lxml-domeqsque Documentation

Release 0.2

Frank Sachsenheim

Jul 26, 2020

CONTENTS

`delb` is a library that provides an ergonomic model for XML encoded text documents (e.g. TEI-XML) for the Python programming language. It fills a gap for the humanities-related field of software development towards the excellent (scientific) communities in the Python ecosystem.

For a more elaborated discussion see the *Design* chapter of the documentation.

CHAPTER

ONE

FEATURES

- Loads documents from various source types. This is customizable and extensible.
- XML DOM types are represented by distinct classes.
- A completely type-annotated API.
- Consistent design regarding names and callables' signatures.
- Shadows comments and processing instructions by default.
- Querying with XPath and CSS expressions.
- Applying XSL Transformations.

CHAPTER

TWO

DEVELOPMENT STATUS

You're invited to submit tests that reflect desired use cases or are merely of theoretical nature. Of course, any kind of proposals for or implementations of improvements are welcome as well.

CHAPTER

THREE

RELATED PROJECTS & TESTIMONIALS

snakesist is an eXist-db client that uses `delb` to expose database resources.

Kurt Raschke noted in 2010:

```
In a DOM-based implementation, it would be relatively easy [...]
But lxml doesn't use text nodes; instead it uses and properties to hold text
content.
```

CHAPTER

FOUR

ROADMAPISH

- gain insights from usage experience
- implement the API in Rust
- provide bindings for Python and Javascript to the Rust implementation, while nurturing the lxml-based implementation as reference for some time
- be finished before the Digital Humanities community realizes how to foster a viable software ecosystem and fund such efforts

4.1 Installation

4.1.1 From the Python Package Index

Before you install the library manually you might consider to use a project management tool like pipenv or poetry, or still use pip:

```
pip install delb
```

At the moment there's only one optional dependency to enable document loading via *https*, to include it use:

```
# in a poetry managed project
poetry add --extras=https-loader delb
# otherwise
pip install delb[https-loader]
```

4.1.2 From source

Prerequisites:

- poetry is available (`pip install --user poetry` should often work)
- a virtual environment of your project is activated
- that virtual environment houses an interpreter for Python 3.6 or later

Obtain the code with roughly one of:

- `git clone --recurse-submodules git@github.com:funkyfuture/delb.git`
- `curl -LosS https://github.com/funkyfuture/delb/archive/master.tar.gz | tar xzf -`

To install it regularly:

```
.../delb $ pip install .
```

For developing purposes of `delb` itself, poetry should be used which install the library in editable mode and all employed development tools:

```
.../delb $ poetry install
```

Hint: Using git submodules is a great way to vendorize a lib for your project and to have a fork for your adjustments. Please offer the latter to upstream if done well.

4.2 API Documentation

Note: There are actually two packages that are installed with *delb*: `delb` and `_delb`. As the underscore indicates, the latter is exposing private parts of the API while the first is re-exposing what is deemed to be public from that one and additional contents. As a thumb of rule, use the public API in applications and the private API in *delb* extensions. By doing so, you can avoid circular dependencies if your extension (or other code that it depends on) uses contents from the `delb` API.

4.2.1 Documents

class delb.**Document**(*source: Any*, *parser: lxml.etree.XMLParser = <lxml.etree.XMLParser object>*, ***config*)

This class is the entrypoint to obtain a representation of an XML encoded text document. For instantiation any object can be passed. A suitable loader must be available for the given source. See *Document loaders* for the default loaders that come with this package. Plugins are capable to alter the available loaders, see *Extending delb*.

Nodes can be tested for membership in a document:

```
>>> document = Document("<root>text</root>")
>>> text_node = document.root[0]
>>> text_node in document
True
>>> text_node.clone() in document
False
```

The string coercion of a document yields an XML encoded stream, but unlike *Document.save()* and *Document.write()*, without an XML declaration:

```
>>> document = Document("<root/>")
>>> str(document)
'<root/>'
```

Parameters

- **source** – Anything that the configured loaders can make sense of to return a parsed document tree.

- **parser** – An optional `lxml.etree.XMLParser` instance that is used to parse a document stream.
- **config** – Additional keyword arguments for the configuration of extension classes.

Properties

namespaces	The namespace mapping of the document's *root* node.
root	The root node of a document tree.
head_nodes	A list-like accessor to the nodes that precede the document's root node.
source_url	The source URL where a loader obtained the document's contents or `None`.
tail_nodes	A list-like accessor to the nodes that follow the document's root node.

Uncategorized methods

`__init__`(source[, parser])	Initialize self.
`__contains__`(node)	
`__str__`()	Return str(self).
cleanup_namespaces([namespaces, retain_prefixes])	Consolidates the namespace declarations in the document by removing unused and redundant ones.
clone()	**return** Another instance with the duplicated contents.
css_select(expression)	This method proxies to the *TagNode.css_select()* method of the document's *root* node.
merge_text_nodes()	This method proxies to the *TagNode.merge_text_nodes()* method of the document's *root* node.
new_tag_node(local_name[, attributes, namespace])	This method proxies to the *TagNode.new_tag_node()* method of the document's root node.
save(path[, pretty])	**param path** The path where the document shall be saved.
write(buffer[, pretty])	**param buffer** A file-like object that the document is written to.
xpath(expression)	This method proxies to the *TagNode.xpath()* method of the document's *root* node.

continues on next page

Table 2 – continued from previous page

xslt(transformation)	**param transformation** A `lxml.etree.XSLT` instance that shall be

cleanup_namespaces(*namespaces: Optional[etree._NSMap] = None*, *retain_prefixes: Optional[Iterable[str]] = None*)

Consolidates the namespace declarations in the document by removing unused and redundant ones.

There are currently some caveats due to lxml/libxml2's implementations:

- prefixes cannot be set for the default namespace
- a namespace cannot be declared as default after a node's creation (where a namespace was specified that had been registered for a prefix with *register_namespace()*)
- there's no way to unregister a prefix for a namespace
- if there are other namespaces used as default namespaces (where a namespace was specified that had *not* been registered for a prefix) in the descendants of the root, their declarations are lost when this method is used

To ensure clean serializations, one should:

- register prefixes for all namespaces except the default one at the start of an application
- use only one default namespace within a document

Parameters

- **namespaces** – An optional mapping of prefixes (keys) to namespaces (values) that will be declared at the root element.
- **retain_prefixes** – An optional iterable that contains prefixes whose declarations shall be kept despite not being used.

clone() → *delb.Document*

Returns Another instance with the duplicated contents.

config: SimpleNamespace

Beside the used `parser`, this property contains the namespaced data that extension classes and loaders may have stored.

css_select(*expression: str*) → _delb.nodes.QueryResults

This method proxies to the *TagNode.css_select()* method of the document's *root* node.

head_nodes

A list-like accessor to the nodes that precede the document's root node. Note that nodes can't be removed or replaced.

merge_text_nodes()

This method proxies to the *TagNode.merge_text_nodes()* method of the document's *root* node.

property namespaces

The namespace mapping of the document's *root* node.

new_tag_node(*local_name: str*, *attributes: Optional[Dict[str, str]] = None*, *namespace: Optional[str] = None*) → _delb.nodes.TagNode
This method proxies to the *TagNode.new_tag_node()* method of the document's root node.

property root
The root node of a document tree.

save(*path: pathlib.Path*, *pretty: bool = False*, ***cleanup_namespaces_args*)

Parameters

- **path** – The path where the document shall be saved.
- **pretty** – Adds indentation for human consumers when `True`.
- **cleanup_namespaces_args** – Arguments that are a passed to *Document.cleanup_namespaces()* before saving.

source_url: Optional[str]
The source URL where a loader obtained the document's contents or `None`.

tail_nodes
A list-like accessor to the nodes that follow the document's root node. Note that nodes can't be removed or replaced.

write(*buffer: IO*, *pretty: bool = False*, ***cleanup_namespaces_args*)

Parameters

- **buffer** – A file-like object that the document is written to.
- **pretty** – Adds indentation for human consumers when `True`.
- **cleanup_namespaces_args** – Arguments that are a passed to *Document.cleanup_namespaces()* before writing.

xpath(*expression: str*) → _delb.nodes.QueryResults
This method proxies to the *TagNode.xpath()* method of the document's *root* node.

xslt(*transformation: lxml.etree.XSLT*) → *delb.Document*

Parameters **transformation** – A `lxml.etree.XSLT` instance that shall be applied to the document.

Returns A new instance with the transformation's result.

4.2.2 Document loaders

If you want or need to manipulate the availability of or order in which loaders are attempted, you can change the `delb.plugins.plugin_manager.plugins.loaders` object which is a `list`. Its state is reflected in your whole application. Please refer to this issue when you require finer controls over these aspects.

Core

The `core_loaders` module provides a set loaders to retrieve documents from various data sources.

`_delb.plugins.core_loaders.`**`buffer_loader`**(*data: Any*, *config: types.SimpleNamespace*) → Union[Tuple[Optional[lxml.etree._ElementTree], Dict[int, _ElementWrappingNode]], str]

This loader loads a document from a file-like object.

`_delb.plugins.core_loaders.`**`etree_loader`**(*data: Any*, *config: types.SimpleNamespace*) → Union[Tuple[Optional[lxml.etree._ElementTree], Dict[int, _ElementWrappingNode]], str]

This loader processes `lxml.etree._Element` and `lxml.etree._ElementTree` instances.

`_delb.plugins.core_loaders.`**`ftp_http_loader`**(*data: Any*, *config: types.SimpleNamespace*) → Union[Tuple[Optional[lxml.etree._ElementTree], Dict[int, _ElementWrappingNode]], str]

Loads a document from a URL with either the `ftp` or `http` schema. The URL will be bound to `source_url` on the document's `Document.config` attribute.

`_delb.plugins.core_loaders.`**`path_loader`**(*data: Any*, *config: types.SimpleNamespace*) → Union[Tuple[Optional[lxml.etree._ElementTree], Dict[int, _ElementWrappingNode]], str]

This loader loads from a file that is pointed at with a `pathlib.Path` instance. That instance will be bound to `source_path` on the document's `Document.config` attribute.

`_delb.plugins.core_loaders.`**`tag_node_loader`**(*data: Any*, *config: types.SimpleNamespace*) → Union[Tuple[Optional[lxml.etree._ElementTree], Dict[int, _ElementWrappingNode]], str]

This loader loads, or rather clones, a *`delb.TagNode`* instance and its descendant nodes.

`_delb.plugins.core_loaders.`**`text_loader`**(*data: Any*, *config: types.SimpleNamespace*) → Union[Tuple[Optional[lxml.etree._ElementTree], Dict[int, _ElementWrappingNode]], str]

Parses a string containing a full document.

Extra

If `delb` is installed with `https-loader` as extra, the required dependencies for this loader are installed as well. See *Installation*.

`_delb.plugins.https_loader.`**`https_loader`**(*data: Any*, *config: types.SimpleNamespace*) → Union[Tuple[Optional[lxml.etree._ElementTree], Dict[int, _ElementWrappingNode]], str]

This loader loads a document from a URL with the `https` scheme. The URL will be bound to `source_url` on the document's `Document.config` attribute.

4.2.3 Nodes

Comment

class delb.**CommentNode**(*etree_element: lxml.etree._Element*, *cache: Dict[int, _ElementWrappingNode]*)

The instances of this class represent comment nodes of a tree.

To instantiate new nodes use *new_comment_node()*.

Properties

depth	The depth (or level) of the node in its tree.
document	The *Document* instances that the node is associated with or None.
first_child	
full_text	The concatenated contents of all text node descendants in document order.
index	The node's index within the parent's collection of child nodes or None when the node has no parent.
last_child	
parent	The node's parent or None.
content	The comment's text.

