Margate Documentation

Release 0.1

Tim Martin

Jul 04, 2017

Contents:

CHAPTER 1

Using from Django

Language compatibility

The Margate language is very similar in style to the built-in Django template engine, but differs in a number of important details.

Most importantly, `{{ }}` expressions (and expressions in `for` loop commands etc.) are treated as arbitrary Python code. This means that they are more flexible than Django template language, but prevents you from taking advantage of the shortcuts that automatically convert object attributes into dictionary member lookup.

For example, instead of writing:

```
{% for tag in blog_post.tags %}
...
{% endfor %}
```

if `blog_post` is a dictionary, you will need to write:

```
{% for tag in blog_post["tags"] %}
...
{% endfor %}
```

Another limitation is that none of the built-in filters are currently supported.

Configuring Django to use the engine

To enable Margate in Django, simply add it to the `TEMPLATES` in `settings.py`:

```
TEMPLATES = [
  {
    'BACKEND': 'margate.django.MargateEngine',
    'DIRS': [],
    'APP_DIRS': True
```

```
    }
]
```

CHAPTER 2

TODO List

Todo

This doesn't really belong in this module. It's here because we're combining two different types: block parser output and code generation.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/margate/envs/latest/lib/python3.5/site-packages/margate-0.0.1-py3.5.egg/margate/code_generation.py:docstring of margate.code_generation.Execution, line 1.)

CHAPTER 3

Reference

The process of building a template into a function has the following steps:

- The template is broken down into blocks (such as literal text and code execution) that are treated differently. This is handled by *the block parser*.
- The resultant sequence of blocks is passed to the *Parser* to be turned into a parse tree.
- The parse tree is processed by *code generation* to make Python bytecode.

Compiler

The compiler module contains the public interface to the library.

class `margate.compiler.`**`Compiler`**(*template_locator=None*)
The Compiler takes a template in string form and returns bytecode that implements the template.

`compile`(*source*)
Compile the template source code into a callable function.

Returns A callable function that returns rendered content as a string when called.

class `margate.compiler.`**`TemplateLocator`**
The template locator abstracts the details of locating templates when one template extends another (such as with the `{% extends %}` tag)self.

Code generation

This module contains the building blocks of the final template function, in the form of bytecode generators.

There are a series of classes in here that are used as nodes in the code generation tree, and each one implements a `make_bytecode()` method.

class margate.code_generation.**Sequence**
A sequence of nodes that occur in a parse tree. Elements in the sequence can themselves be sequences (thus forming a tree).

class margate.code_generation.**ForBlock**(*for_node*)

class margate.code_generation.**IfBlock**(*condition*)
The IfBlock generates code for a conditional expression.

This currently only includes literal *True* and *False* as expressions, and doesn't support an *else* branch.

class margate.code_generation.**ExtendsBlock**(*template*)

class margate.code_generation.**ReplaceableBlock**(*name*)

class margate.code_generation.**VariableExpansion**(*variable_name*)
A variable expansion takes the value of an expression and includes it in the template output.

class margate.code_generation.**Literal**(*contents*)

class margate.code_generation.**Execution**(*expression*)

Todo

This doesn't really belong in this module. It's here because we're combining two different types: block parser output and code generation.

Block parser

The block parser splits a template into the blocks that make it up. There are three different sorts of data in a template that get handled in different ways:

- Literal text, which just gets embedded in the output (but may be skipped or repeated by executing code around it).
- Executable code
- Embedded variable expressions that get expanded into text output.

It's implemented as a state machine, where the template starts out in literal text and transitions to a different state depending on whether it encounters `{{`, `}}`, `{%` or `%}`.

class margate.block_parser.**LiteralState**(*text*)
The literal state is the state the block parser is in when it is processing anything that will be included in the template output as a literal. The template starts out in literal state and transitions back into it every time a block is closed.

class margate.block_parser.**ExecutionState**(*text*)
Execution state is the state when any kind of code execution is occurring. This includes the start and ends of blocks.

class margate.block_parser.**ExpressionState**(*text*)
Expression state occurs when processing a `{{ ... }}` expression that embeds the value of an expression into the output.

Parser

The parser converts the template language into a usable structured form.

There are two layers to the parsing: breaking the template down into blocks (which is done by the *block_parser* module), and parsing the expressions that appear in the execution blocks within the template.

The parser in this module uses a combination of ad hoc parsing, funcparserlib and ast.parse. The top-level rules in the language (`if`, `for`, `endif` etc.) are handled ad hoc since they are not recursive. However, the expression that is given as an argument to `if` is an arbitrary expression and parsed

class `margate.parser.`**`Parser`**(*template_locator=None*)

The Parser is responsible for turning a template in "tokenised" form into a tree structure from which it is straightforward to generate bytecode.

The input is in the form of a flat list of atomic elements of the template, where literal text (of any length) is a single element, and a `{% %}` or `{{ }}` expression is a single element.

Figuring out nesting of starting and ending of loops happens within the parser.

`parse`(*tokens*)

Parse a token sequence into a *Sequence* object.

`margate.parser.`**`parse_expression`**(*expression*)

Parse an expression that appears in an execution node, i.e. a block delimited by `{% %}`.

This can be a compound expression like a `for` statement with several sub-expressions, or it can just be a single statement such as `endif`.

Parameters **`expression`** (*list*) – Tokenised expression.

CHAPTER 4

Introduction

Margate is a templating engine for Python that compiles templates down to Python bytecode. It is mostly Django-compatible in spirit, though it falls short of being a drop-in replacement for Django templates.

Early performance testing suggests that it is around 10 times faster than regular Django templates.

CHAPTER 5

Example

Simply instantiate a *Compiler* and call its *compile()* method with the template source:

```
template_source = """
<p>Hello {{ person }}, my name is {{ me }}
"""

compiler = margate.compiler.Compiler()
template_function = compiler.compile(template_source)
```

You now have a function that can be called to yield the rendered content. Pass variable values in keyword arguments:

```
print(template_function(person="alice",
                        me="a template"))
```

CHAPTER 6

FAQ

Why oh why?

Mostly to learn about Python bytecode.

You don't really expect the speed benefit to be worth it, do you?

Template rendering is extremely unlikely to be the bottleneck in your web application. Optimising it will at best save a constant overhead from each page view, and will have a proportionately lower impact on your slowest pages.

On the other hand, it's free speed. It can probably save you a few milliseconds per page view, which might help when you're trying to get your landing page to load as fast as possible. Assuming the templating language has all the same features, why wouldn't you? Template expansion probably can't be parallelised with anything else your web app is doing, so miliseconds here contribute directly to the bottom line.

What's with the name?

The library was originally called Stencil, but it turns out that lots of people call their templating library Stencil, so I had to change.

I hate spending time thinking of names for projects, so when I get stuck I just use the name of an English seaside town. There are plenty of them and they are reasonably unique and memorable names.

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Python Module Index

m

Index