# rst2hml5 Documentation

***Release 1.8.0***

**André Felipe Dias**

June 06, 2016

Contents

*rst2html5* generates (X)HTML5 documents from standalone reStructuredText sources. It is a complete rewrite of the docutils' *rst2html* and uses new HTML5 constructs such as `<section>` and `<aside>`.

CHAPTER 1

# rst2html5

*rst2html5* generates (X)HTML5 documents from standalone reStructuredText sources. It is a complete rewrite of the docutils' *rst2html* and uses new HTML5 constructs such as `<section>` and `<aside>`.

## 1.1 Installation

```
$ pip install rst2html5
```

## 1.2 Usage

```
$ rst2html5 [options] SOURCE
```

Options:

**--no-indent** Don't indent output

**--stylesheet=<URL or path>** Specify a stylesheet URL to be included. (This option can be used multiple times)

**--script=<URL or path>** Specify a script URL to be included. (This option can be used multiple times)

**--script-defer=<URL or path>** Specify a script URL with a defer attribute to be included in the output HTML file. (This option can be used multiple times)

**--script-async=<URL or path>** Specify a script URL with a async attribute to be included in the output HTML file. (This option can be used multiple times)

**--html-tag-attr=<attribute>** Specify a html tag attribute. (This option can be used multiple times)

**--template=<filename or text>** Specify a filename or text to be used as the HTML5 output template. The template must have the {head} and {body} placeholders. The "<html{html_attr}>" placeholder is recommended.

**--define=<identifier>** Define a case insensitive identifier to be used with ifdef and ifndef directives. There is no value associated with an identifier. (This option can be used multiple times)

### 1.2.1 Example

Consider the following rst snippet:

```
Title
=====

Some text and a target to `Title 2`_. **strong emphasis**:

* item 1
* item 2

Title 2
=======

.. parsed-literal::

    Inline markup is supported, e.g. *emphasis*, **strong**, ``literal
    text``,
    _`hyperlink targets`, and `references <http://www.python.org/>`_
```

The html5 produced is clean and tidy:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
</head>
<body>
    <section id="title">
        <h1>Title</h1>
        <p>Some text and a target to <a href="#title-2">Title 2</a>. <strong>strong emphasis</strong>
        <ul>
            <li>item 1</li>
            <li>item 2</li>
        </ul>
    </section>
    <section id="title-2">
        <h1>Title 2</h1>
        <pre>Inline markup is supported, e.g. <em>emphasis</em>, <strong>strong</strong>, <code>liter
text</code>,
<a id="hyperlink-targets">hyperlink targets</a>, and <a href="http://www.python.org/">references</a>
    </section>
</body>
</html>
```

### 1.2.2 Stylesheets and Scripts

No stylesheets or classes are spread over the html5 by default. However stylesheets and javascripts URLs or paths can be included through `stylesheet` and `script` options:

```
$ rst2html5 example.rst \
--stylesheet css/default.css \
--stylesheet css/special.css \
--script https://ajax.googleapis.com/ajax/libs/jquery/1.7.2/jquery.min.js
```

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <link href="css/default.css" rel="stylesheet" />
    <link href="css/special.css" rel="stylesheet" />
    <script src="https://ajax.googleapis.com/ajax/libs/jquery/1.7.2/jquery.min.js"></script>
...
```

Additional scripts can be included in the result using options `--script`, `--script-defer` or `--script-async`:

```
$ rst2html5 example.rst \
    --script js/test1.js \
    --script-defer js/test2.js \
    --script-async js/test3.js
```

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <script src="js/test1.js"></script>
    <script src="js/test2.js" defer="defer"></script>
    <script src="js/test3.js" async="async"></script>
...
```

Html tag attributes can be included through `html-tag-attr` option:

```
$ rst2html5 --html-tag-attr `lang=''pt-BR''' example.rst
```

```
<!DOCTYPE html>
<html lang="pt-BR">
...
```

### 1.2.3 Templates

Custom html5 template via the `--template` option. Example:

```
$ template='<!DOCTYPE html>
<html{html_attr}>
<head>{head}    <!-- custom links and scripts -->
    <link href=''css/default.css'' rel=''stylesheet'' />
    <link href=''css/pygments.css'' rel=''stylesheet'' />
    <script src=''http://code.jquery.com/jquery-latest.min.js''></script>
</head>
<body>{body}</body>
</html>'

$ echo `one line' > example.rst

$ rst2html5 --template ``$template'' example.rst
```

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <!-- custom links and scripts -->
```

```
    <link href="css/default.css" rel="stylesheet" />
    <link href="css/pygments.css" rel="stylesheet" />
    <script src="http://code.jquery.com/jquery-latest.min.js"></script>
</head>
<body>
    <p>one line</p>
</body>
</html>
```

### 1.2.4 New Directives

`rst2html5` provides some new directives: `define`, `undef`, `ifdef` and `ifndef`, similar to those used in C++. They allow to conditionally include (or not) some rst snippets:

```
.. ifdef:: x

    this line will be included if 'x' was previously defined
```

In case of you check two or more identifiers, there must be an operator (`[and | or]`) defined:

```
.. ifdef:: x y z
    :operator: or

    This line will be included only if 'x', 'y' or 'z' is defined.
```

## 1.3 Links

- Documentation
- Project page at BitBucket

CHAPTER 2

# Changelog

Here you can see the full list of changes between each rst2html5 releases.

- 1.8 - 2016-06-04
  - New directives `define`, `undef`, `ifdef` and `ifndef` to conditionally include (or not) a rst snippet.
- 1.7.5 - 2015-05-14
  - fixes the stripping of leading whitespace from the highlighted code
- 1.7.4 - 2015-04-09
  - fixes deleted blank lines in <table><pre> during Genshi rendering
  - Testing does not depend on ordered tag attributes anymore
- 1.7.3 - 2015-04-04
  - fix some imports
  - Sphinx dependency removed
- 1.7.2 - 2015-03-31
  - Another small bugfix related to imports
- 1.7.1 - 2015-03-31
  - Fix 1.7 package installation. `requirements.txt` was missing
- 1.7 - 2015-03-31
  - Small bufix in setup.py
  - LICENSE file added to the project
  - Sublists are not under <blockquote> anymore
  - Never a <p> as a <li> first child
  - New CodeBlock directive merges docutils and sphinx CodeBlock directives
  - Generated codeblock cleaned up to a more HTML5 style: <pre data-language="...">...</pre>
- 1.6 - 2015-03-09
  - code-block's `:class:` value should go to <pre class="value"> instead of <pre><code class="value">
  - Fix problem with no files uploaded to Pypi in 1.5 version
- 1.5 - 2015-23-02

  - rst2html5 generates html5 comments
  - A few documentation improvementss
- 1.4 - 2014-09-21
  - Improved packaging
  - Using tox for testing management
  - Improved compatibility to Python3
  - Respect initial_header_level_setting
  - Container and compound directives map to div
  - rst2html5 now process field_list nodes
  - Additional tests
  - Multiple-time options should be specified multiple times, not with commas
  - Metatags are declared at the top of head
  - Only one link to mathjax script is generated
- 1.3 - 2014-04-21
  - Fixes #16 | New –template option
  - runtests.sh without parameter should keep current virtualenv
- 1.2 - 2014-02-16
  - Fix doc version
- 1.1 - 2014-02-16
  - rst2html5 works with docutils 0.11 and Genshi 0.7
- 1.0 - 2013-06-17
  - Documentation improvement
  - Added html-tag-attr, script-defer and script-async options
  - Dropped option-limit option
  - Fix bug with caption generation within table
  - Footer should be at the bottom of the page
  - Indent raw html
  - field-limit and option-limit are set to 0 (no limit)
- 0.10 - 2013-05-11
  - Support docutils 0.10
  - Force syntax_hightlight to ‘short’
  - Conforming to PEP8 and PyFlakes
  - Testing structure simplified
  - rst2html5.py refactored
  - Some bugfixes
- 0.9 - 2012-08-03

- First public preview release

CHAPTER 3

# License

rst2html5 is distributed under the MIT License (MIT).