Fetching a single relative node

next_node(*filter)

param filter Any number of *filter* s.

next_node_in_stream(*filter)

param filter Any number of *filter* s.

previous_node(*filter)

param filter Any number of *filter* s.

previous_node_in_stream(*filter)

param filter Any number of *filter* s.

Iterating over relative nodes

ancestors(*filter)

param filter Any number of *filter* s that a node must match to be

child_nodes(*filter[, recurse])	A generator iterator that yields nothing.

continues on next page

Table 5 – continued from previous page

iterate_next_nodes(*filter)	**param filter** Any number of *filter* s that a node must match to be
iterate_next_nodes_in_stream(*filter)	**param filter** Any number of *filter* s that a node must match to be
iterate_previous_nodes(*filter)	**param filter** Any number of *filter* s that a node must match to be
iterate_previous_nodes_in_stream(*filter)	**param filter** Any number of *filter* s that a node must match to be

Adding nodes

add_next(*node[, clone])	Adds one or more nodes to the right of the node this method is called on.
add_previous(*node[, clone])	Adds one or more nodes to the left of the node this method is called on.

Removing a node from its tree

detach()	Removes the node, including its descendants, from its tree.
replace_with(node[, clone])	Removes the node and places the given one in its tree location.

Uncategorized methods

__init__(etree_element, cache)	Initialize self.
clone([deep])	**param deep** Clones the whole subtree if `True`.
new_tag_node(local_name[, attributes, ...])	Creates a new *TagNode* instance in the node's context.
__copy__()	
__deepcopy__([memodict])	
__eq__(other)	Return self==value.
__str__()	Return str(self).
__repr__()	Return repr(self).

add_next (**node: Union[str, NodeBase, _TagDefinition], clone: bool = False*)
Adds one or more nodes to the right of the node this method is called on.

The nodes can be concrete instances of any node type or rather abstract descriptions in the form of strings or objects returned from the `tag()` function that are used to derive `TextNode` respectively `TagNode` instances from.

Parameters

- **node** – The node(s) to be added.
- **clone** – Clones the concrete nodes before adding if `True`.

add_previous (**node: Union[str, NodeBase, _TagDefinition], clone: bool = False*)
Adds one or more nodes to the left of the node this method is called on.

The nodes can be concrete instances of any node type or rather abstract descriptions in the form of strings or objects returned from the `tag()` function that are used to derive `TextNode` respectively `TagNode` instances from.

Parameters

- **node** – The node(s) to be added.
- **clone** – Clones the concrete nodes before adding if `True`.

ancestors (**filter: Callable[[NodeBase], bool]*) → Iterator[_delb.nodes.TagNode]

Parameters **filter** – Any number of *filter* s that a node must match to be yielded.

Returns A generator iterator that yields the ancestor nodes from bottom to top.

child_nodes (**filter: Callable[[NodeBase], bool], recurse: bool = False*) → Iterator[_delb.nodes.NodeBase]
A generator iterator that yields nothing.

clone (*deep: bool = False*) → _delb.nodes._ElementWrappingNode

Parameters **deep** – Clones the whole subtree if `True`.

Returns A copy of the node.

property content
The comment's text.

property depth
The depth (or level) of the node in its tree.

detach () → _ElementWrappingNode
Removes the node, including its descendants, from its tree.

Returns The removed node.

property document
The `Document` instances that the node is associated with or `None`.

property full_text
The concatenated contents of all text node descendants in document order.

property index
The node's index within the parent's collection of child nodes or `None` when the node has no parent.

iterate_next_nodes (**filter: Callable[[NodeBase], bool]*) → Iterator[_delb.nodes.NodeBase]

Parameters **filter** – Any number of *filter* s that a node must match to be yielded.

Returns A generator iterator that yields the siblings to the node's right.

iterate_next_nodes_in_stream(**filter: Callable[[NodeBase], bool]*) → Iterator[_delb.nodes.NodeBase]

Parameters **filter** – Any number of *filter* s that a node must match to be yielded.

Returns A generator iterator that yields the following nodes in document order.

iterate_previous_nodes(**filter: Callable[[NodeBase], bool]*) → Iterator[_delb.nodes.NodeBase]

Parameters **filter** – Any number of *filter* s that a node must match to be yielded.

Returns A generator iterator that yields the siblings to the node's left.

iterate_previous_nodes_in_stream(**filter: Callable[[NodeBase], bool]*) → Iterator[_delb.nodes.NodeBase]

Parameters **filter** – Any number of *filter* s that a node must match to be yielded.

Returns A generator iterator that yields the previous nodes in document order.

new_tag_node(*local_name: str*, *attributes: Optional[Dict[str, str]] = None*, *namespace: Optional[str] = None*, *children: Sequence[Union[str, _delb.nodes.NodeBase, _TagDefinition]] = ()*) → _delb.nodes.TagNode

Creates a new `TagNode` instance in the node's context.

Parameters

- **local_name** – The tag name.
- **attributes** – Optional attributes that are assigned to the new node.
- **namespace** – An optional tag namespace. If none is provided, the context node's namespace is inherited.
- **children** – An optional sequence of objects that will be appended as child nodes. This can be existing nodes, strings that will be inserted as text nodes and in-place definitions of `TagNode` instances from `tag()`. The latter will be assigned to the same namespace.

Returns The newly created tag node.

next_node(**filter: Callable[[NodeBase], bool]*) → Optional[_delb.nodes.NodeBase]

Parameters **filter** – Any number of *filter* s.

Returns The next sibling to the right that matches all filters or `None`.

next_node_in_stream(**filter: Callable[[NodeBase], bool]*) → Optional[_delb.nodes.NodeBase]

Parameters **filter** – Any number of *filter* s.

Returns The next node in document order that matches all filters or `None`.

property parent

The node's parent or `None`.

previous_node(**filter: Callable[[NodeBase], bool]*) → Optional[_delb.nodes.NodeBase]

Parameters **filter** – Any number of *filter* s.

Returns The next sibling to the left that matches all filters or `None`.

previous_node_in_stream(**filter: Callable[[NodeBase], bool]*) → Optional[_delb.nodes.NodeBase]

Parameters **filter** – Any number of *filter* s.

Returns The previous node in document order that matches all filters or `None`.

replace_with(*node: Union[str, NodeBase, _TagDefinition]*, *clone: bool = False*) → _delb.nodes.NodeBase

Removes the node and places the given one in its tree location.

The node can be a concrete instance of any node type or a rather abstract description in the form of a string or an object returned from the *tag()* function that is used to derive a *TextNode* respectively *TagNode* instance from.

Parameters

- **node** – The replacing node.
- **clone** – A concrete, replacing node is cloned if `True`.

Returns The removed node.

Processing instruction

class delb.**ProcessingInstructionNode**(*etree_element: lxml.etree._Element*, *cache: Dict[int, _ElementWrappingNode]*)

The instances of this class represent processing instruction nodes of a tree.

To instantiate new nodes use *new_processing_instruction_node()*.

Properties

depth	The depth (or level) of the node in its tree.
document	The *Document* instances that the node is associated with or `None`.
first_child	
full_text	The concatenated contents of all text node descendants in document order.
index	The node's index within the parent's collection of child nodes or `None` when the node has no parent.
last_child	
parent	The node's parent or `None`.
content	The processing instruction's text.
target	The processing instruction's target.

Fetching a single relative node

next_node(*filter)	**param filter** Any number of *filter* s.
next_node_in_stream(*filter)	**param filter** Any number of *filter* s.
previous_node(*filter)	**param filter** Any number of *filter* s.

continues on next page

Table 10 – continued from previous page

previous_node_in_stream(*filter)	**param filter** Any number of *filter* s.

Iterating over relative nodes

ancestors(*filter)	**param filter** Any number of *filter* s that a node must match to be
child_nodes(*filter[, recurse])	A generator iterator that yields nothing.
iterate_next_nodes(*filter)	**param filter** Any number of *filter* s that a node must match to be
iterate_next_nodes_in_stream(*filter)	**param filter** Any number of *filter* s that a node must match to be
iterate_previous_nodes(*filter)	**param filter** Any number of *filter* s that a node must match to be
iterate_previous_nodes_in_stream(*filter)	**param filter** Any number of *filter* s that a node must match to be

Adding nodes

add_next(*node[, clone])	Adds one or more nodes to the right of the node this method is called on.
add_previous(*node[, clone])	Adds one or more nodes to the left of the node this method is called on.

Removing a node from its tree

detach()	Removes the node, including its descendants, from its tree.
replace_with(node[, clone])	Removes the node and places the given one in its tree location.

Uncategorized methods

`__init__`(etree_element, cache)	Initialize self.
clone([deep])	**param deep** Clones the whole subtree if `True`.
new_tag_node(local_name[, attributes, ...])	Creates a new *TagNode* instance in the node's context.
`__copy__`()	
`__deepcopy__`([memodict])	
`__eq__`(other)	Return self==value.
`__str__`()	Return str(self).
`__repr__`()	Return repr(self).

add_next (**node: Union[str, NodeBase, _TagDefinition], clone: bool = False*)
Adds one or more nodes to the right of the node this method is called on.

The nodes can be concrete instances of any node type or rather abstract descriptions in the form of strings or objects returned from the *tag()* function that are used to derive *TextNode* respectively *TagNode* instances from.

Parameters

- **node** – The node(s) to be added.
- **clone** – Clones the concrete nodes before adding if `True`.

add_previous (**node: Union[str, NodeBase, _TagDefinition], clone: bool = False*)
Adds one or more nodes to the left of the node this method is called on.

The nodes can be concrete instances of any node type or rather abstract descriptions in the form of strings or objects returned from the *tag()* function that are used to derive *TextNode* respectively *TagNode* instances from.

Parameters

- **node** – The node(s) to be added.
- **clone** – Clones the concrete nodes before adding if `True`.

ancestors (**filter: Callable[[NodeBase], bool]*) → Iterator[_delb.nodes.TagNode]

Parameters **filter** – Any number of *filter* s that a node must match to be yielded.

Returns A generator iterator that yields the ancestor nodes from bottom to top.

child_nodes (**filter: Callable[[NodeBase], bool], recurse: bool = False*) → Iterator[_delb.nodes.NodeBase]
A generator iterator that yields nothing.

clone (*deep: bool = False*) → _delb.nodes._ElementWrappingNode

Parameters **deep** – Clones the whole subtree if `True`.

Returns A copy of the node.

property content
The processing instruction's text.

property depth
The depth (or level) of the node in its tree.

detach() → _ElementWrappingNode
Removes the node, including its descendants, from its tree.

Returns The removed node.

property document
The *Document* instances that the node is associated with or None.

property full_text
The concatenated contents of all text node descendants in document order.

property index
The node's index within the parent's collection of child nodes or None when the node has no parent.

iterate_next_nodes(**filter: Callable[[NodeBase], bool]*) → Iterator[_delb.nodes.NodeBase]

Parameters **filter** – Any number of *filter* s that a node must match to be yielded.

Returns A generator iterator that yields the siblings to the node's right.

iterate_next_nodes_in_stream(**filter: Callable[[NodeBase], bool]*) → Iterator[_delb.nodes.NodeBase]

Parameters **filter** – Any number of *filter* s that a node must match to be yielded.

Returns A generator iterator that yields the following nodes in document order.

iterate_previous_nodes(**filter: Callable[[NodeBase], bool]*) → Iterator[_delb.nodes.NodeBase]

Parameters **filter** – Any number of *filter* s that a node must match to be yielded.

