lexor Documentation

Release 0.1.2c0

Manuel Lopez

September 30, 2014

Lexor is a document converter implemented in Python.

CHAPTER 1

Basic Usage

1.1 What is Lexor?

Lexor provides a platform where we can specify how a document is parsed, converted and written. It is an expandable Python package which aims to provide functionality to deal with any text file.

1.1.1 Motivation

Lexor started as a simple HTML file parser which later evolved into a markdown parser. The first versions took inspiration on Python-Markdown. However, the need to modify and extend the functionality let us to find Pandoc.

Today, lexor aims to behave similary to Pandoc, in the sense that it converts documents but it does so to meet the users preferences. For instance, we may want to write an HTML file in a minified form, that is, seeing as spaces and new lines do not matter in almost all HTML tags we could write an HTML in a single line, provided that there are no script or other special tags. Or perhaps we want to let Lexor write it in a different style. Lexor attempts to emulate Pandoc in Python. We should note that Pandoc is written in Haskell and although Pandoc already has lots of document converters, Lexor brings potential for Python users to create their own tools for processing files in a simplified manner.

1.2 Installing Lexor

1.2.1 Pip or Manual Installation

The easiest way to install lexor is to use `pip`. If you wish to perform a global installation and you have admin rights then do

```
sudo pip install lexor
```

or to install in some directory under your user account

```
pip install --user lexor
```

Or if you prefer to do do a manual installation then you may do the following from the command line (where x.y is the version number):

```
wget https://pypi.python.org/packages/source/l/lexor/lexor-x.y.tar.gz
tar xvzf lexor-x.y.tar.gz
cd lexor-x.y/
sudo python setup.py install
```

The last command can be replaced by `python setup.py install --user`. See PyPI for all available versions.

1.2.2 Lexor Languages

The basic lexor installation does not provide any parsers, converters or writers. You must install them manually using the `install` lexor command.

```
lexor install <language>
```

To see the available languages see http://jmlopez-rod.github.io/lexor-lang/.

1.3 Getting Started

As a simple example we will use the HTML language:

```
lexor install html
```

Consider the file `example.html`

```
<!DOCTYPE html>
<html>
    <head>
        <title>Example</title>
</head>
    <body>
            <h1>Example</h1>
            <p>
                This is an example

                </p>
        </body>
</html>
```

Now we can rewrite this file into three different versions:

```
lexor example.html html~plain,min,_~ -wn
```

The `-w` option writes the output to a file by appending the specified style. The `-n` suppress the output in the terminal.

The following are the files written:

example.default.html:

```
<!DOCTYPE html>
<html>
<head>
<title>Example</title>
</head>
<body>
<h1>Example</h1>
<p> This is an example </p>
</body>
</html>
```

example.plain.html:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Example</title>
</head>
    <body>
            <h1>Example</h1>
            <p>
                This is an example

                </p>
        </body>
```

example.min.html:

