`marky` Documentation – `pdf`

Contents

Abstract – `marky` is a preprocessor with an easy and intuitive syntax for execution of embedded pyhon code during rendering `html` and `pdf` documents from Markdown text. This document is created using `marky`, version *0.9*. For more information please refer to the `marky` repository.

1 `marky` Dynamic Markdown

`marky` is a Markdown preprocessor which transforms a Markdown document using python. `marky` implements three statements with extremely easy and intuitive syntax, which are embedded directly in the Markdown text:

1. `<?...?>`: Python code block.
2. `{{...}}`: `f`-string output into Markdown.
3. `___()`: Function for output into Markdown.

Using `<?...?>` and `{{...}}` python processing and `f`-string output is embedded directly inside the Markdown text. Using the `___()` function text is generated from python algorithms and dynamically inserted into the resulting Markdown.

The following example can be produced by just calling `make pdf/file` or `make html/file`.

Example: `md/file.md`

```
---
title: An Example
---
<?
def cap_first(x):
    return " ".join([i[0].upper() + i[1:] for i in i.split()])
for i in ["very", "not so"]:
    ?>
**{{cap_first(i)}} Section**

To day is a {{i}} very nice day.
```

```
The sun is shining {{i}} bright and
the birds are singing {{i}} loud and
fly {{i}} high in the {{i}} blue sky.
    <?
?>
```

Output `build/file.md`

```
---
title: An Example
---

**Very Section**

To day is a very very nice day.
The sun is shining very bright and
the birds are singing very loud and
fly very high in the very blue sky.

**Not So Section**

To day is a not so very nice day.
The sun is shining not so bright and
the birds are singing not so loud and
fly not so high in the not so blue sky.
```

2 How does `marky` work internally?

`marky` uses an extremely simple mechanism for generating a python programm from the Markdown text. Using the `<?...?>` and `{{...}}` statement, Python code is embedded into the Markdown text and translated into a series of calls to the `___()` function using `f`-strings as arguments, where python variables are referenced. This results into a python program which can generate Markdown text algorithmically.

Example: `md/file.md`

```
* This is {first}. <?
x = 1 # this is code
```

```
for i in range(3):
    if x:
        ?>
{{i+1}}. The value is {{{x}}}.
<?
    else:
        ?>{{i+1}}. The value is zero.
<?
    x = 0
?>* This is last.
```

The file produces the following Markdown output.

Output: Markdown

```
* This is {first}.
1. The value is {1}.
2. The value is zero.
3. The value is zero.
* This is last.
```

`marky` transforms the Markdown into Python source code. Execution of the Python source code yields the new Markdown text.

Output: `build/file.py`

```
___(rf"""* This is {{first}}. """, ___);
x = 1 # this is code
for i in range(3):
    if x:
        ___(rf"""
{i+1}. The value is {{{x}}}.
""", ___);
    else:
        ___(rf"""{i+1}. The value is zero.
""", ___);
    x = 0
___(rf"""* This is last.
""", ___);
```

3 Quick Start

3.1 `marky` Dependencies

`marky` depends on `pandoc` and `pyyaml`. `pandoc` is used for rendering the Markdown into `html` and `pdf`. `marky` uses pandoc for rendering `html` and `pdf`. `pandoc>=2.10` releases can be found here. The other packages can be installed with `pip`.

```
pip install pandoc-fignos
pip install pandoc-eqnos
pip install pandoc-secnos
pip install pandoc-tablenos
pip install pandoc-xnos
pip install pyyaml
```

3.2 `marky` Workflow

Workflow for creating `html` or `pdf` using `marky` by invocation of `make scan` and `make all`.

make	1. *write*	2. *build*		3. *render*
`pdf`		`build/file.html.md`	->	`html/file.html`
`-`	`md/file.md` ->			
`hfml`		`build/file.pdf.md`	->	`pdf/file.pdf`

1. **write**: user writes a Markdown text file and places it in `md/*.md` directory with the extension `.md`.
2. **build**: marky`transforms the files in`md/*.md`into regular Markdown text and places the transformed files in`build/‘.
3. **render**: the regular Markdown text in the files `build/*.md` is rendered into `html` and `pdf` using `pandoc`.

The three steps are implemented in a Makefile.

3.3 Download and Initialize

`marky` is supplied as a single-file script which automatically sets up the project structure containing all scripts required for processing and rendering Markdown.