Returns A generator iterator that yields the siblings to the node's left.

iterate_previous_nodes_in_stream(**filter: Callable[[NodeBase], bool]*) → Iterator[_delb.nodes.NodeBase]

Parameters **filter** – Any number of *filter* s that a node must match to be yielded.

Returns A generator iterator that yields the previous nodes in document order.

new_tag_node(*local_name: str*, *attributes: Optional[Dict[str, str]] = None*, *namespace: Optional[str] = None*, *children: Sequence[Union[str, _delb.nodes.NodeBase, _TagDefinition]] = ()*) → _delb.nodes.TagNode
Creates a new *TagNode* instance in the node's context.

Parameters

- **local_name** – The tag name.
- **attributes** – Optional attributes that are assigned to the new node.
- **namespace** – An optional tag namespace. If none is provided, the context node's namespace is inherited.
- **children** – An optional sequence of objects that will be appended as child nodes. This can be existing nodes, strings that will be inserted as text nodes and in-place definitions of *TagNode* instances from *tag()*. The latter will be assigned to the same namespace.

Returns The newly created tag node.

next_node(**filter: Callable[[NodeBase], bool]*) → Optional[_delb.nodes.NodeBase]

Parameters **filter** – Any number of *filter* s.

Returns The next sibling to the right that matches all filters or `None`.

next_node_in_stream(**filter: Callable[[NodeBase], bool]*) → Optional[_delb.nodes.NodeBase]

Parameters **filter** – Any number of *filter* s.

Returns The next node in document order that matches all filters or `None`.

property parent
The node's parent or `None`.

previous_node(**filter: Callable[[NodeBase], bool]*) → Optional[_delb.nodes.NodeBase]

Parameters **filter** – Any number of *filter* s.

Returns The next sibling to the left that matches all filters or `None`.

previous_node_in_stream(**filter: Callable[[NodeBase], bool]*) → Optional[_delb.nodes.NodeBase]

Parameters **filter** – Any number of *filter* s.

Returns The previous node in document order that matches all filters or `None`.

replace_with(*node: Union[str, NodeBase, _TagDefinition]*, *clone: bool = False*) → _delb.nodes.NodeBase
Removes the node and places the given one in its tree location.

The node can be a concrete instance of any node type or a rather abstract description in the form of a string or an object returned from the *tag()* function that is used to derive a *TextNode* respectively *TagNode* instance from.

Parameters

- **node** – The replacing node.
- **clone** – A concrete, replacing node is cloned if `True`.

Returns The removed node.

property target
The processing instruction's target.

Tag

class delb.**TagNode**(*etree_element: lxml.etree._Element*, *cache: Dict[int, _ElementWrappingNode]*)
The instances of this class represent *tag node* s of a tree, the equivalent of DOM's elements.

To instantiate new nodes use *Document.new_tag_node*, *TagNode.new_tag_node*, *TextNode.new_tag_node* or *new_tag_node()*.

Some syntactic sugar is baked in:

Attributes and nodes can be tested for membership in a node.

```
>>> root = Document('<root ham="spam"><child/></root>').root
>>> child = root.first_child
>>> "ham" in root
True
>>> child in root
True
```

Nodes can be copied. Note that this relies on *TagNode.clone()*.

```
>>> from copy import copy, deepcopy
>>> root = Document("<root>Content</root>").root
>>> print(copy(root))
<root/>
>>> print(deepcopy(root))
<root>Content</root>
```

Nodes can be tested for equality regarding their qualified name and attributes.

```
>>> root = Document('<root><foo x="0"/><foo x="0"/><bar x="0"/></root>').root
>>> root[0] == root[1]
True
>>> root[0] == root[2]
False
```

Attribute values and child nodes can be obtained with the subscript notation.

```
>>> root = Document('<root x="0"><child_1/>child_2<child_3/></root>').root
>>> root["x"]
'0'
>>> print(root[0])
<child_1/>
>>> print(root[-1])
<child_3/>
>>> print([str(x) for x in root[1::-1]])
['child_2', '<child_1/>']
```

How much child nodes has this node anyway?

```
>>> root = Document("<root><child_1/><child_2/></root>").root
>>> len(root)
2
>>> len(root[0])
0
```

As seen in the examples above, a tag nodes string representation yields a serialized XML representation of a sub-/tree.

Properties

depth	The depth (or level) of the node in its tree.
document	The *Document* instances that the node is associated with or None.
first_child	The node's first child node.
full_text	The concatenated contents of all text node descendants in document order.
index	The node's index within the parent's collection of child nodes or None when the node has no parent.
last_child	The node's last child node.
parent	The node's parent or None.
attributes	A mapping that can be used to query and alter the node's attributes.
local_name	The node's name.

continues on next page

Table 15 – continued from previous page

location_path	An unambiguous XPath location path that points to this node from its tree root.
namespace	The node's namespace.
namespaces	The prefix to namespace mapping of the node.
prefix	The prefix that the node's namespace is currently mapped to.
qualified_name	The node's qualified name.

Fetching a single relative node

next_node(*filter)

param filter Any number of *filter* s.

next_node_in_stream(*filter)

param filter Any number of *filter* s.

previous_node(*filter)

param filter Any number of *filter* s.

previous_node_in_stream(*filter)

param filter Any number of *filter* s.

Iterating over relative nodes

ancestors(*filter)

param filter Any number of *filter* s that a node must match to be

child_nodes(*filter[, recurse])

param filter Any number of *filter* s that a node must match to be

iterate_next_nodes(*filter)

param filter Any number of *filter* s that a node must match to be

iterate_next_nodes_in_stream(*filter)

param filter Any number of *filter* s that a node must match to be

iterate_previous_nodes(*filter)

param filter Any number of *filter* s that a node must match to be

continues on next page

Table 17 – continued from previous page

iterate_previous_nodes_in_stream(*filter)	**param filter** Any number of *filter* s that a node must match to be

Querying descending nodes

css_select(expression)	Namespace prefixes are delimited with a `|` before a name test, for example `div svg|metadata` selects all descendants of `div` named nodes that belong to the default namespace or have no namespace and whose name is `metadata` and have a namespace that is mapped to the `svg` prefix.
xpath(expression)	Returns all *tag node* s that match the evaluation of an XPath expression.

Adding nodes

add_next(*node[, clone])	Adds one or more nodes to the right of the node this method is called on.
add_previous(*node[, clone])	Adds one or more nodes to the left of the node this method is called on.
append_child(*node[, clone])	Adds one or more nodes as child nodes after any existing to the child nodes of the node this method is called on.
insert_child(index, *node[, clone])	Inserts one or more child nodes.
prepend_child(*node[, clone])	Adds one or more nodes as child nodes before any existing to the child nodes of the node this method is called on.

Removing a node from its tree

detach()	Removes the node, including its descendants, from its tree.
replace_with(node[, clone])	Removes the node and places the given one in its tree location.

Uncategorized methods

`__init__`(etree_element, cache)	Initialize self.
clone([deep])	**param deep** Clones the whole subtree if `True`.
new_tag_node(local_name[, attributes, ...])	Creates a new *TagNode* instance in the node's context.
`__copy__`()	
`__deepcopy__`([memodict])	
`__contains__`(item)	
`__eq__`(other)	Return self==value.
`__getitem__`()	
`__hash__`()	Return hash(self).
`__len__`()	
`__str__`()	Return str(self).
`__repr__`()	Return repr(self).
merge_text_nodes()	Merges all consecutive text nodes in the subtree into one.

add_next (**node: Union[str, NodeBase, _TagDefinition], clone: bool = False*)
Adds one or more nodes to the right of the node this method is called on.

The nodes can be concrete instances of any node type or rather abstract descriptions in the form of strings or objects returned from the *tag()* function that are used to derive *TextNode* respectively *TagNode* instances from.

Parameters

- **node** – The node(s) to be added.
- **clone** – Clones the concrete nodes before adding if `True`.

add_previous (**node: Union[str, NodeBase, _TagDefinition], clone: bool = False*)
Adds one or more nodes to the left of the node this method is called on.

The nodes can be concrete instances of any node type or rather abstract descriptions in the form of strings or objects returned from the *tag()* function that are used to derive *TextNode* respectively *TagNode* instances from.

Parameters

- **node** – The node(s) to be added.
- **clone** – Clones the concrete nodes before adding if `True`.

ancestors (**filter: Callable[[NodeBase], bool]*) → Iterator[_delb.nodes.TagNode]

Parameters **filter** – Any number of *filter* s that a node must match to be yielded.

Returns A generator iterator that yields the ancestor nodes from bottom to top.

append_child (**node: Union[str, NodeBase, _TagDefinition], clone: bool = False*)
Adds one or more nodes as child nodes after any existing to the child nodes of the node this method is called on.

The nodes can be concrete instances of any node type or rather abstract descriptions in the form of strings or objects returned from the *tag()* function that are used to derive *TextNode* respectively *TagNode* instances from.

Parameters

- **node** – The node(s) to be added.
- **clone** – Clones the concrete nodes before adding if `True`.

property attributes

A mapping that can be used to query and alter the node's attributes.

```
>>> node = new_tag_node("node", attributes={"foo": "0", "bar": "0"})
>>> node.attributes
{'foo': '0', 'bar': '0'}
>>> node.attributes.pop("bar")
'0'
>>> node.attributes["foo"] = "1"
>>> node.attributes["peng"] = "1"
>>> print(node)
<node foo="1" peng="1"/>
>>> node.attributes.update({"bar": "2", "zong": "2"})
>>> print(node)
<node foo="1" peng="1" bar="2" zong="2"/>
```

child_nodes(**filter: Callable[[NodeBase], bool], recurse: bool = False*) → Iterator[_delb.nodes.NodeBase]

Parameters

- **filter** – Any number of *filter* s that a node must match to be yielded.
- **recurse** – Also returns the children's children and so on in document order if `True`.

Returns A generator iterator that yields the child nodes of the node.

clone(*deep: bool = False*) → TagNode

Parameters **deep** – Clones the whole subtree if `True`.

Returns A copy of the node.

css_select(*expression: str*) → _delb.nodes.QueryResults

Namespace prefixes are delimited with a `|` before a name test, for example `div svg|metadata` selects all descendants of `div` named nodes that belong to the default namespace or have no namespace and whose name is `metadata` and have a namespace that is mapped to the `svg` prefix.

Parameters **expression** – A CSS selector expression.

Returns A list of matching *tag node* s.

property depth

The depth (or level) of the node in its tree.

detach() → _delb.nodes._ElementWrappingNode

Removes the node, including its descendants, from its tree.

Returns The removed node.

property document

The *Document* instances that the node is associated with or `None`.

property first_child

The node's first child node.

property full_text
The concatenated contents of all text node descendants in document order.

property index
The node's index within the parent's collection of child nodes or `None` when the node has no parent.

insert_child(*index: int*, **node: Union[str, NodeBase, _TagDefinition]*, *clone: bool = False*)
Inserts one or more child nodes.

The nodes can be concrete instances of any node type or rather abstract descriptions in the form of strings or objects returned from the *tag()* function that are used to derive *TextNode* respectively *TagNode* instances from.

Parameters

- **index** – The index at which the first of the given nodes will be inserted, the remaining nodes are added afterwards in the given order.
- **node** – The node(s) to be added.
- **clone** – Clones the concrete nodes before adding if `True`.

iterate_next_nodes(**filter: Callable[[NodeBase], bool]*) → Iterator[_delb.nodes.NodeBase]

Parameters **filter** – Any number of *filter* s that a node must match to be yielded.

Returns A generator iterator that yields the siblings to the node's right.

iterate_next_nodes_in_stream(**filter: Callable[[NodeBase], bool]*) → Iterator[_delb.nodes.NodeBase]

Parameters **filter** – Any number of *filter* s that a node must match to be yielded.

