 'strike', 'tt',...`
frame_tags	**Value:** `frozenset(['frame', 'frameset', 'noframes'])`
nonstandard_tags	**Value:** `frozenset(['blink', 'marque'])`
tags	**Value:** `frozenset(['a', 'abbr', 'acronym', 'address', 'applet', '...`
__package__	**Value:** `None`

B.12 Module lxml.html.diff

B.12.1 Functions

html_annotate(*doclist*, *markup*=`<function default_markup at 0x3010230>`)

doclist should be ordered from oldest to newest, like:

```
>>> version1 = 'Hello World'
>>> version2 = 'Goodbye World'
>>> print(html_annotate([(version1, 'version 1'),
...                     (version2, 'version 2')]))
<span title="version 2">Goodbye</span> <span title="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:

```
>>> print(default_markup('Some Text', 'by Joe'))
<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

object

exceptions.BaseException

exceptions.Exception

exceptions.StandardError

exceptions.LookupError

lxml.html.formfill.FormNotFound

Raised when no form can be found

Methods

Inherited from exceptions.LookupError

__init__(), __new__()

Inherited from exceptions.BaseException

__delattr__(), __getattribute__(), __getitem__(), __getslice__(), __reduce__(), __repr__(), __setattr__(), __setstate__(), __str__(), __unicode__()

Inherited from object

__format__(), __hash__(), __reduce_ex__(), __sizeof__(), __subclasshook__()

Properties

Name	Description
Inherited from exceptions.BaseException args, message	
Inherited from object __class__	

B.13.3 Class DefaultErrorCreator

object ─┐
lxml.html.formfill.DefaultErrorCreator

Methods

__init__(*self*, ***kw*)

x.__init__(...) initializes x; see x.__class__.__doc__ for signature
Overrides: object.__init__ extit(inherited documentation)

__call__(*self*, *el*, *is_block*, *message*)

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

Properties

Name	Description
Inherited from object __class__	

Class Variables

Name	Description
insert_before	**Value:** `True`
block_inside	**Value:** `True`
error_container_tag	**Value:** `'div'`
error_message_class	**Value:** `'error-message'`
error_block_class	**Value:** `'error-block'`
default_message	**Value:** `'Invalid'`

B.14 Module lxml.html.html5parser

An interface to html5lib.

B.14.1 Functions

document_fromstring(*html*, *guess_charset*=`True`, *parser*=`None`)

Parse a whole document into a string.

fragments_fromstring(*html*, *no_leading_text*=`False`, *guess_charset*=`False`, *parser*=`None`)

Parses several HTML elements, returning a list of elements.

The first item in the list may be a string. 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.

If `guess_charset` is `True` and the text was not unicode but a bytestring, the `chardet` library will perform charset guessing on the string.

fragment_fromstring(*html*, *create_parent*=`False`, *guess_charset*=`False`, *parser*=`None`)

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.

fromstring(*html*, *guess_charset*=`True`, *parser*=`None`)

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)

parse(*filename_url_or_file*, *guess_charset*=`True`, *parser*=`None`)

Parse a filename, URL, or file-like object into an HTML document tree. Note: this returns a tree, not an element. Use `parse(...).getroot()` to get the document root.

B.14.2 Variables

Name	Description
xhtml_parser	**Value:** `XHTMLParser()`
html_parser	**Value:** `HTMLParser()`
__package__	**Value:** `'lxml.html'`

B.14.3 Class HTMLParser

object

html5lib.html5parser.HTMLParser

lxml.html.html5parser.HTMLParser

An html5lib HTML parser with lxml as tree.

