xml4h Documentation

Release 0.2.0

James Murty

Sep 27, 2017

Contents

1 Features **3**

2 Installation **5**

3 Links **7**

4 Introduction **9**

5 Why? **11**

6 Development Status: beta **13**

7 History **15**
7.1 0.2.0 . . . 15
7.2 0.1.0 . . . 15

8 User Guide **17**
8.1 Parser . . . 17
8.1.1 Parse function . . . 17
8.1.2 Stripping of Whitespace Nodes . . . 18
8.2 Builder . . . 18
8.2.1 Getting Started . . . 19
8.2.2 Method Chaining . . . 20
8.2.3 Shorthand Methods . . . 21
8.2.4 Access the DOM . . . 21
8.2.5 Building on an Existing DOM . . . 22
8.2.6 Hydra-Builder . . . 22
8.3 Writer . . . 23
8.3.1 Write methods . . . 23
8.3.2 Write to a String . . . 24
8.3.3 Format Output . . . 24
8.3.4 Write using the underlying implementation . . . 25
8.4 DOM Nodes . . . 25
8.4.1 Traversing Nodes . . . 25
8.4.2 “Magical” Node Traversal . . . 26
8.4.3 Searching with Find and XPath . . . 27
8.4.4 Filtering Node Lists . . . 30

8.4.5 Manipulating Nodes and Elements . . . 31
8.4.6 Wrapping and Unwrapping *xml4h* Nodes . . . 35
8.5 Advanced . . . 36
8.5.1 Namespaces . . . 36
8.5.2 *xml4h* Architecture . . . 38
8.6 API . . . 42
8.6.1 Main Interface . . . 42
8.6.2 Builder . . . 43
8.6.3 Writer . . . 46
8.6.4 DOM Nodes API . . . 46
8.6.5 XML Libarary Adapters . . . 56
8.6.6 Custom Exceptions . . . 58

9 Indices and tables **59**

Python Module Index **61**

xml4h is an ISC licensed library for Python to make working with XML a human-friendly activity.

This library exists because Python is awesome, XML is everywhere, and combining the two should be a pleasure. With *xml4h*, it can be.

CHAPTER 1

Features

xml4h is a simplification layer over existing Python XML processing libraries such as *lxml*, *ElementTree* and the *minidom*. It provides:

- a rich pythonic API to traverse and manipulate the XML DOM.
- a document builder to simply and safely construct complex documents with minimal code.
- a writer that serialises XML documents with the structure and format that you expect, unlike the machine- but not human-friendly output you tend to get from other libraries.

The *xml4h* abstraction layer also offers some other benefits, beyond a nice API and tool set:

- A common interface to different underlying XML libraries, so code written against *xml4h* need not be rewritten if you switch implementations.
- You can easily move between *xml4h* and the underlying implementation: parse your document using the fastest implementation, manipulate the DOM with human-friendly code using *xml4h*, then get back to the underlying implementation if you need to.

CHAPTER 2

Installation

Install *xml4h* with pip:

```
$ pip install xml4h
```

Or install the tarball manually with:

```
$ python setup.py install
```

CHAPTER 3

Links

- GitHub for source code and issues: http://github.com/jmurty/xml4h
- ReadTheDocs for documentation: http://xml4h.readthedocs.org
- Install from the Python Package Index: http://pypi.python.org/pypi/xml4h

CHAPTER 4

Introduction

Here is an example of parsing and reading data from an XML document using "magical" element and attribute lookups:

```
>>> import xml4h
>>> doc = xml4h.parse('tests/data/monty_python_films.xml')

>>> for film in doc.MontyPythonFilms.Film[:3]:
...     print film['year'], ':', film.Title.text
1971 : And Now for Something Completely Different
1974 : Monty Python and the Holy Grail
1979 : Monty Python's Life of Brian
```

You can also use a more traditional approach to traverse the DOM:

```
>>> for film in doc.child('MontyPythonFilms').children('Film')[:3]:
...     print film.attributes['year'], ':', film.children.first.text
1971 : And Now for Something Completely Different
1974 : Monty Python and the Holy Grail
1979 : Monty Python's Life of Brian
```

The *xml4h* builder makes programmatic document creation simple, with a method-chaining feature that allows for expressive but sparse code that mirrors the document itself:

```
>>> b = (xml4h.build('MontyPythonFilms')
...     .attributes({'source': 'http://en.wikipedia.org/wiki/Monty_Python'})
...     .element('Film')
...         .attributes({'year': 1971})
...         .element('Title')
...             .text('And Now for Something Completely Different')
...             .up()
...         .elem('Description').t(
...             "A collection of sketches from the first and second TV"
...             " series of Monty Python's Flying Circus purposely"
...             " re-enacted and shot for film.").up()
...         .up()
...     )
```

```
>>> # A builder object can be re-used
>>> b = (b.e('Film')
...      .attrs(year=1974)
...      .e('Title').t('Monty Python and the Holy Grail').up()
...      .e('Description').t(
...          "King Arthur and his knights embark on a low-budget search"
...          " for the Holy Grail, encountering humorous obstacles along"
...          " the way. Some of these turned into standalone sketches."
...          ).up()
...      .up()
... )
```

Pretty-print your XML document with the flexible write() and xml() methods:

```
>>> b.write_doc(indent=4, newline=True)
<?xml version="1.0" encoding="utf-8"?>
<MontyPythonFilms source="http://en.wikipedia.org/wiki/Monty_Python">
    <Film year="1971">
        <Title>And Now for Something Completely Different</Title>
        <Description>A collection of sketches from ...</Description>
    </Film>
    <Film year="1974">
        <Title>Monty Python and the Holy Grail</Title>
        <Description>King Arthur and his knights embark ...</Description>
    </Film>
</MontyPythonFilms>
```

CHAPTER 5

Why?

Python has three popular libraries for working with XML, none of which are particularly easy to use:

- xml.dom.minidom is a light-weight, moderately-featured implementation of the W3C DOM that is included in the standard library. Unfortunately the W3C DOM API is terrible – the very opposite of pythonic – and the *minidom* does not support XPath expressions.
- xml.etree.ElementTree is a fast hierarchical data container that is included in the standard library and can be used to represent XML, mostly. The API is fairly pythonic and supports some basic XPath features, but it lacks some DOM traversal niceties you might expect (e.g. to get an element's parent) and when using it you often feel like your working with something subtly different from XML, because you are.
- lxml is a fast, full-featured XML library with an API based on ElementTree but extended. It is your best choice for doing serious work with XML in Python but it is not included in the standard library, it can be difficult to install, and it gives you the same it's-XML-but-not-quite feeling as its ElementTree forebear.

Given these three options it can be difficult to choose which library to use, especially if you're new to XML processing in Python and haven't already used (struggled with) any of them.

In the past your best bet would have been to go with *lxml* for the most flexibility, even though it might be overkill, because at least then you wouldn't have to rewrite your code if you later find you need XPath support or powerful DOM traversal methods.

This is where *xml4h* comes in. It provides an abstraction layer over the existing XML libraries, taking advantage of their power while offering an improved API and tool set.

This project is heavily inspired by the work of Kenneth Reitz such as the excellent Requests HTTP library.

CHAPTER 6

Development Status: beta

Currently *xml4h* includes adapter implementations for all three of the main XML processing Python libraries.

If you have *lxml* available (highly recommended) it will use that, otherwise it will fall back to use the *(c)ElementTree* then the *minidom* libraries.

CHAPTER 7

History

0.2.0

- Add adapter for the *(c)ElementTree* library versions included as standard with Python 2.7+.
- Improved "magical" node traversal to work with lowercase tag names without always needing a trailing underscore. See also improved docs.
- Fixes for: potential errors ASCII-encoding nodes as strings; default XPath namespace from document node; lookup precedence of xmlns attributes.

0.1.0

- Initial alpha release with support for *lxml* and *minidom* libraries.

CHAPTER 8

User Guide

Parser

The *xml4h* parser is a simple wrapper around the parser provided by an underlying *XML library implementation*.

Parse function

To parse XML documents with *xml4h* you feed the `xml4h.parse()` function an XML text document in one of three forms:

- A file-like object:

```
>>> import xml4h

>>> xml_file = open('tests/data/monty_python_films.xml', 'rb')
>>> doc = xml4h.parse(xml_file)

>>> doc.MontyPythonFilms
<xml4h.nodes.Element: "MontyPythonFilms">
```

- A file path string:

```
>>> doc = xml4h.parse('tests/data/monty_python_films.xml')

>>> doc.root['source']
'http://en.wikipedia.org/wiki/Monty_Python'
```

- A string containing literal XML content:

```
>>> xml_file = open('tests/data/monty_python_films.xml', 'rb')
>>> xml_text = xml_file.read()
>>> doc = xml4h.parse(xml_text)
```

```
>>> len(doc.find('Film'))
7
```

Note: The *parse()* method distinguishes between a file path string and an XML text string by looking for a < character in the value.

Stripping of Whitespace Nodes

By default the *parse* method ignores whitespace nodes in the XML document – or more accurately, it does extra work to remove these nodes after the document has been parsed by the underlying XML library.

Whitespace nodes are rarely interesting, since they are usually the result of XML content that has been serialized with extra whitespace to make it more readable to humans.

However if you need to keep these nodes, or if you want to avoid the extra processing overhead when parsing large documents, you can disable this feature by passing in the `ignore_whitespace_text_nodes=False` flag:

```
>>> # Strip whitespace nodes from document
>>> doc = xml4h.parse('tests/data/monty_python_films.xml')

>>> # No excess text nodes (XML doc lists 7 films)
>>> len(doc.MontyPythonFilms.children)
7
>>> doc.MontyPythonFilms.children[0]
<xml4h.nodes.Element: "Film">


>>> # Don't strip whitespace nodes
>>> doc = xml4h.parse('tests/data/monty_python_films.xml',
...                   ignore_whitespace_text_nodes=False)

>>> # An extra text node is present
>>> len(doc.MontyPythonFilms.children)
8
>>> doc.MontyPythonFilms.children[0]
<xml4h.nodes.Text: "#text">
```

Builder

xml4h includes a document builder tool that makes it easy to create valid, well-formed XML documents using relatively sparse python code. It makes it so easy to create XML that you will no longer be tempted to cobble together documents with error-prone methods like manual string concatenation or a templating library.

Internally, the builder uses the DOM-building features of an underlying XML library which means it is (almost) impossible to construct an invalid document.

Here is some example code to build a document about Monty Python films:

```
>>> import xml4h
>>> xmlb = (xml4h.build('MontyPythonFilms')
...     .attributes({'source': 'http://en.wikipedia.org/wiki/Monty_Python'})
...     .element('Film')
...         .attributes({'year': 1971})
```

```
...          .element('Title')
...              .text('And Now for Something Completely Different')
...              .up()
...          .elem('Description').t(
...              "A collection of sketches from the first and second TV"
...              " series of Monty Python's Flying Circus purposely"
...              " re-enacted and shot for film.")
...              .up()
...          .up()
...      .elem('Film')
...          .attrs(year=1974)
...          .e('Title')
...              .t('Monty Python and the Holy Grail')
...              .up()
...          .e('Description').t(
...              "King Arthur and his knights embark on a low-budget search"
...              " for the Holy Grail, encountering humorous obstacles along"
...              " the way. Some of these turned into standalone sketches."
...              ).up()
...      )
```

The code above produces the following XML document (abbreviated):

```
>>> xmlb.write_doc(indent=True)
<?xml version="1.0" encoding="utf-8"?>
<MontyPythonFilms source="http://en.wikipedia.org/wiki/Monty_Python">
    <Film year="1971">
        <Title>And Now for Something Completely Different</Title>
        <Description>A collection of sketches from the first and second...
    </Film>
    <Film year="1974">
        <Title>Monty Python and the Holy Grail</Title>
        <Description>King Arthur and his knights embark on a low-budget...
    </Film>
</MontyPythonFilms>
```

Getting Started

You typically create a new XML document builder by calling the *xml4h.build()* function with the name of the root element:

```
>>> root_b = xml4h.build('RootElement')
```

The function returns a *Builder* object that represents the *RootElement* and allows you to manipulate this element's attributes or to add child elements.