Returns A generator iterator that yields the following nodes in document order.

iterate_previous_nodes(**filter: Callable[[NodeBase], bool]*) → Iterator[_delb.nodes.NodeBase]

Parameters **filter** – Any number of *filter* s that a node must match to be yielded.

Returns A generator iterator that yields the siblings to the node's left.

iterate_previous_nodes_in_stream(**filter: Callable[[NodeBase], bool]*) → Iterator[_delb.nodes.NodeBase]

Parameters **filter** – Any number of *filter* s that a node must match to be yielded.

Returns A generator iterator that yields the previous nodes in document order.

property last_child
The node's last child node.

property local_name
The node's name.

property location_path
An unambiguous XPath location path that points to this node from its tree root.

merge_text_nodes()
Merges all consecutive text nodes in the subtree into one.

property namespace
The node's namespace. Be aware, that while this property can be set to `None`, serializations will continue to render a previous default namespace declaration if the node had such.

property namespaces
The prefix to namespace mapping of the node.

new_tag_node(*local_name: str, attributes: Optional[Dict[str, str]] = None, namespace: Optional[str] = None, children: Sequence[Union[str, _delb.nodes.NodeBase, _TagDefinition]] = ()*) → _delb.nodes.TagNode

Creates a new `TagNode` instance in the node's context.

Parameters

- **local_name** – The tag name.
- **attributes** – Optional attributes that are assigned to the new node.
- **namespace** – An optional tag namespace. If none is provided, the context node's namespace is inherited.
- **children** – An optional sequence of objects that will be appended as child nodes. This can be existing nodes, strings that will be inserted as text nodes and in-place definitions of `TagNode` instances from `tag()`. The latter will be assigned to the same namespace.

Returns The newly created tag node.

next_node(**filter: Callable[[NodeBase], bool]*) → Optional[_delb.nodes.NodeBase]

Parameters **filter** – Any number of *filter* s.

Returns The next sibling to the right that matches all filters or `None`.

next_node_in_stream(**filter: Callable[[NodeBase], bool]*) → Optional[_delb.nodes.NodeBase]

Parameters **filter** – Any number of *filter* s.

Returns The next node in document order that matches all filters or `None`.

property parent

The node's parent or `None`.

property prefix

The prefix that the node's namespace is currently mapped to.

prepend_child(**node: _delb.nodes.NodeBase, clone: bool = False*) → None

Adds one or more nodes as child nodes before any existing to the child nodes of the node this method is called on.

The nodes can be concrete instances of any node type or rather abstract descriptions in the form of strings or objects returned from the `tag()` function that are used to derive `TextNode` respectively `TagNode` instances from.

Parameters

- **node** – The node(s) to be added.
- **clone** – Clones the concrete nodes before adding if `True`.

previous_node(**filter: Callable[[NodeBase], bool]*) → Optional[_delb.nodes.NodeBase]

Parameters **filter** – Any number of *filter* s.

Returns The next sibling to the left that matches all filters or `None`.

previous_node_in_stream(**filter: Callable[[NodeBase], bool]*) → Optional[_delb.nodes.NodeBase]

Parameters **filter** – Any number of *filter* s.

Returns The previous node in document order that matches all filters or `None`.

property qualified_name

The node's qualified name.

replace_with(*node: Union[str, NodeBase, _TagDefinition], clone: bool = False*) → _delb.nodes.NodeBase

Removes the node and places the given one in its tree location.

The node can be a concrete instance of any node type or a rather abstract description in the form of a string or an object returned from the `tag()` function that is used to derive a `TextNode` respectively `TagNode` instance from.

Parameters

- **node** – The replacing node.
- **clone** – A concrete, replacing node is cloned if `True`.

Returns The removed node.

xpath(*expression: str*) → _delb.nodes.QueryResults

Returns all *tag node* s that match the evaluation of an XPath expression.

Mind to start any the expression with a `.` when the node you call it on is supposed to be the initial context node in the path evaluation.

As this API is for a real programming language, the full XPath specification is not intended to be supported. For example, instead of querying attributes with an XPath expression, one must use a comprehension like:

```
>>> [ x.attributes["target"] for x in root.xpath(".//foo")
...   if "target" in x.attributes ]
```

Instead of:

```
>>> root.xpath(".//foo/@target")
```

Having that said, implementing retrieval of attributes may actually happen if there are convincing user stories. But other things like addressing processing instructions and higher level operations are out of scope.

This method includes a workaround for a bug in XPath 1.0 that concerns its lack of default namespace support. It is extensively described in this lxml issue: https://github.com/lxml/lxml/pull/236

Parameters **expression** – An XPath 1.0 location path.

Text

class delb.**TextNode**(*reference_or_text: Union[lxml.etree._Element, str, TextNode], position: int = 0, cache: Optional[Dict[int, _ElementWrappingNode]] = None*)

TextNodes contain the textual data of a document.

Instances expose all methods of `str`:

```
>>> node = TextNode("Show us the way to the next whisky bar.")
>>> node.split()
['Show', 'us', 'the', 'way', 'to', 'the', 'next', 'whisky', 'bar.']
```

Instances can be tested for inequality with other text nodes and strings:

```
>>> TextNode("ham") == TextNode("spam")
False
>>> TextNode("Patsy") == "Patsy"
True
```

And they can be tested for substrings:

```
>>> "Sir" in TextNode("Sir Bedevere the Wise")
True
```

Attributes that rely to child nodes yield nothing respectively `None`.

Properties

depth	The depth (or level) of the node in its tree.
document	The *Document* instances that the node is associated with or `None`.
first_child	
full_text	The concatenated contents of all text node descendants in document order.
index	The node's index within the parent's collection of child nodes or `None` when the node has no parent.
last_child	
parent	The node's parent or `None`.
content	The node's text content.

Fetching a single relative node

next_node(*filter)

param filter Any number of *filter* s.

next_node_in_stream(*filter)

param filter Any number of *filter* s.

previous_node(*filter)

param filter Any number of *filter* s.

previous_node_in_stream(*filter)

param filter Any number of *filter* s.

Iterating over relative nodes

ancestors(*filter)

param filter Any number of *filter* s that a node must match to be

child_nodes(*filter[, recurse])	A generator iterator that yields nothing.

iterate_next_nodes(*filter)

param filter Any number of *filter* s that a node must match to be

continues on next page

Table 24 – continued from previous page

iterate_next_nodes_in_stream(*filter)	**param filter** Any number of *filter* s that a node must match to be
iterate_previous_nodes(*filter)	**param filter** Any number of *filter* s that a node must match to be
iterate_previous_nodes_in_stream(*filter)	**param filter** Any number of *filter* s that a node must match to be

Adding nodes

add_next(*node[, clone])	Adds one or more nodes to the right of the node this method is called on.
add_previous(*node[, clone])	Adds one or more nodes to the left of the node this method is called on.

Removing a node from its tree

detach()	Removes the node, including its descendants, from its tree.
replace_with(node[, clone])	Removes the node and places the given one in its tree location.

Uncategorized methods

__init__(reference_or_text[, position, cache])	Initialize self.
clone([deep])	**param deep** Clones the whole subtree if `True`.
new_tag_node(local_name[, attributes, ...])	Creates a new *TagNode* instance in the node's context.
__contains__(item)	
__eq__(other)	Return self==value.
__getattr__(item)	
__getitem__(item)	
__repr__()	Return repr(self).
__str__()	Return str(self).

add_next(**node: Union[str, NodeBase, _TagDefinition]*, *clone: bool = False*)

Adds one or more nodes to the right of the node this method is called on.

The nodes can be concrete instances of any node type or rather abstract descriptions in the form of strings or objects returned from the *tag()* function that are used to derive *TextNode* respectively *TagNode* instances from.

Parameters

- **node** – The node(s) to be added.
- **clone** – Clones the concrete nodes before adding if `True`.

add_previous(**node: Union[str, NodeBase, _TagDefinition]*, *clone: bool = False*)
Adds one or more nodes to the left of the node this method is called on.

The nodes can be concrete instances of any node type or rather abstract descriptions in the form of strings or objects returned from the *tag()* function that are used to derive *TextNode* respectively *TagNode* instances from.

Parameters

- **node** – The node(s) to be added.
- **clone** – Clones the concrete nodes before adding if `True`.

ancestors(**filter: Callable[[NodeBase], bool]*) → Iterator[_delb.nodes.TagNode]

Parameters **filter** – Any number of *filter* s that a node must match to be yielded.

Returns A generator iterator that yields the ancestor nodes from bottom to top.

child_nodes(**filter: Callable[[NodeBase], bool]*, *recurse: bool = False*) → Iterator[_delb.nodes.NodeBase]
A generator iterator that yields nothing.

clone(*deep: bool = False*) → _delb.nodes.NodeBase

Parameters **deep** – Clones the whole subtree if `True`.

Returns A copy of the node.

property content
The node's text content.

property depth
The depth (or level) of the node in its tree.

detach() → _delb.nodes.TextNode
Removes the node, including its descendants, from its tree.

Returns The removed node.

property document
The *Document* instances that the node is associated with or `None`.

property full_text
The concatenated contents of all text node descendants in document order.

property index
The node's index within the parent's collection of child nodes or `None` when the node has no parent.

iterate_next_nodes(**filter: Callable[[NodeBase], bool]*) → Iterator[_delb.nodes.NodeBase]

Parameters **filter** – Any number of *filter* s that a node must match to be yielded.

Returns A generator iterator that yields the siblings to the node's right.

iterate_next_nodes_in_stream(**filter: Callable[[NodeBase], bool]*) → Iterator[_delb.nodes.NodeBase]

Parameters **filter** – Any number of *filter* s that a node must match to be yielded.

Returns A generator iterator that yields the following nodes in document order.

iterate_previous_nodes(**filter: Callable[[NodeBase], bool]*) → Iterator[_delb.nodes.NodeBase]

Parameters **filter** – Any number of *filter* s that a node must match to be yielded.

Returns A generator iterator that yields the siblings to the node's left.

iterate_previous_nodes_in_stream(**filter: Callable[[NodeBase], bool]*) → Iterator[_delb.nodes.NodeBase]

Parameters **filter** – Any number of *filter* s that a node must match to be yielded.

Returns A generator iterator that yields the previous nodes in document order.

new_tag_node(*local_name: str*, *attributes: Optional[Dict[str, str]] = None*, *namespace: Optional[str] = None*, *children: Sequence[Union[str, _delb.nodes.NodeBase, _TagDefinition]] = ()*) → _delb.nodes.TagNode

Creates a new `TagNode` instance in the node's context.

Parameters

- **local_name** – The tag name.
- **attributes** – Optional attributes that are assigned to the new node.
- **namespace** – An optional tag namespace. If none is provided, the context node's namespace is inherited.
- **children** – An optional sequence of objects that will be appended as child nodes. This can be existing nodes, strings that will be inserted as text nodes and in-place definitions of `TagNode` instances from `tag()`. The latter will be assigned to the same namespace.

Returns The newly created tag node.

next_node(**filter: Callable[[NodeBase], bool]*) → Optional[_delb.nodes.NodeBase]

Parameters **filter** – Any number of *filter* s.

Returns The next sibling to the right that matches all filters or `None`.

next_node_in_stream(**filter: Callable[[NodeBase], bool]*) → Optional[_delb.nodes.NodeBase]

Parameters **filter** – Any number of *filter* s.

Returns The next node in document order that matches all filters or `None`.

property parent

The node's parent or `None`.

previous_node(**filter: Callable[[NodeBase], bool]*) → Optional[_delb.nodes.NodeBase]

Parameters **filter** – Any number of *filter* s.