```
<!DOCTYPE html><html><head><title>Example</title></head><body><h1>Example</h1><p> This is an example
```

For more information on how to transform files see the lexor command `to`.

CHAPTER 2

Command Line

2.1 Commands

2.1.1 lexor install

Install a parser, writer or converter style.

CHAPTER 3

Lexor API

3.1 API

This is the Lexor API documentation, autogenerated from the source code.

3.1.1 lexor package

To use lexor as a module you should explore in detail the packages provided with lexor. These packages contain many other functions and information which can help you convert your document in the way you desire.

core The core of lexor defines basic objects such as `Document` and provides the main objects that define the functions provided in this module.

command This module is in charge of providing all the available commands to lexor.

In this module we can find useful functions to quickly parse, convert and write files without first creating any of the main lexor objects.

`lexor.`**`lexor`**(*src*, *search=False*, ***keywords*)

Utility function to parse and convert a file or string specified by *src*. If *search* is `True` then it will attemp to search for *src* in the paths specified by the enviromental variable `$LEXORINPUTS`. The following are all the valid keywords and its defaults that this function accepts:

- parser_style: `'_'`
- parser_lang: `None`
- parser_defaults: `None`,
- convert_style: `'_'`,
- convert_from: `None`,
- convert_to: `'html'`,
- convert_defaults: `None`,
- convert: `'true'`

Returns the converted `Document` object and the log `Document` containing possible warning or error messages.

`lexor.`**`parse`**(*text*, *lang='xml'*, *style='default'*)

Parse the *text* in the language specified by *lang* and return its `Document` form and a `Document` log containing the errors encountered during parsing.

`lexor.`**`read`**(*filename*, *style='default'*, *lang=None*)

Read and parse a file. If *lang* is not specified then the language is assummed from the *filename* extension. Returns the `Document` form and a `Document` log containing the errors encountered during parsing.

`lexor.`**`convert`**(*doc*, *lang=None*, *style='default'*)

Convert the `Document` *doc* to another language in a given style. If the *lang* is not specified then the *doc* is tranformed to the same language as the `Document` using the default style.

`lexor.`**`write`**(*doc*, *filename=None*, *mode='w'*, ***options*)

Write *doc* to a file. To write to the standard output use the default parameters, otherwise provide *filename*. If *filename* is provided you have the option of especifying the mode: `'w'` or `'a'`.

You may also provide a file you may have opened yourself in place of filename so that the writer writes to that file, in this case the *mode* is ignored.

The valid *options* depends on the language the document specifies. See the `DEFAULT` values a particular writer style has to obtain the valid options.

`lexor.`**`init`**(***keywords*)

Every lexor style needs to call the `init` function. These are the valid keywords to initialize a style:

- version: `(major, minor, micro, alpha/beta/rc/final, #)`
- lang
- [to_lang]
- type
- description
- author
- author_email
- [url]
- license
- path: Must be set to `__file__`.

3.1.2 lexor.core package

The core of lexor is divided among the modules in this package.

node Provides the most basic structure to create the document object model (DOM).

elements Here we define the basic structures to handle the information provided in files. Make sure to familiarize yourself with all the objects in this module to be able to write extensions for the `Parser`, `Converter` and `Writer`.

parser The parser module provides the `Parser` and the abstract class `NodeParser` which helps us write derived objects for future languages to parse.

converter The converter module provides the `Converter` and the abstract class `NodeConverter` which helps us copy a `Document` we want to convert to another language.

writer The writer module provides the `Writer` and the abstract class `NodeWriter` which once subclassed help us tell the `Writer` how to write a `Node` to a file object.

lexor.core.parser module

Parser Module

Provides the *Parser* object which defines the basic mechanism for parsing character sequences. This involves using objects derived from the abstract class *NodeParser*.

class `lexor.core.parser.`**`NodeParser`**(*parser*)

Bases: `object`

An object that has two methods: *makeNode* and *close*. The first method is required to be overloaded in derived objects.

`close`(*_*)

This method needs to be overloaded if the node parser returns a *Node* with the *make_node* method.

This method will not get called if *make_node* returned a *Node* inside a *list*. The close function takes as input the *Node* object that *make_node* returned and it should decide if the node can be closed or not. If it is indeed time to close the *Node* then return a list with the position where the *Node* is being closed, otherwise return *None*.

If this method is not overloaded then a *NotImplementedError* exception will be raised.

`make_node`()

This method is required to be overloaded by the derived node parser. It returns *None* if the node parser will not be able to create a node from the current information in the parser. Otherwise it creates a *Node* object and returns it.

When returning a node you have the option of informing the parser if the node is complete or not. For instance, if your node parser creates an Element and it does not have any children to be parsed then return a list containing only the single node. This will tell the parser that the node has been closed and it will not call the *close* method of the node parser. If the *Node* does not have a child, say *ProcessingInstruction*, *RawText*, or *Void* then there is no need to wrap the node in a list.

The *Node* object that this method returns also needs to have the property *pos*. This is a list of two integers stating the line and column number where the node was encountered in the text that is being parsed. This property will be removed by the parser once the parser finishes all processing with the node.

If this method is not overloaded as previously stated then a *NotImplementedError* exception will be raised.

`msg`(*code*, *pos*, *arg=None*, *uri=None*)

Send a message to the parser.

class `lexor.core.parser.`**`Parser`**(*lang='xml'*, *style='default'*, *defaults=None*)

Bases: `object`

To see the languages that it is able to parse see the *lexor.lang* module.

`caret_position`

The index in the text the parser is processing. You may use the attribute access *caret* if performance is an issue.

`cdata`

The character sequence data that was last processed by the *parse* method. You may use the attribute access *text* if performance is an issue.

`compute`(*index*)

Returns a position in the text *[line, column]* given an index. Note: This does not modify anything in the parser. It only gives you the line and column where the caret would be given the index. The same applies as in update. Do not use compute with an index less than the current position of the caret.

copy_pos()
: Returns a copy of the current position.

document
: The parsed document. This is a *Document* or *FragmentedDocument* created by the *parse* method.

language
: The language in which the *Parser* object will parse character sequences.

lexor_log
: The *lexor_log* document. See this document after each call to *parse* to see warnings and errors in the text that was parsed.

load_node_parsers()
: Loads the node parsers. This function is called automatically when *parse* is called only if there was a change in the settings.

msg(*mod_name*, *code*, *pos*, *arg=None*, *uri=None*)
: Provide the name of module issuing the message, the code number, the position of caret and optional arguments and uri. This information gets stored in the log.

parse(*text*, *uri=None*)
: parses the given *text*. To see the results of this method see the *document* and *log* property. If no *uri* is given then *document* will return a *DocumentFragment* node.

parsing_style
: The style in which the *Parser* object will parse the character sequences.

position
: Position of caret in the text in terms of line and column. i.e. returns [line, column]. You may use the attribute access *pos* if performance is an issue.

set(*lang*, *style*, *defaults=None*)
: Set the language and style in one call.

update(*index*)
: Changes the position of the *caret* and updates *pos*. This function assumes that you are moving forward. Do not update to an index which is less than the current position of the caret.

uri
: The Uniform Resource Identifier. This is the name that was given to the text that was last parsed.

lexor.core.converter module

Converter Module

Provides the *Converter* object which defines the basic mechanism for converting the objects defined in *lexor.core.elements*. This involves using objects derived from the abstract class *NodeConverter*.