Once you have the first builder instance, every action you perform to add content to the XML document will return another instance of the Builder class:

```
>>> # Add attributes to the root element's Builder
>>> root_b = root_b.attributes({'a': 1, 'b': 2}, c=3)

>>> root_b
<xml4h.builder.Builder object ...
```

The Builder class always represents an underlying element in the DOM. The *dom_element* attribute returns the element node:

```
>>> root_b.dom_element
<xml4h.nodes.Element: "RootElement">

>>> root_b.dom_element.attributes
<xml4h.nodes.AttributeDict: [('a', '1'), ('b', '2'), ('c', '3')]>
```

When you add a new child element, the result is a builder instance representing that child element, *not the original element*:

```
>>> child1_b = root_b.element('ChildElement1')
>>> child2_b = root_b.element('ChildElement2')

>>> # The element method returns a Builder wrapping the new child element
>>> child2_b.dom_element
<xml4h.nodes.Element: "ChildElement2">
>>> child2_b.dom_element.parent
<xml4h.nodes.Element: "RootElement">
```

This feature of the builder can be a little confusing, but it allows for the very convenient method-chaining feature that gives the builder its power.

Method Chaining

Because every builder method that adds content to the XML document returns a builder instance representing the nearest (or newest) element, you can chain together many method calls to construct your document without any need for intermediate variables.

For example, the example code in the previous section used the variables `root_b`, `child1_b` and `child2_b` to represent builder instances but this is not necessary. Here is how you can use method-chaining to build the same document with less code:

```
>>> b = (xml4h
...     .build('RootElement').attributes({'a': 1, 'b': 2}, c=3)
...         .element('ChildElement1').up()  # NOTE the up() method
...         .element('ChildElement2')
...     )

>>> b.write_doc(indent=4)
<?xml version="1.0" encoding="utf-8"?>
<RootElement a="1" b="2" c="3">
    <ChildElement1/>
    <ChildElement2/>
</RootElement>
```

Notice how you can use chained method calls to write code with a structure that mirrors that of the XML document you want to produce? This makes it much easier to spot errors in your code than it would be if you were to concatenate strings.

Note: It is a good idea to wrap the *build()* function call and all following chained methods in parentheses, so you don't need to put backslash (\) characters at the end of every line.

The code above introduces a very important builder method: *up()*. This method returns a builder instance representing the current element's parent, or indeed any ancestor.

Without the `up()` method, every time you created a child element with the builder you would end up deeper in the document structure with no way to return to prior elements to add sibling nodes or hierarchies.

To help reduce the number of `up()` method calls you need to include in your code, this method can also jump up multiple levels or to a named ancestor element:

```
>>> # A builder that references a deeply-nested element:
>>> deep_b = (xml4h.build('Root')
...     .element('Deep')
...         .element('AndDeeper')
...             .element('AndDeeperStill')
...                 .element('UntilWeGetThere')
...     )
>>> deep_b.dom_element
<xml4h.nodes.Element: "UntilWeGetThere">

>>> # Jump up 4 levels, back to the root element
>>> deep_b.up(4).dom_element
<xml4h.nodes.Element: "Root">

>>> # Jump up to a named ancestor element
>>> deep_b.up('Root').dom_element
<xml4h.nodes.Element: "Root">
```

Note: To avoid making subtle errors in your document's structure, we recommend you use *up()* calls to return up one level for every *element()* method (or alias) you call.

Shorthand Methods

To make your XML-producing code even less verbose and quicker to type, the builder has shorthand "alias" methods corresponding to the full names.

For example, instead of calling `element()` to create a new child element, you can instead use the equivalent `elem()` or `e()` methods. Similarly, instead of typing `attributes()` you can use `attrs()` or `a()`.

Here are the methods and method aliases for adding content to an XML document:

XML Node Created	Builder method	Aliases
Element	`element`	`elem`, `e`
Attribute	`attributes`	`attrs`, `a`
Text	`text`	`t`
CDATA	`cdata`	`data`, `d`
Comment	`comment`	`c`
Process Instruction	`processing_instruction`	`inst`, `i`

These shorthand method aliases are convenient and lead to even less cruft around the actual XML content you are interested in. But on the other hand they are much less explicit than the longer versions, so use them judiciously.

Access the DOM

The XML builder is merely a layer of convenience methods that sits on the *xml4h.nodes* DOM API. This means you can quickly access the underlying nodes from a builder if you need to inspect them or manipulate them in a way

the builder doesn't allow:

- The *dom_element* attribute returns a builder's underlying *Element*
- The *root* attribute returns the document's root element.
- The *document* attribute returns a builder's underlying *Document*.

See the *DOM Nodes API* documentation to find out how to work with DOM element nodes once you get them.

Building on an Existing DOM

When you are building an XML document from scratch you will generally use the *build()* function described in *Getting Started*. However, what if you want to add content to a parsed XML document DOM you have already?

To wrap an *Element* DOM node with a builder you simply provide the element node to the same `builder()` method used previously and it will do the right thing.

Here is an example of parsing an existing XML document, locating an element of interest, constructing a builder from that element, and adding some new content. Luckily, the code is simpler than that description...

```
>>> # Parse an XML document
>>> doc = xml4h.parse('tests/data/monty_python_films.xml')

>>> # Find an Element node of interest
>>> lob_film_elem = doc.MontyPythonFilms.Film[2]
>>> lob_film_elem.Title.text
"Monty Python's Life of Brian"

>>> # Construct a builder from the element
>>> lob_builder = xml4h.build(lob_film_elem)

>>> # Add content
>>> b = (lob_builder.attrs(stars=5)
...      .elem('Review').t('One of my favourite films!').up())

>>> # See the results
>>> lob_builder.write(indent=True)
<Film stars="5" year="1979">
    <Title>Monty Python's Life of Brian</Title>
    <Description>Brian is born on the first Christmas, in the stable...
    <Review>One of my favourite films!</Review>
</Film>
```

Hydra-Builder

Because each builder class instance is independent, an advanced technique for constructing complex documents is to use multiple builders anchored at different places in the DOM. In some situations, the ability to add content to different places in the same document can be very handy.

Here is a trivial example of this technique:

```
>>> # Create two Elements in a doc to store even or odd numbers
>>> odd_b = xml4h.build('EvenAndOdd').elem('Odd')
>>> even_b = odd_b.up().elem('Even')

>>> # Populate the numbers from a loop
>>> for i in range(1, 11):
```

```
...     if i % 2 == 0:
...         even_b.elem('Number').text(i)
...     else:
...         odd_b.elem('Number').text(i)
<...

>>> # Check the final document
>>> odd_b.write_doc(indent=True)
<?xml version="1.0" encoding="utf-8"?>
<EvenAndOdd>
    <Odd>
        <Number>1</Number>
        <Number>3</Number>
        <Number>5</Number>
        <Number>7</Number>
        <Number>9</Number>
    </Odd>
    <Even>
        <Number>2</Number>
        <Number>4</Number>
        <Number>6</Number>
        <Number>8</Number>
        <Number>10</Number>
    </Even>
</EvenAndOdd>
```

Writer

The *xml4h* writer produces serialized XML text documents much as you would expect, and in respect that it is a little unlike the writer methods in some of the other Python XML libraries.

Write methods

To write out an XML document with *xml4h* you will generally use the *write()* or *write_doc()* methods available on any *xml4h* node.

The *write()* method outputs the current node and any descendants:

```
>>> import xml4h
>>> doc = xml4h.parse('tests/data/monty_python_films.xml')

>>> first_film_elem = doc.find('Film')[0]
>>> first_film_elem.write(indent=True)
<Film year="1971">
    <Title>And Now for Something Completely Different</Title>
    <Description>A collection of sketches from the first and second...
</Film>
```

The *write_doc()* method outputs the entire document no matter which node you call it on:

```
>>> first_film_elem.write_doc(indent=True)
<?xml version="1.0" encoding="utf-8"?>
<MontyPythonFilms source="http://en.wikipedia.org/wiki/Monty_Python">
    <Film year="1971">
```

```
        <Title>And Now for Something Completely Different</Title>
        <Description>A collection of sketches from the first and second...
    </Film>
 ...
```

The *write* methods send output to `sys.stdout` by default. To send output to a file, or any other writer-like object, provide the target writer as an argument:

```
>>> # Write to a file
>>> with open('/tmp/example.xml', 'wb') as f:
...     first_film_elem.write_doc(f)

>>> # Write to a string (BUT SEE SECTION BELOW...)
>>> from StringIO import StringIO
>>> str_writer = StringIO()
>>> first_film_elem.write_doc(str_writer)
>>> str_writer.getvalue()
'<?xml version="1.0" encoding="utf-8"?><MontyPythonFilms source...
```

Write to a String

Because you will often want to generate a string of XML content directly, *xml4h* includes the convenience methods *xml()* and *xml_doc()* to do this easily.

The *xml()* method works like the *write* method and will return a string of XML content including the current node and its descendants:

```
>>> print first_film_elem.xml()
<Film year="1971">
    <Title>And Now for Something Completely...
```

The *xml_doc()* method works like the *write_doc* method and returns a string for the whole document:

```
>>> print first_film_elem.xml_doc()
<?xml version="1.0" encoding="utf-8"?>
<MontyPythonFilms source="http://en.wikipedia.org/wiki/Monty_Python">
    <Film year="1971">
        <Title>And Now for Something Completely Different</Title>
        <Description>A collection of sketches from the first and second...
    </Film>
    ...
```

Note: *xml4h* assumes that when you directly generate an XML string in this way it is intended for human consumption, so it applies pretty-print formatting by default.

Format Output

The *write* and *xml* methods accept a range of formatting options to control how XML content is serialized. These are useful if you expect a human to read the resulting data.

For the full range of formatting options see the code documentation for *write()* and *xml()* et al. but here are some pointers to get you started:

- Set `indent=True` to write a pretty-printed XML document with four space characters for indentation and `\n` for newlines.
- To use a tab character for indenting and `\r\n` for indents: `indent='\t', newline='\r\n'`.
- *xml4h* writes *utf-8*-encoded documents by default, to write with a different encoding: `encoding='iso-8859-1'`.
- To avoid outputting the XML declaration when writing a document: `omit_declaration=True`.

Write using the underlying implementation

Because *xml4h* sits on top of an underlying *XML library implementation* you can use that library's serialization methods if you prefer, and if you don't mind having some implementation-specific code.

For example, if you are using *lxml* as the underlying library you can use its serialisation methods by accessing the implementation node:

```
>>> # Get the implementation root node, in this case an lxml node
>>> lxml_root_node = first_film_elem.root.impl_node
>>> lxml_root_node.__class__
<type 'lxml.etree._Element'>

>>> # Use lxml features as normal; xml4h is no longer in the picture
>>> from lxml import etree
>>> print etree.tostring(lxml_root_node, encoding='utf-8',
...                      xml_declaration=True, pretty_print=True)
<?xml version='1.0' encoding='utf-8'?>
<MontyPythonFilms source="http://en.wikipedia.org/wiki/Monty_Python"><Film year="1971
↪"><Title>And Now for Something Completely Different</Title>
        <Description>A collection of sketches from the first and second...
    </Film>
    <Film year="1974"><Title>Monty Python and the Holy Grail</Title>
        <Description>King Arthur and his knights embark on a low-budget...
    </Film>
    ...
```

Note: The output from *lxml* is a little quirky, at least on the author's machine. Note for example the single-quote characters in the XML declaration, and the missing newline and indent before the first `<Film>` element. But don't worry, that's why you have *xml4h* ;)

DOM Nodes

xml4h provides node objects and convenience methods that make it easier to work with an in-memory XML document object model (DOM).

This section of the document covers the main features of *xml4h* nodes. For the full API-level documentation see *DOM Nodes API*.

Traversing Nodes

xml4h aims to provide a simple and intuitive API for traversing and manipulating the XML DOM. To that end it includes a number of convenience methods for performing common tasks:

- Get the *Document* or root *Element* from any node via the `document` and `root` attributes respectively.
- You can get the `name` attribute of nodes that have a name, or look up the different name components with `prefix` to get the namespace prefix (if any) and `local_name` to get the name portion without the prefix.
- Nodes that have a value expose it via the `value` attribute.
- A node's `parent` attribute returns its parent, while the `ancestors` attribute returns a list containing its parent, grand-parent, great-grand-parent etc.
- A node's `children` attribute returns the child nodes that belong to it, while the `siblings` attribute returns all other nodes that belong to its parent. You can also get the `siblings_before` or `siblings_after` the current node.
- Look up a node's namespace URI with `namespace_uri` or the alias `ns_uri`.
- Check what type of *Node* you have with Boolean attributes like `is_element`, `is_text`, `is_entity` etc.

