Stencil Documentation

Release 2.0.0

Curtis Maloney

Jul 26, 2017

Contents

1 Writing Templates **3**
- 1.1 Expressions . . . 3
- 1.2 Comments . . . 4
- 1.3 Variables . . . 4
- 1.4 Block Tags . . . 4

2 Using Templates **9**
- 2.1 From Strings . . . 9
- 2.2 From a file . . . 9
- 2.3 Context . . . 10

3 Extending stencil **11**
- 3.1 Filters . . . 11
- 3.2 Tags . . . 12

4 Why? **15**

Contents:

CHAPTER 1

Writing Templates

Templates are just plain text files, with special notation (called 'tags') to indicate where the engine should take action.

There are 3 basic types of tags:

- var
- block
- comment

Expressions

At several points in the syntax, an Expression can be used.

They start with a value, optionally followed by a series of filters.

A value can be:

- an integer
- a float
- a string
- a lookup

A lookup will try to delve into the Context. For example the expression `name` will look for Context['name'].

However, lookups can delve deeper. They will attempt dict lookups, attribute lookup, and list indexing (in that order). Also, if the resulting value is callable, it will be called.

So, to get the `name` attribute of your `user` object, and call its `title` method, the expression would be `name:user:title`.

Filters allow you to pass the value (and possibly more arguments) to helper functions. For example, you might have a dollar format function:

```
def dollar_format(value, currency_symbol='$'):
    return "%s%0.2f" % (currency_symbol, float(value))

# Register the filter
stencil.FILTERS['dollar_format'] = dollar_format
```

You can now in your templates use the expression `product:price|dollar_format`, or even override the currency symbol using `product:price|dollar_format:'¥'`.

Filters can be chained, one after another.

There are currently no default filters provided with Stencil.

Comments

Comment tags are discarded during parsing, and are only for the benefit of future template editors. They have no impact on rendering performance.

Variables

Var tags are used to display the values of variables. They look like this:

```
Hello {{ expr }}
```

Block Tags

Block tags perform some action, may render some output, and may "contain" other tags.

```
{% include 'another.html' %}
```

Built In Tags

for

The `for` tag allows you to loop over a sequence.

```
{% for x in expr %}
...
{% endfor %}
```

The `for` tag also support and `else` block. It will be used if sequence to be iterated is empty.

```
{% for x in empty_list %}
...
{% else %}
Nothing to show.
{% endfor %}
```

if

The `if` tag allows for simple flow control based on a truthy test.

```
{% if expr %}
Success!
{% endif %}
```

It also supports negative cases:

```
{% if not expr %}
Failure!
{% endif %}
```

And, like the `for` tag, it supports an `else` block:

```
{% if expr %}
Success!
{% else %}
Failure!
{% endif %}
```

"Truthiness" is based on the Pythocept. Here are some things that are "truthy":

- True
- non-empty strings
- non-empty lists or dicts
- non-zero values

Conversely, things that are "falsy" are:

- False
- empty strings
- 0 and 0.0
- empty lists and dicts

include

The `include` tag lets you render another template inline, using the current context.

```
{% include expr %}
```

Additionally, you can pass extra expressions to be added to the context whilst the other template is being rendered.

```
{% include form_field.html field=current_field %}
```

load

This tag lets you load other code modules to add new tags to use in this template. See *Tags* for more details.

```
{% load 'myproject.tags' %}
```

The value passed is a Python import path.

extends and block

The `extends` tag allows the use of template inheritance. A *base* template can denote `blocks` of content which can be overridden by templates which `extend` it.

> **Caution:** The `extends` tag only works properly if it is the *very first* thing in your template.

Say we have the following base template:

```
<!DOCTYPE html>
<html lan="en">
    <head>
        <title>{% block title %}Welcome!{% endblock %}</title>
        <link rel="stylesheet" type="text/css" href="/static/css/base.css">
        {% block extra_head %}{% endblock %}
    </head>
    <body>
        <header>
            <h1>{% block header %}Welcome!{% endblock %}</h1>
        </header>
        <main>
        {% block content %}{% endblock %}
        </main>
        <footer>
            <p>&copy; 2016 Me!</p>
        </footer>
        {% block footer_scripts %}{% endblock %}
    </body>
</html>
```

Now, when rendered itself, it will show as:

```
<!DOCTYPE html>
<html lan="en">
    <head>
        <title>Welcome!</title>
        <link rel="stylesheet" type="text/css" href="/static/css/base.css">

    </head>
    <body>
        <header>
            <h1>Welcome!</h1>
        </header>
        <main>


        </main>
        <footer>
            <p>&copy; 2016 Me!</p>
        </footer>

    </body>
</html>
```

However, if we write another template which extends this one, we just have to write now the `blocks` we want to override:

```
{% extends base.html %}

{% block title %}My Title!{% endblock %}

{% block content %}
Welcome to my first page!
{% endblock %}
```

This will override only the two given blocks content.

Any content outside of `block` tags will be ignored.

with

Using `with` you can temporarily assign new values in the context from expressions. This can help avoid repeated work.

```
{% with url=page|make_url %}
<a href="{{ url }}" class="link {% if url|is_current_url %}current{% endif %}">{{
↪page:title }}</a>
{% endwith %}
```

case/when

Allows switching between multiple blocks of template depending on the value of a variable.

```
{% case foo.bar %}
{% when 1 %}
You got one!
{% when 2 %}
You got two!
{% else %}
You got some!
{% endcase %}
```

The optional *{% else %}* clause is used if no when cases match.