class lexor.core.converter.**BaseLog**(*converter*)
: Bases: object

A simple class to provide messages to a converter. You must derive an object from this class in the module which will be issuing the messages. For instance:

class Log(BaseLog): pass

After that you can create a new object and use it in a module.

log = Log(converter)

where *converter* is a *Converter* provided to the module. Make sure that the module contains the objects *MSG* and *MSG_EXPLANATION*.

`msg` (*code*, *arg=None*, *uri=None*)
Send a message to the converter.

class `lexor.core.converter.`**`Converter`** (*fromlang='xml'*, *tolang='xml'*, *style='default'*, *defaults=None*)
Bases: `object`

To see the languages available to the *Converter* see the *lexor.lang* module.

`convert` (*doc*, *namespace=False*)
Convert the *Document* doc.

`convert_from`
The language from which the converter will convert.

`convert_to`
The language to which the converter will convert.

`converting_style`
The converter style.

`document`
The parsed document. This is a *Document* or *FragmentedDocument* created by the *convert* method.

`exec_python` (*node*, *id_num*, *parser*, *error=True*)
Executes the contents of the processing instruction. You must provide an id number identifying the processing instruction, the namespace where the execution takes place and a parser that will parse the output provided by the execution. If *error* is True then any errors generated during the execution will be appended to the output of the document.

`lexor_log`
The *lexorlog* document. See this document after each call to *convert* to see warnings and errors.

`match_info` (*fromlang*, *tolang*, *style*, *defaults=None*)
Check to see if the converter main information matches.

`msg` (*mod_name*, *code*, *node*, *arg=None*, *uri=None*)
Provide the name of module issuing the message, the code number, the node with the error, optional arguments and uri. This information gets stored in the log.

`pop` ()
Remove the last document and last log document and return them.

static `remove_node` (*node*)
Removes the node from the current document it is in. Returns the previous sibling is possible, otherwise it returns an empty Text node.

`set` (*fromlang*, *tolang*, *style*, *defaults=None*)
Sets the languages and styles in one call.

`update_log` (*log*, *after=True*)
Append the messages from a log document to the converters log. Note that this removes the children from log.

class `lexor.core.converter.`**`NodeConverter`** (*converter*)
Bases: `object`

A node converter is an object which determines if the node will be copied (default). To avoid copying the node simply declare

copy = False

when deriving a node converter. Note that by default, the children of the node (if any) will be copied and assigned to the parent. To avoid copying the children then set

copy_children = False

classmethod **`end`**(*node*)
This method gets called after all the children have been copied. Make sure to return the node or the node replacement.

`msg`(*code*, *node*, *arg=None*, *uri=None*)
Send a message to the converter.

classmethod **`start`**(*node*)
This method gets called only if *copy* is set to True (default). By overloading this method you have access to the converter and the node. You can thus set extra variables in the converter or modify the node. DO NOT modify any of the parents of the node. If there is a need to modify any of parents of the node then set a variable in the converter to point to the node so that later on in the *convert* function it can be modified.

`lexor.core.converter.`**`echo`**(*node*)
Allows the insertion of Nodes generated within python embeddings.

<?python comment = PI('!–', 'This is a comment') echo(comment) ?>

`lexor.core.converter.`**`get_converter_namespace`**()
Many converters may be defined during the conversion of a document. In some cases we may need to save references to objects in documents. If this is the case, then call this function to obtain the namespace where you can save those references.

`lexor.core.converter.`**`get_current_node`**()
Return the *Document* node containing the python embeddings currently being executed.

`lexor.core.converter.`**`get_lexor_namespace`**()
The execution of python instructions take place in the namespace provided by this function.

`lexor.core.converter.`**`import_module`**(*mod_path*, *mod_name=None*)
Return a module from a path. If no name is provided then the name of the file loaded will be assigned to the name. When using relative paths, it will find the module relative to the file executing the python embedding.

`lexor.core.converter.`**`include`**(*input_file*, ***keywords*)
Inserts a file into the current node.

lexor.core.writer module

Writer Module

Provides the *Writer* object which defines the basic mechanism for writing the objects defined in *lexor.core.elements*. This involves using objects derived from the abstract class *NodeWriter*. See *lexor.core.dev* for more information on how to write objects derived from *NodeWriter* to be able to write *Documents* in the way you desire.

class `lexor.core.writer.`**`DefaultWriter`**(*writer*)
Bases: `lexor.core.writer.NodeWriter`

If the language does not define a NodeWriter for __default__ then the writer will use this default writer.

`end`(*node*)
Write the end of the node as an xml end tag.

`start`(*node*)
Write the start of the node as a xml tag.

class `lexor.core.writer.`**`NodeWriter`**(*writer*)

Bases: `object`

A node writer is an object which writes a node in three steps: *start*, *data/child*, *end*.

classmethod **`child`**(_)

This method gets called for *Elements* that have children. If it gets overwritten then it will not traverse through child nodes unless you return something other than None.

This method by default returns *True* so that the *Writer* can traverse through the child nodes.

`data`(*node*)

This method gets called only by *CharacterData* nodes. This method should be overloaded to write their attribute *data*, otherwise it will write the node's data as it is.

`end`(*node*)

Overload this method to write part of the *Node* object in the last encounter with the *Node*.

`start`(*node*)

Overload this method to write part of the *Node* object in the first encounter with the *Node*.

`write`(*string*, *split=False*)

Writes the string to a file object. The file object is determined by the *Writer* object that initialized this object (*self*).

class `lexor.core.writer.`**`Writer`**(*lang='xml'*, *style='default'*, *defaults=None*)

Bases: `object`

To see the languages in which a *Writer* object is able to write see the *lexor.lang* module.

`close`()

Close the file.

`disable_raw`()

Turn off raw mode.

`disable_wrap`()

Turn off wrapping.

`enable_raw`()

Use this to set the writing in raw mode.

`enable_wrap`()

Use this to set the writing in wrapping mode.

`endl`(*force=True*, *tot=1*, *tail=False*)

Insert a new line character. By setting *force* to False you may omit inserting a new line character if the last character printed was already the new line character.

`filename`

READ-ONLY: The name of the file to which a *Node* object was last written to.

`flush_buffer`(*tail=True*)

Empty the contents of the buffer.

`get_node_writer`(*name*)

Return one of the NodeWriter objects available to the Writer.

`indent`

The indentation at the beginning of each line.

`language`

The language in which the *Writer* writes *Node* objects.

last()
Returns the last written string with the contents of the buffer.

normalize_buffer()
The term normalize means that the length of the buffer will be less than or equal to the wrapping width. Anything that exceeds the limit will be flushed.

raw_enabled()
Determine if raw mode is enabled or not.

set(*lang*, *style*, *defaults=None*)
Set the language and style in one call.

string_buffer
The current string buffer. This is the string that will be printed after its length exceeds the writer's width.

wrap_enabled()
Determine if wrap mode is enabled or not.

write(*node*, *filename=None*, *mode='w'*)
Write node to a file or string. To write to a string use the default parameters, otherwise provide a file name. If filename is provided you have the option of specifying the mode: 'w' or 'a'.

You may also provide a file you may have opened yourself in place of filename so that the writer writes to that file.

Use the __str__ function to retrieve the contents written to a string.

write_str(*string*, *split=False*)
The write function is meant to be used with Node objects. Use this function to write simple strings while the file descriptor is open.

writing_style
The style in which the *Writer* writes a *Node* object.

lexor.core.writer.**find_whitespace**(*line*, *start*, *lim*)
Attempts to find the index of the first whitespace before lim, if its not found, then it looks ahead.

lexor.core.writer.**replace**(*string*, **key_val*)
Replacement of strings done in one pass. Example:

```
>>> replace("a < b && b < c", ('<', '&lt;'), ('&', '&amp;'))
'a &lt; b &amp;&amp; b &lt; c'
```

Source: <http://stackoverflow.com/a/15221068/788553>

lexor.core.selector module

Selector

This module is trying to simulate jquery selectors. If some code looks similar to that of the Sizzle CSS Selector engine it is because the ideas were taken from it.

In short, credit goes to [Sizzle][1] and CSS for the seletor idea.

[1]: http://sizzlejs.com/

class lexor.core.selector.**Selector**(*selector*, *node*, *results=None*)
Bases: object

JQuery like object.

after(**arg*, ***keywords*)
Insert content, specified by the parameter, after each element in the set of matched elements.

: .after(content [,content])

:: content Type: htmlString or Element or Array or jQuery string, Node, array of Node, or Selector object to insert after each element in the set of matched elements.

:: content Type: htmlString or Element or Array or jQuery One or more additional DOM elements, arrays of elements, HTML strings, or jQuery objects to insert after each element in the set of matched elements.

: .after(function(node, index))

:: function(node, index) A function that returns a string, DOM element(s), or Selector object to insert after each element in the set of matched elements. Receives the element in the set and its index position in the set as its arguments.

: .after(..., lang='html', style='default', 'defaults'=None)

:: lang The language in which strings will be parsed in.

:: style The style in which strings will be parsed in.

:: defaults A dictionary with string keywords and values especifying options for the particular style.

append(**arg*, ***keywords*)
Insert content, specified by the parameter, to the end of each element in the set of matched elements.

Should behave similarly as https://api.jquery.com/append/. Major difference is in the function. When passing a function it should take 2 parameters: node, index. Where node will be the current element to which the return value will be appended to.

before(**arg*, ***keywords*)
Insert content, specified by the parameter, before each element in the set of matched elements.

: .before(content [,content])

:: content Type: htmlString or Element or Array or jQuery string, Node, array of Node, or Selector object to insert before each element in the set of matched elements.

:: content Type: htmlString or Element or Array or jQuery One or more additional DOM elements, arrays of elements, HTML strings, or jQuery objects to insert before each element in the set of matched elements.

: .before(function(node, index))

:: function(node, index) A function that returns a string, DOM element(s), or Selector object to insert before each element in the set of matched elements. Receives the element in the set and its index position in the set as its arguments.

: .before(..., lang='html', style='default', 'defaults'=None)

:: lang The language in which strings will be parsed in.

:: style The style in which strings will be parsed in.

:: defaults A dictionary with string keywords and values especifying options for the particular style.

contents()
Get the children of each element in the set of matched elements, including text and comment nodes.

find(*selector*)
Get the descendants of each element in the current set of matched elements, filtered by a selector.

prepend(**arg*, ***keywords*)
Insert content, specified by the parameter, to the beginning of each element in the setof matched elements.

Should behave similarly as https://api.jquery.com/append/. Major difference is in the function. When passing a function it should take 2 parameters: node, index. Where node will be the current element to which the return value will be appended to.

`lexor.core.selector.`**`clone_obj`**`(`*obj*, *parser*`)`
Utility function to create deep copies of objects used for the Selector object. A parser should be given in case the object is a string.

`lexor.core.selector.`**`get_date`**`()`
Obtain an integer representation of the date.

`lexor.core.selector.`**`mark_function`**`(`*fnc*`)`
Mark a function for special use by Sizzle.

`lexor.core.selector.`**`select`**`(`*selector*, *context*, *results*, *seed*`)`
A low-level selection function that works with Sizzle's compiled selector functions

@param {String|Function} selector A selector or a pre-compiled selector function built with Sizzle.compile

@param {Element} context @param {Array} [results] @param {Array} [seed] A set of elements to match against

`lexor.core.selector.`**`sizzle`**`(`*selector*, *context*, *results=None*, *seed=None*`)`
Function shamelessly borrowed and partially translated to python from http://sizzlejs.com/.

`lexor.core.selector.`**`tokenize`**`(`*selector*, *parse_only=False*`)`
Tokenize...

3.1.3 lexor.core.node module

This module defines the basic object of the document object model (DOM).

class `lexor.core.node.`**`Node`**
Bases: `object`

Primary datatype for the entire Document Object Model.

`__init__``()`
Initializes all data descriptors to `None`. Each descriptor has an associated *READ-ONLY* property. Read the comment on each property to see what each descriptor represents.

`node_name`

Read-Only Property

The name of this node. Its value depends on the node type. This property is associated with the attribute `name`.

`owner_document`

Read-Only Property

The `Document` in which this node resides. This property is associated with the attribute `owner`.

`parent_node`

Read-Only Property

The parent of this node. If the node has been just created or removed from a `Document` then this property is `None`. This property is associated with the attribute `parent`.

node_index

Read-Only Property

The number of preceding siblings.

```
>>> x is x.parent_node[x.node_index]
True
```

This property is associated with the attribute `index`.

node_level

Read-Only Property

The nodes level of containtment in a `Document` object.

This property is associated with the attribute `level`.

element_index

Read-Only Property

The number of preceding element siblings.

previous_sibling

Read-Only Property

The node immediately preceding this node. If this property is not *None* then

```
>>> x.previous_sibling <==> x.parent_node[x.node_index - 1]
```

This property is associated with the attribute `prev`.

next_sibling

Read-Only Property

The node immediately following this node. If this property is not *None* then

```
>>> x.next_sibling <==> x.parent_node[x.node_index + 1]
```

This property is associated with the attribute `next`.

previous_element

Read-Only Property

The last sibling `Element` preceding this node.

`next_element`

Read-Only Property

The sibling `Element` after this node.

`remove_children`()

Remove all the child nodes.

`__repr__`()

```
>>> x.__repr__() == repr(x)
True
```

`__str__`()

```
>>> x.__str__() == str(x)
True
```

`insert_before`(*index*, *new_child*)

Inserts *new_child* to the list of children just before the child specified by *index*.

`extend_before`(*index*, *new_children*)

Inserts the contents of an iterable containing nodes just before the child specified by *index*. The following are equivalent:

```
>>> while doc: node.parent.insert_before(index, doc[0])
```

```
>>> node.extend_before(index, doc)
```

The second form, however, has a more efficient reindexing method.

`append_child`(*new_child*)

Adds the node *new_child* to the end of the list of children of this node. If the node is a `DocumentFragment` then it appends its child nodes. Returns the calling node.

`extend_children`(*new_children*)

Extend the list of children by appending children from an iterable containing nodes.

`append_after`(*new_child*)

Place *new_child* after the node.

`append_nodes_after`(*new_children*)

Place *new_children* after the node.

`prepend_before`(*new_child*)

Place *new_child* before the node.

`prepend_nodes_before`(*new_children*)

Place *new_children* before the node.

`normalize`()

Removes empty `Text` nodes, and joins adjacent `Text` nodes.

`__len__`()

Return the number of child nodes.

```
>>> x.__len__() == len(x)
True
```

__getitem__(*i*)

Return the *i*-th child of this node.

```
>>> x.__getitem__(i) <==> x[i]
>>> x.__getitem__(slice(i, j)) <==> x[i:j]
>>> x.__getitem__(slice(i, j, dt)) <==> x[i:j:dt]
```

When using a slice, the `__getitem__` function will return a list with references to the requested nodes.

__delitem__(*index*)

Delete child nodes.

```
>>> x.__delitem__(index) <==> del x[index]
>>> x.__delitem__(slice(i, j)) <==> del x[i:j]
>>> x.__delitem__(slice(i, j, dt)) <==> del x[i:j:dt]
```

__setitem__(*index*, *node*)

Replace child nodes.

```
>>> x.__setitem__(index) = node <==> x[index] = node
>>> x.__setitem__(slice(i, j)) = dfrag <==> x[i:j] = dfrag
>>> x.__setitem__(slice(i, j, dt)) = dfrag <==> x[i:j:dt] = dfrag
```

When using slices the nodes to be assigned to the indices need to be contained in a `DocumentFragment` node. This function does not support insertion as the regular slice for list does. To insert use a node use `insert_before()` or `append_after()`.

get_nodes_by_name(*name*)

Return a `list` of child nodes that have the given *name*.

set_parent(*parent*, *index*)

Helper Method

Modifies the parent node and takes care of the child node levels.

disconnect()

Helper Method

Reset its attributes.

set_prev(*node*)

Helper Method

Sets the `prev` attribute.

set_next(*node*)

Helper Method

Sets the `next` attribute.

increase_child_level()

Helper Method

Sets the level of the child nodes.

append_child_node(*new_child*)

Helper Method

Use this method to insert a node at a the end of the child list. See `append_child()` and `extend_children()` to see this method in action.

insert_node_before(*index*, *new_child*)

Helper Method

Insert a *new_child* at a given *index*. See `insert_before()` and `extend_before()` to see this method in action.

3.1.4 lexor.core.elements module

This module defines the elements of the document object model (DOM). This implementation follows most of the recommendations of w3.

Inheritance Tree

```
lexor.core.node.Node (__builtin__.object)
    CharacterData
        Text
        ProcessingInstruction
        Comment
        CData
        Entity
        DocumentType
    Element
        RawText (Element, CharacterData)
        Void
        Document
            DocumentFragment
```

class lexor.core.elements.**CharacterData**(*text=''*)

Bases: `lexor.core.node.Node`

A simple interface to deal with strings.

__init__(*text=''*)
Set the data property to the value of *text* and set its name to `'#character-data'`.

node_value
Return or set the value of the node. This property is a wrapper for the `data` attribute.

class `lexor.core.elements.`**Text**(*text=''*)
Bases: `lexor.core.elements.CharacterData`

A node to represent a string object.

__init__(*text=''*)
Call its base constructor and set its name to `'#text'`.

clone_node(*_=True*)
Return a new `Text` node with the same data content.

class `lexor.core.elements.`**ProcessingInstruction**(*target, data=''*)
Bases: `lexor.core.elements.CharacterData`

Represents a "processing instruction", used to keep processor-specific information in the text of the document.

__init__(*target, data=''*)
Create a *Text* node with its *data* set to data.

target
The target of this processing instruction.

clone_node(*_=True*)
Returns a new PI with the same data content.

class `lexor.core.elements.`**Comment**(*data=''*)
Bases: `lexor.core.elements.CharacterData`

A node to store comments.

__init__(*data=''*)
Create a comment node.

comment_type
Type of comment. This property is meant to help with documents that support different styles of comments.

clone_node(*_=True*)
Returns a new comment with the same data content.

class `lexor.core.elements.`**CData**(*data=''*)
Bases: `lexor.core.elements.CharacterData`

Although this node has been deprecated from the DOM, it seems that xml still uses it.

__init__(*data=''*)
Create a CDATA node and set the node name to `'#cdata-section'`.

clone_node(*_=True*)
Returns a new `CData` node with the same data content.

class `lexor.core.elements.`**Entity**(*text=''*)
Bases: `lexor.core.elements.CharacterData`

From merriam-webster definition:

- *Something that exists by itself.*
- *Something that is separate from other things.*

This node acts in the same way as a `Text` node but it has one main difference. The data it contains should contain no white spaces. This node should be reserved for special characters or words that have different meanings across different languages. For instance in HTML you have the `&amp;` to represent `&`. In LaTeX you have to type `\$` to represent `$`. Using this node will help you handle these Entities hopefully more efficiently than simply finding and replacing them in a Text node.

`__init__`(*text=''*)
Create an `Entity` node and set the node name to `#entity`.

`clone_node`(*_=True*)
Returns a new `Entity` with the same data content.

class `lexor.core.elements.`**`DocumentType`**(*data=''*)
Bases: `lexor.core.elements.CharacterData`

A node to store the doctype declaration. This node will not follow the specifications at this point (May 30, 2013). It will simply recieve the string in between `<!doctype` and `>`.

Specs: http://www.w3.org/TR/2012/WD-dom-20121206/#documenttype

`__init__`(*data=''*)
Create a `DocumentType` node and set its name to `#doctype`.

`clone_node`(*_=True*)
Returns a new doctype with the same data content.

class `lexor.core.elements.`**`Element`**(*name, data=None*)
Bases: `lexor.core.node.Node`

Node object configured to have child Nodes and attributes.

`__init__`(*name, data=None*)
The parameter `data` should be a `dict` object. The element will use the keys and values to populate its attributes. You may modify the elements internal dictionary. However, this may unintentially overwrite the attributes defined by the `__setitem__` method. If you wish to add another attribute to the `Element` object use the convention of adding an underscore at the end of the attribute. i.e