Returns The next sibling to the left that matches all filters or `None`.

previous_node_in_stream(**filter: Callable[[NodeBase], bool]*) → Optional[_delb.nodes.NodeBase]

Parameters **filter** – Any number of *filter* s.

Returns The previous node in document order that matches all filters or `None`.

replace_with(*node: Union[str, NodeBase, _TagDefinition]*, *clone: bool = False*) → _delb.nodes.NodeBase

Removes the node and places the given one in its tree location.

The node can be a concrete instance of any node type or a rather abstract description in the form of a string or an object returned from the *tag()* function that is used to derive a *TextNode* respectively *TagNode* instance from.

Parameters

- **node** – The replacing node.
- **clone** – A concrete, replacing node is cloned if `True`.

Returns The removed node.

Node constructors

delb.**new_comment_node**(*content: str*) → _delb.nodes.CommentNode

Creates a new *CommentNode*.

Parameters **content** – The comment's content a.k.a. as text.

Returns The newly created comment node.

delb.**new_processing_instruction_node**(*target: str*, *content: str*) → _delb.nodes.ProcessingInstructionNode

Creates a new *ProcessingInstructionNode*.

Parameters

- **target** – The processing instruction's target name.
- **content** – The processing instruction's text.

Returns The newly created processing instruction node.

delb.**new_tag_node**(*local_name: str*, *attributes: Optional[Dict[str, str]] = None*, *namespace: Optional[str] = None*, *children: Sequence[Union[str, NodeBase, _TagDefinition]] = ()*) → _delb.nodes.TagNode

Creates a new *TagNode* instance outside any context. It is preferable to use *new_tag_node()*, on instances of documents and nodes where the instance is the creation context.

Parameters

- **local_name** – The tag name.
- **attributes** – Optional attributes that are assigned to the new node.
- **namespace** – An optional tag namespace.
- **children** – An optional sequence of objects that will be appended as child nodes. This can be existing nodes, strings that will be inserted as text nodes and in-place definitions of *TagNode* instances from *tag()*. The latter will be assigned to the same namespace.

Returns The newly created tag node.

Query results

class delb.**QueryResults**(*results: Iterator[_delb.nodes._ElementWrappingNode]*)
A sequence with the results of a CSS or XPath query with some helpers for readable Python expressions.

as_list() → List[_delb.nodes.TagNode]
The contained nodes as a new `list`.

as_set() → Set[_delb.nodes.TagNode]
The contained nodes as a new `set`.

property as_tuple
The contained nodes in a `tuple`.

count(*value*) → integer – return number of occurrences of value

filtered_by(**filters: Callable[[NodeBase], bool]*) → _delb.nodes.QueryResults
Returns another *`QueryResults`* instance that contains all nodes filtered by the provided *filter* s.

property first
The first node from the results or `None` if there are none.

index(*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

Supporting start and stop arguments is optional, but recommended.

property last
The last node from the results or `None` if there are none.

property size
The amount of contained nodes.

4.2.4 Filters

Default filters

delb.**altered_default_filters**(**filter: Callable[[NodeBase], bool]*, *extend: bool = False*)
This function can be either used as as context manager or decorator to define a set of `default_filters` for the encapsuled code block or callable. These are then applied in all operations that allow node filtering, like *`TagNode.next_node()`*. Mind that they also affect a node's index property and indexed access to child nodes.

```
>>> root = Document(
...     '<root xmlns="foo"><a/><!--x--><b/><!--y--><c/></root>'
... ).root
>>> with altered_default_filters(is_comment_node):
...     print([x.content for x in root.child_nodes()])
['x', 'y']
```

As the default filters shadow comments and processing instructions by default, use no argument to unset this in order to access all type of nodes.

Parameters **extend** – Extends the currently active filters with the given ones instead of replacing them.

Contributed filters

delb.**any_of**(**filter: Callable[[NodeBase], bool]*) → Callable[[_delb.nodes.NodeBase], bool]
A node filter wrapper that matches when any of the given filters is matching, like a boolean `or`.

delb.**is_comment_node**(*node: _delb.nodes.NodeBase*) → bool
A node filter that matches *CommentNode* instances.

delb.**is_processing_instruction_node**(*node: _delb.nodes.NodeBase*) → bool
A node filter that matches *ProcessingInstructionNode* instances.

delb.**is_tag_node**(*node: _delb.nodes.NodeBase*) → bool
A node filter that matches *TagNode* instances.

delb.**is_text_node**(*node: _delb.nodes.NodeBase*) → bool
A node filter that matches *TextNode* instances.

delb.**not_**(*filter: Callable[[NodeBase], bool]*) → Callable[[_delb.nodes.NodeBase], bool]
A node filter wrapper that matches when the given filter is not matching, like a boolean `not`.

4.2.5 Various helpers

delb.**first**(*iterable: Iterable*) → Optional[Any]
Returns the first item of the given iterable or `None` if it's empty. Note that the first item is consumed when the iterable is an iterator.

delb.**get_traverser**(*from_left=True*, *depth_first=True*, *from_top=True*)
Returns a function that can be used to traverse a (sub)tree with the given node as root. While traversing the given root node is yielded at some point.

The returned functions have this signature:

```
def traverser(root: NodeBase, *filters: Filter) -> Iterator[NodeBase]:
    ...
```

Parameters

- **from_left** – The traverser yields sibling nodes from left to right if `True`, or starting from the right if `False`.
- **depth_first** – The child nodes resp. the parent node are yielded before the siblings of a node by a traverser if `True`. Siblings are favored if `False`.
- **from_top** – The traverser starts yielding nodes with the lowest depth if `True`. When `False`, again, the opposite is in effect.

delb.**last**(*iterable: Iterable*) → Optional[Any]
Returns the last item of the given iterable or `None` if it's empty. Note that the whole iterator is consumed when such is given.

delb.**register_namespace**(*prefix: str*, *namespace: str*)
Registers a namespace prefix that newly created *TagNode* instances in that namespace will use in serializations.

The registry is global, and any existing mapping for either the given prefix or the namespace URI will be removed. It has however no effect on the serialization of existing nodes, see `Document.cleanup_namespace()` for that.

Parameters

- **prefix** – The prefix to register.

- **namespace** – The targeted namespace.

delb.**tag**(*local_name: str*)
delb.**tag**(*local_name: str*, *attributes: Mapping[str, str]*)
delb.**tag**(*local_name: str*, *child: NodeSource*)
delb.**tag**(*local_name: str*, *children: Sequence[NodeSource]*)
delb.**tag**(*local_name: str*, *attributes: Mapping[str, str]*, *child: NodeSource*)
delb.**tag**(*local_name: str*, *attributes: Mapping[str, str]*, *children: Sequence[NodeSource]*)

This function can be used for in-place creation (or call it templating if you want to) of *TagNode* instances as:

- `node` argument to methods that add nodes to a tree
- items in the `children` argument of *new_tag_node()* and `NodeBase.new_tag_node()`

The first argument to the function is always the local name of the tag node. Optionally, the second argument can be a mapping that specifies attributes for that node. The optional last argument is either a single object that will be appended as child node or a sequence of such, these objects can be node instances of any type, strings (for derived *TextNode* instances) or other definitions from this function (for derived *TagNode* instances).

The actual nodes that are constructed always inherit the namespace of the context node they are created in.

```
>>> root = new_tag_node('root', children=[
...     tag("head", {"lvl": "1"}, "Hello!"),
...     tag("items", (
...         tag("item1"),
...         tag("item2"),
...         )
...     )
... ])
>>> str(root)
'<root><head lvl="1">Hello!</head><items><item1/><item2/></items></root>'
>>> root.append_child(tag("addendum"))
>>> str(root)[-26:]
'</items><addendum/></root>'
```

4.2.6 Exceptions

exception delb.exceptions.**InvalidCodePath**

Raised when a code path that is not expected to be executed is reached.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception delb.exceptions.**InvalidOperation**

Raised when an invalid operation is attempted by the client code.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

4.3 Extending delb

Note: There are actually two packages that are installed with *delb*: `delb` and `_delb`. As the underscore indicates, the latter is exposing private parts of the API while the first is re-exposing what is deemed to be public from that one and additional contents. As a thumb of rule, use the public API in applications and the private API in *delb* extensions. By doing so, you can avoid circular dependencies if your extension (or other code that it depends on) uses contents from the `delb` API.

`delb` offers a plugin system to facilitate the extendability of a few of its mechanics with Python packages. A package that extends its functionality must provide entrypoint metadata for an entrypoint group named `delb` that points to modules that contain extensions. The individual extensions have to be decorated with specific methods of the plugin manager object (see the following sections).

Authors are encouraged to prefix their package names with `delb-` in order to increase discoverability.

There are currently two distinct plugin types: *loaders* and *document extension classes*. *Loaders* are functions that try to make sense of any given input value, and if they can they return a parsed document. *Extension classes* add functionality / attributes to the *`delb.Document`* class as *mixin classes* (instead of inheriting from it). That allows applications to rely optionally on the availability of plugins. The designated means of communication between these two extension types is the `config` argument to the loader respectively the instance property of a document instance with that name.

Warning: A module that contains plugins and any module it is explicitly or implicitly importing **must not** import anything from the `delb` module itself, because that would initiate the collection of plugin implementations. And these wouldn't have been completely registered at that point.

Caution: Mind to re-install a package in development when its entrypoint specification changed.

There's a repository that outlines the mechanics as developer reference: https://github.com/funkyfuture/delb-reference-plugins

There's also the snakesist project that implements both plugin types to interact with eXist-db as storage.

4.3.1 Document loaders

Loaders are registered with this decorator:

_delb.plugins.plugin_manager.**register_loader**(*before: Optional[Union[Callable[[Any, types.SimpleNamespace], Union[Tuple[Optional[lxml.etree._ElementTree], Dict[int, _ElementWrappingNode]], str]], Iterable[Callable[[Any, types.SimpleNamespace], Union[Tuple[Optional[lxml.etree._ElementTree], Dict[int, _ElementWrappingNode]], str]]]]] = None*, *after: Optional[Union[Callable[[Any, types.SimpleNamespace], Union[Tuple[Optional[lxml.etree._ElementTree], Dict[int, _ElementWrappingNode]], str]], Iterable[Callable[[Any, types.SimpleNamespace], Union[Tuple[Optional[lxml.etree._ElementTree], Dict[int, _ElementWrappingNode]], str]]]] = None*) → Callable

Registers a document loader.

An example module that is specified as `delb` plugin for an IPFS loader might look like this:

```
from os import getenv
from types import SimpleNamespace
from typing import Any

from _delb.plugins import plugin_manager
from _delb.plugins.https_loader import https_loader
from _delb.typing import LoaderResult


IPFS_GATEWAY = getenv("IPFS_GATEWAY_PREFIX", "https://ipfs.io/ipfs/")


@plugin_manager.register_loader()
def ipfs_loader(source: Any, config: SimpleNamespace) -> LoaderResult:
    if isinstance(source, str) and source.startswith("ipfs://"):

        config.source_url = source
        config.ipfs_gateway_source_url = IPFS_GATEWAY + source[7:]

        return https_loader(config.ipfs_gateway_source_url, config)

    # return an indication why this loader didn't attempt to load in order
    # to support debugging
    return "The input value is not an URL with the ipfs scheme."
```

The `source` argument is what a `Document` instance is initialized with as input data.

Note that the `config` argument that is passed to a loader function contains configuration data, it's the *`delb.Document.config`* property after *`_delb.plugins.DocumentExtensionHooks._init_config()`* has been processed.

Loaders that retrieve a document from an URL should add the origin as string to the `config` object as

source_url.