"Magical" Node Traversal

To make it easy to traverse XML documents with a known structure *xml4h* performs some minor magic when you look up attributes or keys on Document and Element nodes. If you like, you can take advantage of magical traversal to avoid peppering your code with `find` and `xpath` searches, or with `child` and `children` node attribute lookups.

The principle is simple:

- Child elements are available as Python attributes of the parent element class.
- XML element attributes are available as a Python dict in the owning element.

Here is an example of retrieving information from our Monty Python films document using element names as Python attributes (`MontyPythonFilms`, `Film`, `Title`) and XML attribute names as Python keys (`year`):

```
>>> # Parse an example XML document about Monty Python films
>>> import xml4h
>>> doc = xml4h.parse('tests/data/monty_python_films.xml')

>>> for film in doc.MontyPythonFilms.Film:
...     print film['year'], ':', film.Title.text
1971 : And Now for Something Completely Different
1974 : Monty Python and the Holy Grail
...
```

Python class attribute lookups of child elements work very well when your XML document contains only camel-case tag names `LikeThisOne` or `LikeThat`. However, if your document contains lower-case tag names there is a chance the element names will clash with existing Python attribute or method names in the *xml4h* classes.

To work around this potential issue you can add an underscore (_) character at the end of a magical attribute lookup to avoid the naming clash; *xml4h* will remove that character before looking for a child element. For example, to look up a child of the element `elem1` which is named `child`, the code `elem1.child_` will return the child element whereas `elem1.child` would access the *child()* Node method instead.

Note: Not all XML child element tag names are accessible using magical traversal. Names with leading underscore characters will not work, and nor will names containing hyphens because they are not valid Python attribute names. If you have to deal with XML names like this use the full API methods like *child()* and *children()* instead.

All the gory details about how magical traversal works are documented at *NodeAttrAndChildElementLookupsMixin*. Depending on how you feel about magical behaviour this

feature might feel like a great convenience, or black magic that makes you wary. The right attitude probably lies somewhere in the middle...

Warning: The behaviour of namespaced XML elements and attributes is inconsistent. You can do magical traversal of elements regardless of what namespace the elements are in, but to look up XML attributes with a namespace prefix you must include that prefix in the name e.g. `prefix:attribute-name`.

Searching with Find and XPath

There are two ways to search for elements within an *xml4h* document: `find` and `xpath`.

The find methods provided by the library are easy to use but can only perform relatively simple searches that return *Element* results, whereas you need to be familiar with XPath query syntax to search effectively with the `xpath` method but you can perform more complex searches and get results other than just elements.

Find Methods

xml4h provides three different find methods:

- *find()* searches descendants of the current node for elements matching the given constraints. You can search by element name, by namespace URI, or with no constraints at all:

```
>>> # Find ALL elements in the document
>>> elems = doc.find()
>>> [e.name for e in elems]
[u'MontyPythonFilms', u'Film', u'Title', u'Description', u'Film', u'Title', u
↪'Description',...

>>> # Find the seven <Film> elements in the XML document
>>> film_elems = doc.find('Film')
>>> [e.Title.text for e in film_elems]
['And Now for Something Completely Different', 'Monty Python and the Holy Grail',.
↪..
```

  Note that the *find()* method only finds descendants of the node you run it on:

```
>>> # Find <Title> elements in a single <Film> element; there's only one
>>> film_elem = doc.find('Film', first_only=True)
>>> film_elem.find('Title')
[<xml4h.nodes.Element: "Title">]
```

- *find_first()* searches descendants of the current node but only returns the first result element, not a list. If there are no matching element results this method returns *None*:

```
>>> # Find the first <Film> element in the document
>>> doc.find_first('Film')
<xml4h.nodes.Element: "Film">

>>> # Search for an element that does not exist
>>> print doc.find_first('OopsWrongName')
None
```

  If you were paying attention you may have noticed in the example above that you can make the *find()* method do exactly same thing as *find_first()* by passing the keyword argument `first_only=True`.

- *find_doc()* is a convenience method that searches the entire document no matter which node you run it on:

```
>>> # Normal find only searches descendants of the current node
>>> len(film_elem.find('Title'))
1

>>> # find_doc searches the entire document
>>> len(film_elem.find_doc('Title'))
7
```

  This method is exactly like calling `xml4h_node.document.find()`, which is actually what happens behind the scenes.

XPath Querying

xml4h provides a single XPath search method which is available on *Document* and *Element* nodes:

xpath() takes an XPath query string and returns the result which may be a list of elements, a list of attributes, a list of values, or a single value. The result depends entirely on the kind of query you perform.

Note: XPath querying is currently only available if you use the *lxml* or *ElementTree* implementation libraries. You can check whether the XPath feature is available with *has_feature()*.

Note: Although *ElementTree* supports XPath queries, this support is very limited and most of the example XPath queries below **will not work**. If you want to use XPath, you should install *lxml* for better support.

XPath queries are powerful and complex so we cannot describe them in detail here, but we can at least present some useful examples. Here are queries that perform the same work as the find queries we saw above:

```
>>> # Query for ALL elements in the document
>>> elems = doc.xpath('//*')
>>> [e.name for e in elems]
[u'MontyPythonFilms', u'Film', u'Title', u'Description', u'Film', u'Title', u
↪'Description',...

>>> # Query for the seven <Film> elements in the XML document
>>> film_elems = doc.xpath('//Film')
>>> [e.Title.text for e in film_elems]
['And Now for Something Completely Different', 'Monty Python and the Holy Grail',...

>>> # Query for the first <Film> element in the document (returns list)
>>> doc.xpath('//Film[1]')
[<xml4h.nodes.Element: "Film">]

>>> # Query for <Title> elements in a single <Film> element; there's only one
>>> film_elem = doc.xpath('Film[1]')[0]
>>> film_elem.xpath('Title')
[<xml4h.nodes.Element: "Title">]
```

You can also do things with XPath queries that you simply cannot with the *find* method, such as find all the attributes of a certain name or apply rich constraints to the query:

```
>>> # Query for all year attributes
>>> doc.xpath('//@year')
```

```
['1971', '1974', '1979', '1982', '1983', '2009', '2012']

>>> # Query for the title of the film released in 1982
>>> doc.xpath('//Film[@year="1982"]/Title/text()')
['Monty Python Live at the Hollywood Bowl']
```

Namespaces and XPath

Finally, let's discuss how you can run XPath queries on documents with namespaces, because unfortunately this is not a simple subject.

First, you need to understand that if you are working with a namespaced document your XPath queries must refer to those namespaces or they will not find anything:

```
>>> # Parse a namespaced version of the Monty Python Films doc
>>> ns_doc = xml4h.parse('tests/data/monty_python_films.ns.xml')
>>> ns_doc.write(indent=True)
<?xml version="1.0" encoding="utf-8"?>
<MontyPythonFilms source="http://en.wikipedia.org/wiki/Monty_Python" xmlns="uri:monty-
↪python" xmlns:work="uri:artistic-work">
    <work:Film year="1971">
        <Title>And Now for Something Completely Different</Title>
        ...

>>> # XPath queries without prefixes won't find namespaced elements
>>> ns_doc.xpath('//Film')
[]
```

To refer to namespaced nodes in your query the namespace must have a prefix alias assigned to it. You can specify prefixes when you call the *xpath* method by providing a `namespaces` keyword argument with a dictionary of alias-to-URI mappings:

```
>>> # Specify explicit prefix alias mappings
>>> films = ns_doc.xpath('//x:Film', namespaces={'x': 'uri:artistic-work'})
>>> len(films)
7
```

Or, preferably, if your document node already has prefix mappings you can use them directly:

```
>>> # Our root node already has a 'work' prefix defined...
>>> ns_doc.root['xmlns:work']
'uri:artistic-work'

>>> # ...so we can use this prefix directly
>>> films = ns_doc.root.xpath('//work:Film')
>>> len(films)
7
```

Another gotcha is when a document has a default namespace. The default namespace applies to every descendent node without its own namespace, but XPath doesn't have a good way of dealing with this since there is no such thing as a "default namespace" prefix alias.

xml4h helps out by providing just such an alias: the underscore (_):

```
>>> # Our document root has a default namespace
>>> ns_doc.root.ns_uri
```

```
'uri:monty-python'

>>> # You need a prefix alias that refers to the default namespace
>>> ns_doc.xpath('//Title')
[]

>>> # You could specify it explicitly...
>>> titles = ns_doc.xpath('//x:Title',
...                       namespaces={'x': ns_doc.root.ns_uri})
>>> len(titles)
7

>>> # ...or use xml4h's special default namespace prefix: _
>>> titles = ns_doc.xpath('//_:Title')
>>> len(titles)
7
```

Filtering Node Lists

Many *xml4h* node attributes return a list of nodes as a *NodeList* object which confers some special filtering powers. You get this special node list object from attributes like `children`, `ancestors`, and `siblings`, and from the `find` search method if it has element results.

Here are some examples of how you can easily filter a *NodeList* to get just the nodes you need:

- Get the first child node using the `filter` method:

```
>>> # Filter to get just the first child
>>> doc.root.children.filter(first_only=True)
<xml4h.nodes.Element: "Film">

>>> # The document has 7 <Film> element children of the root
>>> len(doc.root.children)
7
```

- Get the first child node by treating `children` as a callable:

```
>>> doc.root.children(first_only=True)
<xml4h.nodes.Element: "Film">
```

  When you treat the node list as a callable it calls the `filter` method behind the scenes, but since doing it the callable way is quicker and clearer in code we will use that approach from now on.

- Get the first child node with the `child` filtering method, which accepts the same constraints as the `filter` method:

```
>>> doc.root.child()
<xml4h.nodes.Element: "Film">

>>> # Apply filtering with child
>>> print doc.root.child('WrongName')
None
```

- Get the first of a set of children with the `first` attribute:

```
>>> doc.root.children.first
<xml4h.nodes.Element: "Film">
```

- Filter the node list by name:

```
>>> for n in doc.root.children('Film'):
...     print n.Title.text
And Now for Something Completely Different
Monty Python and the Holy Grail
Monty Python's Life of Brian
Monty Python Live at the Hollywood Bowl
Monty Python's The Meaning of Life
Monty Python: Almost the Truth (The Lawyer's Cut)
A Liar's Autobiography: Volume IV

>>> len(doc.root.children('WrongName'))
0
```

Note: Passing a node name as the first argument will match the *local* name of a node. You can match the full node name, which might include a prefix for example, with a call like: `.children(name='SomeName')`.

- Filter with a custom function:

```
>>> # Filter to films released in the year 1979
>>> for n in doc.root.children('Film',
...         filter_fn=lambda node: node.attributes['year'] == '1979'):
...     print n.Title.text
Monty Python's Life of Brian
```

Manipulating Nodes and Elements

xml4h provides simple methods to manipulate the structure and content of an XML DOM. The methods available depend on the kind of node you are interacting with, and by far the majority are for working with *Element* nodes.

Delete a Node

Any node can be removes from its owner document with *delete()*:

```
>>> # Before deleting a Film element there are 7 films
>>> len(doc.MontyPythonFilms.Film)
7

>>> doc.MontyPythonFilms.children('Film')[-1].delete()
>>> len(doc.MontyPythonFilms.Film)
6
```

Note: By default deleting a node also destroys it, but it can optionally be left intact after removal from the document by including the `destroy=False` option.

Name and Value Attributes

Many nodes have low-level name and value properties that can be read from and written to. Nodes with names and values include Text, CDATA, Comment, ProcessingInstruction, Attribute, and Element nodes.

Here is an example of accessing the low-level name and value properties of a Text node:

```
>>> text_node = doc.MontyPythonFilms.child('Film').child('Title').child()
>>> text_node.is_text
True

>>> text_node.name
u'#text'
>>> text_node.value
u'And Now for Something Completely Different'
```

And here is the same for an Attribute node:

```
>>> # Access the name/value properties of an Attribute node
>>> year_attr = doc.MontyPythonFilms.child('Film').attribute_node('year')
>>> year_attr.is_attribute
True

>>> year_attr.name
u'year'
>>> year_attr.value
u'1971'
```

The name attribute of a node is not necessarily a plain string, in the case of nodes within a defined namespaced the `name` attribute may comprise two components: a `prefix` that represents the namespace, and a `local_name` which is the plain name of the node ignoring the namespace. For more information on namespaces see *Namespaces*.

Import a Node and its Descendants

In addition to manipulating nodes in a single XML document directly, you can also import a node (and all its descendant) from another document using a node clone or transplant operation.

There are two ways to import a node and its descendants:

- Use the *`clone_node()`* Node method or *`clone()`* Builder method to copy a node into your document without removing it from its original document.
- Use the *`transplant_node()`* Node method or *`transplant()`* Builder method to transplant a node into your document and remove it from its original document.

Here is an example of transplanting a node into a document (which also happens to undo the damage we did to our example DOM in the `delete()` example above):

```
>>> # Build a new document containing a Film element
>>> film_builder = (xml4h.build('DeletedFilm')
...     .element('Film').attrs(year='1971')
...         .element('Title')
...             .text('And Now for Something Completely Different').up()
...         .element('Description').text(
...             "A collection of sketches from the first and second TV"
...             " series of Monty Python's Flying Circus purposely"
...             " re-enacted and shot for film.")
...     )

>>> # Transplant the Film element from the new document
>>> node_to_transplant = film_builder.root.child('Film')
>>> doc.MontyPythonFilms.transplant_node(node_to_transplant)
```

```
>>> len(doc.MontyPythonFilms.Film)
7
```

When you transplant a node from another document it is removed from that document:

```
>>> # After transplanting the Film node it is no longer in the original doc
>>> len(film_builder.root.find('Film'))
0
```

If you need to leave the original document unchanged when importing a node use the clone methods instead.

Working with Elements

Element nodes have the most methods to access and manipulate their content, which is fitting since this is the most useful type of node and you will deal with elements regularly.

The leaf elements in XML documents often have one or more *Text* node children that contain the element's data content. While you could iterate over such text nodes as child nodes, *xml4h* provides the more convenient text accessors you would expect:

```
>>> title_elem = doc.MontyPythonFilms.Film[0].Title
>>> orig_title = title_elem.text
>>> orig_title
'And Now for Something Completely Different'

>>> title_elem.text = 'A new, and wrong, title'
>>> title_elem.text
'A new, and wrong, title'

>>> # Let's put it back the way it was...
>>> title_elem.text = orig_title
```

Elements also have attributes that can be manipulated in a number of ways.

Look up an element's attributes with:

- the *attributes()* attribute (or aliases `attrib` and `attrs`) that return an ordered dictionary of attribute names and values:

  ```
  >>> film_elem = doc.MontyPythonFilms.Film[0]
  >>> film_elem.attributes
  <xml4h.nodes.AttributeDict: [('year', '1971')]>
  ```

- or by obtaining an element's attributes as *Attribute* nodes, though that is only likely to be useful in unusual circumstances:

  ```
  >>> film_elem.attribute_nodes
  [<xml4h.nodes.Attribute: "year">]

  >>> # Get a specific attribute node by name or namespace URI
  >>> film_elem.attribute_node('year')
  <xml4h.nodes.Attribute: "year">
  ```

- and there's also the "magical" keyword lookup technique discussed in *"Magical" Node Traversal* for quickly grabbing attribute values.

Set attribute values with:

- the *set_attributes()* method, which allows you to add attributes without replacing existing ones. This method also supports defining XML attributes as a dictionary, list of name/value pairs, or keyword arguments:

```
>>> # Set/add attributes as a dictionary
>>> film_elem.set_attributes({'a1': 'v1'})

>>> # Set/add attributes as a list of name/value pairs
>>> film_elem.set_attributes([('a2', 'v2')])

>>> # Set/add attributes as keyword arguments
>>> film_elem.set_attributes(a3='v3', a4=4)

>>> film_elem.attributes
<xml4h.nodes.AttributeDict: [('a1', 'v1'), ('a2', 'v2'), ('a3', 'v3'), ('a4', '4
↪'), ('year', '1971')]>
```

- the setter version of the *attributes* attribute, which replaces any existing attributes with the new set:

```
>>> film_elem.attributes = {'year': '1971', 'note': 'funny'}
>>> film_elem.attributes
<xml4h.nodes.AttributeDict: [('note', 'funny'), ('year', '1971')]>
```

Delete attributes from an element by:

- using Python's delete-in-dict technique:

```
>>> del(film_elem.attributes['note'])
>>> film_elem.attributes
<xml4h.nodes.AttributeDict: [('year', '1971')]>
```

- or by calling the `delete()` method on an *Attribute* node.

Finally, the *Element* class provides a number of methods for programmatically adding child nodes, for cases where you would rather work directly with nodes instead of using a *Builder*.

The most complex of these methods is *add_element()* which allows you to add a named child element, and to optionally to set the new element's namespace, text content, and attributes all at the same time. Let's try an example:

```
>>> # Add a Film element with an attribute
>>> new_film_elem = doc.MontyPythonFilms.add_element(
...     'Film', attributes={'year': 'never'})

>>> # Add a Description element with text content
>>> desc_elem = new_film_elem.add_element(
...     'Description', text='Just testing...')

>>> # Add a Title element with text *before* the description element
>>> title_elem = desc_elem.add_element(
...     'Title', text='The Film that Never Was', before_this_element=True)

>>> print doc.MontyPythonFilms.Film[-1].xml()
<Film year="never">
    <Title>The Film that Never Was</Title>
    <Description>Just testing...</Description>
</Film>
```

There are similar methods for handling simpler cases like adding text nodes, comments etc. Here is an example of adding text nodes:

```
>>> # Add a text node
>>> title_elem = doc.MontyPythonFilms.Film[-1].Title
>>> title_elem.add_text(', and Never Will Be')

>>> title_elem.text
'The Film that Never Was, and Never Will Be'
```

Refer to the *Element* documentation for more information about the other methods for adding nodes.

Wrapping and Unwrapping *xml4h* Nodes

You can easily convert to or from *xml4h*'s wrapped version of an implementation node. For example, if you prefer the *lxml* library's ElementMaker document builder approach to the *xml4h Builder*, you can create a document in *lxml*...

```
>>> from lxml.builder import ElementMaker
>>> E = ElementMaker()
>>> lxml_doc = E.DocRoot(
...     E.Item(
...         E.Name('Item 1'),
...         E.Value('Value 1')
...     ),
...     E.Item(
...         E.Name('Item 2'),
...         E.Value('Value 2')
...     )
... )
>>> lxml_doc
<Element DocRoot at ...
```

...and then convert (or, more accurately, wrap) the *lxml* nodes with the appropriate adapter to make them *xml4h* versions:

```
>>> # Convert lxml Document to xml4h version
>>> xml4h_doc = xml4h.LXMLAdapter.wrap_document(lxml_doc)
>>> xml4h_doc.children
[<xml4h.nodes.Element: "Item">, <xml4h.nodes.Element: "Item">]

>>> # Get an element within the lxml document
>>> lxml_elem = list(lxml_doc)[0]
>>> lxml_elem
<Element Item at ...

>>> # Convert lxml Element to xml4h version
>>> xml4h_elem = xml4h.LXMLAdapter.wrap_node(lxml_elem, lxml_doc)
>>> xml4h_elem
<xml4h.nodes.Element: "Item">
```

You can reach the underlying XML implementation document or node at any time from an *xml4h* node:

```
>>> # Get an xml4h node's underlying implementation node
>>> xml4h_elem.impl_node
<Element Item at ...
>>> xml4h_elem.impl_node == lxml_elem
True

>>> # Get the underlying implementatation document from any node
```

```
>>> xml4h_elem.impl_document
<Element DocRoot at ...
>>> xml4h_elem.impl_document == lxml_doc
True
```

Advanced

Namespaces

xml4h supports using XML namespaces in a number of ways, and tries to make this sometimes complex and fiddly aspect of XML a little easier to deal with.

Namespace URIs

XML document nodes can be associated with a *namespace URI* which uniquely identifies the namespace. At bottom a URI is really just a name to identifiy the namespace, which may or may not point at an actual resource.

Namespace URIs are the core piece of the namespacing puzzle, everything else is extras.

Namespace URI values are assigned to a node in one of three ways:

- an `xmlns` attribute on an element assigns a *namespace URI* to that element, and may also define a shorthand *prefix* for the namespace:

  ```
  <AnElement xmlns:my-prefix="urn:example-uri">
  ```

  Note: Technically the `xmlns` attribute must itself also be in the special XML namespacing namespace http://www.w3.org/2000/xmlns/. You needn't care about this.

- a tag or attribute name includes a *prefix* alias portion that specifies the namespace the item belongs to:

  ```
  <my-prefix:AnotherElement attr1="x" my-prefix:attr2="i am namespaced">
  ```

  A prefix alias can be defined using an "xmlns" attribute as described above, or by using the Builder `ns_prefix()` or Node `set_ns_prefix()` methods.

- in an apparent effort to reduce confusion around namespace URIs and prefixes, some XML libraries avoid prefix aliases altogether and instead require you to specify the full *namespace URI* as a prefix to tag and attribute names using a special syntax with braces:

  ```
  >>> tagname = '{urn:example-uri}YetAnotherWayToNamespace'
  ```

  Note: In the author's opinion, using a non-standard way to define namespaces does not reduce confusion. *xml4h* supports this approach technically but not philosphically.

xml4h allows you to assign namespace URIs to document nodes when using the Builder:

```
>>> # Assign a default namespace with ns_uri
>>> import xml4h
>>> b = xml4h.build('Doc', ns_uri='ns-uri')
>>> root = b.root
```

```
>>> # Descendent without a namespace inherit their ancestor's default one
>>> elem1 = b.elem('Elem1').dom_element
>>> elem1.namespace_uri
'ns-uri'

>>> # Define a prefix alias to assign a new or existing namespace URI
>>> elem2 = b.ns_prefix('my-ns', 'second-ns-uri') \
...     .elem('my-ns:Elem2').dom_element
>>> print root.xml()
<Doc xmlns="ns-uri" xmlns:my-ns="second-ns-uri">
    <Elem1/>
    <my-ns:Elem2/>
</Doc>

>>> # Or use the explicit URI prefix approach, if you must
>>> elem3 = b.elem('{third-ns-uri}Elem3').dom_element
>>> elem3.namespace_uri
'third-ns-uri'
```

And when adding nodes with the API:

```
>>> # Define the ns_uri argument when creating a new element
>>> elem4 = root.add_element('Elem4', ns_uri='fourth-ns-uri')

>>> # Attributes can be namespaced too
>>> elem4.set_attributes({'my-ns:attr1': 'value'})

>>> print elem4.xml()
<Elem4 my-ns:attr1="value" xmlns="fourth-ns-uri"/>
```

Filtering by Namespace

xml4h allows you to find and filter nodes based on their namespace.

The *find()* method takes a `ns_uri` keyword argument to return only elements in that namespace:

```
>>> # By default, find ignores namespaces...
>>> [n.local_name for n in root.find()]
[u'Elem1', u'Elem2', u'Elem3', u'Elem4']
>>> # ...but will filter by namespace URI if you wish
>>> [n.local_name for n in root.find(ns_uri='fourth-ns-uri')]
[u'Elem4']
```

Similarly, a node's children listing can be filtered:

```
>>> len(root.children)
4
>>> root.children(ns_uri='ns-uri')
[<xml4h.nodes.Element: "Elem1">]
```

XPath queries can also filter by namespace, but the `xpath()` method needs to be given a dictionary mapping of prefix aliases to URIs:

```
>>> root.xpath('//ns4:*', namespaces={'ns4': 'fourth-ns-uri'})
[<xml4h.nodes.Element: "Elem4">]
```

Note: Normally, because XPath queries rely on namespace prefix aliases, they cannot find namespaced nodes in the default namespace which has an "empty" prefix name. *xml4h* works around this limitation by providing the special empty/default prefix alias '_'.

Element Names: Local and Prefix Components

When you use a namespace prefix alias to define the namespace an element or attribute belongs to, the name of that node will be made up of two components:

- *prefix* - the namespace alias.
- *local* - the real name of the node, without the namespace alias.

xml4h makes the full (qualified) name, and the two components, available at node attributes:

```
>>> # Elem2's namespace was defined earlier using a prefix alias
>>> elem2
<xml4h.nodes.Element: "my-ns:Elem2">

# The full node name...
>>> elem2.name
u'my-ns:Elem2'
>>> # ...comprises a prefix...
>>> elem2.prefix
u'my-ns'
>>> # ...and a local name component
>>> elem2.local_name
u'Elem2'

>>> # Here is an element without a prefix alias
>>> elem1.name
u'Elem1'
>>> elem1.prefix == None
True
>>> elem1.local_name
u'Elem1'
```

xml4h Architecture

To best understand the *xml4h* library and to use it appropriately in demanding situations, you should appreciate what the library is not.

xml4h is not a full-fledged XML library in its own right, far from it. Instead of implementing low-level document parsing and manipulation tools, it operates as an abstraction layer on top of the pre-existing XML processing libraries you already know.

This means the improved API and tool suite provided by *xml4h* work by mediating operations you perform, asking the underlying XML library to do the work, and packaging up the results of this work as wrapped *xml4h* objects.

This approach has a number of implications, good and bad.

On the good side:

- you can start using and benefiting from *xml4h* in an existing projects that already use a supported XML library without any impact, it can fit right in.

- *xml4h* can take advantage of the existing powerful and fast XML libraries to do its work.
- by providing an abstraction layer over multiple libraries, *xml4h* can make it (relatively) easy to switch the underlying library without you needing to rewrite your own XML handling code.
- by building on the shoulders of giants, *xml4h* itself can remain relatively lightweight and focussed on simplicity and usability.
- the author of *xml4h* does not have to write XML-handling code in C...

On the bad side:

- if the underlying XML libraries available in the Python environment do not support a feature (like XPath querying) then that feature will not be available in *xml4h*.
- *xml4h* cannot provide radical new XML processing features, since the bulk of its work must be done by the underlying library.
- the abstraction layer *xml4h* uses to do its work requires more resources than it would to use the underlying library directly, so if you absolutely need maximal speed or minimal memory use the library might prove too expensive.
- *xml4h* sometimes needs to jump through some hoops to maintain the shared abstraction interface over multiple libraries, which means extra work is done in Python instead of by the underlying library code in C.

The author believes the benefits of using *xml4h* outweighs the drawbacks in the majority of real-world situations, or he wouldn't have created the library in the first place, but ultimately it is up to you to decide where you should or should not use it.

Library Adapters

To provide an abstraction layer over multiple underlying XML libraries, *xml4h* uses an "adapter" mechanism to mediate operations on documents. There is an adapter implementation for each library *xml4h* can work with, each of which extends the `XmlImplAdapter` class. This base class includes some standard behaviour, and defines the interface for adapter implementations (to the extent you can define such interfaces in Python).

The current version of *xml4h* includes adapter implementations for the three main XML processing libraries for Python:

- `LXMLAdapter` works with the excellent lxml library which is very full-featured and fast, but which is not included in the standard library.
- `cElementTreeAdapter` and `ElementTreeAdapter` work with the *ElementTree* libraries included with the standard library of Python versions 2.7 and later. *ElementTree* is fast and includes support for some basic XPath expressions. If the C-based version of ElementTree is available, the former adapter is made available and should be used for best performance.
- `XmlDomImplAdapter` works with the minidom W3C-style XML library included with the standard library. This library is always available but is slower and has fewer features than alternative libraries (e.g. no support for XPath)

The adapter layer allows the rest of the *xml4h* library code to remain almost entirely oblivious to the underlying XML library that happens to be available at the time. The *xml4h* Builder, Node objects, writer etc. call adapter methods to perform document operations, and the adapter is responsible for doing the necessary work with the underlying library.

"Best" Adapter

While *xml4h* can work with multiple underlying XML libraries, some of these libraries are better (faster, more fully-featured) than others so it would be smart to use the best of the libraries available.

xml4h does exactly that: unless you explicitly choose an adapter (see below) *xml4h* will find the supported libraries in the Python environment and choose the "best" adapter for you in the circumstances.

Here is the list of libraries *xml4h* will choose from, best to least-best:

- *lxml*
- *(c)ElementTree*
- *ElementTree*
- *minidom*

The `xml4h.best_adapter` attribute stores the adapter class that *xml4h* considers to be the best.

Choose Your Own Adapter

By default, *xml4h* will choose an adapter and underlying XML library implementation that it considers the best available. However, in some cases you may need to have full control over which underlying implementation *xml4h* uses, perhaps because you will use features of the underlying XML implementation later on, or because you need the performance characteristics only available in a particular library.

For these situations it is possible to tell *xml4h* which adapter implementation, and therefore which underlying XML library, it should use.

To use a specific adapter implementation when parsing a document, or when creating a new document using the builder, simply provide the optional `adapter` keyword argument to the relevant method:

- Parsing:

```
>>> # Explicitly use the minidom adapter to parse a document
>>> minidom_doc = xml4h.parse('tests/data/monty_python_films.xml',
...                           adapter=xml4h.XmlDomImplAdapter)
>>> minidom_doc.root.impl_node
<DOM Element: MontyPythonFilms at ...
```

- Building:

```
>>> # Explicitly use the lxml adapter to build a document
>>> lxml_b = xml4h.build('MyDoc', adapter=xml4h.LXMLAdapter)
>>> lxml_b.root.impl_node
<Element {http://www.w3.org/2000/xmlns/}MyDoc at ...
```

- Manipulating:

```
>>> # Use xml4h with a cElementTree document object
>>> import xml.etree.ElementTree as ET
>>> et_doc = ET.parse('tests/data/monty_python_films.xml')
>>> et_doc
<xml.etree.ElementTree.ElementTree object ...
>>> doc = xml4h.cElementTreeAdapter.wrap_document(et_doc)
>>> doc.root
<xml4h.nodes.Element: "MontyPythonFilms">
```

Check Feature Support

Because not all underlying XML libraries support all the features exposed by *xml4h*, the library includes a simple mechanism to check whether a given feature is available in the current Python environment or with the current adapter.

To check for feature support call the *has_feature()* method on a document node, or `has_feature()` on an adapter class.

List of features that are not available in all adapters:

- `xpath` - Can perform XPath queries using the `xpath()` method.
- More to come later, probably...

For example, here is how you would test for XPath support in the *minidom* adapter, which doesn't include it:

```
>>> minidom_doc.root.has_feature('xpath')
False
```

If you forget to check for a feature and use it anyway, you will get a *FeatureUnavailableException*:

```
>>> try:
...     minidom_doc.root.xpath('//*')
... except Exception, e:
...     e
FeatureUnavailableException('xpath',)
```

Adapter & Implementation Quirks

Although *xml4h* aims to provide a seamless abstraction over underlying XML library implementations this isn't always possible, or is only possible by performing lots of extra work that affects performance. This section describes some implementation-specific quirks or differences you may encounter.

LXMLAdapter - *lxml*

- *lxml* does not have full support for CDATA nodes, which devolve into plain text node values when written (by *xml4h* or by *lxml*'s writer).
- Namespaces defined by adding `xmlns` element attributes are not properly represented in the underlying implementation due to the *lxml* library's immutable `nsmap` namespace map. Such namespaces are written correcly by the *xml4h* writer, but to avoid quirks it is best to specify namespace when creating nodes by setting the `ns_uri` keyword attribute.
- When *xml4h* writes *lxml*-based documents with namespaces, some node tag names may have unnecessary namespace prefix aliases.

(c)ElementTreeAdapter - *ElementTree*

- Only the versions of (c)ElementTree included with Python version 2.7 and later are supported.
- *ElementTree* supports only a very limited subset of XPath for querying, so although the `has_feature('xpath')` check returns `True` don't expect to get the full power of XPath when you use this adapter.
- *ElementTree* does not have full support for CDATA nodes, which devolve into plain text node values when written (by *xml4h* or by *ElementTree*'s writer).
- Because *ElementTree* doesn't retain information about a node's parent, *xml4h* needs to build and maintain its own records of which nodes are parents of which children. This extra overhead might harm performance or memory usage.

- *ElementTree* doesn't normally remember explicit namespace definition directives when parsing a document. *xml4h* works around this when it is asked to parse XML data, but if you parse data outside of *xml4h* then use the library on the resultant document the namespace definitions will get messed up.

XmlImplAdapter - *minidom*

- No support for performing XPath queries.
- Slower than alternative C-based implementations.

API

Main Interface

`xml4h.`**`parse`**(*to_parse*, *ignore_whitespace_text_nodes=True*, *adapter=None*)

Parse an XML document into an *xml4h*-wrapped DOM representation using an underlying XML library implementation.

Parameters

- **`to_parse`** (*`a file-like object or string`*) – an XML document file, document string, or the path to an XML file. If a string value is given that contains a < character it is treated as literal XML data, otherwise a string value is treated as a file path.
- **`ignore_whitespace_text_nodes`** (*`bool`*) – if `True` pure whitespace nodes are stripped from the parsed document, since these are usually noise introduced by XML docs serialized to be human-friendly.
- **`adapter`** (*`adapter class or None`*) – the *xml4h* implementation adapter class used to parse the document and to interact with the resulting nodes. If None, *`best_adapter`* will be used.

Returns an *`xml4h.nodes.Document`* node representing the parsed document.

Delegates to an adapter's `parse_string()` or `parse_file()` implementation.

`xml4h.`**`build`**(*tagname_or_element*, *ns_uri=None*, *adapter=None*)

Return a *`Builder`* that represents an element in a new or existing XML DOM and provides "chainable" methods focussed specifically on adding XML content.

Parameters

- **`tagname_or_element`** (string or *`Element`* node) – a string name for the root node of a new XML document, or an *`Element`* node in an existing document.
- **`ns_uri`** (*`string or None`*) – a namespace URI to apply to the new root node. This argument has no effect this method is acting on an element.
- **`adapter`** (*`adapter class or None`*) – the *xml4h* implementation adapter class used to interact with the document DOM nodes. If None, *`best_adapter`* will be used.

Returns a *`Builder`* instance that represents an *`Element`* node in an XML DOM.

`xml4h.`**`best_adapter`**

alias of `cElementTreeAdapter`

Builder

Builder is a utility class that makes it easy to create valid, well-formed XML documents using relatively sparse python code. The builder class works by wrapping an *xml4h.nodes.Element* node to provide "chainable" methods focussed specifically on adding XML content.

Each method that adds content returns a Builder instance representing the current or the newly-added element. Behind the scenes, the builder uses the *xml4h.nodes* node traversal and manipulation methods to add content directly to the underlying DOM.

You will not generally create Builder instances directly, but will instead call the *xml4h.builder()* method with the name for a new root element or with an existing *xml4h.nodes.Element* node.

class xml4h.builder.**Builder**(*element*)

Builder class that wraps an *xml4h.nodes.Element* node with methods for adding XML content to an underlying DOM.

a(**args*, ***kwargs*)

Add one or more attributes to the *xml4h.nodes.Element* node represented by this Builder.

Returns the current Builder.

Delegates to *xml4h.nodes.Element.set_attributes()*.

attributes(**args*, ***kwargs*)

Add one or more attributes to the *xml4h.nodes.Element* node represented by this Builder.

Returns the current Builder.

Delegates to *xml4h.nodes.Element.set_attributes()*.

attrs(**args*, ***kwargs*)

Add one or more attributes to the *xml4h.nodes.Element* node represented by this Builder.

Returns the current Builder.

Delegates to *xml4h.nodes.Element.set_attributes()*.

c(*text*)

Add a coment node to the *xml4h.nodes.Element* node represented by this Builder.

Returns the current Builder.

Delegates to *xml4h.nodes.Element.add_comment()*.

cdata(*text*)

Add a CDATA node to the *xml4h.nodes.Element* node represented by this Builder.

Returns the current Builder.

Delegates to *xml4h.nodes.Element.add_cdata()*.

clone(*node*)

Clone a node from another document to become a child of the *xml4h.nodes.Element* node represented by this Builder.

Returns a new Builder that represents the current element (not the cloned node).

Delegates to *xml4h.nodes.Node.clone_node()*.

comment(*text*)

Add a coment node to the *xml4h.nodes.Element* node represented by this Builder.

Returns the current Builder.

Delegates to *xml4h.nodes.Element.add_comment()*.

d(*text*)
Add a CDATA node to the *xml4h.nodes.Element* node represented by this Builder.

Returns the current Builder.