For example, download `marky` from github.

```
git clone https://lehmann7.github.com/marky.git
cd marky
```

After download, the `marky` environment is initialized using `marky`.

```
./marky.py --init
# mkdir build/
# mkdir data
# mkdir md/
# WRITE Makefile
# WRITE pandoc-run
# WRITE md/marky.md
# WRITE .gitignore
# USAGE
make help
```

3.4 `marky` Environment

During initialization, `marky` creates directories and files. After initialization, the following structure is auto-generated in the project directory. `marky` shows the project structure when invoking `make tree`.

```
PROJECT TREE
###############
<working_dir>
|- marky.py              - marky executable
|- Makefile         (*) - marky Makefile
|- pandoc-run       (*) - pandoc wrapper
|- md/              (*) - user Markdown dir
|  |- *.md              - user Markdown text
|- data/            (*) - user data dir
|  |- *.*                 user data files
|- build/           (*) - build Markdown dir
|  |- *.py          (*) - Markdown marky code
|  |- *.make        (*) - Makefile rules
|  |- *.html.md     (*) - Markdown for html format
|  |- *.pdf.md      (*) - Markdown for pdf format
|- html/*.html      (*) - rendered html dir
|- pdf/*.pdf        (*) - rendered pdf dir

(*) directories/files are auto-generated using
```

```
`./marky.py --init; make scan; make all´
```

The script `pandoc-run` can be adjusted in case specific `pandoc` options are required for rendering the `html` and `pdf` documents.

3.5 Document Rendering

By invoking `make all` all files `md/*.md` are transformed into corresponding `html/*.html` and `pdf/*.pdf` files. By invoking `make httpd` a python web server is started in `html/`.

All user-generated Markdown content goes into `md/*` user-generated data files go into `data/*`.

ATTENTION: The files in the directories `build/*` are **auto-generated**. All user files have to be placed inside the directory `md/*`. Invoking `make clean` will **delete all files** in `html/`, `build/` and `pdf/`.

3.6 Integrated Documentation

`marky` has an integrated environment. Using `make help` displays a short info about the `marky` dependencies, make targets and examples.

```
marky DEPENDENCIES
####################
* pandoc >= 2.10
* pip install pandoc-fignos
* pip install pandoc-eqnos
* pip install pandoc-secnos
* pip install pandoc-tablenos
* pip install pandoc-xnos
* pip install pyyaml

ATTENTION
############
All files in `build/*.md` and `html/*.html` are auto-generated!
User files `*.md` have to be placed in `md/*.md`!
`make clean` deletes all files in `build/`, `html/` and `pdf/`.

marky UTILS
```

```
#############
* make help            - show this *Help Message*
* make tree            - show the *Project Tree*
* make httpd           - run python -m httpd.server in `html/`
* make clean           - delete: `build/*`, `html/*`, `pdf/*`
* make scan            - build make deps: `build/*.make`
* make list            - list all scanned files and targets

marky BUILD ALL
#################
* make build           -> `build/*.{html,pdf}.md`
* make tex             -> `build/*.tex`
* make html            -> `html/*.html`
* make pdf             -> `pdf/*.pdf`
* make all             -> `html/*.html`, `pdf/*.pdf`

marky BUILD FILE
##################
* make build/file      -> `build/file.{html,pdf}.md`
* make build/file.tex  -> `build/file.tex`
* make html/file       -> `html/file.html`
* make pdf/file        -> `pdf/pdf.html`

EXAMPLE
#########
1. run `make scan; make html/file.html httpd`:
   * generate `build/file.make`
   * transform `md/file.md` -> `html/file.html`
   * start a python httpd server in `html`
2. run `make scan; make pdf/file.pdf`
   * generate `build/file.make`
   * transform `md/file.md` -> `pdf/file.pdf`
```

4 marky Features

Place a new file in `md/file.md` and run the following commands.

```
touch md/file.md
```

`marky` discovers the new document when invoking `make scan`.

```
make scan
# WRITE build/file.make
```

`marky` renders `html` and `pdf` using make targets.

```
make html/file
make pdf/file
```

4.1 Meta Data in Front Matter

If document starts with `---`, yaml is used to parse the front matter block delimited by `---`. All meta data keys will be exposed into the python scope as a local variable, unless the variable already exists.

```
---
title: "My Documet"
author: ...
date: 2022-01-01
---
The title of this document is {{title}}.
```

4.2 Embedding Python Code

Python code blocks are embedded into Markdown using `<?...?>` and `{{...}}`. All code blocks span one large scope sharing functions and local variables. Meta data is imported from Markdown front matter as local variables in the python scope. The `import` statement can be used in python code in order to access installed python packages as usual.

4.2.1 Visible Code

Using `<?!...?>` code is executed and also shown in Markdown.

Example

```
<?!
x = 42 # visible code
print("Hello console!")
?>
```

Run and Output

```
x = 42 # visible code
print("Hello console!")
```

ATTENTION: Using the `print()` function the text will be printed to the console and **not** inside the resulting Markdown text.

4.2.2 Hidden Code

Using `<?...?>` code is executed but not shown in Markdown.

Example

```
<?
x = 41 # hidden code
___(f"Output to Markdown. x = {x}!")
?>
```

Run and Output

```
Output to Markdown. x = 41!
```

ATTENTION: Using the `___()` function the text will be printed inside the resulting Markdown text **and not** on the console.

4.3 The `___()` Function

Using the `print()` statement the text will be printed to the console. When using the `___()` statement new Markdown text is inserted dynamically into the document during preprocessing.

Example: Line Break

```
<?
x = 40 # hidden code
___("Output in", ___)
___("single line! ", ___)
___(f"x = {x}")
?>
```

Run and Output

```
Output in single line! x = 40
```

Example: Shift, Crop, Return

```
<?
result = ___("""
   * text is cropped and shifted
         * shift and crop
            * can be combined
          * returning the result
""", shift="########", crop=True, ret=True)
___(result)
?>
```

Run and Output

```
########* text is cropped and shifted
########      * shift and crop
########         * can be combined
########       * returning the result
```

4.4 Algorithmic Table Example

Table 1 is generated using the following python clode block.

```
n = 5
table = ""
dec = ["*%s*", "**%s**", "~~%s~~", "`%s`",
       r"$\times^%s$", "$\infty_%s$"]
table += "|".join("X"*n) + "\n" + "|".join("-"*n) + "\n"
for i in range(n):
    fill = [chr(ord("A")+(2*i+3*k)%26) for k in range(i+1)]
    fill = [dec[(l+i)%len(dec)]%k for l, k in enumerate(fill)]
    text = list("O")*n
    text[(n>>1)-(i>>1):(n>>1)+(i>>1)] = fill
    table += "|".join(text) + "\n"
```

Table 1: Table is generated using code and the `___()` statement.

X	X	X	X	X
0	0	A	0	0
0	0	**C**	~~F~~	0
0	~~E~~	`H`	$\times^{K}$	0
0	`G`	$\times^{J}$	∞_{M}	P
$\times^{I}$	∞_{L}	O	**R**	~~U~~

4.5 Inline Formatted Output

The `{{...}}` statement uses sntax similar to python `f`-strings for formatted output of variables and results of expressions into Markdown text. The `marky` operator `{{<expression>[:<format>]}}` uses the syntax of `f`-strings.

Example 1

```
`x` is {{x}} and {{",".join([str(i) for i in range(x-10,x)])}}.
```

Output

> `x` is 40 and 30,31,32,33,34,35,36,37,38,39.

Example 2

```
x = int(1)
y = float(2.3)
z = 0
a = [1, 2, 3]
b = (4, 5)

This is a paragraph and x is {{x:03d}} and y is {{y:.2f}}.
Other content is: a = {{a}}, b = {{b}}.
```

Output

> This is a paragraph and x is 001 and y is 2.30. Other content is: a = [1, 2, 3], b = (4, 5).

4.6 Format Link Extension

When writing multiple documents, often documents are referenced between each other using links. In order to refer to external `html` and `pdf` documents the Markdown link statement is used.

```
[Link Caption](path/to/file.html)
[Link Caption](path/to/file.pdf)
```

One link statement cannot be used for rendering `html` and `pdf` with consistent paths. Using the `marky` format link `.???` file extension results in consistent links for `html` and `pdf` documents.

Example

```
[Link to this Document](marky.???)
```

Output

Link to this Document

4.7 Format Codes

Often when writing markdown for `html` and `pdf` documents, the output needs to be tweaked accordingly. `marky` supports format specific tweaking by injecting raw `html` or `tex` code into Markdown using format codes.

In order to inject format specific code the `fmtcode` class is used. The `fmtcode` class manages injection of `html` and `tex` code depending on the output format.

ATTENTION: `tex` packages have to be included for `pdf` as well as JavaScript and style sheets for `html` using the meta data fields `header-includes--pdf` and `header-includes--html` respectively.

Example: `fmtcode`

```
F = fmtcode(html="H<sup>T</sup><sub>M</sub>L", pdf=r"\LaTeX")

Invocation of format code results in: {{F()}}.
```

Output

Invocation of format code results in: LaTeX.

Example: Color

```
C = lambda color: fmtcode(
    html="<span style='color:%s;'>{0}</span>" % color,
    pdf=r"\textcolor{{%s}}{{{0}}}" % color
)
B = C("blue")
R = C("red")

Text with {{B("blue")}} and {{R("RED")}}.
```

Output

Text with blue and RED.