## 3.1 rst2html5 License

CHAPTER 4

# Contributing to rst2html5

Contributions are welcome! So don't be afraid to contribute with anything that you think will be helpful. Help with maintaining the English documentation are particularly appreciated.

The bugtracker, wiki and Mercurial repository can be found at the rst2html5 projects's page on BitBucket.

## 4.1 How to contribute

Please, follow the procedure:

1. Check for the open issues or open a new issue on the BitBucket issue tracker to start a discussion about a feature or a bug.
2. Fork the rst2html5 project on BitBucket and start making your modifications.
3. Send a pull request.

## 4.2 Installing OS Packages

You will need:

1. pip. A tool for installing and managing Python packages.
2. virtualenvwrapper. A set of extensions to Ian Bicking's virtualenv tool. Using a virtual environment will make the installation easier, and will help to avoid clutter in your system-wide libraries.
3. Mercurial. Version control used by rst2html5 project.

```
sudo apt-get install python-dev python-pip mercurial
sudo pip install virtualenvwrapper
```

Add these two lines to `~/.bashrc`:

```
export WORKON_HOME=$HOME/.virtualenvs
source /usr/local/bin/virtualenvwrapper.sh
```

## 4.3 Project Setup

1. Clone the repository:

```
$ hg clone http://www.bitbucket.org/andre_felipe_dias/rst2html5
$ cd rst2html5
```

2. Make a new virtual enviroment for development:

```
$ mkvirtualenv rst2html5
```

3. Install project's requirements:

```
$ pip install -r requirements.txt -r dev_test_requirements.txt
```

Now you are ready!

**Note:** To come back to the virtualenv in another session, use the command `workon rst2html5`.

**See also:**

- Virtualenvwrapper command reference

## 4.4 Running the test suite

To run the tests, just type the following on a terminal:

```
$ nosetests
```

To get a complete test verification, run:

```
$ tox
```

The complete tests save some interesting metrics at `rst2html5/.tox/metrics/log`.

**Important:** Before sending a patch or a pull request, ensure that all tests pass and there is no flake8 error or warning codes.

## 4.5 Documentation

Contributing to documentation is as simple as editing the specified file in the `docs` directory. We use restructuredtext markup and Sphinx for building the documentation.

## 4.6 Reporting an issue

Proposals, enhancements, bugs or tasks should be directly reported on BitBucket issue tracker.

If there are issues please let us know so we can improve rst2html5. If you don't report it, we probably won't fix it. When creating a bug issue, try to provide the following information at least:

1. Steps to reproduce the bug
2. The produced output
3. The expected output

**Tip:** See https://bitbucket.org/andre_felipe_dias/rst2html5/issue/1 as a reference.

For proposals or enhancements, you should provide input and output examples. Whenever possible, you should also provide external references to articles or documentation that endorses your request.

While it's handy to provide useful code snippets in an issue, it is better for you as a developer to submit pull requests. By submitting pull request your contribution to `rst2html5` will be recorded by BitBucket.

## 4.7 Contacting the author

`rst2html5` is written and maintained by André Felipe Dias. You can reach me at google plus or twitter.

CHAPTER 5

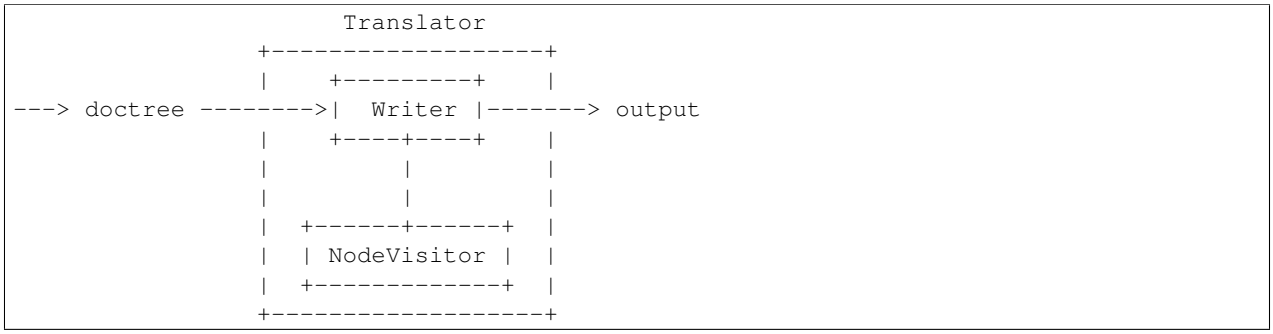
# rst2html5 Design Notes

The following documentation describes the knowledge collected durint rst2html5 implementation. Probably, it isn't complete or even exact, but it might be helpful to other people who want to create another rst converter.

## 5.1 Docutils

Docutils is a set of tools for processing plaintext documentation in restructuredText markup (rst) into other formats such as HTML, PDF and Latex. Its documents design issues and implementation details are described at http://docutils.sourceforge.net/docs/peps/pep-0258.html

In the early stages of the translation process, the rst document is analyzed and transformed into an intermediary format called *doctree* which is then passed to a translator to be transformed into the desired formatted output:




### 5.1.1 Doctree

The doctree is a hierarchical structure of the elements of a rst document. It is defined at `docutils.nodes` and is used internally by Docutils components.

The command `rst2pseudoxml.py` produces a textual representation of a doctree that is very useful to visualize the nesting of the elements of a rst document. This information was of great help to both `rst2html5` design and tests.

Given the following rst snippet:

```
Title
=====

Text and more text
```

The textual representation produced by `rst2pseudoxml` is:

```
<document ids="title" names="title" source="snippet.rst" title="Title">
    <title>
        Title
    <paragraph>
        Text and more text
```

### 5.1.2 Translator, Writer e NodeVisitor

A translator is comprised of two parts: a `Writer` and a `NodeVisitor`. The `Writer` is responsible to prepare and to coordinate the translation made by the `NodeVisitor`. The `NodeVisitor` is used when visiting each doctree node and it performs all actions needed to translate the node to the desired format according to its type and content.

