Djula HTML templating system Documentation

Release 0.2

Nick Allen

July 30, 2014

CONTENTS

Djula is an HTML templating system similar to Django templates for Common Lisp.

Contents:

CHAPTER ONE

INTRODUCTION

Djula is an HTML templating system similar to Django templates for Common Lisp.

Djula's template language is designed to strike a balance between power and ease. It's designed to feel comfortable to those used to working with HTML.

Philosophy

If you have a background in programming, or if you're used to languages which mix programming code directly into HTML, you'll want to bear in mind that the Djula template system is not simply Common Lisp code embedded into HTML. This is by design: the template system is meant to express presentation, not program logic.

The Djula template system provides tags which function similarly to some programming constructs – an `if` tag for boolean tests, a `for` tag for looping, etc. – but these are not simply executed as the corresponding Lisp code, and the template system will not execute arbitrary Lisp expressions. Only the tags, filters and syntax listed below are supported by default (although you can add `your own extensions` to the template language as needed).

1.1 Prerequisites

TODO: list of Common Lisp compilers Djula works on.

1.2 Installation

Djula is available on Quicklisp:

```
(ql:quickload :djula)
```

CHAPTER

TWO

BASICS

A template is simply a text file. It can generate any text-based format (HTML, XML, CSV, etc.).

A template contains **variables**, which get replaced with values when the template is evaluated, and **tags**, which control the logic of the template.

Below is a minimal template that illustrates a few basics. Each element will be explained later in this document.

```
{% extends "base_generic.html" %}

{% block title %}{{ section.title }}{% endblock %}

{% block content %}
<h1>{{ section.title }}</h1>

{% for story in story_list %}
<h2>
  <a href="{{ story.get_absolute_url }}">
    {{ story.headline|upper }}
  </a>
</h2>
<p>{{ story.tease|truncatewords:"100" }}</p>
{% endfor %}
{% endblock %}
```

CHAPTER

THREE

VARIABLES

Variables look like this: `{{ variable }}`. When the template engine encounters a variable, it evaluates that variable and replaces it with the result. Variable names consist of any combination of alphanumeric characters and the underscore (`"_"`). The dot (`"."`) also appears in variable sections, although that has a special meaning, as indicated below. Importantly, *you cannot have spaces or punctuation characters in variable names.*

Use a dot (`.`) to access attributes of a variable.

Behind the scenes

Technically, when the template system encounters a dot, it tries the following lookups, in this order:

- Dictionary lookup
- Attribute lookup
- Method call
- List-index lookup

This can cause some unexpected behavior with objects that override dictionary lookup. For example, consider the following code snippet that attempts to loop over a `collections.defaultdict`:

```
{% for k, v in defaultdict.iteritems %}
    Do something with k and v here...
{% endfor %}
```

Because dictionary lookup happens first, that behavior kicks in and provides a default value instead of using the intended `.iteritems()` method. In this case, consider converting to a dictionary first.

In the above example, `{{ section.title }}` will be replaced with the `title` attribute of the `section` object.

If you use a variable that doesn't exist, the template system will insert the value of the `TEMPLATE_STRING_IF_INVALID` setting, which is set to `''` (the empty string) by default.

Note that "bar" in a template expression like `{{ foo.bar }}` will be interpreted as a literal string and not using the value of the variable "bar", if one exists in the template context.

CHAPTER

FOUR

USAGE

To render our templates, they need to be compiled first. We do that with the COMPILE-TEMPLATE* function. For inheritance to work, we need to put all the templates in the same directory so that Djula can find them when resolving templates inheritance.

In the following example we use *TEMPLATE-FOLDER* as the templates folder, and then several templates are compiled with the COMPILE-TEMPLATE* function.

```
(defparameter *templates-folder*
  (asdf:system-relative-pathname "webapp" "templates/"))

(defparameter +base.html+ (djula:compile-template*
                            (princ-to-string
                              (merge-pathnames "base.html" *templates-folder*))))

(defparameter +welcome.html+ (djula:compile-template*
                               (princ-to-string
                                 (merge-pathnames "welcome.html" *templates-folder*))))

(defparameter +contact.html+ (djula:compile-template*
                               (princ-to-string
                               (merge-pathnames "contact.html" *templates-folder*))))
```

Then we can render our compiled templates using the RENDER-TEMPLATE* function:

```
(djula:render-template* +welcome.html+ s
                         :title "Ukeleles"
                         :project-name "Ukeleles"
                         :mode "welcome")
```

4.1 API

generic (**compile-template***compiler name &optional error-p*)
Provides a hook to customize template compilation.