Delegates to *xml4h.nodes.Element.add_cdata()*.

data(*text*)
Add a CDATA node to the *xml4h.nodes.Element* node represented by this Builder.

Returns the current Builder.

Delegates to *xml4h.nodes.Element.add_cdata()*.

document

Returns the *xml4h.nodes.Document* node that contains the element represented by this Builder.

dom_element

Returns the *xml4h.nodes.Element* node represented by this Builder.

e(**args*, ***kwargs*)
Add a child element to the *xml4h.nodes.Element* node represented by this Builder.

Returns a new Builder that represents the child element.

Delegates to *xml4h.nodes.Element.add_element()*.

elem(**args*, ***kwargs*)
Add a child element to the *xml4h.nodes.Element* node represented by this Builder.

Returns a new Builder that represents the child element.

Delegates to *xml4h.nodes.Element.add_element()*.

element(**args*, ***kwargs*)
Add a child element to the *xml4h.nodes.Element* node represented by this Builder.

Returns a new Builder that represents the child element.

Delegates to *xml4h.nodes.Element.add_element()*.

find(***kwargs*)
Find descendants of the element represented by this builder that match the given constraints.

Returns a list of *xml4h.nodes.Element* nodes

Delegates to *xml4h.nodes.Node.find()*

find_doc(***kwargs*)
Find nodes in this element's owning *xml4h.nodes.Document* that match the given constraints.

Returns a list of *xml4h.nodes.Element* nodes

Delegates to *xml4h.nodes.Node.find_doc()*.

i(*target*, *data*)
Add a processing instruction node to the *xml4h.nodes.Element* node represented by this Builder.

Returns the current Builder.

Delegates to *xml4h.nodes.Element.add_instruction()*.

instruction(*target*, *data*)
Add a processing instruction node to the *xml4h.nodes.Element* node represented by this Builder.

Returns the current Builder.

Delegates to *xml4h.nodes.Element.add_instruction()*.

ns_prefix(*prefix*, *ns_uri*)
Set the namespace prefix of the *xml4h.nodes.Element* node represented by this Builder.

Returns the current Builder.

Delegates to *xml4h.nodes.Element.set_ns_prefix()*.

processing_instruction(*target*, *data*)
Add a processing instruction node to the *xml4h.nodes.Element* node represented by this Builder.

Returns the current Builder.

Delegates to *xml4h.nodes.Element.add_instruction()*.

root

Returns the *xml4h.nodes.Element* root node ancestor of the element represented by this Builder

t(*text*)
Add a text node to the *xml4h.nodes.Element* node represented by this Builder.

Returns the current Builder.

Delegates to *xml4h.nodes.Element.add_text()*.

text(*text*)
Add a text node to the *xml4h.nodes.Element* node represented by this Builder.

Returns the current Builder.

Delegates to *xml4h.nodes.Element.add_text()*.

transplant(*node*)
Transplant a node from another document to become a child of the *xml4h.nodes.Element* node represented by this Builder.

Returns a new Builder that represents the current element (not the transplanted node).

Delegates to *xml4h.nodes.Node.transplant_node()*.

up(*count=1*, *to_name=None*)

Returns a builder representing an ancestor of the current element, by default the parent element.

Parameters

- **count** (*integer >= 1 or None*) – return the n'th ancestor element; defaults to 1 which means the immediate parent. If *count* is greater than the number of number of ancestors return the document's root element.
- **to_name** (*string or None*) – return the nearest ancestor element with the matching name, or the document's root element if there are no matching elements. This argument trumps the `count` argument.

write(**args*, ***kwargs*)
Write XML text for the element represented by this builder.

Delegates to *xml4h.nodes.Node.write()*.

write_doc(**args*, ***kwargs*)
Write XML text for the Document containing the element represented by this builder.

Delegates to *xml4h.nodes.Node.write_doc()*.

Writer

Writer to serialize XML DOM documents or sections to text.

xml4h.writer.**write_node**(*node*, *writer=None*, *encoding='utf-8'*, *indent=0*, *newline='\n'*, *omit_declaration=False*, *node_depth=0*, *quote_char='"'*)

Serialize an *xml4h* DOM node and its descendants to text, writing the output to a given *writer* or to stdout.

Parameters

- **node** (an *xml4h.nodes.Node* or subclass) – the DOM node whose content and descendants will be serialized.
- **writer** (*a file, stream, etc or None*) – an object such as a file or stream to which XML text is sent. If *None* text is sent to `sys.stdout`.
- **encoding** (*string*) – the character encoding for serialized text.
- **indent** (*string, int, bool, or None*) – indentation prefix to apply to descendent nodes for pretty-printing. The value can take many forms:
  - *int*: the number of spaces to indent. 0 means no indent.
  - *string*: a literal prefix for indented nodes, such as `\t`.
  - *bool*: no indent if *False*, four spaces indent if *True*.
  - *None*: no indent.
- **newline** (*string, bool, or None*) – the string value used to separate lines of output. The value can take a number of forms:
  - *string*: the literal newline value, such as `\n` or `\r`. An empty string means no newline.
  - *bool*: no newline if *False*, `\n` newline if *True*.
  - *None*: no newline.
- **omit_declaration** (*boolean*) – if *True* the XML declaration header is omitted, otherwise it is included. Note that the declaration is only output when serializing an *xml4h.nodes.Document* node.
- **node_depth** (*int*) – the indentation level to start at, such as 2 to indent output as if the given *node* has two ancestors. This parameter will only be useful if you need to output XML text fragments that can be assembled into a document. This parameter has no effect unless indentation is applied.
- **quote_char** (*string*) – the character that delimits quoted content. You should never need to mess with this.

DOM Nodes API

class xml4h.nodes.**Attribute**(*node*, *adapter*)

Node representing an attribute of a *Document* or *Element* node.

class xml4h.nodes.**AttributeDict**(*attr_impl_nodes*, *impl_element*, *adapter*)

Dictionary-like object of element attributes that always reflects the state of the underlying element node, and that allows for in-place modifications that will immediately affect the element.

__weakref__
list of weak references to the object (if defined)

element

Returns the *Element* that contains these attributes.

impl_attributes

Returns the attribute node objects from the underlying XML implementation.

items()

Returns a list of name/value attribute pairs sorted by attribute name.

keys()

Returns a list of attribute name strings.

namespace_uri(*name*)

Parameters **name** (*string*) – the name of an attribute to look up.

Returns the namespace URI associated with the named attribute, or None.

prefix(*name*)

Parameters **name** (*string*) – the name of an attribute to look up.

Returns the prefix component of the named attribute's name, or None.

to_dict

Returns an `OrderedDict` of attribute name/value pairs.

values()

Returns a list of attribute value strings.

class xml4h.nodes.**CDATA**(*node*, *adapter*)
Node representing character data in an XML document.

class xml4h.nodes.**Comment**(*node*, *adapter*)
Node representing a comment in an XML document.

class xml4h.nodes.**Document**(*node*, *adapter*)
Node representing an entire XML document.

class xml4h.nodes.**DocumentFragment**(*node*, *adapter*)
Node representing an XML document fragment.

class xml4h.nodes.**DocumentType**(*node*, *adapter*)
Node representing the type of an XML document.

class xml4h.nodes.**Element**(*node*, *adapter*)
Node representing an element in an XML document, with support for manipulating and adding content to the element.

add_cdata(*data*)
Add a character data node to this element.

Parameters **data** (*string*) – text content to add as character data.

add_comment(*text*)
Add a comment node to this element.

Parameters **text** (*string*) – text content to add as a comment.

add_element(*name*, *ns_uri=None*, *attributes=None*, *text=None*, *before_this_element=False*)
Add a new child element to this element, with an optional namespace definition. If no namespace is provided the child will be assigned to the default namespace.

Parameters

- **name** (*string*) – a name for the child node. The name may be used to apply a namespace to the child by including:
  - a prefix component in the name of the form `ns_prefix:element_name`, where the prefix has already been defined for a namespace URI (such as via *set_ns_prefix()*).
  - a literal namespace URI value delimited by curly braces, of the form `{ns_uri}element_name`.
- **ns_uri** (*string or None*) – a URI specifying the new element's namespace. If the `name` parameter specifies a namespace this parameter is ignored.
- **attributes** (*dict, list, tuple, or None*) – collection of attributes to assign to the new child.
- **text** (*string or None*) – text value to assign to the new child.
- **before_this_element** (*bool*) – if *True* the new element is added as a sibling preceding this element, instead of as a child. In other words, the new element will be a child of this element's parent node, and will immediately precent this element in the DOM.

Returns the new child as a an *Element* node.

add_instruction(*target*, *data*)
Add an instruction node to this element.

Parameters **text** (*string*) – text content to add as an instruction.

add_text(*text*)
Add a text node to this element.

Adding text with this method is subtly different from assigning a new text value with *text()* accessor, because it "appends" to rather than replacing this element's set of text nodes.

Parameters

- **text** – text content to add to this element.
- **type** – string or anything that can be coerced by `unicode()`.

attrib
Get or set this element's attributes as name/value pairs.

Note: Setting element attributes via this accessor will **remove** any existing attributes, as opposed to the *set_attributes()* method which only updates and replaces them.

---

attribute_node(*name*, *ns_uri=None*)

Parameters

- **name** (*string*) – the name of the attribute to return.
- **ns_uri** (*string or None*) – a URI defining a namespace constraint on the attribute.

Returns this element's attributes that match `ns_uri` as *Attribute* nodes.

attribute_nodes

Returns a list of this element's attributes as *Attribute* nodes.

attributes
Get or set this element's attributes as name/value pairs.

Note: Setting element attributes via this accessor will **remove** any existing attributes, as opposed to the *set_attributes()* method which only updates and replaces them.

---

attrs
Get or set this element's attributes as name/value pairs.

Note: Setting element attributes via this accessor will **remove** any existing attributes, as opposed to the *set_attributes()* method which only updates and replaces them.

---

builder

Returns a *Builder* representing this element with convenience methods for adding XML content.

set_attributes(*attr_obj=None*, *ns_uri=None*, ***attr_dict*)
Add or update this element's attributes, where attributes can be specified in a number of ways.

Parameters

- **attr_obj** (*dict, list, tuple, or None*) – a dictionary or list of attribute name/value pairs.
- **ns_uri** (*string or None*) – a URI defining a namespace for the new attributes.
- **attr_dict** (*dict*) – attribute name and values specified as keyword arguments.

set_ns_prefix(*prefix*, *ns_uri*)
Define a namespace prefix that will serve as shorthand for the given namespace URI in element names.

Parameters

- **prefix** (*string*) – prefix that will serve as an alias for a the namespace URI.
- **ns_uri** (*string*) – namespace URI that will be denoted by the prefix.

text
Get or set the text content of this element.

class xml4h.nodes.**Entity**(*node*, *adapter*)
Node representing an entity in an XML document.

class xml4h.nodes.**EntityReference**(*node*, *adapter*)
Node representing an entity reference in an XML document.

class xml4h.nodes.**NameValueNodeMixin**(*node*, *adapter*)
Provide methods to access node name and value attributes, where the node name may also be composed of "prefix" and "local" components.

local_name

Returns the local component of a node name excluding any prefix.

name

Get or set the name of a node, possibly including prefix and local components.

prefix

Returns the namespace prefix component of a node name, or None.

value
Get or set the value of a node.

class xml4h.nodes.**Node**(*node*, *adapter*)
Base class for *xml4h* DOM nodes that represent and interact with a node in the underlying XML implementation.

__init__(*node*, *adapter*)
Construct an object that represents and wraps a DOM node in the underlying XML implementation.

Parameters

- **node** – node object from the underlying XML implementation.
- **adapter** – the xml4h.impls.XmlImplAdapter subclass implementation to mediate operations on the node in the underlying XML implementation.

__weakref__
list of weak references to the object (if defined)

_convert_nodelist(*impl_nodelist*)
Convert a list of underlying implementation nodes into a list of *xml4h* wrapper nodes.

adapter

Returns the xml4h.impls.XmlImplAdapter subclass implementation that mediates operations on the node in the underlying XML implementation.

adapter_class

Returns the class of the xml4h.impls.XmlImplAdapter subclass implementation that mediates operations on the node in the underlying XML implementation.

ancestors

Returns the ancestors of this node in a list ordered by proximity to this node, that is: parent, grandparent, great-grandparent etc.

child(*local_name=None*, *name=None*, *ns_uri=None*, *node_type=None*, *filter_fn=None*)

Returns the first child node matching the given constraints, or *None* if there are no matching child nodes.