---

**Important:** To develop a new docutils translator, one needs to specialize these two classes.

---

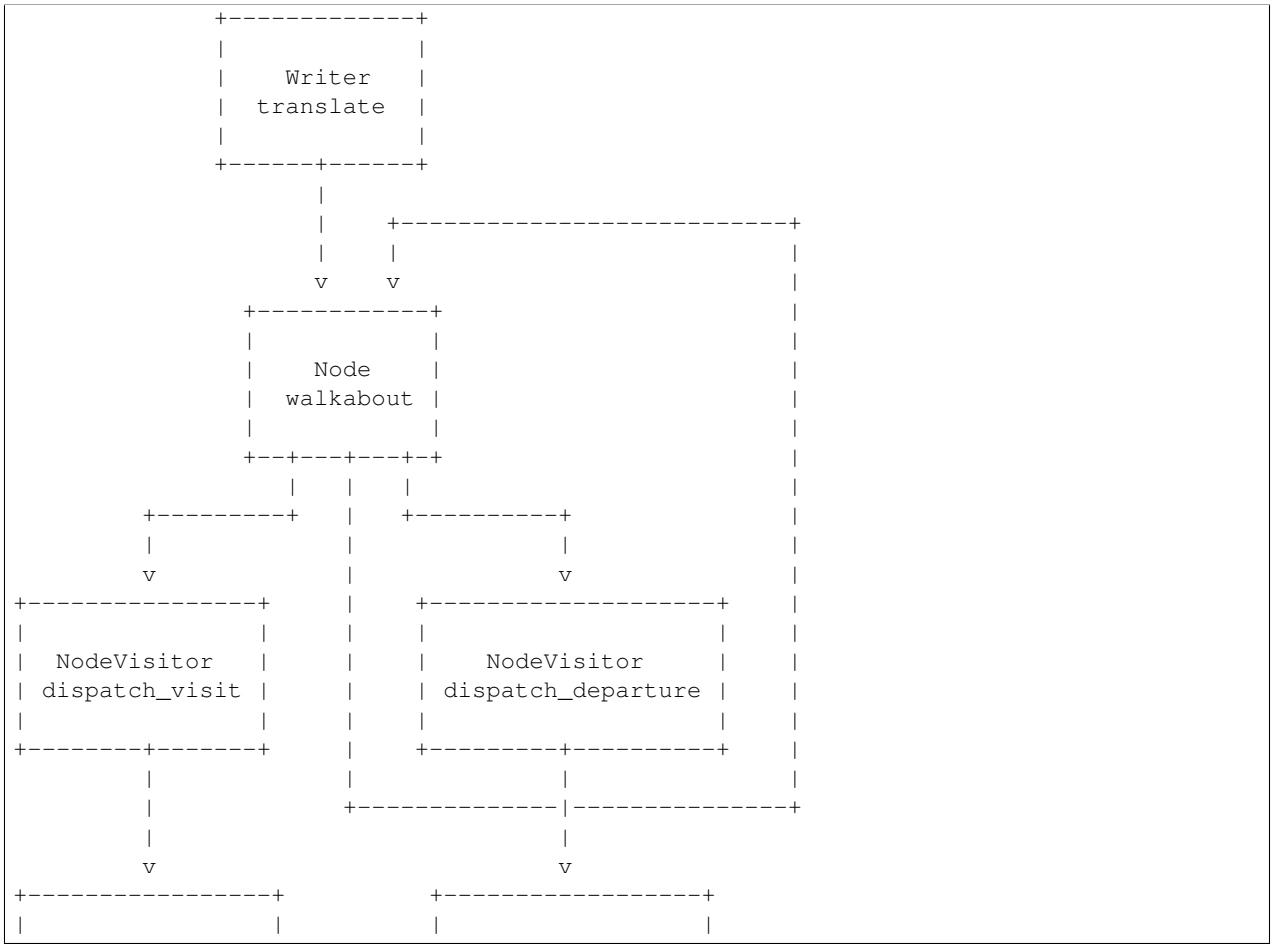
---

**Note:** Those classes correspond to a variation of the Visitor pattern, called "Extrinsic Visitor" that is more commonly used in Python. See The "Visitor Pattern", Revisited.

---

**See also:**

Double Dispatch and the "Visitor" Pattern.



```
|   NodeVisitor    |           |   NodeVisitor    |
| visit_NODE_TYPE  |           | depart_NODE_TYPE |
|                  |           |                  |
+------------------+           +------------------+
```

During the doctree traversal through *docutils.nodes.Node.walkabout()*, there are two `NodeVisitor` dispatch methods called: *dispatch_visit()* and *dispatch_departure()*. The former is called early in the node visitation. Then, all children nodes *walkabout()* are visited and lastly the latter dispatch method is called. Each dispatch method calls another method whose name follows the pattern `visit_NODE_TYPE` or `depart_NODE_TYPE` such as `visit_paragraph` or `depart_title`, that should be implemented by the `NodeVisitor` subclass object.

## 5.2 rst2html5

In `rst2html5`, `Writer` and `NodeVisitor` are specialized through `HTML5Writer` and `HTML5Translator` classes.

`rst2html5.HTML5Translator` is a `NodeVisitor` subclass that implements all `visit_NODE_TYPE` and `depart_NODE_TYPE` methods needed to translate a doctree to its HTML5 content. The `rst2html5.HTML5Translator` uses an object of the:class:*~rst2html5.ElemStack* helper class that controls a context stack to handle indentation and the nesting of the doctree traversal:

```
                    rst2html5
            +-----------------------+
            |    +-------------+    |
doctree ---|--->| HTML5Writer |----|-->  HTML5
            |    +------+------+    |
            |           |           |
            |           |           |
            |  +--------+--------+  |
            |  | HTML5Translator |  |
            |  +--------+--------+  |
            |           |           |
            |           |           |
            |     +-----+-----+     |
            |     | ElemStack |     |
            |     +-----------+     |
            +-----------------------+
```

The standard `visit_NODE_TYPE` action is initiate a new node context:

```
    def default_visit(self, node):
        '''
        Initiate a new context to store inner HTML5 elements.
        '''
        if 'ids' in node and self.once_attr('expand_id_to_anchor', default=True):
            # create an anchor <a id=id></a> for each id found before the
            # current element.
            for id in node['ids'][1:]:
                self.context.begin_elem()
                self.context.commit_elem(tag.a(id=id))
            node.attributes['ids'] = node.attributes['ids'][0:1]
        self.context.begin_elem()
        return
```

The standard `depart_NODE_TYPE` action is to create the HTML5 element according to the saved context:

```
    def default_departure(self, node):
        '''
        Create the node's corresponding HTML5 element and combine it with its
        stored context.
        '''
        tag_name, indent, attributes = self.parse(node)
        elem = getattr(tag, tag_name)(**attributes)
        self.context.commit_elem(elem, indent)
        return
```