Example: Classes

```
class color:
    def __init__(self, color):
        self.color = color
    def upper(self, x):
        return self.text(x.upper())
    def lower(self, x):
        return self.text(x.lower())

class html(color):
    def text(self, x):
        return f"<span style='color:{self.color};'>{x}</span>"

class pdf(color):
    def text(self, x):
        return rf"\textcolor{{{self.color}}}{{{x}}}"

CC = lambda x: fmtcode(html=html(x), pdf=pdf(x))
BB = CC("blue")
RR = CC("red")

Text with {{BB.upper("blue")}} and {{RR.lower("RED")}}.
```

Output

Text with BLUE and red.

5 Meta Data in Front Matter

Meta data is annotated in the front matter of a Markdown text document. The front matter must start in the first line with `---` and precedes all other text being fenced by `---`. The meta data is in `yaml` format. The `yaml` block is parsed using `python-pyyaml`. All meta data is imported into the preprocessed document.

5.1 Pandoc Front Matter

Example

```yaml
---
title: My Document
date: 2022-01-01
author: ...
link-citations: true
bibliography: data/marky.bib
header-includes: >
   \hypersetup{colorlinks=false,
   allbordercolors={0 0 0},
   pdfborderstyle={/S/U/W 1}}
xnos-cleveref: true
xnos-capitalise: true
fontsize: 11pt
---
```

The meta data fields `title`, `date`, `author`, `link-citations`, `bibliography` and `header-includes` are processed by `pandoc` during document rendering. `fontsize` adjusts the font size in `html` and `pdf` documents. The `xnos-cleveref` and `xnos-capitalise` fields are used by the `pandoc-xnos` extensions for referencing figures, tables, sections and equations.

5.2 `marky` Format Fields

Example

```yaml
---
header-includes--pdf: >
  \hypersetup{
  colorlinks=false,
  allbordercolors={0 0 0},
```

```
  pdfborderstyle={/S/U/W 1}}
header-includes--html: >
  <style>* { box-sizing: border-box; }</style>
---
```

The pandoc `header-includes` field is used for `pdf` and `html` documents, therefore it must contain corresponding tex and `html` code.

The field `header-includes` ending with `--pdf` or `--html` specifies corresponding options for the generation of `pdf` and `html` documents. During make, `marky` scans all meta data fields, and fields which end with `--pdf` and `--html` are selected and forwarded to `pandoc` based on the format to be rendered.

6 Scientific Writing in Markdown

Markdown is a markup language for technical writing, with emphasis on readability. Markdown can be rendered in many formats including `html` and `pdf` by using `pandoc` for example.

Using various Markdown extensions of `pandoc` a sufficient structure for writing scientific documents is reflected using Markdown syntax. `marky` by default uses the following `pandoc` Markdown extensions.

- parsing extensions
  - all_symbols_escapable
  - intraword_underscores
  - escaped_line_breaks
  - space_in_atx_header
  - lists_without_preceding_blankline
- styling extensions
  - inline_code_attributes
  - strikeout
- structuring extensions
  - yaml_metadata_block
  - pipe_tables
  - line_blocks
  - implicit_figures
  - abbreviations
  - inline_notes
- code injection

- raw_html
- raw_tex

`pandoc` supports equations rendered inline and single-line in tex-style using `$...$` and `$$...$$`, bibliography using the `--citeproc` option, section numbering using the `--number-sections` option and table of contents using the `--table-of-contents` option.

`pandoc` supports `xnos` filters for referencing document content like figures, equations, tables, sections by using the `--filter pandoc-xnos` option. `xnos` integrates clever references, which means "Fig.", "Sec.", "Eq." and "Tab." are added automatically to the corresponding element. If the prefix is to be omitted, the reference is written as `\!@ref:label`.

Example

```
## Referenced Section {#sec:label}

This is a reference to @sec:label.

![This is the caption](
UhEUgAAAAUAAAAFCAYAAACNbyblAAAAHElEQVQI12P4//8/w38GIAXDIBKE0DH
xgljNBAAO9TXL0Y4OHwAAAABJRU5ErkJggg==){#fig:label}

This is a reference to @fig:label.

A  |B  |C  |D
---|---|---|---
000|111|444|555
222|333|666|777

Table: This is the caption {#tbl:label}

This is a reference to @tbl:label.

$$\mbox{e}^{\mbox{i}\pi}+1=0$${#eq:label}

This is a reference to @eq:label.

This is a citation [@Muller1993].
```

The file `marky.bib` is specified in the meta data in the front matter of the Markdown text.

6.1 Referenced Section

This is a reference to Section 6.1.

Figure 1: This is the caption

This is a reference to Fig. 1.

Table 2: This is the caption.

A	B	C	D
000	111	444	555
222	333	666	777

This is a reference to Table 2.

$$e^{i\pi} + 1 = 0 \tag{1}$$

This is a reference to Eq. 1.

This is a citation (Muller 1993).

References

Muller, Peter. 1993. "The Title of the Work." *The Name of the Journal* 4 (2): 201–13.