commonmark.py Documentation

Release 0.9.1

Roland Shoemaker, Bibek Kafle

Oct 04, 2019

Contents

CHAPTER 1

commonmark.py

commonmark.py is a pure Python port of jgm's commonmark.js, a Markdown parser and renderer for the CommonMark specification, using only native modules. Once both this project and the CommonMark specification are stable we will release the first `1.0` version and attempt to keep up to date with changes in `commonmark.js`.

commonmark.py is tested against the CommonMark spec with Python versions 2.7, 3.4, 3.5, 3.6, and 3.7.

Current version: 0.9.1

1.1 Installation

```
$ pip install commonmark
```

1.2 Usage

```
>>> import commonmark
>>> commonmark.commonmark('*hello!*')
'<p><em>hello!</em></p>\n'
```

Or, without the syntactic sugar:

```
import commonmark
parser = commonmark.Parser()
ast = parser.parse("Hello *World*")

renderer = commonmark.HtmlRenderer()
html = renderer.render(ast)
print(html) # <p>Hello <em>World</em><p/>
```

(continues on next page)

(continued from previous page)

```
# inspecting the abstract syntax tree
json = commonmark.dumpJSON(ast)
commonmark.dumpAST(ast) # pretty print generated AST structure
```

There is also a CLI:

```
$ cmark README.md -o README.html
$ cmark README.md -o README.json -aj # output AST as JSON
$ cmark README.md -a # pretty print generated AST structure
$ cmark -h
usage: cmark [-h] [-o [O]] [-a] [-aj] [infile]

Process Markdown according to the CommonMark specification.

positional arguments:
  infile      Input Markdown file to parse, defaults to stdin

optional arguments:
  -h, --help  show this help message and exit
  -o [O]      Output HTML/JSON file, defaults to stdout
  -a          Print formatted AST
  -aj         Output JSON AST
```

1.3 Contributing

If you would like to offer suggestions/optimizations/bugfixes through pull requests please do! Also if you find an error in the parser/renderer that isn't caught by the current test suite please open a new issue and I would also suggest you send the commonmark.js project a pull request adding your test to the existing test suite.

1.4 Tests

To work on commonmark.py, you will need to be able to run the test suite to make sure your changes don't break anything. To run the tests, you can do something like this:

```
$ pyvenv venv
$ ./venv/bin/python setup.py develop test
```

The tests script, `commonmark/tests/run_spec_tests.py`, is pretty much a devtool. As well as running all the tests embedded in `spec.txt` it also allows you to run specific tests using the `-t` argument, provide information about passed tests with `-p`, percentage passed by category of test with `-s`, and enter markdown interactively with `-i` (In interactive mode end a block by inputting a line with just `end`, to quit do the same but with `quit`). `-d` can be used to print call tracing.

```
$ ./venv/bin/python commonmark/tests/run_spec_tests.py -h
usage: run_spec_tests.py [-h] [-t T] [-p] [-f] [-i] [-d] [-np] [-s]

script to run the CommonMark specification tests against the commonmark.py
parser.

optional arguments:
  -h, --help  show this help message and exit
```

(continues on next page)

(continued from previous page)

```
  -t T          Single test to run or comma separated list of tests (-t 10 or -t 10,11,
↪12,13)
  -p            Print passed test information
  -f            Print failed tests (during -np...)
  -i            Interactive Markdown input mode
  -d            Debug, trace calls
  -np           Only print section header, tick, or cross
  -s            Print percent of tests passed by category
```

1.5 Authors

- Bibek Kafle
- Roland Shoemaker
- Nikolas Nyby

1.5.1 API

HTML

class commonmark.render.html.**HtmlRenderer**(*options={}*)

out(*s*)
Concatenate a string to the buffer possibly escaping the content.

Concrete renderer implementations should override this method.

@param str {String} The string to concatenate.

tag(*name*, *attrs=None*, *selfclosing=None*)
Helper function to produce an HTML tag.

reStructuredText

class commonmark.render.rst.**ReStructuredTextRenderer**(*indent_char=' '*)
Render reStructuredText from Markdown

Example:

```
import commonmark

parser = commonmark.Parser()
ast = parser.parse('Hello `inline code` example')

renderer = commonmark.ReStructuredTextRenderer()
rst = renderer.render(ast)
print(rst)  # Hello ``inline code`` example
```

lit(*s*)
Concatenate a literal string to the buffer.

@param str {String} The string to concatenate.

Parser

class commonmark.blocks.**Parser**(*options={}*)

add_child(*tag*, *offset*)
Add block of type tag as a child of the tip. If the tip can't accept children, close and finalize it and try its parent, and so on til we find a block that can accept children.

add_line()
Add a line to the block at the tip. We assume the tip can accept lines – that check should be done before calling this.

close_unmatched_blocks()
Finalize and close any unmatched blocks.

finalize(*block*, *line_number*)
Finalize a block. Close it and do any necessary postprocessing, e.g. creating string_content from strings, setting the 'tight' or 'loose' status of a list, and parsing the beginnings of paragraphs for reference definitions. Reset the tip to the parent of the closed block.

incorporate_line(*ln*)
Analyze a line of text and update the document appropriately.

We parse markdown text by calling this on each line of input, then finalizing the document.

parse(*my_input*)
The main parsing function. Returns a parsed document AST.

process_inlines(*block*)
Walk through a block & children recursively, parsing string content into inline content where appropriate.

Node

class commonmark.node.**NodeWalker**(*root*)

nxt()
for backwards compatibility

class commonmark.node.**Node**(*node_type*, *sourcepos*)

Index