Flask-Commonmark

Release 1.0.4

Doug Shawhan

Jun 27, 2021

CONTENTS:

Add CommonMark processing filter to your *Flask* app. https://commonmark.org/

CHAPTER

ONE

FLASK_COMMONMARK

Commonmark filter class for Flask. One may notice a similarity to Dan Colish's Flask-Markdown, from which I shamelessly copied a bunch of this. Does not have all the nice provisions for extension baked in, but probably does what you need. See https://commonmark.org/ for details.

Usage

```
from flask_commonmark import Commonmark
cm = Commonmark(app)

# or, if you are using the factory pattern

cm = Commonmark()
cm.init_app(app)

# Create routes in the usual way
@app.route("/commonmark")
def display_commonmark():
    mycm = u"Hello, *commonmark* block."
    return render_template("commonmark.html", mycm=mycm)
```

Templates

```
# one can just place raw markdown in the template. The filter expects
# your markdown to be fully left-aligned! Otherwise expect plaintext.
{% filter commonmark %}
# Nagasaki
1. Chew Terbaccy
1. Wicky-waky-woo
{% endfilter %}

# block style
{% filter commonmark %}{{ mycm }}{% endfilter %}

# inline style
{{mycm|commonmark}}
```

class flask_commonmark.**Commonmark**(*app: Union[bool, flask.app.Flask] = False*, *auto_escape: bool = False*)
Bases: object

Wrapper class for Commonmark (aka "common markdown"), objects.

Parameters

- **app** (*obj*) – Flask app instance
- **auto_escape** (*bool*) – Use Jinja2 auto_escape, default False

__build_filter(*app_auto_escape: bool*) → Callable
Jinja2 __build_filter

Parameters **app_auto_escape** (*bool*) – auto_escape value (default False)

Returns context filter

Return type commonmark_filter (obj)

init_app(*app: Union[bool, flask.app.Flask]*, *auto_escape: bool = False*)
Create parser and renderer objects and auto_escape value. Set filter.

Parameters

- **app** (*Union[bool, flask.app.Flask]*) – Flask app
- **auto_escape** (*bool*) – Shall we auto escape?

class flask_commonmark.**Markup**(*base: Any = ''*, *encoding: Optional[str] = None*, *errors: str = 'strict'*)
Bases: str

A string that is ready to be safely inserted into an HTML or XML document, either because it was escaped or because it was marked safe.

Passing an object to the constructor converts it to text and wraps it to mark it safe without escaping. To escape the text, use the *escape()* class method instead.

```
>>> Markup("Hello, <em>World</em>!")
Markup('Hello, <em>World</em>!')
>>> Markup(42)
Markup('42')
>>> Markup.escape("Hello, <em>World</em>!")
Markup('Hello &lt;em&gt;World&lt;/em&gt;!')
```

This implements the __html__() interface that some frameworks use. Passing an object that implements __html__() will wrap the output of that method, marking it safe.

```
>>> class Foo:
...     def __html__(self):
...         return '<a href="/foo">foo</a>'
...
>>> Markup(Foo())
Markup('<a href="/foo">foo</a>')
```

This is a subclass of str. It has the same methods, but escapes their arguments and returns a Markup instance.

```
>>> Markup("<em>%s</em>") % ("foo & bar",)
Markup('<em>foo &amp;amp; bar</em>')
>>> Markup("<em>Hello</em> ") + "<foo>"
Markup('<em>Hello</em> &lt;foo&gt;')
```

capitalize()
Return a capitalized version of the string.

More specifically, make the first character have upper case and the rest lower case.

center(*width*, *fillchar=' '*, */*)
Return a centered string of length width.

Padding is done using the specified fill character (default is a space).

classmethod escape(*s: Any*) → *markupsafe.Markup*
Escape a string. Calls *escape()* and ensures that for subclasses the correct type is returned.

expandtabs(*tabsize=8*)
Return a copy where all tab characters are expanded using spaces.

If tabsize is not given, a tab size of 8 characters is assumed.

format(**args*, ***kwargs*) → str
Return a formatted version of S, using substitutions from args and kwargs. The substitutions are identified by braces ('{' and '}').

join(*seq: Iterable[Union[str, HasHTML]]*) → *Markup*
Concatenate any number of strings.

The string whose method is called is inserted in between each given string. The result is returned as a new string.