Methods

__**init**__(*self*, *strict*=`False`)

strict - raise an exception when a parse error is encountered

tree - a treebuilder class controlling the type of tree that will be returned. Built in treebuilders can be accessed through html5lib.treebuilders.getTreeBuilder(treeType)

tokenizer - a class that provides a stream of tokens to the treebuilder. This may be replaced for e.g. a sanitizer which converts some tags to text Overrides: object.__init__ extit(inherited documentation)

Inherited from html5lib.html5parser.HTMLParser

normalizeToken(), normalizedTokens(), parse(), parseError(), parseFragment(), parseRCDataCData(), resetInsertionMode()

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

Properties

Name	Description
Inherited from object __class__	

B.14.4 Class XHTMLParser

object

html5lib.html5parser.HTMLParser

html5lib.liberalxmlparser.XMLParser

html5lib.liberalxmlparser.XHTMLParser

lxml.html.html5parser.XHTMLParser

An html5lib XHTML Parser with lxml as tree.

Methods

__init__(*self*, *strict*=`False`)

strict - raise an exception when a parse error is encountered

tree - a treebuilder class controlling the type of tree that will be returned. Built in treebuilders can be accessed through html5lib.treebuilders.getTreeBuilder(treeType)

tokenizer - a class that provides a stream of tokens to the treebuilder. This may be replaced for e.g. a sanitizer which converts some tags to text
Overrides: object.__init__ extit(inherited documentation)

Inherited from html5lib.liberalxmlparser.XHTMLParser

normalizeToken()

Inherited from html5lib.liberalxmlparser.XMLParser

parseRCDataCData()

Inherited from html5lib.html5parser.HTMLParser

normalizedTokens(), parse(), parseError(), parseFragment(), resetInsertionMode()

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

Properties

Name	Description
Inherited from object __class__	

B.15 Module lxml.html.soupparser

External interface to the BeautifulSoup HTML parser.

B.15.1 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 diffent 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 ElemenTree using the BeautifulSoup parser.

You can pass a different BeautifulSoup parser through the `beautifulsoup` keyword, and a diffent 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.16 Module lxml.html.usedoctest

Doctest module for HTML comparison.

Usage:

```
>>> import lxml.html.usedoctest
>>> # now do your HTML doctests ...
```

See `lxml.doctestcompare`.

B.17 Module lxml.objectify

The `lxml.objectify` module implements a Python object API for XML. It is based on `lxml.etree`. **Version:** 2.2.6

B.17.1 Functions

DataElement(*_value*, *attrib*=`None`, *nsmap*=`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 nsmap are reused unless they are redefined in attrib and/or keyword arguments.

Element(*_tag*, *attrib*=`None`, *nsmap*=`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`, *base_url*=`None`)

Objectify specific version of the lxml.etree XML() literal factory that uses the objectify parser.

You can pass a different parser as second 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, ...).

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`, *xsi_nil*=`False`)

Recursively de-annotate the elements of an XML tree by removing 'py:pytype' and/or 'xsi:type' attributes and/or 'xsi:nil' attributes.

If the 'pytype' keyword argument is True (the default), 'py:pytype' attributes will be removed. If the 'xsi' keyword argument is True (the default), 'xsi:type' attributes will be removed. If the 'xsi_nil' keyword argument is True (default: False), 'xsi:nil' attributes will be removed.

Note that this does not touch the namespace declarations. If you want to remove unused namespace declarations from the tree, use `lxml.etree.cleanup_namespaces()`.

dump(...)

dump(_Element element not None)

Return a recursively generated string representation of an element.

enable_recursive_str(*on*=`True`)

Enable a recursively generated tree representation for str(element), based on objectify.dump(element).

fromstring(*xml*, *parser*=`None`, *base_url*=`None`)

Objectify specific version of the lxml.etree fromstring() function that uses the objectify parser.

You can pass a different parser as second 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, ...).

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 the 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`, *base_ url*=`None`)

Parse a file or file-like object with the objectify parser.

You can pass a different parser as second argument.

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.

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.

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.17.2 Variables

Name	Description
E	**Value:** `<lxml.objectify.ElementMaker object at 0x2ab5960>`
PYTYPE_ATTRIBUTE	**Value:** `'{http://codespeak.net/lxml/objectify/pytype}pytype'`

B.17.3 Class BoolElement

object

lxml.etree._Element

lxml.etree.ElementBase

lxml.objectify.ObjectifiedElement

lxml.objectify.ObjectifiedDataElement

lxml.objectify.NumberElement

lxml.objectify.IntElement

lxml.objectify.BoolElement

Boolean type base on string values: 'true' or 'false'.