CHAPTER 2

Using Templates

To use stencil templates there is very little to do.

From Strings

To build a tempate from a string, just create a `stencil.Template` instance:

```
>>> from stencil import Template

>>> t = Template('''Hello, {{name}}!''')
```

And to render it:

```
>>> t.render({'name': 'Bob'})
'Hello, Bob!'
```

From a file

First you'll need to create a `TemplateLoader`, passing it a list of paths to search for templates.

```
>>> from stencil import TemplateLoader
>>> loader = TemplateLoader(['templates/'])
```

You can ask it to load a template freshly calling `TemplateLoader.load`

```
>>> t = loader.load('base.html')
```

The `TemplateLoader` can also cache loaded, parsed templates if you treat it as a dict:

```
>>> t = loader['base.html']
# Loads template from file.
>>> s = loader['base.html']
# Returns the same template instance.
```

Context

When rendering a template, you need to pass it a `Context` - this is the limit of information the template can access.

Custom filters

Filtering functions for applying to values in expressions can be defined in the globally shared dict `stencil.FILTERS`.

CHAPTER 3

Extending stencil

Stencil allows you to easily add new tags and filters.

Filters

As noted in *Custom filters*, you can easily register new filter functions.

Here is an example of adding an `escape` filter:

```
escape_html = lambda text: (
    text.replace('&', '&amp;')
        .replace("<", "&lt;")
        .replace(">", "&gt;")
        .replace('"', "&quot;")
        .replace("'", "&#x27;")
)


_js_escapes = {
    ord('\\'): '\\u005C',
    ord("'"): '\\u0027',
    ord('"'): '\\u0022',
    ord('>'): '\\u003E',
    ord('<'): '\\u003C',
    ord('&'): '\\u0026',
    ord('='): '\\u003D',
    ord('-'): '\\u002D',
    ord(';'): '\\u003B',
    ord('\u2028'): '\\u2028',
    ord('\u2029'): '\\u2029'
}

# Escape every ASCII character with a value less than 32.
_js_escapes.update((ord('%c' % z), '\\u%04X' % z) for z in range(32))
```

```
escape_js = lambda text: text.translate(_js_escapes)

def escape(value, mode='html'):
    if mode == 'html':
        return escape_html(value)
    elif mode == 'js':
        return escape_js(value)
    raise ValueError('Invalid escape mode: %r' % mode)

stencil.FILTERS['escape'] = escape
```

And we use it in our template:

```
<input type="text" value="{{ value|escape }}">
```

Now we can use it:

```
>>> from stencil import TemplateLoader, Context
>>> ctx = Context({'value': '<script>alert("BOO");</script>'})
>>> tmp.render(ctx)
u'<input type="text" value="&lt;script&gt;alert(&quot;BOO&quot;);&lt;/script&gt;">'
```

Tags

All tags derive from the `stencil.BlockNode` class, and self-register with `stencil` on declaration.

```
from stencil import BlockNode

class MyTag(BlockNode):
    name = 'my'  # This is matched in {% my %}
```

When `stencil` finds a tag matching this name, it will call the `BlockNode.parse` classmethod, passing it the rest of the tag content, and the template instance. This method must return a BlockNode sub-class instance.

```
class MyTag(BlockNode):

    @classmethod
    def parse(cls, content, parser):
        return cls(content)
```

The default action is to just return an instance of the class, passed the tag content.

When a template is rendered, a blocks `render` method will be called, passed a `Context` instance, and a file-like object to output to.

Tags with children

Some tags contain *child* nodes (e.g. `for`, `if`, `block`).

To do this they build a `Nodelist`:

```
class MyBlock(BlockNode):

    @classmethod
    def parse(self, content, parser):
        nodelist = parser.parse_nodelist({'endmyblock',})
        return cls(nodelist)
```

This will consume tags until it reaches one with a name found in the list. The tags are added to a `Nodelist` instance, except the matching one which it stored in `Nodelist.endnode`.

A `Nodelist` can be rendered easily by calling their `render` method, which works just like a `BlockNode`.

```
nodelist.render(context, output)
```

Expressions

To have an argument resolved as an expression, use the `parse_expression` function. This will parse then value passed, and construct an `Expression` instance.

Then in render, call `Expression.resolve(context)` to get its value.

For more fine grained parsing, and to parse `key=expr` syntax, use a `Tokens` class.

```
tokens = Tokens(content)
```

This provides several useful methods:

```
value = tokens.parse_argument()
```

Parses a single argument, be it a string, float or int literal, or a lookup. The result is suitable for passing as the second argument to `resolve_lookup`, or as the first to `Expression`.

```
value = resolve_lookup(value)
```

```
value, filters = tokens.parse_filter_expression()
```

Parse a filter expression, returning a value (as from `parse_argument`, and a list of (filter name, *args) tuples.

```
kwargs = tokens.parse_kwargs()
```

Parse *key=filter-expression* sequences, and construct a dict of *key: Expression()* items.

```
tokens.assert_end()
```

Asserts the current token to be parsed is an end marker, or raises and assertion error with a message showing where the token was.

CHAPTER 4

Why?

There are plenty of template engines in Python, and I've even written my own powerful, super-fast one (knights-templater), so why write another?

I was experimenting with AWS' Serverless concept, and was saddened to learn it only supports Python 2.7 currently. I wanted templating, but felt back-porting K-T to Py2 just wasn't warranted.

So I figured, why not see how small I can make a functional template language?

Apparently, "under 400 lines of code" is the answer...