Not all rst elements follow this procedure. The `Text` element, for example, is a leaf-node and thus doesn't need a specific context. Other elements have a common processing and can share the same `visit_` and/or `depart_` method. To take advantage of theses similarities, the `rst_terms` dict maps a node type to a `visit_` and `depart_` methods:

```
    def append(self, element, indent=True):
        '''
        Append to current element
        '''
        self.stack[-1].append(self._indent_elem(element, indent))
        return

    def begin_elem(self):
        '''
        Start a new element context
        '''
        self.stack.append([])
        self.indent_level += 1
        return

    def commit_elem(self, elem, indent=True):
        '''
        A new element is create by removing its stack to make a tag.
        This tag is pushed back into its parent's stack.
        '''
        pop = self.stack.pop()
        elem(*pop)
        self.indent_level -= 1
        self.append(elem, indent)
        return

    def pop(self):
        return self.pop_elements(1)[0]

    def pop_elements(self, num_elements):
        assert num_elements > 0
        parent_stack = self.stack[-1]
        result = []
        for x in range(num_elements):
            pop = parent_stack.pop()
            elem = pop[0 if len(pop) == 1 else self.indent_output]
            result.append(elem)
        result.reverse()
        return result


dv = 'default_visit'
dp = 'default_departure'
```

```
pass_ = 'no_op'


class HTML5Translator(nodes.NodeVisitor):

    rst_terms = {
        # 'term': ('tag', 'visit_func', 'depart_func', use_term_in_class,
        #          indent_elem)
        # use_term_in_class and indent_elem are optionals.
        # If not given, the default is False, True
        'Text': (None, 'visit_Text', None),
        'abbreviation': ('abbr', dv, dp),
        'acronym': (None, dv, dp),
        'address': (None, 'visit_address', None),
        'admonition': ('aside', 'visit_aside', 'depart_aside', True),
        'attention': ('aside', 'visit_aside', 'depart_aside', True),
        'attribution': ('p', dv, dp, True),
        'author': (None, 'visit_bibliographic_field', None),
        'authors': (None, 'visit_authors', None),
        'block_quote': ('blockquote', 'visit_blockquote', dp),
        'bullet_list': ('ul', dv, dp, False),
        'caption': ('figcaption', dv, dp, False),
        'caution': ('aside', 'visit_aside', 'depart_aside', True),
        'citation': (None, 'visit_citation', 'depart_citation', True),
        'citation_reference': ('a', 'visit_citation_reference',
                               'depart_reference', True, False),
        'classifier': (None, 'visit_classifier', None),
        'colspec': (None, pass_, 'depart_colspec'),
        'comment': (None, 'visit_comment', None),
        'compound': ('div', dv, dp),
        'contact': (None, 'visit_bibliographic_field', None),
        'container': ('div', dv, dp),
        'copyright': (None, 'visit_bibliographic_field', None),
        'danger': ('aside', 'visit_aside', 'depart_aside', True),
        'date': (None, 'visit_bibliographic_field', None),
        'decoration': (None, 'do_nothing', None),
        'definition': ('dd', dv, dp),
        'definition_list': ('dl', dv, dp),
        'definition_list_item': (None, 'do_nothing', None),
        'description': ('td', dv, dp),
        'docinfo': (None, 'do_nothing', None),
        'doctest_block': ('pre', 'visit_literal_block', 'depart_literal_block', True),
        'document': (None, 'visit_document', 'depart_document'),
        'emphasis': ('em', dv, dp, False, False),
        'entry': (None, dv, 'depart_entry'),
        'enumerated_list': ('ol', dv, 'depart_enumerated_list'),
        'error': ('aside', 'visit_aside', 'depart_aside', True),
        'field': (None, 'visit_field', None),
        'field_body': (None, 'do_nothing', None),
        'field_list': (None, 'do_nothing', None),
        'field_name': (None, 'do_nothing', None),
        'figure': (None, 'visit_figure', dp),
        'footer': (None, dv, dp),
        'footnote': (None, 'visit_citation', 'depart_citation', True),
        'footnote_reference': ('a', 'visit_citation_reference', 'depart_reference', True, False),
        'generated': (None, 'do_nothing', None),
        'header': (None, dv, dp),
        'hint': ('aside', 'visit_aside', 'depart_aside', True),
```

```
        'image': ('img', dv, dp),
        'important': ('aside', 'visit_aside', 'depart_aside', True),
        'inline': ('span', dv, dp, False, False),
        'label': ('th', 'visit_reference', 'depart_label'),
        'legend': ('div', dv, dp, True),
        'line': (None, 'visit_line', None),
        'line_block': ('pre', 'visit_line_block', 'depart_line_block', True),
        'list_item': ('li', dv, dp),
        'literal': ('code', 'visit_literal', 'depart_literal', False, False),
        'literal_block': ('pre', 'visit_literal_block', 'depart_literal_block'),
        'math': (None, 'visit_math_block', None),
        'math_block': (None, 'visit_math_block', None),
        'meta': (None, 'visit_meta', None),
        'note': ('aside', 'visit_aside', 'depart_aside', True),
        'option': ('kbd', 'visit_option', dp, False, False),
        'option_argument': ('var', 'visit_option_argument', dp, False, False),
        'option_group': ('td', 'visit_option_group', 'depart_option_group'),
        'option_list': (None, 'visit_option_list', 'depart_option_list', True),
        'option_list_item': ('tr', dv, dp),
```

### 5.2.1 HTML5 Tag Construction

HTML5 Tags are constructed by the `genshi.builder.tag` object.

**Genshi Builder**

Support for programmatically generating markup streams from Python code using a very simple syntax. The main entry point to this module is the *tag* object (which is actually an instance of the `ElementFactory` class). You should rarely (if ever) need to directly import and use any of the other classes in this module.
Elements can be created using the *tag* object using attribute access. For example:

```
>>> doc = tag.p('Some text and ', tag.a('a link', href='http://example.org/'), '.')
>>> doc
<Element "p">
```

This produces an *Element* instance which can be further modified to add child nodes and attributes. This is done by "calling" the element: positional arguments are added as child nodes (alternatively, the *Element.append* method can be used for that purpose), whereas keywords arguments are added as attributes:

```
>>> doc(tag.br)
<Element "p">
>>> print(doc)
<p>Some text and <a href="http://example.org/">a link</a>.<br/></p>
```

If an attribute name collides with a Python keyword, simply append an underscore to the name:

```
>>> doc(class_='intro')
<Element "p">
>>> print(doc)
<p class="intro">Some text and <a href="http://example.org/">a link</a>.<br/></p>
```

As shown above, an *Element* can easily be directly rendered to XML text by printing it or using the Python `str()` function. This is basically a shortcut for converting the *Element* to a stream and serializing that stream:

```
>>> stream = doc.generate()
>>> stream
<genshi.core.Stream object at ...>
>>> print(stream)
<p class="intro">Some text and <a href="http://example.org/">a link</a>.<br/></p>
```

The *tag* object also allows creating "fragments", which are basically lists of nodes (elements or text) that don't have a parent element. This can be useful for creating snippets of markup that are attached to a parent element later (for example in a template). Fragments are created by calling the *tag* object, which returns an object of type *Fragment*:

```
>>> fragment = tag('Hello, ', tag.em('world'), '!')
>>> fragment
<Fragment>
>>> print(fragment)
Hello, <em>world</em>!
```

### 5.2.2 ElemStack

For the previous doctree example, the sequence of `visit_...` and `depart_...` calls is:

```
1. visit_document
    2. visit_title
        3. visit_Text
        4. depart_Text
    5. depart_title
    6. visit_paragraph
        7. visit_Text
        8. depart_Text
```

```
    9. depart_paragraph
10. depart_document
```

For this sequence, the behavior of a ElemStack context object is:

0. **Initial State**. The context stack is empty:

```
    context = []
```

1. **visit_document**. A new context for `document` is reserved:

```
    context = [ [] ]
                \
                 document
                 context
```

2. **visit_title**. A new context for `title` is pushed into the context stack:

```
                   title
                   context
                    /
    context = [ [], [] ]
                \
                 document
                 context
```

3. **visit_Text**. A `Text` node doesn't need a new context because it is a leaf-node. Its text is simply added to the context of its parent node:

```
                  title
                  context
                 /
context = [ [], ['Title'] ]
            \
             document
             context
```

4. **depart_Text**. No action performed. The context stack remains the same.

5. **depart_title**. This is the end of the title processing. The title context is popped from the context stack to form an `h1` tag that is then inserted into the context of the title parent node (`document context`):

```
    context = [ [tag.h1('Title')] ]
                \
                 document
                 context
```

6. **visit_paragraph**. A new context is added:

```
                                    paragraph
                                    context
                                   /
    context = [ [tag.h1('Title')], [] ]
                \
                 document
                 context
```

7. **visit_Text**. Again, the text is inserted into its parent's node context:

```
                                    paragraph
                                    context
```

```
                                        /
    context = [ [tag.h1('Title')], ['Text and more text'] ]
                 \
                  document
                  context
```

8. **depart_Text**. No action performed.

9. **depart_paragraph**. Follows the standard procedure where the current context is popped and form a new tag that is appended into the context of the parent node:

```
    context = [ [tag.h1('Title'), tag.p('Text and more text')] ]
                 \
                  document
                  context
```

10. **depart_document**. The document node doesn't have an HTML tag. Its context is simply combined to the outer context to form the body of the HTML5 document:

```
    context = [tag.h1('Title'), tag.p('Text and more text')]
```

## 5.3 rst2html5 Tests

The tests executed in `rst2html5.tests.test_html5writer` are bases on generators (veja http://nose.readthedocs.org/en/latest/writing_tests.html#test-generators). The test cases are in `tests/cases.py`. Each test case is a dictionary whose main keys are:

**rst** text snippet in rst format

**out** expected output

**part** specifies which part of `rst2html5` output will be compared to `out`. Possible values are `head`, `body` or `whole`.

All other keys are `rst2html5` configuration settings such as `indent_output`, `script`, `script-defer`, `html-tag-attr` or `stylesheet`.

When test fails, three auxiliary files are saved on the temporary directory (`/tmp`):

1. `TEST_CASE.rst` com o trecho de texto rst do caso de teste;
2. `TEST_CASE.result` com resultado produzido pelo `rst2html5` e
3. `TEST_CASE.expected` com o resultado esperado pelo caso de teste.

Their differences can be easily visualized:

```
$ kdiff3 /tmp/TEST_CASE.result /tmp/TEST_CASE.expected
```

CHAPTER 6

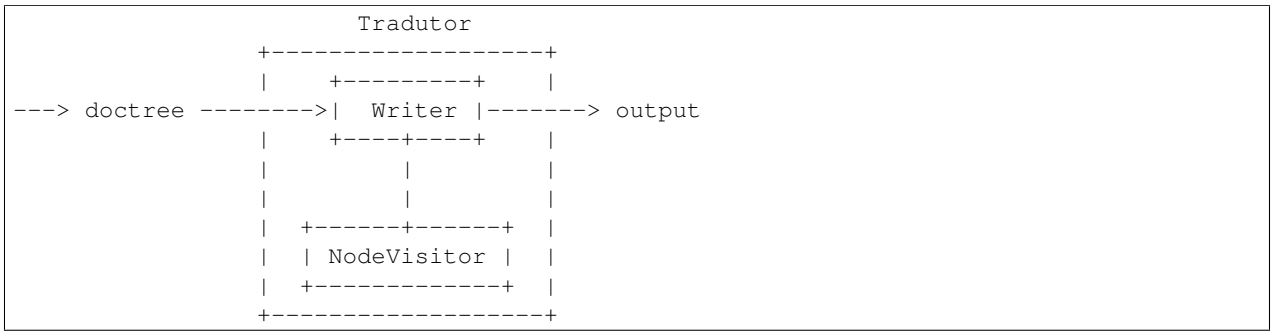
# Notas de Design (pt_BR)

O texto a seguir descreve o conhecimento coletado durante a implementação do rst2html5. Certamente não está completo e talvez nem esteja exato, mas pode ser de grande utilidade para outras pessoas que desejem criar um novo tradutor de rst para algum outro formato.

## 6.1 Docutils

O Docutils é um conjunto de ferramentas para processamento de documentação em texto simples em marcação reStructuredText (rst) para outros formatos tais como HTML, PDF e Latex. Seu funcionamento básico está descrito em http://docutils.sourceforge.net/docs/peps/pep-0258.html

Nas primeiras etapas do processo de tradução, o documento rst é analisado e convertido para um formato intermediário chamado de *doctree*, que então é passado a um tradutor para ser transformado na saída formatada desejada:




### 6.1.1 Doctree

O doctree é uma estrutura hierárquica dos elementos que compõem o documento rst, usada internamente pelos componentes do `Docutils`. Está definida no módulo `docutils.nodes`.

O comando/aplicativo `rst2pseudoxml.py` gera uma representação textual da *doctree* que é muito útil para visualizar o aninhamento dos elementos de um documento rst. Essa informação foi de grande ajuda tanto para o *design* quanto para os testes do `rst2html5`.

Dado o trecho de texto rst abaixo:

```
Título
======

Texto e mais texto
```

A sua representação textual produzida pelo `rst2pseudoxml` é:

```
<document ids="titulo" names="título" source="snippet.rst" title="Título">
    <title>
        Título
    <paragraph>
        Texto e mais texto
```

### 6.1.2 Tradutor, Writer e NodeVisitor

Um tradutor é formado por duas partes: `Writer` e `NodeVisitor`. A responsabilidade do `Writer` é preparar e coordenar a tradução feita pelo `NodeVisitor`. O `NodeVisitor` é responsável por visitar cada nó da doctree e executar a ação necessária de tradução para o formato desejado de acordo com o tipo e conteúdo do nó.

---

**Note:** A classe `NodeVisitor` corresponde à superclasse abstrata do padrão de projeto "Visitor" *[GoF95]*.

---

---

**Note:** Estas classes correspondem a uma variação do padrão de projeto "Visitor" conhecida como "Extrinsic Visitor" que é mais comumente usada em Python. Veja The "Visitor Pattern", Revisited.