Specializers

- `(toplevel-compiler common-lisp:t)`
- `(compiler common-lisp:t)`

function (**compile-template****name*)
Compiles template `NAME` with compiler in *CURRENT-COMPILER*

function (`render-template*`*template stream &rest *template-arguments**)
Render `TEMPLATE` into `STREAM` passing *TEMPLATE-ARGUMENTS*

CHAPTER FIVE

TAGS

5.1 Overview

Tags look like this: `{% tag %}`. Tags are more complex than variables: Some create text in the output, some control flow by performing loops or logic, and some load external information into the template to be used by later variables.

Some tags require beginning and ending tags (i.e. `{% tag %} ... tag contents ... {% endtag %}`).

Djula ships with about two dozen built-in template tags. You can read all about them in the *built-in tag reference*. To give you a taste of what's available, here are some of the more commonly used tags:

`for` Loop over each item in an array. For example, to display a list of athletes provided in `athlete_list`:

```
<ul>
{% for athlete in athlete_list %}
    <li>{{ athlete.name }}</li>
{% endfor %}
</ul>
```

`if`, `elif`, and `else` Evaluates a variable, and if that variable is "true" the contents of the block are displayed:

```
{% if athlete_list %}
    Number of athletes: {{ athlete_list|length }}
{% elif athlete_in_locker_room_list %}
    Athletes should be out of the locker room soon!
{% else %}
    No athletes.
{% endif %}
```

In the above, if `athlete_list` is not empty, the number of athletes will be displayed by the `{{ athlete_list|length }}` variable. Otherwise, if `athlete_in_locker_room_list` is not empty, the message "Athletes should be out..." will be displayed. If both lists are empty, "No athletes." will be displayed.

You can also use filters and various operators in the `if` tag:

```
{% if athlete_list|length > 1 %}
   Team: {% for athlete in athlete_list %} ... {% endfor %}
{% else %}
   Athlete: {{ athlete_list.0.name }}
{% endif %}
```

While the above example works, be aware that most template filters return strings, so mathematical comparisons using filters will generally not work as you expect. `length` is an exception.

`block` and `extends` Set up **'template inheritance'_** (see below), a powerful way of cutting down on "boilerplate" in templates.

5.2 List of tags

Tags

- block
- extends
- super
- comment
- csrf_token
- cycle
- debug
- filter
- firstof
- for
- for ... empty
- if
- Boolean operators
- == operator
- != operator
- < operator
- > operator
- <= operator
- >= operator
- `in` operator
- `not in` operator
- Filters
- ifchanged
- ifequal
- ifnotequal
- include

5.2.1 block

Defines a block that can be overridden by child templates.

Sample usage:

```
{% block stylesheets %}
    ...
{% endblock %}
```

See *Template inheritance* for more information.

5.2.2 extends

Extends a template

Sample usage:

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

5.2.3 super

Gets the content of the block from the parent template. You have to pass the name of the block of the parent template you want to access.

Sample usage:

```
{% super "stylesheets" %}
```

5.2.4 comment

Ignores everything between `{% comment %}` and `{% endcomment %}`. An optional note may be inserted in the first tag. For example, this is useful when commenting out code for documenting why the code was disabled.

Sample usage:

```
<p>Rendered text with {{ pub_date|date:"c" }}</p>
{% comment "Optional note" %}
    <p>Commented out text with {{ create_date|date:"c" }}</p>
{% endcomment %}
```

`comment` tags cannot be nested.

5.2.5 csrf_token

This tag is used for CSRF protection, as described in the documentation for `Cross Site Request Forgeries`.

5.2.6 cycle

Produces one of its arguments each time this tag is encountered. The first argument is produced on the first encounter, the second argument on the second encounter, and so forth. Once all arguments are exhausted, the tag cycles to the first argument and produces it again.

