Knights Templater Documentation

Release 0.1

Curtis Maloney

February 06, 2016

Contents

Contents:

CHAPTER 1

First Steps

Install using pip:

```
$ pip install knights-templater
```

Compile a template from a string:

```
>>> from knights import kompile
>>> t = kompile('''Hello {{ name }}, how are you?''')
>>> t
<Template object at 0x101362358>
```

Template objects are callable. To render, just call them with a dict of values for their rendering context.

```
>>> t({'name': 'Bob'})
'Hello Bob, how are you?'
```

The {{ var }} token supports Python syntax, and so is very powerful:

```
>>> t = kompile('''Hello {{ name.title() }}!''')
>>> t({'name': 'wally west'})
'Hello Wally West!'
```

The rendering process will cast everything to strings, so you don't have to.

```
>>> t = kompile('''Progress: {{ done * 100.0 / total }}% ''')
>>> t({'total': 300, 'done': 180})
'Progress: 60.0% '
```

Note, however, this is done only after the expression is evaluated.

```
>>> t({'total': 300, 'done': 'some'})
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<compiler>", line 1, in __call__
  File "<compiler>", line 1, in _root
TypeError: can't multiply sequence by non-int of type 'float'
```

CHAPTER 2

Flow Control

By default the syntax supports if/else and for.

2.1 If/Else

Simple `if` expressions work just like in Python:

```
{% if value is True %}It is true!{% endif %}
```

You can also use `else`:

```
Today is an {% if date // 2 %} even {% else %} odd {% endif %} date.
```

Currently `elif` is not supported – nested `if` in `else` is the best solution.

2.2 For

The `for` block works just like in Python also.

```
{% for item in sequence %}{{ item }}{% endfor %}
```

Just like in Python, it can unpack items in a sequence:

```
{% for key, value in mydict.items() %}{{ key }}: {{ value }}{% endfor %}
```

CHAPTER 3

Helpers

Template are compiled in their own module, with almost nothing else in their namespace.

Some extra helper functions are available inside the _ object.

```
{{ _.escape_html(foo) }}
```

Additional helpers can be loaded using the `{% load %}` tag.

CHAPTER 4

Inheritance

Templates are able to extend other templates, allowing you to avoid duplication.

The base template much declare `{% block %}` which can be overriden. Each block has an unique name, which becomes a method on the class.

```
<!DOCTYPE html>
<html>
    <head>
        <title>{% block title %}default title{% endblock %}</title>
    </head>
    <body>{% block content %}{% endblock %}</body>
</html>
```

Any blocks not overriden by an inheriting template will default to their parent classes implementation, as you'd expect.

```
{% extends base.html %}
{% block title %}My Title {% endblock%}
```

The `super` tag allows you to access blocks from the parent class.

```
{% extends base.html %}
{% block title %}{% super title %} - Extra title{% endblock %}
```

CHAPTER 5

Loading Templates

Templates can be loaded from files. The `loader.TemplateLoader` class makes it easy to access a list of directories for templates.

First you need to create a `TemplateLoader`

```
>>> from knights.loader import TemplateLoader
>>> loader = TemplateLoader(['templates'])
```

The list of paths provided will be resolved relative to the CWD.

Now you can ask the loader to find a template in any of the supplied directories:

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

Additionally, the loader will act as a cache if used like a dict:

```
>>> t = loader['index.html']  # Will load and parse the class
>>> s = loader['index.html']  # Will reuse the existing instance
```

CHAPTER 6

Advanced Usage

Since Templates are Python classes, you can use them like any other class.

6.1 Inheriting

You can add your own methods for providing extra functionality just like with any other Python class.

If you want your method to work with *{% block %}* it must accept `context` as its first positional argument, and will be called in a *yield from* clause:

```
yield from self.blockname(context)
```

6.2 Re-use

Logically, you can also call blocks on a template the same way. Pass a dict-like object as context, and they return a generator.

CHAPTER 7

Loader

Load templates from files.

class knights.loader.**TemplateNotFound**(*Exception*)
The exception class raised when a template can not be found.

class knights.loader.**TemplateLoader**(*paths*)
Provides a way to load templates from a list of directories.

paths is a list of paths to search for template files. Relative paths will be resolved relative to CWD.

A TemplateLoader will also act as a cache if accessed as a dict.

```
>>> loader = TemplateLoader(['templates'])
>>> tmpl = loader['index.html']
```

load(*name*, *raw=False*)
Find the template *name* in one of the *paths* and compile it.

If the *raw* flag is passed, the returned value will be the class, not an instance.

CHAPTER 8

Default Helpers

Helpers provide utility functions and classes for templates.

Any loaded template library can provide additional helpers.

knights.helpers.**stringfilter**(*func*)
A decorator which ensures the first argument to func is cast as a str()

The default set of helpers are as follows:

knights.helpers.**addslashes**(*value*)
Escape , ", and ' characters by placing a before them.

knights.helpers.**capfirst**(*value*)
Capitalise only the first character of the string.

knights.helpers.**escape**(*value*, *mode='html'*)
Escape unsafe characters in the string.

By default applies HTML escaping on &, <, >, " and ' characters, using HTML entities instead.

Also supports 'js' mode.

CHAPTER 9

Default Tags

Default tags.

load name

```
{% load foo.bar.baz %}
```

Loads template library `name`.

The name is treated as a python import path, and the loaded module is expected to have a 'register' member, being an instance of :library:Library

extends name

```
{% extends base.html %}
```

Make this Template class extend the template in `name`

super name

```
{% super blockname %}
```

Renders the contents of `blockname` from the parent template.

block name

```
{% block foo %}
...
{% endblock %}
```

Declares a new overridable block in the template.

This will result in a new method on the class of the same name, so names must be valid Python identifiers.

if expr

Implements a simple if conditional.

```
{% if ...expr... %}
...
{% endif %}
```

Optionally, it may contain an else:

```
    {% if ...expr... %}
    ...
    {% else %}
    ...
    {% endif %}
```

knights.tags.**for**()

```
    {% for a in sequence %}
    ...
    {% endfor %}
```

A python compatible {% for %} tag.

```
    {% for a, b, c in foo.get(other=1) %}
    ...
    {% endfor %}
```

The target values will be stacked on the scope for the duration, and removed once the loop exits.

Also you can provide an 'empty' block for when the list is empty.

```
    {% for a in sequence %}
    ...
    {% empty %}
    sequence is empty
    {% endfor %}
```

knights.tags.**include**()

Include another template in situ, using the current context.

```
    {% include "othertemplate.html" %}
```

Optionally, you can update the context by passing keyword arguments:

```
    {% include "other.html" foo=1, bar=baz * 6 %}
```

knights.tags.**with**()

Temporarily augment the current context.

```
    {% with ...kwargs... %}
    ...
    {% endwith %}
```

CHAPTER 10

Library

Provides a class for defining custom template utility libraries.

class `knights.library.`**Library**

tags = {}
A dict of template tag handler functions

helpers = {}
A dict of helpers to be added to the template modules global scope.

10.1 Custom tags

To define a custom tag or helper, you first need a library.

```
from knights.library import Library

register = Library()
```

You must call the instance `regsiser` as the Parser will only look for that name, currently.

Next, you can register helpers as follows:

```
@register.helper
def addone(value):
    return value + 1
```

If the name you want to use is reserved or a builtin, you can pass it to the decorator:

```
@register.helper(name='sun')
def addone(value):
    return value + 1
```

Custom tag handlers are more complex, as they require you to construct AST. Howerver, they are just as simple to register.

```
@register.tag
def mytag(parser, token):
   ...
```

Tags are parsed the parser, and the rest of the token text after their name was split from the front.

CHAPTER 11

Internals

11.1 Lexer

The Lexer processes the source string by breaking it into a sequence of Tokens.

class knights.lexer.**TokenType**
An Emum of token types. Valid values are:

- comment
- text
- var
- block

class knights.lexer.**Token**

mode
A TokenType.

content
The raw text content of the token.

lineno
An estimate of the source line.

knights.lexer.**tokenise**(*source*)
A generator yielding Tokens from the source.

This uses *re.finditer* to break up the source string for tag, var and comments, inferring text nodes between.

11.2 Parser

The Parser processes the Token stream from the *lexer* and produces a Template class.

class knights.parser.**Parser**(*source*)
Used to parse token streams and produce a template class.

stream
A lexer.tokenise generator.

parent = None
The parent class to use for this Template

methods
A list of ast.Method nodes

tags
Tag generators loaded from tag libraries.

helpers
Helper functions loaded from tag libraries.

load_library(*path*)
Load a custom tag library from `path`.

It is assumed it will contain an object called *register*, which will have dict properties *tags* and *helpers* to be updated with the parsers.

build_method(*name*, *endnodes=None*)
Build a new method called *name*, and make its body the set of nodes up to any listed in *endnodes*, or the end if None.

The method is appended to self.methods

parse_node(*endnodes=None*)
Yield a stream of AST Nodes based on self.stream

text nodes will produce effectively:

```
yield token
```

var nodes will produce code to evaluate the expression and yield their value.

block nodes will be resolved through the registered tags, unless they match any listed in `endnodes`, in which case the raw name will be yielded before terminating the loop.

parse_nodes_until(**endnodes*)
Return two values - a list of nodes, and the name of the matching end node. This is used for implementing

build_class()
Construct a Class definition from the current state.

The base class will either be 'object' if self.parent is None, else 'parent'.

parse_expression(*expr*)
Helper method to parse an expression and convert raw variable references to context lookups.

parse_args(*expr*)
Helper method to parse an expression and yield a list of args and kwargs.

knights.parser.**wrap_name_in_context**(*name*)
Utility function to turn an ast.Name() node into code to affect:

```
context['name']
```

class knights.parser.**VarVisitor**
A subclass of :ast:NodeTransformer which applies *wrap_name_in_context* to all Name nodes in the AST it visits.

11.3 Compiler

Contains the main compilation function for driving template construction.

To avoid clashing with the built-in *compile* this method is called *kompile*.

knights.compiler.**kompile**(*source*, *raw=False*)

Compiles a template from the provided source string.

If `raw` is True, the template class will be returned, not an instance.

Constructs a `knights.parser.Parser` class, loads the default *tags* and *helpers*, and builds a `__call__` method for the class which is effectively:

```
    return ''.join(str(x) for x in self._root(context))
```

where `context` is the sole argument to `__call__`.

Next, it calls parser.build_method('_root') to consume the tokens.

If, after this, parser.parent is not None, it will remove the _root method, relying on the parent to provide one.

Next is calls parser.build_class(), wraps it in an ast.Module, and compiles it.

Finally it executes the code, putting the parser.parent and parser.helpers into the global context.

It returns the 'Template' object from the resulting global context.

CHAPTER 12

1.3 (2016-02-06)

Features:

- Replaced *loader.load_template* with *loader.TemplateLoader*

  All templates that need to load templates [*extends*, *include*, etc] MUST now have a loader passed to their Parser.

Fixes:

- Don't resolve builtins via the context

1.2 (2015-05-16)

Syntax Changes

- Make `extends` and `load` require string argument.

Optimisations

- Inline _iterable into __call__
- Changed escape_html and escape_js to lambdas

Fixes:

- Count line numbers from 1
- Set line numbers on all nodes returned from parse_node

CHAPTER 14

1.1 (2015-04-28)

Features:

- Added _iterator() method
- Added {% super %} tag
- Added `static`, `now` and `url` helpers to compat.django
- Fixed Django engine wrapper, and renamed to dj
- `loader.load_template` now calls *os.path.abspath* on dirs

Fixes:

- Moved more code generation into Parser from kompiler.
- Removed debug flag
- Don't look up 'self' through context
- Fixed non-trivial for source values

CHAPTER 15

1.0 (2015-04-20)

Initial release.

CHAPTER 16

Requirements

Python 3.4+

CHAPTER 17

Introduction

Knights Templater is a light-weight, super-fast template engine which compiles your templates into Python classes. The syntax is based on Django's DTL, but as it allows raw Python the need for filter piping has been obviated.

CHAPTER 18

Quick Start

Compile and render a template from a string:

```
>>> import knights
>>> tmpl = knights.kompile('Hello {{ name }}, how are you?')
>>> print(tmpl({'name': 'Bob'}))
Hello Bob, how are you?
```

Load a template from a directory:

```
>>> from knights.loader import TemplateLoader
>>> loader = TemplateLoader(['templates'])
>>> tmpl = loader['index.html']
>>> tmpl({....})
...
```

Since WSGI wants an iterable for its content:

```
>>> content = tmpl._iterator(context)
```

CHAPTER 19

Thanks

Many thanks to Markus Holterman for soundboarding many of my ideas.

See Green Tree Snakes for an excellent introductin to Python AST.

Logo provided by Cherry Jam

k