---

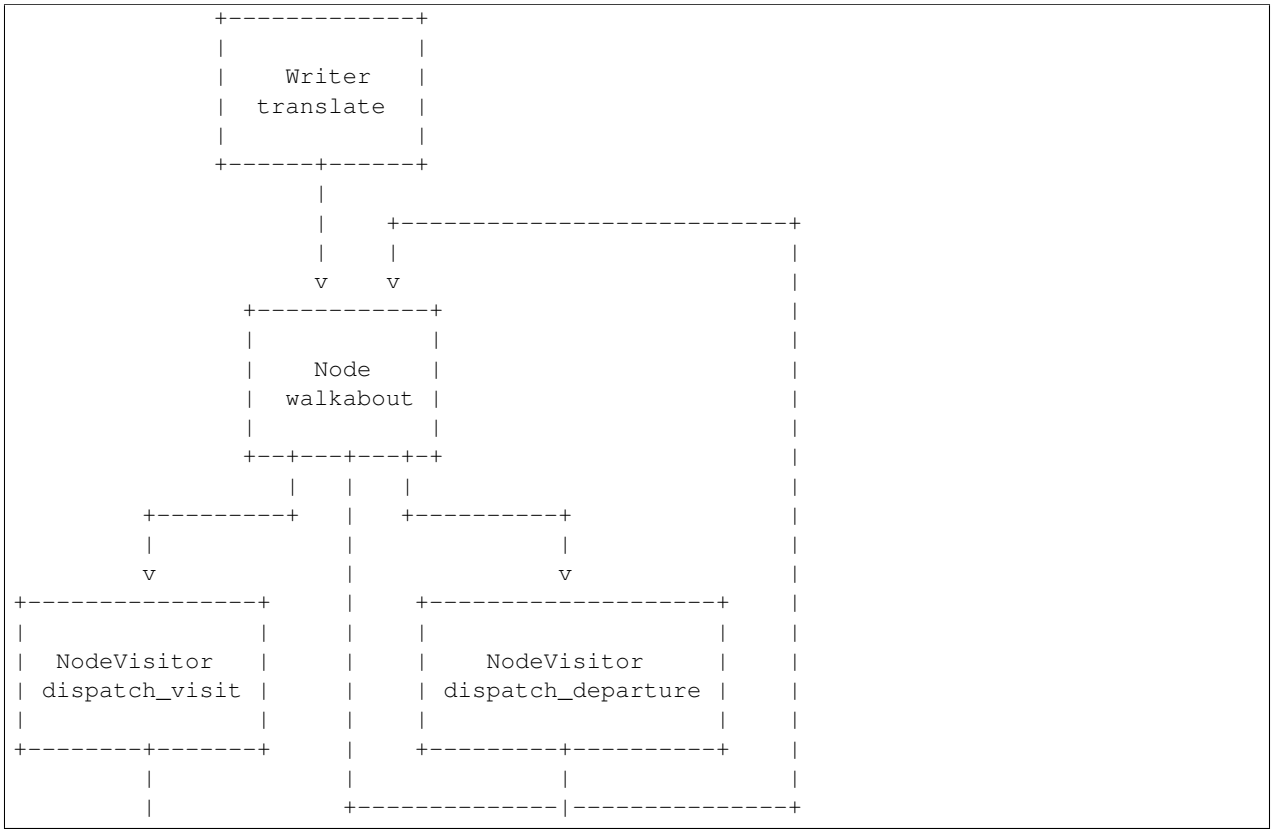
---

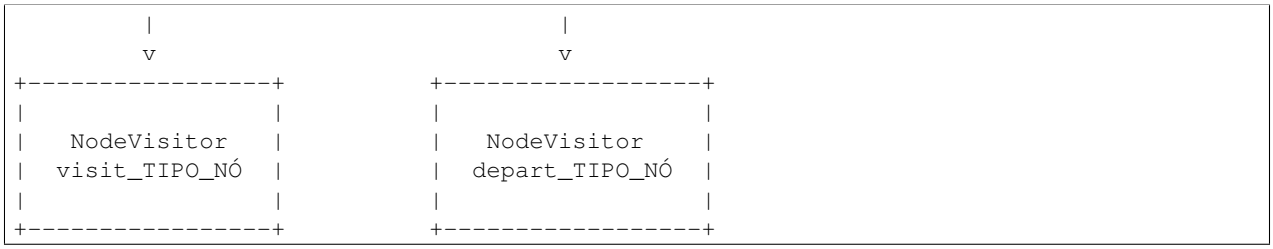
**Important:** Para desenvolver um novo tradutor para o *docutils*, é necessário especializar estas duas classes.

---

**See also:**

Double Dispatch and the "Visitor" Pattern.






During doctree traversal through *docutils.nodes.Node.walkabout()*, there are two `NodeVisitor` dispatch methods called: *dispatch_visit()* and *dispatch_departure()*. The former is called early in the node visitation. Then, all children nodes *walkabout()* are visited and lastly the latter dispatch method is called. Each dispatch method calls a specific `visit_NODE_TYPE` or `depart_NODE_TYPE` method such as `visit_paragraph` or `depart_title`, that should be implemented by the `NodeVisitor` subclass object.

Durante a travessia da doctree feita através do método *docutils.nodes.Node.walkabout()*, há dois métodos `dispatch` de `NodeVisitor` chamados: *dispatch_visit()* e *dispatch_departure()*. O primeiro é chamado logo no começo da visitação do nó. Em seguida, todos os nós-filho são visitados e, por último, o método `dispatch_departure` é chamado. Cada um desses métodos chama um método cujo nome segue o padrão `visit_NODE_TYPE` ou `depart_NODE_TYPE`, tal como `visit_paragraph` ou `depart_title`, que deve ser implementado na subclasse de `NodeVisitor`.

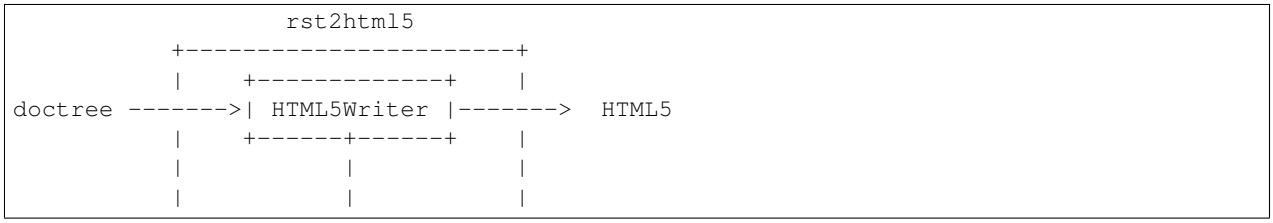
Para a *doctree* do exemplo anterior, a sequência de chamadas `visit_...` e `depart_...` seria:

```
1. visit_document
    2. visit_title
        3. visit_Text
        4. depart_Text
    5. depart_title
    6. visit_paragraph
        7. visit_Text
        8. depart_Text
    9. depart_paragraph
10. depart_document
```

**Note:** São nos métodos `visit_...` e `depart_...` onde deve ser feita a tradução de cada nó de acordo com seu tipo e conteúdo.

## 6.2 rst2html5

O módulo `rst2html5` segue as recomendações originais e especializa as classes `Writer` e `NodeVisitor` através das classes `HTML5Writer` e `HTML5Translator`. `rst2html5.HTML5Translator` é a subclasse de `NodeVisitor` criada para implementar todos os métodos `visit_TIPO_NÓ` e `depart_TIPO_NÓ` necessários para traduzir uma doctree em seu correspondente HTML5. Isto é feito com ajuda de um outro objeto da classe auxiliar `ElemStack` que controla uma pilha de contextos para lidar com o aninhamento da visitação dos nós da doctree e com a endentação:



```
|  +--------+--------+  |
|  | HTML5Translator |  |
|  +--------+--------+  |
|           |           |
|           |           |
|     +-----+-----+     |
|     | ElemStack |     |
|     +-----------+     |
+-----------------------+
```

A ação padrão de um método `visit_TIPO_NÓ` é iniciar um novo contexto para o nó sendo tratado:

```
    def default_visit(self, node):
        '''
        Initiate a new context to store inner HTML5 elements.
        '''
        if 'ids' in node and self.once_attr('expand_id_to_anchor', default=True):
            # create an anchor <a id=id></a> for each id found before the
            # current element.
            for id in node['ids'][1:]:
                self.context.begin_elem()
                self.context.commit_elem(tag.a(id=id))
            node.attributes['ids'] = node.attributes['ids'][0:1]
        self.context.begin_elem()
        return
```

A ação padrão no `depart_TIPO_NÓ` é criar o elemento HTML5 de acordo com o contexto salvo:

```
    def default_departure(self, node):
        '''
        Create the node's corresponding HTML5 element and combine it with its
        stored context.
        '''
        tag_name, indent, attributes = self.parse(node)
        elem = getattr(tag, tag_name)(**attributes)
        self.context.commit_elem(elem, indent)
        return
```

Nem todos os elementos rst seguem o este processamento. O elemento `Text`, por exemplo, é um nó folha e, por isso, não requer a criação de um contexto específico. Basta adicionar o texto correspondente ao elemento pai.

Outros tipos de nós têm um processamento comum e podem compartilhar o mesmo método `visit_` e/ou `depart_`. Para aproveitar essas similaridades, é feito um mapeamento entre o nó rst e os métodos correspondentes pelo dicionário `rst_terms`:

```
    def append(self, element, indent=True):
        '''
        Append to current element
        '''
        self.stack[-1].append(self._indent_elem(element, indent))
        return

    def begin_elem(self):
        '''
        Start a new element context
        '''
        self.stack.append([])
        self.indent_level += 1
        return
```

```
    def commit_elem(self, elem, indent=True):
        '''
        A new element is create by removing its stack to make a tag.
        This tag is pushed back into its parent's stack.
        '''
        pop = self.stack.pop()
        elem(*pop)
        self.indent_level -= 1
        self.append(elem, indent)
        return

    def pop(self):
        return self.pop_elements(1)[0]

    def pop_elements(self, num_elements):
        assert num_elements > 0
        parent_stack = self.stack[-1]
        result = []
        for x in range(num_elements):
            pop = parent_stack.pop()
            elem = pop[0 if len(pop) == 1 else self.indent_output]
            result.append(elem)
        result.reverse()
        return result


dv = 'default_visit'
dp = 'default_departure'
pass_ = 'no_op'


class HTML5Translator(nodes.NodeVisitor):

    rst_terms = {
        # 'term': ('tag', 'visit_func', 'depart_func', use_term_in_class,
        #          indent_elem)
        # use_term_in_class and indent_elem are optionals.
        # If not given, the default is False, True
        'Text': (None, 'visit_Text', None),
        'abbreviation': ('abbr', dv, dp),
        'acronym': (None, dv, dp),
        'address': (None, 'visit_address', None),
        'admonition': ('aside', 'visit_aside', 'depart_aside', True),
        'attention': ('aside', 'visit_aside', 'depart_aside', True),
        'attribution': ('p', dv, dp, True),
        'author': (None, 'visit_bibliographic_field', None),
        'authors': (None, 'visit_authors', None),
        'block_quote': ('blockquote', 'visit_blockquote', dp),
        'bullet_list': ('ul', dv, dp, False),
        'caption': ('figcaption', dv, dp, False),
        'caution': ('aside', 'visit_aside', 'depart_aside', True),
        'citation': (None, 'visit_citation', 'depart_citation', True),
        'citation_reference': ('a', 'visit_citation_reference',
                               'depart_reference', True, False),
        'classifier': (None, 'visit_classifier', None),
        'colspec': (None, pass_, 'depart_colspec'),
        'comment': (None, 'visit_comment', None),
```

```
        'compound': ('div', dv, dp),
        'contact': (None, 'visit_bibliographic_field', None),
        'container': ('div', dv, dp),
        'copyright': (None, 'visit_bibliographic_field', None),
        'danger': ('aside', 'visit_aside', 'depart_aside', True),
        'date': (None, 'visit_bibliographic_field', None),
        'decoration': (None, 'do_nothing', None),
        'definition': ('dd', dv, dp),
        'definition_list': ('dl', dv, dp),
        'definition_list_item': (None, 'do_nothing', None),
        'description': ('td', dv, dp),
        'docinfo': (None, 'do_nothing', None),
        'doctest_block': ('pre', 'visit_literal_block', 'depart_literal_block', True),
        'document': (None, 'visit_document', 'depart_document'),
        'emphasis': ('em', dv, dp, False, False),
        'entry': (None, dv, 'depart_entry'),
        'enumerated_list': ('ol', dv, 'depart_enumerated_list'),
        'error': ('aside', 'visit_aside', 'depart_aside', True),
        'field': (None, 'visit_field', None),
        'field_body': (None, 'do_nothing', None),
        'field_list': (None, 'do_nothing', None),
        'field_name': (None, 'do_nothing', None),
        'figure': (None, 'visit_figure', dp),
        'footer': (None, dv, dp),
        'footnote': (None, 'visit_citation', 'depart_citation', True),
        'footnote_reference': ('a', 'visit_citation_reference', 'depart_reference', True, False),
        'generated': (None, 'do_nothing', None),
        'header': (None, dv, dp),
        'hint': ('aside', 'visit_aside', 'depart_aside', True),
        'image': ('img', dv, dp),
        'important': ('aside', 'visit_aside', 'depart_aside', True),
        'inline': ('span', dv, dp, False, False),
        'label': ('th', 'visit_reference', 'depart_label'),
        'legend': ('div', dv, dp, True),
        'line': (None, 'visit_line', None),
        'line_block': ('pre', 'visit_line_block', 'depart_line_block', True),
        'list_item': ('li', dv, dp),
        'literal': ('code', 'visit_literal', 'depart_literal', False, False),
        'literal_block': ('pre', 'visit_literal_block', 'depart_literal_block'),
        'math': (None, 'visit_math_block', None),
        'math_block': (None, 'visit_math_block', None),
        'meta': (None, 'visit_meta', None),
        'note': ('aside', 'visit_aside', 'depart_aside', True),
        'option': ('kbd', 'visit_option', dp, False, False),
        'option_argument': ('var', 'visit_option_argument', dp, False, False),
        'option_group': ('td', 'visit_option_group', 'depart_option_group'),
        'option_list': (None, 'visit_option_list', 'depart_option_list', True),
        'option_list_item': ('tr', dv, dp),
```

### 6.2.1 Construção de Tags HTML5

A construção das *tags* do HTML5 é feita através do objeto `tag` do módulo *genshi.builder*.

**Genshi Builder**

Support for programmatically generating markup streams from Python code using a very simple syntax. The main entry point to this module is the *tag* object (which is actually an instance of the `ElementFactory` class). You should rarely (if ever) need to directly import and use any of the other classes in this module.
Elements can be created using the *tag* object using attribute access. For example:

```
>>> doc = tag.p('Some text and ', tag.a('a link', href='http://example.org/'), '.')
>>> doc
<Element "p">
```

This produces an *Element* instance which can be further modified to add child nodes and attributes. This is done by "calling" the element: positional arguments are added as child nodes (alternatively, the *Element.append* method can be used for that purpose), whereas keywords arguments are added as attributes:

```
>>> doc(tag.br)
<Element "p">
>>> print(doc)
<p>Some text and <a href="http://example.org/">a link</a>.<br/></p>
```

If an attribute name collides with a Python keyword, simply append an underscore to the name:

```
>>> doc(class_='intro')
<Element "p">
>>> print(doc)
<p class="intro">Some text and <a href="http://example.org/">a link</a>.<br/></p>
```

As shown above, an *Element* can easily be directly rendered to XML text by printing it or using the Python `str()` function. This is basically a shortcut for converting the *Element* to a stream and serializing that stream:

```
>>> stream = doc.generate()
>>> stream
<genshi.core.Stream object at ...>
>>> print(stream)
<p class="intro">Some text and <a href="http://example.org/">a link</a>.<br/></p>
```

The *tag* object also allows creating "fragments", which are basically lists of nodes (elements or text) that don't have a parent element. This can be useful for creating snippets of markup that are attached to a parent element later (for example in a template). Fragments are created by calling the *tag* object, which returns an object of type *Fragment*:

```
>>> fragment = tag('Hello, ', tag.em('world'), '!')
>>> fragment
<Fragment>
>>> print(fragment)
Hello, <em>world</em>!
```

### 6.2.2 ElemStack

Como a travessia da *doctree* não é feita por recursão, é necessária uma estrutura auxiliar de pilha para armazenar os contextos prévios. A classe auxiliar `ElemStack` é uma pilha que registra os contextos e controla o nível de endentação.