This tag is particularly useful in a loop:

```
{% for o in some_list %}
    <tr class="{% cycle 'row1' 'row2' %}">
        ...
    </tr>
{% endfor %}
```

The first iteration produces HTML that refers to class `row1`, the second to `row2`, the third to `row1` again, and so on for each iteration of the loop.

You can use variables, too. For example, if you have two template variables, `rowvalue1` and `rowvalue2`, you can alternate between their values like this:

```
{% for o in some_list %}
    <tr class="{% cycle rowvalue1 rowvalue2 %}">
        ...
    </tr>
{% endfor %}
```

Variables included in the cycle will be escaped. You can disable auto-escaping with:

```
{% for o in some_list %}
    <tr class="{% autoescape off %}{% cycle rowvalue1 rowvalue2 %}{% endautoescape %}
        ...
    </tr>
{% endfor %}
```

You can mix variables and strings:

```
{% for o in some_list %}
    <tr class="{% cycle 'row1' rowvalue2 'row3' %}">
        ...
    </tr>
{% endfor %}
```

In some cases you might want to refer to the current value of a cycle without advancing to the next value. To do this, just give the `{% cycle %}` tag a name, using "as", like this:

```
{% cycle 'row1' 'row2' as rowcolors %}
```

From then on, you can insert the current value of the cycle wherever you'd like in your template by referencing the cycle name as a context variable. If you want to move the cycle to the next value independently of the original `cycle` tag, you can use another `cycle` tag and specify the name of the variable. So, the following template:

```
<tr>
    <td class="{% cycle 'row1' 'row2' as rowcolors %}">...</td>
    <td class="{{ rowcolors }}">...</td>
</tr>
<tr>
    <td class="{% cycle rowcolors %}">...</td>
    <td class="{{ rowcolors }}">...</td>
</tr>
```

would output:

```
<tr>
    <td class="row1">...</td>
    <td class="row1">...</td>
</tr>
<tr>
    <td class="row2">...</td>
    <td class="row2">...</td>
</tr>
```

You can use any number of values in a `cycle` tag, separated by spaces. Values enclosed in single quotes (`'`) or double quotes (`"`) are treated as string literals, while values without quotes are treated as template variables.

By default, when you use the `as` keyword with the cycle tag, the usage of `{% cycle %}` that initiates the cycle will itself produce the first value in the cycle. This could be a problem if you want to use the value in a nested loop or an included template. If you only want to declare the cycle but not produce the first value, you can add a `silent` keyword as the last keyword in the tag. For example:

```
{% for obj in some_list %}
    {% cycle 'row1' 'row2' as rowcolors silent %}
    <tr class="{{ rowcolors }}">{% include "subtemplate.html" %}</tr>
{% endfor %}
```

This will output a list of `<tr>` elements with `class` alternating between `row1` and `row2`. The subtemplate will have access to `rowcolors` in its context and the value will match the class of the `<tr>` that encloses it. If the `silent` keyword were to be omitted, `row1` and `row2` would be emitted as normal text, outside the `<tr>` element.

When the silent keyword is used on a cycle definition, the silence automatically applies to all subsequent uses of that specific cycle tag. The following template would output *nothing*, even though the second call to `{% cycle %}` doesn't specify `silent`:

```
{% cycle 'row1' 'row2' as rowcolors silent %}
{% cycle rowcolors %}
```

For backward compatibility, the `{% cycle %}` tag supports the much inferior old syntax from previous Django versions. You shouldn't use this in any new projects, but for the sake of the people who are still using it, here's what it looks like:

```
{% cycle row1,row2,row3 %}
```

In this syntax, each value gets interpreted as a literal string, and there's no way to specify variable values. Or literal commas. Or spaces. Did we mention you shouldn't use this syntax in any new projects?

5.2.7 debug

Outputs a whole load of debugging information, including the current context and imported modules.

5.2.8 filter

Filters the contents of the block through one or more filters. Multiple filters can be specified with pipes and filters can have arguments, just as in variable syntax.

Note that the block includes *all* the text between the `filter` and `endfilter` tags.

Sample usage:

```
{% filter force_escape|lower %}
    This text will be HTML-escaped, and will appear in all lowercase.
{% endfilter %}
```

Note: The `escape` and `safe` filters are not acceptable arguments. Instead, use the `autoescape` tag to manage autoescaping for blocks of template code.