```
>>> strong = Element('strong')
>>> strong.message_ = 'An internal message'
>>> strong['message'] = 'Attribute message'
```

`__call__`(*selector*)
Return a `lexor.core.selector.Selector` object.

`update_attributes`(*node*)
Copies the attributes of the input node into the calling node.

`__getitem__`(*k*)
Return the *k*-th child of this node if *k* is an integer. Otherwise return the attribute of name with value of *k*.

```
>>> x.__getitem__(k) is x[k]
True
```

`get`(*k, val=''*)
Return the attribute of name with value of *k*.

`__setitem__`(*k, val*)
Overloaded array operator. Appends or modifies an attribute. See its base method `lexor.core.node.Node.__setitem__()` for documentation on when *val* is not string.

```
>>> x.__setitem__(attname) = 'att' <==> x[attname] = 'att'
```

`__delitem__`(*k*)

Remove a child or attribute.

```
>>> x.__delitem__(k) <==> del x[k]
```

`__contains__`(*obj*)

Return `True` if *obj* is a node and it is a child of this element or if *obj* is an attribute of this element. Return `False` otherwise.

```
>>> x.__contains__(obj) == obj in x
True
```

`contains`(*obj*)

Unlike `__contains__`, this method returns `True` if *obj* is any of the desendents of the node.

`__iter__`()

Iterate over the element attributes names.

```
>>> for attribute_name in node: ...
```

`attlen`

The number of attributes.

`attributes`

Return a list of the attribute names in the element.

`values`

Return a list of the attribute values in the Element.

`attribute`(*index*)

Return the name of the attribute at the specified index.

`attr`(*index*)

Return the value of the attribute at the specified index.

`items`()

return all the items.

`update`(*dict_*)

update with the values of *dict_*. useful when the element is empty and you created an Attr object. then just update the values.

`rename`(*old_name*, *new_name*)

Renames an attribute.

```
>>> from lexor.core.elements import Element
>>> node = Element('div')
>>> node['att1'] = 'val1'
>>> node
div[0x10a090750 att1="val1"]:
>>> node.rename('att1', 'new-att-name')
>>> node
div[0x10a090750 new-att-name="val1"]:
```

`clone_node`(*deep=False*, *normalize=True*)

Returns a new node. When deep is True, it will clone also clone all the child nodes.

`get_elements_by_class_name`(*classname*)

Return a list of all child elements which have all of the given class names.

children (*children=None*, ***keywords*)
Set the elements children by providing a list of nodes or a string. If using a string then you may provide any of the following keywords to dictate how to parse and convert:

- parser_style: `'_'`
- parser_lang: `'html`
- parser_defaults: `None`,
- convert_style: `'_'`,
- convert_from: `None`,
- convert_to: `'html'`,
- convert_defaults: `None`,
- convert: `'false'`

If no children are provided then it returns a string of the children written in plain html. To change this behavior provide the following keywords:

- writer_style: `'plain'`
- writer_lang: `'html`

Important: This requires the installation of lexor styles.

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

class `lexor.core.elements.`**RawText** (*name*, *data='‘*, *att=None*)
Bases: `lexor.core.elements.Element`, `lexor.core.elements.CharacterData`

A few elements do not have children, instead they have data. Such elements exist in HTML: `script`, `title` among others.

__init__ (*name*, *data='‘*, *att=None*)
You may provide *att* as a `dict` object.

clone_node (*deep=True*, *normalize=True*)
Returns a new `RawText` element

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

class `lexor.core.elements.`**Void** (*name*, *att=None*)
Bases: `lexor.core.elements.Element`

An element with no children.

__init__ (*name*, *att=None*)
You may provide *att* as a *dict* object.

clone_node (*_=True*, *normalize=True*)
Returns a new `Void` element.

class `lexor.core.elements.`**Document** (*lang='xml'*, *style='default'*)
Bases: `lexor.core.elements.Element`

Contains information about the document that it holds.

__init__ (*lang='xml'*, *style='default'*)
Creates a new document object and sets its name to `#document`.

clone_node (*deep=False*, *normalize=True*)
Returns a new Document. Note: it does not copy the default values.

language
The current document's language. This property is used by the writer to determine how to write the document.

This property is a wrapper for the `lang` attribute.

writing_style
The current document's style. This property is used by the writer to determine how to write the document.

This property is a wrapper for the `style` attribute.

uri
The Uniform Resource Identifier. This property may become useful if the document represents a file. This property should be set by the a `Parser` object telling us the location of the file that it parsed into the Document object.

static **create_element** (*tagname*, *data=None*)
Utility function to avoid having to import `lexor.core.elements` module. Returns an element object.

get_element_by_id (*element_id*)
Return the first element, in tree order, within the document whose ID is element_id, or None if there is none.

class `lexor.core.elements.`**DocumentFragment** (*lang='xml'*, *style='default'*)
Bases: `lexor.core.elements.Document`

Takes in an element and "steals" its children. This element should only be used as a temporary container. Note that the `__str__` method may not yield the expected results since all the function will do is use the `__str__` method in each of its children. First assign this object to an actual Document.

append_child (*new_child*)
Adds the node new_child to the end of the list of children of this node. The children contained in a `DocumentFragment` only have a parent (the `DocumentFragment`). As opposed as `lexor.core.node.Node.append_child()` which also takes care of the `prev` and `next` attributes.

__repr__ ()