Note that this inherits from IntElement to mimic the behaviour of Python's bool type.

Methods

__eq__(*x*, *y*)

x==y Overrides: lxml.objectify.NumberElement.__eq__

__ge__(*x*, *y*)

x>=y Overrides: lxml.objectify.NumberElement.__ge__

__gt__(*x*, *y*)

x>y Overrides: lxml.objectify.NumberElement.__gt__

__le__(*x*, *y*)

x<=y Overrides: lxml.objectify.NumberElement.__le__

__lt__(*x*, *y*)

x<y Overrides: lxml.objectify.NumberElement.__lt__

__ne__(*x*, *y*)

x!=y Overrides: lxml.objectify.NumberElement.__ne__

__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__

__repr__(*self*)

repr(x) Overrides: object.__repr__ extit(inherited documentation)

__str__(*...*)

str(x) Overrides: object.__str__ extit(inherited documentation)

Inherited from lxml.objectify.NumberElement(Section B.17.9)

__abs__(), __add__(), __and__(), __complex__(), __div__(), __float__(), __hex__(), __int__(), __invert__(), __long__(), __lshift__(), __mod__(), __mul__(), __neg__(), __oct__(), __or__(), __pos__(), __pow__(), __radd__(), __rand__(), __rdiv__(), __rlshift__(), __rmod__(), __rmul__(), __ror__(), __rpow__(), __rrshift__(), __rshift__(), __rsub__(), __rtruediv__(), __rxor__(), __sub__(), __truediv__(), __xor__()

Inherited from lxml.objectify.ObjectifiedElement(Section B.17.12)

__delattr__(), __delitem__(), __getattr__(), __getattribute__(), __getitem__(), __iter__(), __len__(), __setattr__(), __setitem__(), addattr(), countchildren(), descendantpaths(), find(), findall(), findtext(), getchildren(), iterfind()

Inherited from lxml.etree.ElementBase(Section B.6.17)

__init__()

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

__format__(), __hash__(), __reduce__(), __reduce_ex__(), __sizeof__(), __subclasshook__()

Properties

Name	Description
pyval	
Inherited from lxml.objectify.ObjectifiedElement (Section B.17.12) text	
Inherited from lxml.etree._Element attrib, base, nsmap, prefix, sourceline, tag, tail	
Inherited from object __class__	

B.17.4 Class ElementMaker

object

lxml.objectify.ElementMaker

ElementMaker(self, namespace=None, nsmap=None, annotate=True, makeelement=None)

An ElementMaker that can be used for constructing trees.

Example:

```
>>> M = ElementMaker(annotate=False)
>>> html = M.html( M.body( M.p(’hello’, M.br, ’objectify’) ) )

>>> from lxml.etree import tostring
>>> print(tostring(html, method=’html’).decode(’ASCII’))
<html><body><p>hello<br>objectify</p></body></html>
```

Note that this module has a predefined ElementMaker instance called `E`.

Methods

__getattr__(...)

__getattribute__(...)

x.__getattribute__('name') <==> x.name Overrides: object.__getattribute__

__init__(*self*, *namespace*=`None`, *nsmap*=`None`, *annotate*=`True`, *makeelement*=`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__(), __format__(), __hash__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

Properties

Name	Description
Inherited from object __class__	

B.17.5 Class FloatElement

object
lxml.etree._Element
lxml.etree.ElementBase
lxml.objectify.ObjectifiedElement
lxml.objectify.ObjectifiedDataElement
lxml.objectify.NumberElement
lxml.objectify.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.17.9)

__abs__(), __add__(), __and__(), __complex__(), __div__(), __eq__(), __float__(), __ge__(), __gt__(), __hex__(), __int__(), __invert__(), __le__(), __long__(), __lshift__(), __lt__(), __mod__(), __mul__(), __ne__(), __neg__(), __nonzero__(), __oct__(), __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.17.12)

__delattr__(), __delitem__(), __getattr__(), __getattribute__(), __getitem__(), __iter__(), __len__(), __setattr__(), __setitem__(), addattr(), countchildren(), descendantpaths(), find(), findall(), findtext(), getchildren(), iterfind()

Inherited from lxml.etree.ElementBase(Section B.6.17)

__init__()

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

__format__(), __hash__(), __reduce__(), __reduce_ex__(), __sizeof__(), __subclasshook__()

Properties

Name	Description
Inherited from lxml.objectify.NumberElement (Section B.17.9) pyval	
Inherited from lxml.objectify.ObjectifiedElement (Section B.17.12) text	
Inherited from lxml.etree._Element attrib, base, nsmap, prefix, sourceline, tag, tail	
Inherited from object __class__	

B.17.6 Class IntElement

object
lxml.etree._Element
lxml.etree.ElementBase
lxml.objectify.ObjectifiedElement
lxml.objectify.ObjectifiedDataElement
lxml.objectify.NumberElement
lxml.objectify.IntElement

Known Subclasses: lxml.objectify.BoolElement

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.17.9)

__abs__(), __add__(), __and__(), __complex__(), __div__(), __eq__(), __float__(), __ge__(), __gt__(), __hex__(), __int__(), __invert__(), __le__(), __long__(), __lshift__(), __lt__(), __mod__(), __mul__(), __ne__(), __neg__(), __nonzero__(), __oct__(), __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.17.12)

__delattr__(), __delitem__(), __getattr__(), __getattribute__(), __getitem__(), __iter__(), __len__(), __setattr__(), __setitem__(), addattr(), countchildren(), descendantpaths(), find(), findall(), findtext(), getchildren(), iterfind()

Inherited from lxml.etree.ElementBase(Section B.6.17)

__init__()

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

__format__(), __hash__(), __reduce__(), __reduce_ex__(), __sizeof__(), __subclasshook__()

Properties

Name	Description
Inherited from lxml.objectify.NumberElement (Section B.17.9) pyval	
Inherited from lxml.objectify.ObjectifiedElement (Section B.17.12) text	
Inherited from lxml.etree._Element attrib, base, nsmap, prefix, sourceline, tag, tail	
Inherited from object __class__	

B.17.7 Class LongElement

object
lxml.etree._Element
lxml.etree.ElementBase
lxml.objectify.ObjectifiedElement
lxml.objectify.ObjectifiedDataElement
lxml.objectify.NumberElement
lxml.objectify.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.17.9)

__abs__(), __add__(), __and__(), __complex__(), __div__(), __eq__(), __float__(), __ge__(), __gt__(), __hex__(), __int__(), __invert__(), __le__(), __long__(), __lshift__(), __lt__(), __mod__(), __mul__(), __ne__(), __neg__(), __nonzero__(), __oct__(), __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.17.12)

__delattr__(), __delitem__(), __getattr__(), __getattribute__(), __getitem__(), __iter__(), __len__(), __setattr__(), __setitem__(), addattr(), countchildren(), descendantpaths(), find(), findall(), findtext(), getchildren(), iterfind()

Inherited from lxml.etree.ElementBase(Section B.6.17)

__init__()

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

__format__(), __hash__(), __reduce__(), __reduce_ex__(), __sizeof__(), __subclasshook__()

Properties

Name	Description
Inherited from lxml.objectify.NumberElement (Section B.17.9) pyval	
Inherited from lxml.objectify.ObjectifiedElement (Section B.17.12) text	
Inherited from lxml.etree._Element attrib, base, nsmap, prefix, sourceline, tag, tail	
Inherited from object __class__	

B.17.8 Class NoneElement

object
lxml.etree._Element
lxml.etree.ElementBase
lxml.objectify.ObjectifiedElement
lxml.objectify.ObjectifiedDataElement
lxml.objectify.NoneElement

Methods

__eq__(*x*, *y*)

x==y

__ge__(*x*, *y*)

x>=y

__gt__(*x*, *y*)

x>y

__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__

__nonzero__(*x*)

x != 0 Overrides: lxml.etree._Element.__nonzero__

__repr__(*self*)

repr(x) Overrides: object.__repr__ extit(inherited documentation)

__str__(*...*)

str(x) Overrides: object.__str__ extit(inherited documentation)

Inherited from lxml.objectify.ObjectifiedElement(Section B.17.12)

__delattr__(), __delitem__(), __getattr__(), __getattribute__(), __getitem__(), __iter__(), __len__(), __setattr__(), __setitem__(), addattr(), countchildren(), descendantpaths(), find(), findall(), findtext(), getchildren(), iterfind()

Inherited from lxml.etree.ElementBase(Section B.6.17)

__init__()

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

__format__(), __hash__(), __reduce__(), __reduce_ex__(), __sizeof__(), __subclasshook__()

Properties

Name	Description
pyval	
Inherited from lxml.objectify.ObjectifiedElement (Section B.17.12) text	
Inherited from lxml.etree._Element attrib, base, nsmap, prefix, sourceline, tag, tail	
Inherited from object __class__	

B.17.9 Class NumberElement

object
lxml.etree._Element
lxml.etree.ElementBase
lxml.objectify.ObjectifiedElement
lxml.objectify.ObjectifiedDataElement
lxml.objectify.NumberElement

Known Subclasses: lxml.objectify.IntElement, lxml.objectify.FloatElement, lxml.objectify.LongEleme

Methods

__abs__(...)

__add__(...)

__and__(...)

__complex__(...)

__div__(...)

__eq__(*x*, *y*)

x==y

__float__(...)

__ge__(*x*, *y*)

x>=y

__gt__(*x*, *y*)

x>y

__hex__(...)

__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 != 0 Overrides: lxml.etree._Element.__nonzero__

__oct__(*...*)

__or__(*...*)

__pos__(*...*)

__pow__(*x*, *y*, *z*=. . .)

pow(x, y[, z])

__radd__(*x*, *y*)

y+x

__rand__(*x*, *y*)

y&x

__rdiv__(*x*, *y*)

y/x

__repr__(*self*)

repr(x) Overrides: object.__repr__ extit(inherited documentation)

__rlshift__(*x*, *y*)

y«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__ extit(inherited documentation)

__sub__(...)

__truediv__(...)

__xor__(...)

Inherited from lxml.objectify.ObjectifiedElement(Section B.17.12)

__delattr__(), __delitem__(), __getattr__(), __getattribute__(), __getitem__(), __iter__(), __len__(), __setattr__(), __setitem__(), addattr(), countchildren(), descendantpaths(), find(), findall(), findtext(), getchildren(), iterfind()

Inherited from lxml.etree.ElementBase(Section B.6.17)

__init__()

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

__format__(), __hash__(), __reduce__(), __reduce_ex__(), __sizeof__(), __subclasshook__()

Properties

Name	Description
pyval	
Inherited from lxml.objectify.ObjectifiedElement (Section B.17.12) text	
Inherited from lxml.etree._Element attrib, base, nsmap, prefix, sourceline, tag, tail	
Inherited from object __class__	

B.17.10 Class ObjectPath

object ─┐
 lxml.objectify.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 it 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__ extit(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*)

setattr(*self*, *root*, *value*)

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__(), __format__(), __getattribute__(), __hash__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __subclasshook__()

Properties

Name	Description
find	
Inherited from object __class__	

B.17.11 Class ObjectifiedDataElement

object
lxml.etree._Element
lxml.etree.ElementBase
lxml.objectify.ObjectifiedElement
lxml.objectify.ObjectifiedDataElement

Known Subclasses: lxml.objectify.NumberElement, lxml.objectify.NoneElement, lxml.objectify.String

This is the base class for all data type Elements. Subclasses should override the 'pyval' property and possibly the __str__ method.

Methods

__new__(*T*, *S*, *...*)

Return Value

a new object with type S, a subtype of T

Overrides: object.__new__

__repr__(*self*)

repr(x) Overrides: object.__repr__ extit(inherited documentation)

__str__(...)

str(x) Overrides: object.__str__ extit(inherited documentation)

Inherited from lxml.objectify.ObjectifiedElement(Section B.17.12)

__delattr__(), __delitem__(), __getattr__(), __getattribute__(), __getitem__(), __iter__(), __len__(), __setattr__(), __setitem__(), addattr(), countchildren(), descendantpaths(), find(), findall(), findtext(), getchildren(), iterfind()

Inherited from lxml.etree.ElementBase(Section B.6.17)

__init__()

Inherited from lxml.etree._Element

__contains__(), __copy__(), __deepcopy__(), __nonzero__(), __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

__format__(), __hash__(), __reduce__(), __reduce_ex__(), __sizeof__(), __subclasshook__()

Properties

Name	Description
pyval	
Inherited from lxml.objectify.ObjectifiedElement (Section B.17.12) text	
Inherited from lxml.etree._Element	

continued on next page

Name	Description
attrib, base, nsmap, prefix, sourceline, tag, tail	
Inherited from object __class__	

B.17.12 Class ObjectifiedElement

object
lxml.etree._Element
lxml.etree.ElementBase
lxml.objectify.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 = XML("<root><c1><c2>0</c2><c2>1</c2></c1></root>")
>>> second_c2 = root.c1.c2[1]
>>> print(second_c2.text)
1
```

Note that you cannot (and must not) instantiate this class or its subclasses.

Methods

__delattr__(...)

x.__delattr__('name') <==> del x.name Overrides: object.__delattr__

__delitem__(*x*, *y*)

del x[y] Overrides: lxml.etree._Element.__delitem__

__getattr__(...)

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.

__getattribute__(...)

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__ extit(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.ElementBase(Section B.6.17)

__init__()

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

__format__(), __hash__(), __reduce__(), __reduce_ex__(), __sizeof__(), __subclasshook__()

Properties

Name	Description
text	Text before the first subelement. This is either a string or the value None, if there was no text.
Inherited from lxml.etree._Element attrib, base, nsmap, prefix, sourceline, tag, tail	
Inherited from object __class__	

B.17.13 Class ObjectifyElementClassLookup

object

lxml.etree.ElementClassLookup

lxml.objectify.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__(), __format__(), __getattribute__(), __hash__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

Properties

Name	Description
Inherited from object	
__class__	

B.17.14 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__ extit(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__(), __format__(), __getattribute__(), __hash__(), __reduce__(), __reduce_ex__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

Properties

Name	Description
name	
stringify	
type_check	
xmlSchemaTypes	The list of XML Schema datatypes this Python type maps to. Note that this must be set before registering the type!
Inherited from object	

continued on next page

Name	Description
__class__	

B.17.15 Class StringElement

object
lxml.etree._Element
lxml.etree.ElementBase
lxml.objectify.ObjectifiedElement
lxml.objectify.ObjectifiedDataElement
lxml.objectify.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

__add__(...)

__complex__(...)

__eq__(*x*, *y*)

x==y

__float__(...)

__ge__(*x*, *y*)

x>=y

__gt__(*x*, *y*)

x>y

__int__(...)

__le__(*x*, *y*)

x<=y