5.2.9 firstof

Outputs the first argument variable that is not `False`. Outputs nothing if all the passed variables are `False`.

Sample usage:

```
{% firstof var1 var2 var3 %}
```

This is equivalent to:

```
{% if var1 %}
    {{ var1|safe }}
{% elif var2 %}
    {{ var2|safe }}
{% elif var3 %}
    {{ var3|safe }}
{% endif %}
```

You can also use a literal string as a fallback value in case all passed variables are False:

```
{% firstof var1 var2 var3 "fallback value" %}
```

This tag auto-escapes variable values. You can disable auto-escaping with:

```
{% autoescape off %}
    {% firstof var1 var2 var3 "<strong>fallback value</strong>" %}
{% endautoescape %}
```

Or if only some variables should be escaped, you can use:

```
{% firstof var1 var2|safe var3 "<strong>fallback value</strong>"|safe %}
```

5.2.10 for

Loops over each item in an array, making the item available in a context variable. For example, to display a list of athletes provided in `athlete_list`:

```
<ul>
{% for athlete in athlete_list %}
    <li>{{ athlete.name }}</li>
{% endfor %}
</ul>
```

You can loop over a list in reverse by using `{% for obj in list reversed %}`.

If you need to loop over a list of lists, you can unpack the values in each sublist into individual variables. For example, if your context contains a list of (x,y) coordinates called `points`, you could use the following to output the list of points:

```
{% for x, y in points %}
    There is a point at {{ x }},{{ y }}
{% endfor %}
```

This can also be useful if you need to access the items in a dictionary. For example, if your context contained a dictionary `data`, the following would display the keys and values of the dictionary:

```
{% for key, value in data.items %}
    {{ key }}: {{ value }}
{% endfor %}
```

The for loop sets a number of variables available within the loop:

Variable	Description
`forloop.counter`	The current iteration of the loop (1-indexed)
`forloop.counter0`	The current iteration of the loop (0-indexed)
`forloop.revcounter`	The number of iterations from the end of the loop (1-indexed)
`forloop.revcounter0`	The number of iterations from the end of the loop (0-indexed)
`forloop.first`	True if this is the first time through the loop
`forloop.last`	True if this is the last time through the loop
`forloop.parentloop`	For nested loops, this is the loop surrounding the current one

5.2.11 for ... empty

The `for` tag can take an optional `{% empty %}` clause whose text is displayed if the given array is empty or could not be found:

```
<ul>
{% for athlete in athlete_list %}
    <li>{{ athlete.name }}</li>
{% empty %}
    <li>Sorry, no athletes in this list.</li>
{% endfor %}
</ul>
```

The above is equivalent to – but shorter, cleaner, and possibly faster than – the following:

```
<ul>
  {% if athlete_list %}
    {% for athlete in athlete_list %}
      <li>{{ athlete.name }}</li>
    {% endfor %}
  {% else %}
    <li>Sorry, no athletes in this list.</li>
  {% endif %}
</ul>
```

5.2.12 if

The `{% if %}` tag evaluates a variable, and if that variable is "true" (i.e. exists, is not empty, and is not a false boolean value) the contents of the block are output:

```
{% if athlete_list %}
    Number of athletes: {{ athlete_list|length }}
{% elif athlete_in_locker_room_list %}
    Athletes should be out of the locker room soon!
{% else %}
    No athletes.
{% endif %}
```

In the above, if `athlete_list` is not empty, the number of athletes will be displayed by the `{{ athlete_list|length }}` variable.

As you can see, the `if` tag may take one or several `{% elif %}` clauses, as well as an `{% else %}` clause that will be displayed if all previous conditions fail. These clauses are optional.

5.2.13 Boolean operators

`if` tags may use `and`, `or` or `not` to test a number of variables or to negate a given variable:

```
{% if athlete_list and coach_list %}
    Both athletes and coaches are available.
{% endif %}

{% if not athlete_list %}
    There are no athletes.
{% endif %}

{% if athlete_list or coach_list %}
    There are some athletes or some coaches.
{% endif %}

{% if not athlete_list or coach_list %}
```

```
    There are no athletes or there are some coaches (OK, so
    writing English translations of boolean logic sounds
    stupid; it's not our fault).
{% endif %}

{% if athlete_list and not coach_list %}
    There are some athletes and absolutely no coaches.
{% endif %}
```

Use of both `and` and `or` clauses within the same tag is allowed, with `and` having higher precedence than `or` e.g.:

```
{% if athlete_list and coach_list or cheerleader_list %}
```

will be interpreted like:

```
if (athlete_list and coach_list) or cheerleader_list
```

Use of actual parentheses in the `if` tag is invalid syntax. If you need them to indicate precedence, you should use nested `if` tags.