```
>>> x.__repr__() <==> repr(x)
```

__str__ ()

```
>>> x.__str__() <==> str(x)
```

3.1.5 lexor.command package

Submodules

lexor.command.config module

Config

This module is in charge of providing all the necessary settings to the rest of the modules in lexor.

class `lexor.command.config.`**`ConfigDispAction`**(*option_strings*, *dest*, *nargs=None*, *const=None*, *default=None*, *type=None*, *choices=None*, *required=False*, *help=None*, *metavar=None*)

Bases: `argparse.Action`

Derived argparse Action class to use when displaying the configuration file and location.

`lexor.command.config.`**`add_parser`**(*subp*, *fclass*)

Add a parser to the main subparser.

`lexor.command.config.`**`get_cfg`**(*names*, *defaults=None*)

Obtain settings from the configuration file.

`lexor.command.config.`**`read_config`**()

Read a configuration file.

`lexor.command.config.`**`run`**()

Run command.

`lexor.command.config.`**`set_style_cfg`**(*obj*, *name*, *defaults*)

Given an obj, this can be a Parser, Converter or Writer. It sets the attribute defaults to the specified defaults in the configuration file or by the user by overwriting values in the parameter defaults.

`lexor.command.config.`**`update_single`**(*cfg*, *name*, *defaults=None*)

Helper function for get_cfg.

`lexor.command.config.`**`value_completer`**(***_*)

value completer.

`lexor.command.config.`**`var_completer`**(***_*)

var completer.

`lexor.command.config.`**`write_config`**(*cfg_file*)

Write the configuration file.

lexor.command.defaults module

Defaults

Print the default values for each command.

`lexor.command.defaults.`**`add_parser`**(*subp*, *fclass*)

Add a parser to the main subparser.

`lexor.command.defaults.`**`name_completer`**(***_*)

var completer.

`lexor.command.defaults.`**`run`**()

Run command.

lexor.command.develop module

Develop

Routine to append a path to the develop section in the configuration file.

`lexor.command.develop.`**`add_parser`**(*subp*, *fclass*)

Add a parser to the main subparser.

`lexor.command.develop.`**`run`**()

Append the path to the develop section in the configuration file.

lexor.command.dist module

Distribute

Package a style along with auxiliary and test files.

`lexor.command.dist.`**`add_parser`**`(`*subp*, *fclass*`)`
  Add a parser to the main subparser.

`lexor.command.dist.`**`run`**`()`
  Run the command.

`lexor.command.dist.`**`style_completer`**`(`*parsed_args*, ***_*`)`
  Return a list of valid files to edit.

lexor.command.document module

Document

Routine to create an xml file with the documentation of a lexor style.

`lexor.command.document.`**`add_parser`**`(`*subp*, *fclass*`)`
  Add a parser to the main subparser.

`lexor.command.document.`**`append_main`**`(`*doc*, *mod*`)`
  Append the main module. Return a dictionary containing all the modules used in the style.

`lexor.command.document.`**`check_filename`**`(`*arg*`)`
  Check if the inputfile exists.

`lexor.command.document.`**`full_class_name`**`(`*cls*`)`
  Obtain the full class name.

`lexor.command.document.`**`get_class_node`**`(`*cls*`)`
  Return a class node. Assumes that cls is a valid class.

`lexor.command.document.`**`get_defaults_node`**`(`*obj*`)`
  Obtain defaults node.

`lexor.command.document.`**`get_function_node`**`(`*func*`)`
  Return a function node.

`lexor.command.document.`**`get_info_node`**`(`*info*`)`
  Generate the info node.

`lexor.command.document.`**`get_mapping_node`**`(`*mapping*`)`
  Generate the mapping node.

`lexor.command.document.`**`get_member_node`**`(`*member*`)`
  Return a property node.

`lexor.command.document.`**`get_property_node`**`(`*prop*`)`
  Return a property node.

`lexor.command.document.`**`make_module_node`**`(`*mod*, *name=None*`)`
  Create a module node documenation.

`lexor.command.document.`**`run`**`()`
  Run the command.

`lexor.command.document.`**`separate_objects`**`(`*mod*, *remove=None*`)`
  Given a module, it separates the objects into a dictionary.

`lexor.command.document.`**`xml_style`**`(`*lang_str*`)`
Parses a style string.

lexor.command.edit module

Edit

Module to open files with an editor.

`lexor.command.edit.`**`add_parser`**`(`*subp*, *fclass*`)`
Add a parser to the main subparser.

`lexor.command.edit.`**`run`**`()`
Run the edit command.

`lexor.command.edit.`**`valid_files`**`(`*parsed_args*, **_`)`
Return a list of valid files to edit.

lexor.command.install module

Install

Routine to install a parser/writer/converter style.

`lexor.command.install.`**`add_parser`**`(`*subp*, *fclass*`)`
Add a parser to the main subparser.

`lexor.command.install.`**`download_file`**`(`*url*, *base='.'*`)`
Download a file.

`lexor.command.install.`**`install_style`**`(`*style*, *install_dir*`)`
Install a given style to the install_dir path.

`lexor.command.install.`**`run`**`()`
Run the command.

`lexor.command.install.`**`unzip_file`**`(`*local_name*`)`
Extract the contents of a zip file.

lexor.command.lang module

Language

This module provides functions to load the different languages parsers, writers and converters.

## Constants

LEXOR_PATH: The paths where lexor looks for the parsing, writing and converting styles.

`lexor.command.lang.`**`add_parser`**`(`*subp*, *fclass*`)`
Add a parser to the main subparser.

`lexor.command.lang.`**`get_style_module`**`(`*type_*, *lang*, *style*, *to_lang=None*`)`
Return a parsing/writing/converting module.

`lexor.command.lang.`**`load_aux`**`(`*info*`)`