You might want to specify a loader to be considered before or after another one. Let's assume a loader shall figure out what to load from a remote XML resource that contains a reference to the actual document. That one would have to be considered before the one that loads XML documents from a URL with the *https* scheme:

```
from _delb.plugins import plugin_manager
from _delb.plugins.https_loader import https_loader


@plugin_manager.register_loader(before=https_loader)
def mets_loader(source, config) -> LoaderResult:
    # loading logic here
    pass
```

4.3.2 Document extensions

Document extension classes are registered with *_delb.plugins.plugin_manager.register_document_extension()*:

_delb.plugins.plugin_manager.**register_document_extension**(*extension: Type*) → Type

This decorator registers document extension classes which are supposed to add additional attributes to a document, e.g. derived data or methods to interact with storage systems. All attributes of an extension should share a common prefix that terminates with an underscore, e.g. *storage_load*, *storage_save*, etc.

There are hook methods that an extension can implement, they are declared in *_delb.plugins.DocumentExtensionHooks*.

Extension classes are *mixin classes* in Python OOP jargon.

They can implement methods that are called from builtin *delb.Document* methods:

class _delb.plugins.**DocumentExtensionHooks**

This class acts as termination for methods that can be implemented by extension classes. Any implementation of a method must call a base class' one with `super()`.

_init_config(*config_args: Dict[str, Any]*)

The `config_args` contains the additional keyword arguments that a `Document` instance is called with. Extension classes that expect configuration data *must* process their specific arguments by clearing them from the `config_args` dictionary, e.g. with `dict.pop()`, and preferably storing the final configuration data in a `types.SimpleNamespace` and bind it to the instance's `Document.config` property with the extension's name. The initially mentioned keyword arguments *should* be prefixed with that name as well. This method is called before the loaders try to read and parse the given source for a document.

4.4 Design

4.4.1 tl;dr

lxml resp. libxml2 are powerful tools, but have an unergonomic data model to work with encoded text. Let's build a DOM API inspired wrapper around it.

4.4.2 Aspects & Caveats

The library is partly opinionated to encourage good practices and to be more pythonic. Therefore its behaviour deviates from lxml and ignores stuff:

- Serializations of documents are UTF-8 encoded by default and always start with an XML declaration.
- Comment and processing instruction nodes are shadowed by default, see *delb.altered_default_filters()* on how to make them accessible.
- CDATA nodes are not accessible at all, but are retained and appear in serializations; unless you **[DANGER ZONE]** manipulate the tree (and you want that often). Depending on your actions you might encounter no alterations or a complete loss of these nodes within the root node. **[/DANGER ZONE]**

If you need to apply bad practices anyway, you can fall back to tinker with the lxml objects that are bound to `TagNode._etree_obj`.

4.4.3 Reasoning

XML can be used to encode text documents, examples for such uses would be the Open Document Format and XML-TEI. It's more prevalent use however is to encode data that is to be consumed by algorithms as configuration, measurements, application events, various metadata and so on.

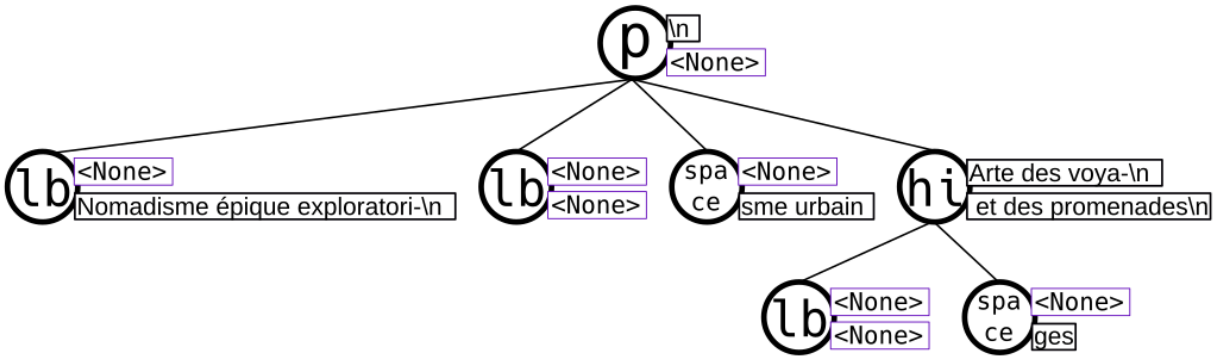
Python is a high-level, general programming language with a vast ecosystem, notably including diverse scientific communities and tools. As such it is well suited to solve and cause problems in the humanities related field of Research Software Engineering by programmers with different educational background and expertise.

The commonly used Python library to parse and interact with a representation of an XML document is lxml. Other libraries like the `xml.etree.ElementTree` module from the Python standard library shall not be discussed due to their insignificance and shortcomings. It is notable that at least these two share significant design aspects of Java APIs which is perceived as weird and clumsy in Python code. lxml is a wrapper around libxml2 which was developed by the GNOME developers for other data than text documents. Data that is strictly structured and expectable. Text documents are different in these regards as the encoding mixes different abstracted encapsulations of logical and physical text fragments. And they are formulated and structured for human consumers, and often printing devices.

So, what's wrong with lxml? Not much, it's a rock-solid, fast API for XML documents with known issues and known workarounds that represents the full glory of what a full-fledged specification implies - of which a lot is not of concern for the problems at hand and occasionally make solutions complicated. The one aspect that's very wrong in the context of text processing is unfortunately its central model of elements and data/text. In particular the notion of an element *tail* makes the whole enchilada tricky to traverse / navigate. The existence of this attribute is due to the insignificance of these fragments of an XML stream in the aforementioned, common uses of XML. Now it is time for an example, given this document snippet:

```
<p rendition="#justify">
  <lb/>Nomadisme épique exploratori-
  <lb/><space dim="horizontal" quantity="2" units="chars"/>sme urbain <hi rendition="
↪#b">Art des voya-
  <lb/><space dim="horizontal" quantity="2" units="chars"/>ges</hi> et des promenades
</p>
```

Here's a graphical representation of the markup with etree's elements and their text and tail attributes:



When thinking about a paragraph of text, a way to conceptualize it is as a sequence of sentences, formed by a series of words, a sequence of graphemes, and punctuation. That's a quite simple cascade of categories which can be very well anticipated when processing text. With that mental model, line beginnings would rather be considered to be on the same level as signs, but "Nomadisme …" turns out *not* to be a sibling object of the object that represents the line beginning and is *not* in direct relation with the paragraph. In lxml's model it is rather an attribute `tail` assigned to that line beginning. The text contents of the object that represents the `hi` element and its children give a good impression how hairy simple tasks can become.

An algorithm that shall remove line beginnings, space representations and concatenate broken words would need a function that removes the element objects in question while preserving the text fragments in its meaningful sequence attached to the `text` and `tail` properties. In case these have no content, their value of `None` leads to different operations to concatenate strings. Here's a working implementation from the inxs library that is used by a variety of more specific functions:

```
def remove_elements(*elements: etree.ElementBase, keep_children=False,
                    preserve_text=False,
                    preserve_tail=False) -> None:
    """ Removes the given elements from its tree. Unless ``keep_children`` is
        passed as ``True``, its children vanish with it into void. If
        ``preserve_text`` is ``True``, the text and tail of a deleted element
        will be preserved either in its left sibling's tail or its parent's
        text. """
    for element in elements:
        if preserve_text and element.text:
            previous = element.getprevious()
            if previous is None:

                parent = element.getparent()
                if parent.text is None:
                    parent.text = ''
                parent.text += element.text
            else:
                if previous.tail is None:
                    previous.tail = element.text
                else:
                    previous.tail += element.text

        if preserve_tail and element.tail:
            if keep_children and len(element):
                if element[-1].tail:
                    element[-1].tail += element.tail
                else:
                    element[-1].tail = element.tail
```

(continues on next page)

(continued from previous page)

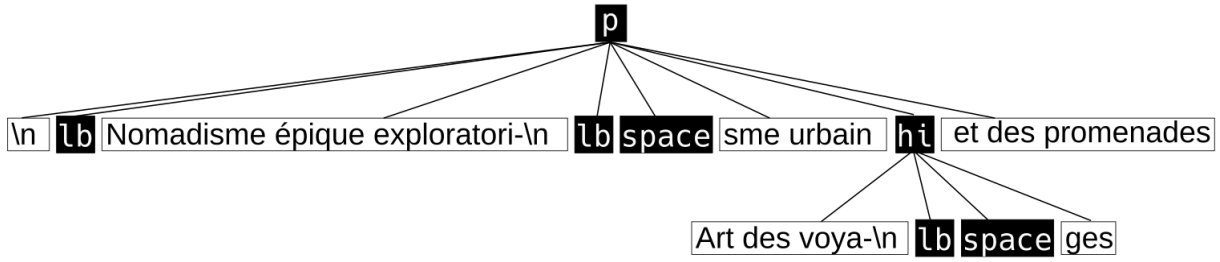
```
            else:
                previous = element.getprevious()
                if previous is None:
                    parent = element.getparent()
                    if parent.text is None:
                        parent.text = ''
                    parent.text += element.tail
                else:
                    if len(element):
                        if element[-1].tail is None:
                            element[-1].tail = element.tail
                        else:
                            element[-1].tail += element.tail
                    else:
                        if previous.tail is None:
                            previous.tail = ''
                        previous.tail += element.tail

        if keep_children:
            for child in element:
                element.addprevious(child)
        element.getparent().remove(element)
```

That by itself is enough to simply remove the `space` elements, but also considering word-breaking dashes to wrap everything up is a similar piece of routine of its own. And these quirks come back to you steadily while actual markup is regularly more complex.

Now obviously, the data model that lxml / libxml2 provides is not up to standard Python ergonomics to solve text encoding problems at hand.

There must be a better way.

There is a notable other markup parser that wraps around lxml, BeautifulSoup4. It carries some interesting ideas, but is overall too opinionated and partly ambiguous to implement a stringent data model. A notable specification of a solid model for text documents is the DOM API that is even implemented in the standard library's `xml.dom.minidom` module. But it lacks an XPath interface and rumours say it's slow. To illustrate the more accessible model with a better locatability, here's another graphical representation of the markup example from above with two different types of nodes:



Note that text containing attributes appear in document order which promises an eased lookaround. So, the obvious (?) idea is to wrap lxml in a layer that takes the DOM API as paradigmatic inspiration, looks and behaves pythonic while keeping the wrapped powers accessible.

Now with that API at hand, this is what an equivalent of the horribly complicated function would look like:

```
def remove_nodes(*nodes: NodeBase, keep_children=False):
    """ Removes the given nodes from its tree. Unless ``keep_children`` is
```

(continues on next page)

(continued from previous page)

```
        passed as ``True``, its children vanish with it into void. """

    for node in nodes:
        parent = node.parent
        index = node.index
        node.detach()
        if keep_children:
            with altered_default_filters():
                parent.insert_node(
                    index,
                    *tuple(x.detach() for x in node.child_nodes())
                )
```

4.5 Glossary

filter Filter functions can be used as arguments with various methods on node instances that return other nodes. They are called with a node instance as only argument and they should return a `bool` to indicate whether the node matches the filter. Have a look at the *Filters* source code for examples.

tag node Tag nodes are the equivalent to the DOM's element node. Its name shall make it distinguishable from the ElementTree API and relates to the nodes' functionality of tagging text.

4.6 Index

4.7 License

```
                    GNU AFFERO GENERAL PUBLIC LICENSE
                       Version 3, 19 November 2007

 Copyright (C) 2007 Free Software Foundation, Inc. <https://fsf.org/>
 Everyone is permitted to copy and distribute verbatim copies
 of this license document, but changing it is not allowed.

                            Preamble

  The GNU Affero General Public License is a free, copyleft license for
software and other kinds of works, specifically designed to ensure
cooperation with the community in the case of network server software.

  The licenses for most software and other practical works are designed
to take away your freedom to share and change the works.  By contrast,
our General Public Licenses are intended to guarantee your freedom to
share and change all versions of a program--to make sure it remains free
software for all its users.

  When we speak of free software, we are referring to freedom, not
price.  Our General Public Licenses are designed to make sure that you
have the freedom to distribute copies of free software (and charge for
them if you wish), that you receive source code or can get it if you
want it, that you can change the software or use pieces of it in new
free programs, and that you know you can do these things.
```

(continues on next page)

(continued from previous page)

```
  Developers that use our General Public Licenses protect your rights
with two steps: (1) assert copyright on the software, and (2) offer
you this License which gives you legal permission to copy, distribute
and/or modify the software.

  A secondary benefit of defending all users' freedom is that
improvements made in alternate versions of the program, if they
receive widespread use, become available for other developers to
incorporate.  Many developers of free software are heartened and
encouraged by the resulting cooperation.  However, in the case of
software used on network servers, this result may fail to come about.
The GNU General Public License permits making a modified version and
letting the public access it on a server without ever releasing its
source code to the public.

  The GNU Affero General Public License is designed specifically to
ensure that, in such cases, the modified source code becomes available
to the community.  It requires the operator of a network server to
provide the source code of the modified version running there to the
users of that server.  Therefore, public use of a modified version, on
a publicly accessible server, gives the public access to the source
code of the modified version.

  An older license, called the Affero General Public License and
published by Affero, was designed to accomplish similar goals.  This is
a different license, not a version of the Affero GPL, but Affero has
released a new version of the Affero GPL which permits relicensing under
this license.

  The precise terms and conditions for copying, distribution and
modification follow.

                       TERMS AND CONDITIONS

  0. Definitions.

  "This License" refers to version 3 of the GNU Affero General Public License.

  "Copyright" also means copyright-like laws that apply to other kinds of
works, such as semiconductor masks.

  "The Program" refers to any copyrightable work licensed under this
License.  Each licensee is addressed as "you".  "Licensees" and
"recipients" may be individuals or organizations.

  To "modify" a work means to copy from or adapt all or part of the work
in a fashion requiring copyright permission, other than the making of an
exact copy.  The resulting work is called a "modified version" of the
earlier work or a work "based on" the earlier work.

  A "covered work" means either the unmodified Program or a work based
on the Program.

  To "propagate" a work means to do anything with it that, without
permission, would make you directly or secondarily liable for
infringement under applicable copyright law, except executing it on a
```

(continues on next page)

(continued from previous page)

```
computer or modifying a private copy.  Propagation includes copying,
distribution (with or without modification), making available to the
public, and in some countries other activities as well.

  To "convey" a work means any kind of propagation that enables other
parties to make or receive copies.  Mere interaction with a user through
a computer network, with no transfer of a copy, is not conveying.

  An interactive user interface displays "Appropriate Legal Notices"
to the extent that it includes a convenient and prominently visible
feature that (1) displays an appropriate copyright notice, and (2)
tells the user that there is no warranty for the work (except to the
extent that warranties are provided), that licensees may convey the
work under this License, and how to view a copy of this License.  If
the interface presents a list of user commands or options, such as a
menu, a prominent item in the list meets this criterion.

  1. Source Code.

  The "source code" for a work means the preferred form of the work
for making modifications to it.  "Object code" means any non-source
form of a work.

  A "Standard Interface" means an interface that either is an official
standard defined by a recognized standards body, or, in the case of
interfaces specified for a particular programming language, one that
is widely used among developers working in that language.

  The "System Libraries" of an executable work include anything, other
than the work as a whole, that (a) is included in the normal form of
packaging a Major Component, but which is not part of that Major
Component, and (b) serves only to enable use of the work with that
Major Component, or to implement a Standard Interface for which an
implementation is available to the public in source code form.  A
"Major Component", in this context, means a major essential component
(kernel, window system, and so on) of the specific operating system
(if any) on which the executable work runs, or a compiler used to
produce the work, or an object code interpreter used to run it.

  The "Corresponding Source" for a work in object code form means all
the source code needed to generate, install, and (for an executable
work) run the object code and to modify the work, including scripts to
control those activities.  However, it does not include the work's
System Libraries, or general-purpose tools or generally available free
programs which are used unmodified in performing those activities but
which are not part of the work.  For example, Corresponding Source
includes interface definition files associated with source files for
the work, and the source code for shared libraries and dynamically
linked subprograms that the work is specifically designed to require,
such as by intimate data communication or control flow between those
subprograms and other parts of the work.

  The Corresponding Source need not include anything that users
can regenerate automatically from other parts of the Corresponding
Source.

  The Corresponding Source for a work in source code form is that
```

(continues on next page)

(continued from previous page)

```
same work.

  2. Basic Permissions.

  All rights granted under this License are granted for the term of
copyright on the Program, and are irrevocable provided the stated
conditions are met.  This License explicitly affirms your unlimited
permission to run the unmodified Program.  The output from running a
covered work is covered by this License only if the output, given its
content, constitutes a covered work.  This License acknowledges your
rights of fair use or other equivalent, as provided by copyright law.

  You may make, run and propagate covered works that you do not
convey, without conditions so long as your license otherwise remains
in force.  You may convey covered works to others for the sole purpose
of having them make modifications exclusively for you, or provide you
with facilities for running those works, provided that you comply with
the terms of this License in conveying all material for which you do
not control copyright.  Those thus making or running the covered works
for you must do so exclusively on your behalf, under your direction
and control, on terms that prohibit them from making any copies of
your copyrighted material outside their relationship with you.

  Conveying under any other circumstances is permitted solely under
the conditions stated below.  Sublicensing is not allowed; section 10
makes it unnecessary.

  3. Protecting Users' Legal Rights From Anti-Circumvention Law.

  No covered work shall be deemed part of an effective technological
measure under any applicable law fulfilling obligations under article
11 of the WIPO copyright treaty adopted on 20 December 1996, or
similar laws prohibiting or restricting circumvention of such
measures.

  When you convey a covered work, you waive any legal power to forbid
circumvention of technological measures to the extent such circumvention
is effected by exercising rights under this License with respect to
the covered work, and you disclaim any intention to limit operation or
modification of the work as a means of enforcing, against the work's
users, your or third parties' legal rights to forbid circumvention of
technological measures.

  4. Conveying Verbatim Copies.

  You may convey verbatim copies of the Program's source code as you
receive it, in any medium, provided that you conspicuously and
appropriately publish on each copy an appropriate copyright notice;
keep intact all notices stating that this License and any
non-permissive terms added in accord with section 7 apply to the code;
keep intact all notices of the absence of any warranty; and give all
recipients a copy of this License along with the Program.

  You may charge any price or no price for each copy that you convey,
and you may offer support or warranty protection for a fee.

  5. Conveying Modified Source Versions.
```

(continues on next page)

(continued from previous page)

```
  You may convey a work based on the Program, or the modifications to
produce it from the Program, in the form of source code under the
terms of section 4, provided that you also meet all of these conditions:

    a) The work must carry prominent notices stating that you modified
    it, and giving a relevant date.

    b) The work must carry prominent notices stating that it is
    released under this License and any conditions added under section
    7.  This requirement modifies the requirement in section 4 to
    "keep intact all notices".

    c) You must license the entire work, as a whole, under this
    License to anyone who comes into possession of a copy.  This
    License will therefore apply, along with any applicable section 7
    additional terms, to the whole of the work, and all its parts,
    regardless of how they are packaged.  This License gives no
    permission to license the work in any other way, but it does not
    invalidate such permission if you have separately received it.

    d) If the work has interactive user interfaces, each must display
    Appropriate Legal Notices; however, if the Program has interactive
    interfaces that do not display Appropriate Legal Notices, your
    work need not make them do so.

  A compilation of a covered work with other separate and independent
works, which are not by their nature extensions of the covered work,
and which are not combined with it such as to form a larger program,
in or on a volume of a storage or distribution medium, is called an
"aggregate" if the compilation and its resulting copyright are not
used to limit the access or legal rights of the compilation's users
beyond what the individual works permit.  Inclusion of a covered work
in an aggregate does not cause this License to apply to the other
parts of the aggregate.

  6. Conveying Non-Source Forms.

  You may convey a covered work in object code form under the terms
of sections 4 and 5, provided that you also convey the
machine-readable Corresponding Source under the terms of this License,
in one of these ways:

    a) Convey the object code in, or embodied in, a physical product
    (including a physical distribution medium), accompanied by the
    Corresponding Source fixed on a durable physical medium
    customarily used for software interchange.

    b) Convey the object code in, or embodied in, a physical product
    (including a physical distribution medium), accompanied by a
    written offer, valid for at least three years and valid for as
    long as you offer spare parts or customer support for that product
    model, to give anyone who possesses the object code either (1) a
    copy of the Corresponding Source for all the software in the
    product that is covered by this License, on a durable physical
    medium customarily used for software interchange, for a price no
    more than your reasonable cost of physically performing this
```

(continues on next page)

(continued from previous page)

```
    conveying of source, or (2) access to copy the
    Corresponding Source from a network server at no charge.

    c) Convey individual copies of the object code with a copy of the
    written offer to provide the Corresponding Source.  This
    alternative is allowed only occasionally and noncommercially, and
    only if you received the object code with such an offer, in accord
    with subsection 6b.

    d) Convey the object code by offering access from a designated
    place (gratis or for a charge), and offer equivalent access to the
    Corresponding Source in the same way through the same place at no
    further charge.  You need not require recipients to copy the
    Corresponding Source along with the object code.  If the place to
    copy the object code is a network server, the Corresponding Source
    may be on a different server (operated by you or a third party)
    that supports equivalent copying facilities, provided you maintain
    clear directions next to the object code saying where to find the
    Corresponding Source.  Regardless of what server hosts the
    Corresponding Source, you remain obligated to ensure that it is
    available for as long as needed to satisfy these requirements.

    e) Convey the object code using peer-to-peer transmission, provided
    you inform other peers where the object code and Corresponding
    Source of the work are being offered to the general public at no
    charge under subsection 6d.

  A separable portion of the object code, whose source code is excluded
from the Corresponding Source as a System Library, need not be
included in conveying the object code work.

  A "User Product" is either (1) a "consumer product", which means any
tangible personal property which is normally used for personal, family,
or household purposes, or (2) anything designed or sold for incorporation
into a dwelling.  In determining whether a product is a consumer product,
doubtful cases shall be resolved in favor of coverage.  For a particular
product received by a particular user, "normally used" refers to a
typical or common use of that class of product, regardless of the status
of the particular user or of the way in which the particular user
actually uses, or expects or is expected to use, the product.  A product
is a consumer product regardless of whether the product has substantial
commercial, industrial or non-consumer uses, unless such uses represent
the only significant mode of use of the product.

  "Installation Information" for a User Product means any methods,
procedures, authorization keys, or other information required to install
and execute modified versions of a covered work in that User Product from
a modified version of its Corresponding Source.  The information must
suffice to ensure that the continued functioning of the modified object
code is in no case prevented or interfered with solely because
modification has been made.

  If you convey an object code work under this section in, or with, or
specifically for use in, a User Product, and the conveying occurs as
part of a transaction in which the right of possession and use of the
User Product is transferred to the recipient in perpetuity or for a
fixed term (regardless of how the transaction is characterized), the
```

(continues on next page)