Delegates to *NodeList.filter()*.

children

Returns a *NodeList* of this node's child nodes.

clone_node(*node*)
Clone a node from another document to become a child of this node, by copying the node's data into this document but leaving the node untouched in the source document. The node to be cloned can be a *Node* based on the same underlying XML library implementation and adapter, or a "raw" node from that implementation.

Parameters **node** (*xml4h or implementation node*) – the node in another document to clone.

delete(*destroy=True*)
Delete this node from the owning document.

Parameters **destroy** (*bool*) – if True the child node will be destroyed in addition to being removed from the document.

Returns the removed child node, or *None* if the child was destroyed.

document

Returns the *Document* node that contains this node, or `self` if this node is the document.

find(*name=None*, *ns_uri=None*, *first_only=False*)
Find *Element* node descendants of this node, with optional constraints to limit the results.

Parameters

- **name** (*string or None*) – limit results to elements with this name. If *None* or `'*'` all element names are matched.
- **ns_uri** (*string or None*) – limit results to elements within this namespace URI. If *None* all elements are matched, regardless of namespace.
- **first_only** (*bool*) – if *True* only return the first result node or *None* if there is no matching node.

Returns a list of *Element* nodes matching any given constraints, or a single node if `first_only=True`.

find_doc(*name=None*, *ns_uri=None*, *first_only=False*)
Find *Element* node descendants of the document containing this node, with optional constraints to limit the results.

Delegates to *find()* applied to this node's owning document.

find_first(*name=None*, *ns_uri=None*)
Find the first *Element* node descendant of this node that matches any optional constraints, or None if there are no matching elements.

Delegates to *find()* with `first_only=True`.

has_feature(*feature_name*)

Returns *True* if a named feature is supported by the adapter implementation underlying this node.

impl_document

Returns the document object from the underlying XML implementation that contains the node represented by this *xml4h* node.

impl_node

Returns the node object from the underlying XML implementation that is represented by this *xml4h* node.

is_attribute

Returns *True* if this is an *Attribute* node.

is_cdata

Returns *True* if this is a *CDATA* node.

is_comment

Returns *True* if this is a *Comment* node.

is_document

Returns *True* if this is a *Document* node.

is_document_fragment

Returns *True* if this is a *DocumentFragment* node.

is_document_type

Returns *True* if this is a *DocumentType* node.

is_element

Returns *True* if this is an *Element* node.

is_entity

Returns *True* if this is an *Entity* node.

is_entity_reference

Returns *True* if this is an *EntityReference* node.

is_notation

Returns *True* if this is a *Notation* node.

is_processing_instruction

Returns *True* if this is a *ProcessingInstruction* node.

is_root

Returns *True* if this node is the document's root element

is_text

Returns *True* if this is a *Text* node.

is_type(*node_type_constant*)

Returns *True* if this node's int type matches the given value.

namespace_uri

Returns this node's namespace URI or *None*.

node_type

Returns an int constant value that identifies the type of this node, such as `ELEMENT_NODE` or `TEXT_NODE`.

ns_uri

Returns this node's namespace URI or *None*.

parent

Returns the parent of this node, or *None* of the node has no parent.

root

Returns the root *Element* node of the document that contains this node, or `self` if this node is the root element.

siblings

Returns a list of this node's sibling nodes.

Return type *NodeList*

siblings_after

Returns a list of this node's siblings that occur *after* this node in the DOM.

siblings_before

Returns a list of this node's siblings that occur *before* this node in the DOM.

transplant_node(*node*)

Transplant a node from another document to become a child of this node, removing it from the source document. The node to be transplanted can be a *Node* based on the same underlying XML library implementation and adapter, or a "raw" node from that implementation.

Parameters **node** (*xml4h or implementation node*) – the node in another document to transplant.

write(*writer=None*, *encoding='utf-8'*, *indent=0*, *newline='\n'*, *omit_declaration=False*, *node_depth=0*, *quote_char='"'*)

Serialize this node and its descendants to text, writing the output to a given *writer* or to stdout.

Parameters

- **writer** (*a file, stream, etc or None*) – an object such as a file or stream to which XML text is sent. If *None* text is sent to `sys.stdout`.
- **encoding** (*string*) – the character encoding for serialized text.
- **indent** (*string, int, bool, or None*) – indentation prefix to apply to descendent nodes for pretty-printing. The value can take many forms:
  - *int*: the number of spaces to indent. 0 means no indent.
  - *string*: a literal prefix for indented nodes, such as `\t`.
  - *bool*: no indent if *False*, four spaces indent if *True*.
  - *None*: no indent
- **newline** (*string, bool, or None*) – the string value used to separate lines of output. The value can take a number of forms:
  - *string*: the literal newline value, such as `\n` or `\r`. An empty string means no newline.
  - *bool*: no newline if *False*, `\n` newline if *True*.
  - *None*: no newline.
- **omit_declaration** (*boolean*) – if *True* the XML declaration header is omitted, otherwise it is included. Note that the declaration is only output when serializing an *xml4h.nodes.Document* node.
- **node_depth** (*int*) – the indentation level to start at, such as 2 to indent output as if the given *node* has two ancestors. This parameter will only be useful if you need to output XML text fragments that can be assembled into a document. This parameter has no effect unless indentation is applied.
- **quote_char** (*string*) – the character that delimits quoted content. You should never need to mess with this.

Delegates to *xml4h.writer.write_node()* applied to this node.

write_doc(**args*, ***kwargs*)

Serialize to text the document containing this node, writing the output to a given *writer* or stdout.

Delegates to *write()*

xml(*indent=4*, ***kwargs*)

Returns this node as XML text.

Delegates to *write()*

xml_doc(***kwargs*)

Returns the document containing this node as XML text.

Delegates to *xml()*

class xml4h.nodes.**NodeAttrAndChildElementLookupsMixin**
Perform "magical" lookup of a node's attributes via dict-style keyword reference, and child elements via class attribute reference.

__getattr__(*child_name*)
Retrieve this node's child element by tag name regardless of the elements namespace, assuming the name given doesn't match an existing attribute or method.

Parameters **child_name** (*string*) – tag name of the child element to look up. To avoid name clashes with class attributes the child name may includes a trailing underscore (_) character, which is removed to get the real child tag name. The child name must not begin with underscore characters.

Returns

the type of the return value depends on how many child elements match the name:

- a single *Element* node if only one child element matches
- a list of *Element* nodes if there is more than 1 match.

Raise AttributeError if the node has no child element with the given name, or if the given name does not match the required pattern.

__getitem__(*attr_name*)
Retrieve this node's attribute value by name using dict-style keyword lookup.

Parameters **attr_name** (*string*) – name of the attribute. If the attribute has a namespace prefix that must be included, in other words the name must be a qname not local name.

Raise KeyError if the node has no such attribute.

__weakref__
list of weak references to the object (if defined)

class xml4h.nodes.**NodeList**
Custom implementation for *Node* lists that provides additional functionality, such as node filtering.

__call__(*local_name=None*, *name=None*, *ns_uri=None*, *node_type=None*, *filter_fn=None*, *first_only=False*)
Apply filters to the set of nodes in this list.

Parameters

- **local_name** (*string or None*) – a local name used to filter the nodes.
- **name** (*string or None*) – a name used to filter the nodes.
- **ns_uri** (*string or None*) – a namespace URI used to filter the nodes. If *None* all nodes are returned regardless of namespace.
- **node_type** (*int node type constant, class, or None*) – a node type definition used to filter the nodes.
- **filter_fn** (*function or None*) – an arbitrary function to filter nodes in this list. This function must accept a single *Node* argument and return a bool indicating whether to include the node in the filtered results.

Note: if `filter_fn` is provided all other filter arguments are ignore.

Returns

the type of the return value depends on the value of the `first_only` parameter and how many nodes match the filter:

- if `first_only=False` return a *NodeList* of filtered nodes, which will be empty if there are no matching nodes.
- if `first_only=True` and at least one node matches, return the first matching *Node*
- if `first_only=True` and there are no matching nodes, return *None*

__weakref__
list of weak references to the object (if defined)

filter(*local_name=None*, *name=None*, *ns_uri=None*, *node_type=None*, *filter_fn=None*, *first_only=False*)
Apply filters to the set of nodes in this list.

Parameters

- **local_name** (*string or None*) – a local name used to filter the nodes.
- **name** (*string or None*) – a name used to filter the nodes.
- **ns_uri** (*string or None*) – a namespace URI used to filter the nodes. If *None* all nodes are returned regardless of namespace.
- **node_type** (*int node type constant, class, or None*) – a node type definition used to filter the nodes.
- **filter_fn** (*function or None*) – an arbitrary function to filter nodes in this list. This function must accept a single *Node* argument and return a bool indicating whether to include the node in the filtered results.

Note: if `filter_fn` is provided all other filter arguments are ignore.

Returns

the type of the return value depends on the value of the `first_only` parameter and how many nodes match the filter:

- if `first_only=False` return a *NodeList* of filtered nodes, which will be empty if there are no matching nodes.
- if `first_only=True` and at least one node matches, return the first matching *Node*
- if `first_only=True` and there are no matching nodes, return *None*

first

Returns the first of the available children nodes, or *None* if there are no children.

class xml4h.nodes.**Notation**(*node*, *adapter*)
Node representing a notation in an XML document.

class xml4h.nodes.**ProcessingInstruction**(*node*, *adapter*)
Node representing a processing instruction in an XML document.

data
Get or set the value of a node.

target

Get or set the name of a node, possibly including prefix and local components.

class xml4h.nodes.**Text**(*node*, *adapter*)
Node representing text content in an XML document.

class xml4h.nodes.**XPathMixin**
Provide *xpath()* method to nodes that support XPath searching.

__weakref__
list of weak references to the object (if defined)

xpath(*xpath*, ***kwargs*)
Perform an XPath query on the current node.

Parameters

- **xpath** (*string*) – XPath query.
- **kwargs** (*dict*) – Optional keyword arguments that are passed through to the underlying XML library implementation.

Returns results of the query as a list of *Node* objects, or a list of base type objects if the XPath query does not reference node objects.

XML Libarary Adapters

class xml4h.impls.interface.**XmlImplAdapter**(*document*)
Base class that defines how *xml4h* interacts with an underlying XML library that the adaptor "wraps" to provide additional (or at least different) functionality.

This class should be treated as an abstract class. It provides some common implementation code used by all *xml4h* adapter implementations, but mostly it sketches out the methods the real implementaiton subclasses must provide.

clear_caches()
Clear any in-adapter cached data, for cases where cached data could become outdated e.g. by making DOM changes directly outside of *xml4h*.

This is a no-op if the implementing adapter has no cached data.

find_node_elements(*node*, *name='*'*, *ns_uri='*'*)

Returns element node descendents of the given node that match the search constraints.

Parameters

- **node** – a node object from the underlying XML library.
- **name** (*string*) – only elements with a matching name will be returned. If the value is * all names will match.
- **ns_uri** (*string*) – only elements with a matching namespace URI will be returned. If the value is * all namespaces will match.

get_ns_info_from_node_name(*name*, *impl_node*)
Return a three-element tuple with the prefix, local name, and namespace URI for the given element/attribute name (in the context of the given node's hierarchy). If the name has no associated prefix or namespace information, None is return for those tuple members.

classmethod has_feature(*feature_name*)

Returns *True* if a named feature is supported by this adapter.

classmethod **ignore_whitespace_text_nodes**(*wrapped_node*)
Find and delete any text nodes containing nothing but whitespace in in the given node and its descendents.

This is useful for cleaning up excess low-value text nodes in a document DOM after parsing a pretty-printed XML document.

classmethod **is_available**()

Returns *True* if this adapter's underlying XML library is available in the Python environment.

class xml4h.impls.lxml_etree.**LXMLAdapter**(*document*)
Adapter to the lxml XML library implementation.

find_node_elements(*node*, *name='*'*, *ns_uri='*'*)

Returns element node descendents of the given node that match the search constraints.

Parameters

- **node** – a node object from the underlying XML library.
- **name** (*string*) – only elements with a matching name will be returned. If the value is * all names will match.
- **ns_uri** (*string*) – only elements with a matching namespace URI will be returned. If the value is * all namespaces will match.

xpath_on_node(*node*, *xpath*, ***kwargs*)
Return result of performing the given XPath query on the given node.