Wrapper around load_mod for easy use when developing styles. The only parameter is the dictionary *INFO* that needs to exist with every style. *INFO* is returned by the init function in the lexor module.

`lexor.command.lang.`**`load_mod`**`(`*modbase*, *dirpath*`)`
Return a dictionary containing the modules located in *dirpath*. The name *modbase* must be provided so that each module may have a unique identifying name. The result will be a dictionary of modules. Each of the modules will have the name "modbase_modname" where modname is a module in the directory.

`lexor.command.lang.`**`load_rel`**`(`*path*, *module*`)`
Load relative to a path. If path is the name of a file the filename will be dropped.

`lexor.command.lang.`**`map_explanations`**`(`*mod*, *exp*`)`
Helper function to create a map of msg codes to explanations in the lexor language modules.

`lexor.command.lang.`**`run`**`()`
Run the command.

lexor.command.paste module

Paste

Routine to append paste templates.

`lexor.command.paste.`**`add_parser`**`(`*subp*, *fclass*`)`
Add a parser to the main subparser.

`lexor.command.paste.`**`lang_completer`**`(`***_*`)`
Return the meta var.

`lexor.command.paste.`**`make_auxilary`**`(`*base*, *type_*, *fmt*, *aux_type='‘*`)`
Creates a new node parser module.

`lexor.command.paste.`**`make_style`**`(`*base*, *type_*, *fmt*`)`
Creates a new style file.

`lexor.command.paste.`**`run`**`()`
Run the command.

`lexor.command.paste.`**`style_completer`**`(`***_*`)`
Return the meta var.

lexor.command.test module

Lexor Test

This module contains various test for lexor.

`lexor.command.test.`**`add_parser`**`(`*subp*, *fclass*`)`
Add a parser to the main subparser.

`lexor.command.test.`**`compare_with`**`(`*str_obj*, *expected*`)`
Calls **nose.eq_** to compare the strings and prints a custom message.

`lexor.command.test.`**`find_failed`**`(`*tests*, *lang*, *style*`)`
Run the tests and return a list of the tests that fail.

`lexor.command.test.`**`nose_msg_explanations`**`(`*lang*, *type_*, *style*, *name*`)`
Gather the MSG_EXPLANATION list and run the tests it contains.

`lexor.command.test.`**`parse_convert_write`**`(`*callerfile*, *in_*, *out_*, *style*, *tolang*`)`
Provide the filename as the input and the style you wish to compare it against.

`lexor.command.test.`**`parse_msg`**`(`*msg*`)`
Obtain the tests embedded inside the messages declared in a style. The format of the messages is as follows:

> <tab>[A-Z][0-9]*: <msg>

or

> <tab>([A-Z][0-9]*|Okay): <tab><tab>msg ... <tab><tab>msg continues <tab>([A-Z][0-9]*|Okay): msg

Where <tab> consists of 4 whitespaces. This function returns the message without the tests and a list of tuples of the form *(code, msg)* along with the message

`lexor.command.test.`**`parse_write`**(*callerfile*, *in_*, *out_*, *style*, *lang*)
: Provide the filename as the input and the style you wish to compare it against.

`lexor.command.test.`**`print_log`**(*node*)
: Display the error obtained from parsing.

`lexor.command.test.`**`run`**()
: Run command.

`lexor.command.test.`**`run_develop`**(*param*, *cfg*, *verbose*)
: Run develop tests.

`lexor.command.test.`**`run_installed`**(*param*, *cfg*, *verbose*)
: Run installed tests.

lexor.command.to module

to

Execute lexor by transforming a file “to” another language.

`lexor.command.to.`**`add_parser`**(*subp*, *fclass*)
: Add a parser to the main subparser.

`lexor.command.to.`**`convert_and_write`**(*f_name*, *parser*, *in_lang*, *log*, *arg*)
: Auxiliary function to reduce the number of branches in run.

`lexor.command.to.`**`get_input`**(*input_file*, *cfg*, *default='_'*)
: Returns the text to be parsed along with the name assigned to that text. The last output is the extension of the file.

`lexor.command.to.`**`input_language`**(*tolang*)
: Parses the tolang argument.

`lexor.command.to.`**`language_style`**(*lang_str*)
: Parses a language string. In particular, the options –from and –log.

`lexor.command.to.`**`parse_styles`**(*lang_str*)
: Parses a language string. In particular, the options –from and –log.

`lexor.command.to.`**`run`**()
: Run the command.

`lexor.command.to.`**`run_converter`**(*param*)
: Auxiliary function for convert and write.

`lexor.command.to.`**`run_writer`**(*param*)
: Auxiliary function for convert and write.

`lexor.command.to.`**`split_at`**(*delimiter*, *text*, *opens='[<(‘*, *closes=']>)'*, *quotes='"\'‘*)
: Custom function to split at commas. Taken from stackoverflow http://stackoverflow.com/a/20599372/788553

`lexor.command.to.`**`style_parameters`**`(`*style*`)`
Parsers a style name along with its parameters.

`lexor.command.to.`**`write_document`**`(`*writer*, *doc*, *fname*, *arg*`)`
Auxiliary function for convert_and_write.

`lexor.command.to.`**`write_log`**`(`*writer*, *log*, *quiet*`)`
Write the log file to stderr.

Module contents

Command

Collection of functions to create lexor's command line utility.

`lexor.command.`**`date`**`(`*short=False*`)`
Return the current date as a string.

`lexor.command.`**`error`**`(`*msg*`)`
Print a message to the standard error stream and exit.

`lexor.command.`**`exec_cmd`**`(`*cmd*, *verbose=False*`)`
Run a subprocess and return its output and errors.

`lexor.command.`**`import_mod`**`(`*name*`)`
Return a module by string.

`lexor.command.`**`warn`**`(`*msg*`)`
Print a message to the standard error

CHAPTER 4

Indices and tables

- *genindex*
- *modindex*
- *search*

This version of the documentation was built September 30, 2014.