Example: '.'.join(['ab', 'pq', 'rs']) -> 'ab.pq.rs'

ljust(*width*, *fillchar=' '*, */*)
Return a left-justified string of length width.

Padding is done using the specified fill character (default is a space).

lower()
Return a copy of the string converted to lowercase.

lstrip(*chars=None*, */*)
Return a copy of the string with leading whitespace removed.

If chars is given and not None, remove characters in chars instead.

partition(*sep: str*) → Tuple[*markupsafe.Markup*, *markupsafe.Markup*, *markupsafe.Markup*]
Partition the string into three parts using the given separator.

This will search for the separator in the string. If the separator is found, returns a 3-tuple containing the part before the separator, the separator itself, and the part after it.

If the separator is not found, returns a 3-tuple containing the original string and two empty strings.

replace(*old*, *new*, *count=-1*, */*)
Return a copy with all occurrences of substring old replaced by new.

count Maximum number of occurrences to replace. -1 (the default value) means replace all occurrences.

If the optional argument count is given, only the first count occurrences are replaced.

rjust(*width*, *fillchar=' '*, */*)
Return a right-justified string of length width.

Padding is done using the specified fill character (default is a space).

rpartition(*sep: str*) → Tuple[*markupsafe.Markup*, *markupsafe.Markup*, *markupsafe.Markup*]
Partition the string into three parts using the given separator.

This will search for the separator in the string, starting at the end. If the separator is found, returns a 3-tuple containing the part before the separator, the separator itself, and the part after it.

If the separator is not found, returns a 3-tuple containing two empty strings and the original string.

rsplit(*sep: Optional[str] = None*, *maxsplit: int = -1*) → List[*markupsafe.Markup*]
Return a list of the words in the string, using sep as the delimiter string.

sep The delimiter according which to split the string. None (the default value) means split according to any whitespace, and discard empty strings from the result.

maxsplit Maximum number of splits to do. -1 (the default value) means no limit.

Splits are done starting at the end of the string and working to the front.

rstrip(*chars=None*, */*)
Return a copy of the string with trailing whitespace removed.

If chars is given and not None, remove characters in chars instead.

split(*sep: Optional[str] = None*, *maxsplit: int = -1*) → List[*markupsafe.Markup*]
Return a list of the words in the string, using sep as the delimiter string.

sep The delimiter according which to split the string. None (the default value) means split according to any whitespace, and discard empty strings from the result.

maxsplit Maximum number of splits to do. -1 (the default value) means no limit.

splitlines(*keepends: bool = False*) → List[*markupsafe.Markup*]
Return a list of the lines in the string, breaking at line boundaries.

Line breaks are not included in the resulting list unless keepends is given and true.

strip(*chars=None*, */*)
Return a copy of the string with leading and trailing whitespace removed.

If chars is given and not None, remove characters in chars instead.

striptags() → str
unescape() the markup, remove tags, and normalize whitespace to single spaces.

```
>>> Markup("Main &raquo;  <em>About</em>").striptags()
'Main » About'
```

swapcase()
Convert uppercase characters to lowercase and lowercase characters to uppercase.

title()
Return a version of the string where each word is titlecased.

More specifically, words start with uppercased characters and all remaining cased characters have lower case.

translate(*table*, */*)
Replace each character in the string using the given translation table.

table Translation table, which must be a mapping of Unicode ordinals to Unicode ordinals, strings, or None.

The table must implement lookup/indexing via __getitem__, for instance a dictionary or list. If this operation raises LookupError, the character is left untouched. Characters mapped to None are deleted.

unescape() → str
Convert escaped markup back into a text string. This replaces HTML entities with the characters they represent.

```
>>> Markup("Main &raquo; <em>About</em>").unescape()
'Main » <em>About</em>'
```

upper()
Return a copy of the string converted to uppercase.

zfill(*width*, */*)
Pad a numeric string with zeros on the left, to fill a field of the given width.

The string is never truncated.

flask_commonmark.**pass_eval_context**(*f: jinja2.utils.F*) → jinja2.utils.F
Pass the `EvalContext` as the first argument to the decorated function when called while rendering a template. See eval-context.

Can be used on functions, filters, and tests.

If only `EvalContext.environment` is needed, use `pass_environment()`.

New in version 3.0.0: Replaces `evalcontextfunction` and `evalcontextfilter`.

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX