django-templateaddons documentation

Release 0.1

Benoît Bryon

October 21, 2016

Contents

django-templateaddons is a set of tools for use with Django's templates, template tags and context processors. It provides additional template tags, context processors and utilities for template tag development.

CHAPTER 1

Links

- Follow the project on BitBucket at http://bitbucket.org/benoitbryon/django-templateaddons
- Read this documentation in HTML format at http://packages.python.org/django-templateaddons/

CHAPTER 2

Contents

2.1 Installation

The code is published under the BSD license. See LICENSE for details.

2.1.1 Requirements

This application requires:

- Django-1.2.1. The application targets the latest stable release of branch 1.x. It may work with previous versions, but tests focus on the latest one.

2.1.2 Get the code

The code is published under the BSD license. See LICENSE.txt for details.

Automatic installation

This application is known as "django-templateaddons" on Pypi.

So you can install it with pip or easy_install. As an example:

```
# pip install -U django-templateaddons
```

Manual installation

The main project page is http://bitbucket.org/benoitbryon/django-templateaddons.

You can clone the repository with the following command line:

```
$ hg clone http://bitbucket.org/benoitbryon/django-templateaddons
```

Copy the templateaddons folder somewhere in your PYTHON_PATH. It may be in your project's directory.

2.1.3 Update settings

- add 'templateaddons' to your INSTALLED_APPS

2.2 Template tag libraries

This application provides several template tags packaged into libraries.

2.2.1 Assign

The "assign" template tag library provides 1 template tag:

- assign: captures the output of template code and saves it in a context variable. Requires a {% endassign %} closing tag.

assign

The {% assign %} template tag is useful when you want to capture some template code output and use the result later.

It captures whatever is between the {% assign %} and {% endassign %} pair.

It takes two optional input parameters:

- name. A string. The context variable name where to store the result. Defaults to "assign".
- silent. A boolean. Whether to only capture the output or both capture and display it. Defaults to False.

The following template code:

```
{% load assign %}
{% assign name="sample_code" %}1234{% endassign %}
5678
{{ sample_code }}
```

... gives the following output:

```
5678
1234
```

Note: the {% assign %} template tag allows you to override the value of an existing context variable, so choose the "name" parameter with care.

The default value for the "name" parameter is "assign". This means that we could have written the previous example as below:

```
{% load assign %}
{% assign %}1234{% endassign %}
5678
{{ assign }}
```

You can set the "silent" parameter to False if you want to capture and display the output at the same time. The following template code:

```
{% load assign %}
{% assign name="sample_code" silent=0 %}1234{% endassign %}
5678
{{ sample_code }}
```

... gives the following output:

```
1234
5678
1234
```

2.2.2 Counter

The "counter" template tag library provides 2 template tags:

- counter: increments a counter on each call

counter

The {% counter %} template tag is useful when you want to use a custom counter. It can be used outside a specific loop, or used over multiple loops...

The {% counter %} template tag accepts several optional parameters:

- name. A string. It identifies the counter. Default value is 'default'.
- start. An integer. The value of the counter at the very first call. Default value is 0.
- step. An integer. The increment or decrement amount. Default value is 1.
- ascending. A boolean. Whether the counter is incremented (True) or decremented (False). Default value is True.
- silent. Whether to render (False) or not (True) the current counter value. Default value is False.
- assign. A string. If not empty, the current value of the counter is assigned to the context variable with the corresponding name. Default value is "" (do not assign).

Notice that the start, ascending and step values are only parsed on counter initialization.

The list of available counters is stored in the context, under the variable name settings.TEMPLATEADDONS_COUNTERS_VARIABLE, which is "_templateaddons_counters" by default.

2.2.3 Heading

The "heading" template tag library provides 1 template tag:

- headingcontext: helps you manage heading levels in HTML code. Requires a {% endheadingcontext %} closing tag.

headingcontext

With cascading templates, includes and bases, some parts of template code could be reused in different heading contexts.

As an example, consider the following "home page" code in home.html:

```
<h1>My beautiful website</h1>
<h2>News</h2>
<p>... links to news ...</p>
```

And consider the following "news page" code in news.html

```
<h1>News</h1>
<p>... links to news ...</p>
```

You cannot reuse (include) news.html code into home.html, because the heading level does not match.

The "headingcontext" template tag allows you to solve this problem. Here is modified home.html:

```
{% load heading %}
<h1>My beautiful website</h1>
{% headingcontext %}
{% include "news.html" %}
{% endheadingcontext %}
```

Ok. Now, what if the news.html code was using h5 in place of h1 like that:

```
<h5>News</h5>
<p>... links to news ...</p>
```

You can use the additional "source_level" parameter in home.html:

```
{% load heading %}
<h1>My beautiful website</h1>
{% headingcontext source_level=5 %}
{% include "news.html" %}
{% endheadingcontext %}
```

This causes all heading of level 5 and greater in news.html to be relative to the current heading level (2).

You can use nested {% headingcontext %}{% endheadingcontext %} calls. As an example, news.html could be:

```
<h5>News</h5>
<p>... links to news ...</p>
{% load heading %}
{% headingcontext source_level=3 target_level=6 %}
{% include "another_template_fragment_which_contains_some_h3.html" %}
{% endheadingcontext %}
```

Notice the use of the additional "target_level" parameter, which forces ouput levels to start at 6.

You can read the provided test cases to observe what does this template tag at tests.HeadingContextTemplateTagTestCase.

2.2.4 Javascript

The "javascript" template tag library provides 3 template tags:

- javascript_assign: registers some Javascript code. Requires a {% endjavascript_assign %} closing tag.
- javascript_render: displays all registered Javascript code
- javascript_reset: empties the Javascript registry

This template tag library has been written to help template designers implement the following pattern:

- in templates, write Javascript fragments along with the corresponding HTML code. This is done with the {% javascript_assign %} template tag.
- display the Javascript code at the end of the HTML document. This is done by calling the {% javascript_render %} template tag at the end of the HTML document.
- remove duplicate code fragments, e.g. do not call the same library twice. This is done by the {% javascript_render %} template tag. Notice that, at this time, only *strict* duplicates are ignored (i.e. if two Javascript fragments have whitespace or attributes order differences, they won't be considered as duplicate, even if they have the same meaning), so you may have to respect some coding conventions about Javascript.

The main advantage of this template tag is that you can manage Javascript code on a per-template basis. You no longer have to maintain both a specific template for HTML code and a global template for all Javascript calls.

Let's review an example. We have 3 templates:

- base.html: the base template
- menu.html: include that display menus
- home.html: called when requesting / URL

base.html, template code:

```
<html>
  {% load javascript %}
  <head>
{% javascript_assign %}<script type="text/javascript" src="/first_lib.js" />{% endjavascript_assign %}
{% javascript_assign %}
<script type="text/javascript">
    /* some javascript code that uses "first_lib.js" */
    var a = 1;
    /* Notice that the "left aligned" indentation helps avoiding whitespace differences between two c
</script>
{% endjavascript_assign %}
  </head>
  <body>
    <div id="menu">
      {% include "menu.html" %}
    </div>
    <div id="content">
      {% block content %}{% endblock content %}
    </div>
    <!--  JAVASCRIPT CODE -->
    {% javascript_render %}
  </body>
</html>
```

menu.html, template code:

```
{% load javascript %}
{% javascript_assign %}<script type="text/javascript" src="/second_lib.js" />{% endjavascript_assign %}
{% javascript_assign %}
<script type="text/javascript">
    /* some javascript code that uses "second_lib.js" */
    var b = 2;
</script>
{% endjavascript_assign %}
<ul>
  <li><a href="/">Home</a></li>
  <!-- the menu... -->
</ul>
```

home.html, template code:

```
{% extends "base.html" %}
{% load javascript %}

{% block content %}
{% javascript_assign %}<script type="text/javascript" src="/first_lib.js" />{% endjavascript_assign %}
{% javascript_assign %}<script type="text/javascript" src="/second_lib.js" />{% endjavascript_assign %}
{% javascript_assign %}<script type="text/javascript">
    /* some javascript code that uses both "first_lib.js" and "second_lib.js" */
    var c = 3;
</script>
{% endjavascript_assign %}
```

```
<p>This is the content</p>
{% endblock content %}
```

HTML output when requesting / URL (indentation and linebreaks have been cleaned for improved lisibility):

```
<html>
  <head>
  </head>
  <body>
    <div id="menu">
      <ul>
        <li><a href="/">Home</a></li>
        <!-- the menu... -->
      </ul>
    </div>
    <div id="content">
      <p>This is the content</p>
    </div>
    <!--  JAVASCRIPT CODE -->
    <script type="text/javascript" src="/first_lib.js" />
    <script type="text/javascript">
        /* some javascript code that uses "first_lib.js" */
        var a = 1;
        /* Notice that the "left aligned" indentation helps avoiding whitespace differences between t
    </script>
    <script type="text/javascript" src="/second_lib.js" />
    <script type="text/javascript">
        /* some javascript code that uses "second_lib.js" */
        var b = 2;
    </script>
    <script type="text/javascript">
        /* some javascript code that uses both "first_lib.js" and "second_lib.js" */
        var c = 3;
    </script>
  </body>
</html>
```

As you can see in the example above, the content of each {% javascript_assign %} block has been saved. Then the {% javascript_render %} call removes duplicates and displays all the code fragments, by order of appearance.

javascript_assign

Use it as container of each Javascript code fragment needed in your templates. You should create separate {% javascript_assign %} blocks for each call to a Javascript library.

javascript_render

Use it to render the Javascript code fragments *previously* registered in {% javascript_assign %} blocks.

javascript_reset

Call it to empty the Javascript code fragments *previously* registered in {% javascript_assign %} blocks. In general use case, you display Javascript code once, so you do not need to use this tag.

2.2.5 Replace

The "replace" template tag library provides 1 template tag and 1 template filter.

Template tag:

- replace: replaces a string by another in the content of the block. Requires a {% endreplace %} closing tag.

Template filter:

- escape_regexp: makes it possible to escape a regular expression special characters in a string.

replace

This template tag provides a simple string replacement functionality. The following template code...:

```
{% load replace %}
{% replace search="pa" replacement="to" %}pamato{% endreplace %}
```

... gives the following output:

```
tomato
```

The "search" parameter is considered as a regular expression by default (options re.UNICODE and re.DOTALL are enabled).

If you do not want to use regular expression, use the enable_regexp=0 parameter. The following template code...:

```
{% load replace %}
{% replace search="(to)" replacement="au" %}(to)to{% endreplace %}
{% replace search="(to)" replacement="au" use_regexp=0 %}(to)to{% endreplace %}
```

... gives the following output:

```
(au)au
auto
```

You can also use backreferences in the replacement parameter. The following template code...:

```
{% load replace %}
{% replace search="([a-z]+)" replacement="*\1*" %}123abc456def{% endreplace %}
```

... gives the following output:

```
123*abc*456*def*
```

Notice, if you write the previous template code in Python code (i.e. as a string), then backreference syntax is \1 rather than 1.

2.3 Developer tools

See inline documentation for templateaddons.utils.parse_tag_argument() and templateaddons.utils.decode_tag_arguments().

Additional documentation is needed.

2.4 Context processors

Documentation needed.

CHAPTER 3

Credits and license

This application is published under the BSD license. See LICENSE and AUTHORS for details.