All known namespace prefix-to-URI mappings in the document are automatically included in the XPath invocation.

If an empty/default namespace (i.e. None) is defined, this is converted to the prefix name '_' so it can be used despite empty namespace prefixes being unsupported by XPath.

class xml4h.impls.xml_etree_elementtree.**ElementTreeAdapter**(*document*)
Adapter to the ElementTree XML library.

This code *must* work with either the base ElementTree pure python implementation or the C-based cElementTree implementation, since it is reused in the *cElementTree* class defined below.

find_node_elements(*node*, *name='*'*, *ns_uri='*'*)

Returns element node descendents of the given node that match the search constraints.

Parameters

- **node** – a node object from the underlying XML library.
- **name** (*string*) – only elements with a matching name will be returned. If the value is * all names will match.
- **ns_uri** (*string*) – only elements with a matching namespace URI will be returned. If the value is * all namespaces will match.

xpath_on_node(*node*, *xpath*, ***kwargs*)
Return result of performing the given XPath query on the given node.

All known namespace prefix-to-URI mappings in the document are automatically included in the XPath invocation.

If an empty/default namespace (i.e. None) is defined, this is converted to the prefix name '_' so it can be used despite empty namespace prefixes being unsupported by XPath.

class xml4h.impls.xml_etree_elementtree.**cElementTreeAdapter**(*document*)
Adapter to the C-based implementation of the ElementTree XML library.

class xml4h.impls.xml_dom_minidom.**XmlDomImplAdapter**(*document*)
Adapter to the minidom XML library implementation.

get_node_text(*node*)
Return contatenated value of all text node children of this element

set_node_text(*node*, *text*)
Set text value as sole Text child node of element; any existing Text nodes are removed

Custom Exceptions

Custom *xml4h* exceptions.

exception xml4h.exceptions.**FeatureUnavailableException**
User has attempted to use a feature that is available in some *xml4h* implementations/adapters, but is not available in the current one.

exception xml4h.exceptions.**IncorrectArgumentTypeException**(*arg*, *expected_types*)
Richer flavour of a ValueError that describes exactly what argument types are expected.

exception xml4h.exceptions.**UnknownNamespaceException**
User has attempted to refer to an unknown or undeclared namespace by prefix or URI.

exception xml4h.exceptions.**Xml4hException**
Base exception class for all non-standard exceptions raised by *xml4h*.

exception xml4h.exceptions.**Xml4hImplementationBug**
xml4h implementation has a bug, probably.

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

Python Module Index

X

xml4h, 42
xml4h.builder, 43
xml4h.exceptions, 58
xml4h.impls.interface, 56
xml4h.impls.lxml_etree, 57
xml4h.impls.xml_dom_minidom, 58
xml4h.impls.xml_etree_elementtree, 57
xml4h.nodes, 46
xml4h.writer, 46

Index

Symbols

__call__() (xml4h.nodes.NodeList method), 54
__getattr__() (xml4h.nodes.NodeAttrAndChildElementLookupsMixin method), 54
__getitem__() (xml4h.nodes.NodeAttrAndChildElementLookupsMixin method), 54
__init__() (xml4h.nodes.Node method), 50
__weakref__ (xml4h.nodes.AttributeDict attribute), 46
__weakref__ (xml4h.nodes.Node attribute), 50
__weakref__ (xml4h.nodes.NodeAttrAndChildElementLookupsMixin attribute), 54
__weakref__ (xml4h.nodes.NodeList attribute), 55
__weakref__ (xml4h.nodes.XPathMixin attribute), 56
_convert_nodelist() (xml4h.nodes.Node method), 50

A

a() (xml4h.builder.Builder method), 43
adapter (xml4h.nodes.Node attribute), 50
adapter_class (xml4h.nodes.Node attribute), 50
add_cdata() (xml4h.nodes.Element method), 47
add_comment() (xml4h.nodes.Element method), 47
add_element() (xml4h.nodes.Element method), 47
add_instruction() (xml4h.nodes.Element method), 48
add_text() (xml4h.nodes.Element method), 48
ancestors (xml4h.nodes.Node attribute), 50
attrib (xml4h.nodes.Element attribute), 48
Attribute (class in xml4h.nodes), 46
attribute_node() (xml4h.nodes.Element method), 48
attribute_nodes (xml4h.nodes.Element attribute), 48
AttributeDict (class in xml4h.nodes), 46
attributes (xml4h.nodes.Element attribute), 49
attributes() (xml4h.builder.Builder method), 43
attrs (xml4h.nodes.Element attribute), 49
attrs() (xml4h.builder.Builder method), 43

B

best_adapter (in module xml4h), 42
build() (in module xml4h), 42
Builder (class in xml4h.builder), 43
builder (xml4h.nodes.Element attribute), 49

C

c() (xml4h.builder.Builder method), 43
CDATA (class in xml4h.nodes), 47
cdata() (xml4h.builder.Builder method), 43
cElementTreeAdapter (class in xml4h.impls.xml_etree_elementtree), 57
child() (xml4h.nodes.Node method), 50
children (xml4h.nodes.Node attribute), 50
clear_caches() (xml4h.impls.interface.XmlImplAdapter method), 56
clone() (xml4h.builder.Builder method), 43
clone_node() (xml4h.nodes.Node method), 50
Comment (class in xml4h.nodes), 47
comment() (xml4h.builder.Builder method), 43

D

d() (xml4h.builder.Builder method), 43
data (xml4h.nodes.ProcessingInstruction attribute), 55
data() (xml4h.builder.Builder method), 44
delete() (xml4h.nodes.Node method), 50
Document (class in xml4h.nodes), 47
document (xml4h.builder.Builder attribute), 44
document (xml4h.nodes.Node attribute), 51
DocumentFragment (class in xml4h.nodes), 47
DocumentType (class in xml4h.nodes), 47
dom_element (xml4h.builder.Builder attribute), 44

E

e() (xml4h.builder.Builder method), 44
elem() (xml4h.builder.Builder method), 44
Element (class in xml4h.nodes), 47
element (xml4h.nodes.AttributeDict attribute), 47
element() (xml4h.builder.Builder method), 44
ElementTreeAdapter (class in xml4h.impls.xml_etree_elementtree), 57
Entity (class in xml4h.nodes), 49
EntityReference (class in xml4h.nodes), 49

F

FeatureUnavailableException, 58
filter() (xml4h.nodes.NodeList method), 55
find() (xml4h.builder.Builder method), 44
find() (xml4h.nodes.Node method), 51
find_doc() (xml4h.builder.Builder method), 44
find_doc() (xml4h.nodes.Node method), 51
find_first() (xml4h.nodes.Node method), 51
find_node_elements() (xml4h.impls.interface.XmlImplAdapter method), 56
find_node_elements() (xml4h.impls.lxml_etree.LXMLAdapter method), 57
find_node_elements() (xml4h.impls.xml_etree_elementtree.ElementTreeAdapter method), 57
first (xml4h.nodes.NodeList attribute), 55

G

get_node_text() (xml4h.impls.xml_dom_minidom.XmlDomImplAdapter method), 58
get_ns_info_from_node_name() (xml4h.impls.interface.XmlImplAdapter method), 56

H

has_feature() (xml4h.impls.interface.XmlImplAdapter class method), 56
has_feature() (xml4h.nodes.Node method), 51

I

i() (xml4h.builder.Builder method), 44
ignore_whitespace_text_nodes() (xml4h.impls.interface.XmlImplAdapter class method), 56
impl_attributes (xml4h.nodes.AttributeDict attribute), 47
impl_document (xml4h.nodes.Node attribute), 51
impl_node (xml4h.nodes.Node attribute), 51
IncorrectArgumentTypeException, 58
instruction() (xml4h.builder.Builder method), 44
is_attribute (xml4h.nodes.Node attribute), 51
is_available() (xml4h.impls.interface.XmlImplAdapter class method), 57
is_cdata (xml4h.nodes.Node attribute), 51
is_comment (xml4h.nodes.Node attribute), 51
is_document (xml4h.nodes.Node attribute), 51
is_document_fragment (xml4h.nodes.Node attribute), 51
is_document_type (xml4h.nodes.Node attribute), 52
is_element (xml4h.nodes.Node attribute), 52
is_entity (xml4h.nodes.Node attribute), 52
is_entity_reference (xml4h.nodes.Node attribute), 52
is_notation (xml4h.nodes.Node attribute), 52
is_processing_instruction (xml4h.nodes.Node attribute), 52
is_root (xml4h.nodes.Node attribute), 52
is_text (xml4h.nodes.Node attribute), 52
is_type() (xml4h.nodes.Node method), 52
items() (xml4h.nodes.AttributeDict method), 47

K

keys() (xml4h.nodes.AttributeDict method), 47

L

local_name (xml4h.nodes.NameValueNodeMixin attribute), 49
LXMLAdapter (class in xml4h.impls.lxml_etree), 57

N

name (xml4h.nodes.NameValueNodeMixin attribute), 49
namespace_uri (xml4h.nodes.Node attribute), 52
namespace_uri() (xml4h.nodes.AttributeDict method), 47
NameValueNodeMixin (class in xml4h.nodes), 49
Node (class in xml4h.nodes), 50
node_type (xml4h.nodes.Node attribute), 52
NodeAttrAndChildElementLookupsMixin (class in xml4h.nodes), 54
NodeList (class in xml4h.nodes), 54
Notation (class in xml4h.nodes), 55
ns_prefix() (xml4h.builder.Builder method), 45
ns_uri (xml4h.nodes.Node attribute), 52

P

parent (xml4h.nodes.Node attribute), 52
parse() (in module xml4h), 42
prefix (xml4h.nodes.NameValueNodeMixin attribute), 49
prefix() (xml4h.nodes.AttributeDict method), 47
processing_instruction() (xml4h.builder.Builder method), 45
ProcessingInstruction (class in xml4h.nodes), 55

R

root (xml4h.builder.Builder attribute), 45
root (xml4h.nodes.Node attribute), 52

S

set_attributes() (xml4h.nodes.Element method), 49
set_node_text() (xml4h.impls.xml_dom_minidom.XmlDomImplAdapter method), 58
set_ns_prefix() (xml4h.nodes.Element method), 49
siblings (xml4h.nodes.Node attribute), 52
siblings_after (xml4h.nodes.Node attribute), 52
siblings_before (xml4h.nodes.Node attribute), 52

T

t() (xml4h.builder.Builder method), 45
target (xml4h.nodes.ProcessingInstruction attribute), 55
Text (class in xml4h.nodes), 56
text (xml4h.nodes.Element attribute), 49

text() (xml4h.builder.Builder method), 45
to_dict (xml4h.nodes.AttributeDict attribute), 47
transplant() (xml4h.builder.Builder method), 45
transplant_node() (xml4h.nodes.Node method), 53

U

UnknownNamespaceException, 58
up() (xml4h.builder.Builder method), 45

V

value (xml4h.nodes.NameValueNodeMixin attribute), 50
values() (xml4h.nodes.AttributeDict method), 47

W

write() (xml4h.builder.Builder method), 45
write() (xml4h.nodes.Node method), 53
write_doc() (xml4h.builder.Builder method), 45
write_doc() (xml4h.nodes.Node method), 53
write_node() (in module xml4h.writer), 46

X

xml() (xml4h.nodes.Node method), 53
xml4h (module), 42
xml4h.builder (module), 43
xml4h.exceptions (module), 58
xml4h.impls.interface (module), 56
xml4h.impls.lxml_etree (module), 57
xml4h.impls.xml_dom_minidom (module), 58
xml4h.impls.xml_etree_elementtree (module), 57
xml4h.nodes (module), 46
xml4h.writer (module), 46
Xml4hException, 58
Xml4hImplementationBug, 58
xml_doc() (xml4h.nodes.Node method), 53
XmlDomImplAdapter (class in xml4h.impls.xml_dom_minidom), 58
XmlImplAdapter (class in xml4h.impls.interface), 56
xpath() (xml4h.nodes.XPathMixin method), 56
xpath_on_node() (xml4h.impls.lxml_etree.LXMLAdapter method), 57
xpath_on_node() (xml4h.impls.xml_etree_elementtree.ElementTreeAdapter method), 57
XPathMixin (class in xml4h.nodes), 56