(continued from previous page)

```
Corresponding Source conveyed under this section must be accompanied
by the Installation Information.  But this requirement does not apply
if neither you nor any third party retains the ability to install
modified object code on the User Product (for example, the work has
been installed in ROM).

  The requirement to provide Installation Information does not include a
requirement to continue to provide support service, warranty, or updates
for a work that has been modified or installed by the recipient, or for
the User Product in which it has been modified or installed.  Access to a
network may be denied when the modification itself materially and
adversely affects the operation of the network or violates the rules and
protocols for communication across the network.

  Corresponding Source conveyed, and Installation Information provided,
in accord with this section must be in a format that is publicly
documented (and with an implementation available to the public in
source code form), and must require no special password or key for
unpacking, reading or copying.

  7. Additional Terms.

  "Additional permissions" are terms that supplement the terms of this
License by making exceptions from one or more of its conditions.
Additional permissions that are applicable to the entire Program shall
be treated as though they were included in this License, to the extent
that they are valid under applicable law.  If additional permissions
apply only to part of the Program, that part may be used separately
under those permissions, but the entire Program remains governed by
this License without regard to the additional permissions.

  When you convey a copy of a covered work, you may at your option
remove any additional permissions from that copy, or from any part of
it.  (Additional permissions may be written to require their own
removal in certain cases when you modify the work.)  You may place
additional permissions on material, added by you to a covered work,
for which you have or can give appropriate copyright permission.

  Notwithstanding any other provision of this License, for material you
add to a covered work, you may (if authorized by the copyright holders of
that material) supplement the terms of this License with terms:

    a) Disclaiming warranty or limiting liability differently from the
    terms of sections 15 and 16 of this License; or

    b) Requiring preservation of specified reasonable legal notices or
    author attributions in that material or in the Appropriate Legal
    Notices displayed by works containing it; or

    c) Prohibiting misrepresentation of the origin of that material, or
    requiring that modified versions of such material be marked in
    reasonable ways as different from the original version; or

    d) Limiting the use for publicity purposes of names of licensors or
    authors of the material; or

    e) Declining to grant rights under trademark law for use of some
```

(continues on next page)

(continued from previous page)

```
    trade names, trademarks, or service marks; or

    f) Requiring indemnification of licensors and authors of that
    material by anyone who conveys the material (or modified versions of
    it) with contractual assumptions of liability to the recipient, for
    any liability that these contractual assumptions directly impose on
    those licensors and authors.

  All other non-permissive additional terms are considered "further
restrictions" within the meaning of section 10.  If the Program as you
received it, or any part of it, contains a notice stating that it is
governed by this License along with a term that is a further
restriction, you may remove that term.  If a license document contains
a further restriction but permits relicensing or conveying under this
License, you may add to a covered work material governed by the terms
of that license document, provided that the further restriction does
not survive such relicensing or conveying.

  If you add terms to a covered work in accord with this section, you
must place, in the relevant source files, a statement of the
additional terms that apply to those files, or a notice indicating
where to find the applicable terms.

  Additional terms, permissive or non-permissive, may be stated in the
form of a separately written license, or stated as exceptions;
the above requirements apply either way.

  8. Termination.

  You may not propagate or modify a covered work except as expressly
provided under this License.  Any attempt otherwise to propagate or
modify it is void, and will automatically terminate your rights under
this License (including any patent licenses granted under the third
paragraph of section 11).

  However, if you cease all violation of this License, then your
license from a particular copyright holder is reinstated (a)
provisionally, unless and until the copyright holder explicitly and
finally terminates your license, and (b) permanently, if the copyright
holder fails to notify you of the violation by some reasonable means
prior to 60 days after the cessation.

  Moreover, your license from a particular copyright holder is
reinstated permanently if the copyright holder notifies you of the
violation by some reasonable means, this is the first time you have
received notice of violation of this License (for any work) from that
copyright holder, and you cure the violation prior to 30 days after
your receipt of the notice.

  Termination of your rights under this section does not terminate the
licenses of parties who have received copies or rights from you under
this License.  If your rights have been terminated and not permanently
reinstated, you do not qualify to receive new licenses for the same
material under section 10.

  9. Acceptance Not Required for Having Copies.
```

(continues on next page)

(continued from previous page)

```
  You are not required to accept this License in order to receive or
run a copy of the Program.  Ancillary propagation of a covered work
occurring solely as a consequence of using peer-to-peer transmission
to receive a copy likewise does not require acceptance.  However,
nothing other than this License grants you permission to propagate or
modify any covered work.  These actions infringe copyright if you do
not accept this License.  Therefore, by modifying or propagating a
covered work, you indicate your acceptance of this License to do so.

  10. Automatic Licensing of Downstream Recipients.

  Each time you convey a covered work, the recipient automatically
receives a license from the original licensors, to run, modify and
propagate that work, subject to this License.  You are not responsible
for enforcing compliance by third parties with this License.

  An "entity transaction" is a transaction transferring control of an
organization, or substantially all assets of one, or subdividing an
organization, or merging organizations.  If propagation of a covered
work results from an entity transaction, each party to that
transaction who receives a copy of the work also receives whatever
licenses to the work the party's predecessor in interest had or could
give under the previous paragraph, plus a right to possession of the
Corresponding Source of the work from the predecessor in interest, if
the predecessor has it or can get it with reasonable efforts.

  You may not impose any further restrictions on the exercise of the
rights granted or affirmed under this License.  For example, you may
not impose a license fee, royalty, or other charge for exercise of
rights granted under this License, and you may not initiate litigation
(including a cross-claim or counterclaim in a lawsuit) alleging that
any patent claim is infringed by making, using, selling, offering for
sale, or importing the Program or any portion of it.

  11. Patents.

  A "contributor" is a copyright holder who authorizes use under this
License of the Program or a work on which the Program is based.  The
work thus licensed is called the contributor's "contributor version".

  A contributor's "essential patent claims" are all patent claims
owned or controlled by the contributor, whether already acquired or
hereafter acquired, that would be infringed by some manner, permitted
by this License, of making, using, or selling its contributor version,
but do not include claims that would be infringed only as a
consequence of further modification of the contributor version.  For
purposes of this definition, "control" includes the right to grant
patent sublicenses in a manner consistent with the requirements of
this License.

  Each contributor grants you a non-exclusive, worldwide, royalty-free
patent license under the contributor's essential patent claims, to
make, use, sell, offer for sale, import and otherwise run, modify and
propagate the contents of its contributor version.

  In the following three paragraphs, a "patent license" is any express
agreement or commitment, however denominated, not to enforce a patent
```

(continues on next page)

(continued from previous page)

```
(such as an express permission to practice a patent or covenant not to
sue for patent infringement).  To "grant" such a patent license to a
party means to make such an agreement or commitment not to enforce a
patent against the party.

  If you convey a covered work, knowingly relying on a patent license,
and the Corresponding Source of the work is not available for anyone
to copy, free of charge and under the terms of this License, through a
publicly available network server or other readily accessible means,
then you must either (1) cause the Corresponding Source to be so
available, or (2) arrange to deprive yourself of the benefit of the
patent license for this particular work, or (3) arrange, in a manner
consistent with the requirements of this License, to extend the patent
license to downstream recipients.  "Knowingly relying" means you have
actual knowledge that, but for the patent license, your conveying the
covered work in a country, or your recipient's use of the covered work
in a country, would infringe one or more identifiable patents in that
country that you have reason to believe are valid.

  If, pursuant to or in connection with a single transaction or
arrangement, you convey, or propagate by procuring conveyance of, a
covered work, and grant a patent license to some of the parties
receiving the covered work authorizing them to use, propagate, modify
or convey a specific copy of the covered work, then the patent license
you grant is automatically extended to all recipients of the covered
work and works based on it.

  A patent license is "discriminatory" if it does not include within
the scope of its coverage, prohibits the exercise of, or is
conditioned on the non-exercise of one or more of the rights that are
specifically granted under this License.  You may not convey a covered
work if you are a party to an arrangement with a third party that is
in the business of distributing software, under which you make payment
to the third party based on the extent of your activity of conveying
the work, and under which the third party grants, to any of the
parties who would receive the covered work from you, a discriminatory
patent license (a) in connection with copies of the covered work
conveyed by you (or copies made from those copies), or (b) primarily
for and in connection with specific products or compilations that
contain the covered work, unless you entered into that arrangement,
or that patent license was granted, prior to 28 March 2007.

  Nothing in this License shall be construed as excluding or limiting
any implied license or other defenses to infringement that may
otherwise be available to you under applicable patent law.

  12. No Surrender of Others' Freedom.

  If conditions are imposed on you (whether by court order, agreement or
otherwise) that contradict the conditions of this License, they do not
excuse you from the conditions of this License.  If you cannot convey a
covered work so as to satisfy simultaneously your obligations under this
License and any other pertinent obligations, then as a consequence you may
not convey it at all.  For example, if you agree to terms that obligate you
to collect a royalty for further conveying from those to whom you convey
the Program, the only way you could satisfy both those terms and this
License would be to refrain entirely from conveying the Program.
```

(continues on next page)

(continued from previous page)

```
  13. Remote Network Interaction; Use with the GNU General Public License.

  Notwithstanding any other provision of this License, if you modify the
Program, your modified version must prominently offer all users
interacting with it remotely through a computer network (if your version
supports such interaction) an opportunity to receive the Corresponding
Source of your version by providing access to the Corresponding Source
from a network server at no charge, through some standard or customary
means of facilitating copying of software.  This Corresponding Source
shall include the Corresponding Source for any work covered by version 3
of the GNU General Public License that is incorporated pursuant to the
following paragraph.

  Notwithstanding any other provision of this License, you have
permission to link or combine any covered work with a work licensed
under version 3 of the GNU General Public License into a single
combined work, and to convey the resulting work.  The terms of this
License will continue to apply to the part which is the covered work,
but the work with which it is combined will remain governed by version
3 of the GNU General Public License.

  14. Revised Versions of this License.

  The Free Software Foundation may publish revised and/or new versions of
the GNU Affero General Public License from time to time.  Such new versions
will be similar in spirit to the present version, but may differ in detail to
address new problems or concerns.

  Each version is given a distinguishing version number.  If the
Program specifies that a certain numbered version of the GNU Affero General
Public License "or any later version" applies to it, you have the
option of following the terms and conditions either of that numbered
version or of any later version published by the Free Software
Foundation.  If the Program does not specify a version number of the
GNU Affero General Public License, you may choose any version ever published
by the Free Software Foundation.

  If the Program specifies that a proxy can decide which future
versions of the GNU Affero General Public License can be used, that proxy's
public statement of acceptance of a version permanently authorizes you
to choose that version for the Program.

  Later license versions may give you additional or different
permissions.  However, no additional obligations are imposed on any
author or copyright holder as a result of your choosing to follow a
later version.

  15. Disclaimer of Warranty.

  THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY
APPLICABLE LAW.  EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT
HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY
OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO,
THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
PURPOSE.  THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM
IS WITH YOU.  SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF
```

(continues on next page)

(continued from previous page)

```
ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

  16. Limitation of Liability.

  IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING
WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS
THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY
GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE
USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF
DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD
PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS),
EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF
SUCH DAMAGES.

  17. Interpretation of Sections 15 and 16.

  If the disclaimer of warranty and limitation of liability provided
above cannot be given local legal effect according to their terms,
reviewing courts shall apply local law that most closely approximates
an absolute waiver of all civil liability in connection with the
Program, unless a warranty or assumption of liability accompanies a
copy of the Program in return for a fee.
```

PYTHON MODULE INDEX

INDEX

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

P

Q

R

S

T

W

X