O comportamento do objeto ElemStack é ilustrado a seguir, através da visualização da estrutura de pilha durante a análise do trecho rst que vem sendo usado como exemplo. As chamadas `visit_...` e `depart_...` acontecerão na seguinte ordem:

```
1. visit_document
    2. visit_title
```

```
        3. visit_Text
        4. depart_Text
    5. depart_title
    6. visit_paragraph
        7. visit_Text
        8. depart_Text
    9. depart_paragraph
10. depart_document
```

0. **Estado inicial**. A pilha de contexto está vazia:

```
    context = []
```

1. **visit_document**. Um novo contexto para `document` é criado:

```
    context = [ [] ]
                \
                 document
                 context
```

2. **visit_title**. Um novo contexto é criado para o elemento `title`:

```
                   title
                   context
                    /
    context = [ [], [] ]
                \
                 document
                 context
```

3. **visit_Text**. O nó do tipo `Text` não precisa de um novo contexto pois é um nó-folha. O texto é simplesmente adicionado ao contexto do seu nó-pai:

```
                     title
                     context
                    /
    context = [ [], ['Title'] ]
                \
                 document
                 context
```

4. **depart_Text**. Nenhuma ação é executada neste passo. A pilha permanece inalterada.

5. **depart_title**. Representa o fim do processamento do título. O contexto do título é extraído da pilha e combinado com uma tag `h1` que é inserida no contexto do nó-pai (`document context`):

```
    context = [ [tag.h1('Title')] ]
                \
                 document
                 context
```

6. **visit_paragraph**. Um novo contexto é criado:

```
                                    paragraph
                                    context
                                   /
    context = [ [tag.h1('Title')], [] ]
                \
                 document
                 context
```

7. **visit_Text**. Mais uma vez, o texto é adicionado ao contexto do nó-pai:

```
                                    paragraph
                                    context
                                   /
    context = [ [tag.h1('Title')], ['Text and more text'] ]
                  \
                   document
                   context
```

8. **depart_Text**. Nenhuma ação é necessária.

9. **depart_paragraph**. Segue o comportamento padrão, isto é, o contexto é combinado com a tag do elemento rst atual e então é inserida no contexto do nó-pai:

```
    context = [ [tag.h1('Title'), tag.p('Text and more text')] ]
                  \
                   document
                   context
```

10. **depart_document**. O nó da classe `document` não tem um correspondente em HTML5. Seu contexto é simplesmente combinado com o contexto mais geral que será o `body`:

```
context = [tag.h1('Title'), tag.p('Text e more text')]
```

## 6.3 Testes

Os testes executados no módulo `rst2html5.tests.test_html5writer` são baseados em geradores (veja http://nose.readthedocs.org/en/latest/writing_tests.html#test-generators). Os casos de teste são registrados no arquivo `tests/cases.py`. Cada caso de teste fica registrado em uma variável do tipo dicionário cujas entradas principais são:

**rst** Trecho de texto rst a ser transformado

**out** Saída esperada

**part** A qual parte da saída produzida pelo `rst2html5` será usada na comparação com `out`. As partes possíveis são: `head`, `body` e `whole`.

Todas as demais entradas são consideradas opções de configuração do `rst2html5`. Exemplos: `indent_output`, `script`, `script-defer`, `html-tag-attr` e `stylesheet`.

Em caso de falha no teste, três arquivos auxiliares são gravados no diretório temporário (`/tmp` no Linux):

1. `NOME_CASO_TESTE.rst` com o trecho de texto rst do caso de teste;
2. `NOME_CASO_TESTE.result` com resultado produzido pelo `rst2html5` e
3. `NOME_CASO_TESTE.expected` com o resultado esperado pelo caso de teste.

Em que `NOME_CASO_TESTE` é o nome da variável que contém o dicionário do caso de teste.

A partir desses arquivos é mais fácil comparar as diferenças:

```
$ kdiff3 /tmp/NOME_CASO_TESTE.result /tmp/NOME_CASO_TESTE.expected
```

CHAPTER 7

# Reference

## 7.1 Methods

`Writer.`**`translate`**`()`
Do final translation of *self.document* into *self.output*. Called from *write*. Override in subclasses.

Usually done with a *docutils.nodes.NodeVisitor* subclass, in combination with a call to *docutils.nodes.Node.walk()* or *docutils.nodes.Node.walkabout()*. The `NodeVisitor` subclass must support all standard elements (listed in *docutils.nodes.node_class_names*) and possibly non-standard elements used by the current Reader as well.

`NodeVisitor.`**`dispatch_visit`**`(`*node*`)`
Call self."`visit_` + node class name" with *node* as parameter. If the `visit_...` method does not exist, call self.unknown_visit.

`NodeVisitor.`**`dispatch_departure`**`(`*node*`)`
Call self."`depart_` + node class name" with *node* as parameter. If the `depart_...` method does not exist, call self.unknown_departure.

`Node.`**`walk`**`(`*visitor*`)`
Traverse a tree of *Node* objects, calling the *dispatch_visit()* method of *visitor* when entering each node. (The *walkabout()* method is similar, except it also calls the *dispatch_departure()* method before exiting each node.)

This tree traversal supports limited in-place tree modifications. Replacing one node with one or more nodes is OK, as is removing an element. However, if the node removed or replaced occurs after the current node, the old node will still be traversed, and any new nodes will not.

Within `visit` methods (and `depart` methods for *walkabout()*), *TreePruningException* subclasses may be raised (*SkipChildren*, *SkipSiblings*, *SkipNode*, *SkipDeparture*).

Parameter *visitor*: A *NodeVisitor* object, containing a `visit` implementation for each *Node* subclass encountered.

Return true if we should stop the traversal.

`Node.`**`walkabout`**`(`*visitor*`)`
Perform a tree traversal similarly to *Node.walk()* (which see), except also call the *dispatch_departure()* method before exiting each node.

Parameter *visitor*: A *NodeVisitor* object, containing a `visit` and `depart` implementation for each *Node* subclass encountered.

Return true if we should stop the traversal.

HTML5Translator.**default_visit**(*node*)
Initiate a new context to store inner HTML5 elements.

HTML5Translator.**default_departure**(*node*)
Create the node's corresponding HTML5 element and combine it with its stored context.

HTML5Translator.**visit_Text**(*node*)

## 7.2 Bibliography

[GoF95] Gamma, Helm, Johnson, Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, USA, 1995.

## g

## D



## G



## T



## V



## W