__long__(...)

__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__ extit(inherited documentation)

__rmod__(*x*, *y*)

y%x

__rmul__(*x*, *y*)

y*x

strlen(...)

Inherited from lxml.objectify.ObjectifiedDataElement(Section B.17.11)

__str__()

Inherited from lxml.objectify.ObjectifiedElement(Section B.17.12)

__delattr__(), __delitem__(), __getattr__(), __getattribute__(), __getitem__(), __iter__(), __len__(), __setattr__(), __setitem__(), addattr(), countchildren(), descendantpaths(), find(), findall(), findtext(), getchildren(), iterfind()

Inherited from lxml.etree.ElementBase(Section B.6.17)

__init__()

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

__format__(), __hash__(), __reduce__(), __reduce_ex__(), __sizeof__(), __subclasshook__()

Properties

Name	Description
pyval	
Inherited from lxml.objectify.ObjectifiedElement (Section B.17.12) text	

continued on next page

Name	Description
Inherited from lxml.etree._Element attrib, base, nsmap, prefix, sourceline, tag, tail	
Inherited from object __class__	

B.18 Module lxml.pyclasslookup

B.18.1 Variables

Name	Description
__package__	**Value:** `'lxml'`

B.19 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.19.1 Functions

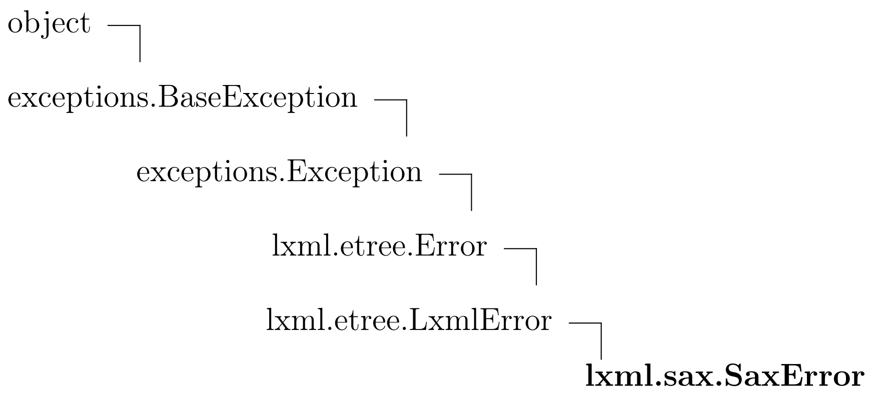
saxify(*element_or_tree*, *content_handler*)

One-shot helper to generate SAX events from an XML tree and fire them against a SAX ContentHandler.

B.19.2 Variables

Name	Description
__package__	**Value:** `'lxml'`

B.19.3 Class SaxError



General SAX error.

Methods

Inherited from lxml.etree.LxmlError(Section B.6.31)

__init__()

Inherited from exceptions.Exception

__new__()

Inherited from exceptions.BaseException

__delattr__(), __getattribute__(), __getitem__(), __getslice__(), __reduce__(), __repr__(), __setattr__(), __setstate__(), __str__(), __unicode__()

Inherited from object

__format__(), __hash__(), __reduce_ex__(), __sizeof__(), __subclasshook__()

Properties

Name	Description
Inherited from exceptions.BaseException args, message	
Inherited from object __class__	

B.19.4 Class ElementTreeContentHandler

xml.sax.handler.ContentHandler ┐
lxml.sax.ElementTreeContentHandler

Build an lxml ElementTree from SAX events.

Methods

__init__(*self*, *makeelement*=`None`)

Overrides: xml.sax.handler.ContentHandler.__init__

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 extit(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 extit(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 extit(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 extit(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 extit(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 extit(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 extit(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 xml.sax.handler.ContentHandler

skippedEntity()

Properties

Name	Description
etree	Contains the generated ElementTree after parsing is finished.

B.19.5 Class ElementTreeProducer

object ─┐
lxml.sax.ElementTreeProducer

Produces SAX events for an element and children.

Methods

__init__(*self*, *element_or_tree*, *content_handler*)

x.__init__(...) initializes x; see x.__class__.__doc__ for signature Overrides: object.__init__ extit(inherited documentation)

saxify(*self*)

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

Properties

Name	Description
Inherited from object	
__class__	

B.20 Module lxml.usedoctest

Doctest module for XML comparison.

Usage:

```
>>> import lxml.usedoctest
>>> # now do your XML doctests ...
```

See `lxml.doctestcompare`