`if` tags may also use the operators ==, !=, <, >, <=, >= and `in` which work as follows:

5.2.14 == operator

Equality. Example:

```
{% if somevar == "x" %}
  This appears if variable somevar equals the string "x"
{% endif %}
```

5.2.15 != operator

Inequality. Example:

```
{% if somevar != "x" %}
  This appears if variable somevar does not equal the string "x",
  or if somevar is not found in the context
{% endif %}
```

5.2.16 < operator

Less than. Example:

```
{% if somevar < 100 %}
  This appears if variable somevar is less than 100.
{% endif %}
```

5.2.17 > operator

Greater than. Example:

```
{% if somevar > 0 %}
  This appears if variable somevar is greater than 0.
{% endif %}
```

5.2.18 <= operator

Less than or equal to. Example:

```
{% if somevar <= 100 %}
  This appears if variable somevar is less than 100 or equal to 100.
{% endif %}
```

5.2.19 >= operator

Greater than or equal to. Example:

```
{% if somevar >= 1 %}
  This appears if variable somevar is greater than 1 or equal to 1.
{% endif %}
```

5.2.20 `in` operator

Contained within. This operator is supported by many Python containers to test whether the given value is in the container. The following are some examples of how `x in y` will be interpreted:

```
{% if "bc" in "abcdef" %}
  This appears since "bc" is a substring of "abcdef"
{% endif %}

{% if "hello" in greetings %}
  If greetings is a list or set, one element of which is the string
  "hello", this will appear.
{% endif %}

{% if user in users %}
  If users is a QuerySet, this will appear if user is an
  instance that belongs to the QuerySet.
{% endif %}
```

5.2.21 `not in` operator

Not contained within. This is the negation of the `in` operator.

The comparison operators cannot be 'chained' like in Python or in mathematical notation. For example, instead of using:

```
{% if a > b > c %}  (WRONG)
```

you should use:

```
{% if a > b and b > c %}
```

5.2.22 Filters

You can also use filters in the `if` expression. For example:

```
{% if messages|length >= 100 %}
   You have lots of messages today!
{% endif %}
```

5.2.23 ifchanged

Check if a value has changed from the last iteration of a loop.

The `{% ifchanged %}` block tag is used within a loop. It has two possible uses.

1. Checks its own rendered contents against its previous state and only displays the content if it has changed. For example, this displays a list of days, only displaying the month if it changes:

   ```
   <h1>Archive for {{ year }}</h1>

   {% for date in days %}
       {% ifchanged %}<h3>{{ date|date:"F" }}</h3>{% endifchanged %}
       <a href="{{ date|date:"M/d"|lower }}/">{{ date|date:"j" }}</a>
   {% endfor %}
   ```

2. If given one or more variables, check whether any variable has changed. For example, the following shows the date every time it changes, while showing the hour if either the hour or the date has changed:

   ```
   {% for date in days %}
       {% ifchanged date.date %} {{ date.date }} {% endifchanged %}
       {% ifchanged date.hour date.date %}
           {{ date.hour }}
       {% endifchanged %}
   {% endfor %}
   ```

The `ifchanged` tag can also take an optional `{% else %}` clause that will be displayed if the value has not changed:

```
{% for match in matches %}
    <div style="background-color:
        {% ifchanged match.ballot_id %}
            {% cycle "red" "blue" %}
        {% else %}
            gray
        {% endifchanged %}
    ">{{ match }}</div>
{% endfor %}
```

5.2.24 ifequal

Output the contents of the block if the two arguments equal each other.

Example:

```
{% ifequal user.pk comment.user_id %}
    ...
{% endifequal %}
```

As in the `if` tag, an `{% else %}` clause is optional.

The arguments can be hard-coded strings, so the following is valid:

```
{% ifequal user.username "adrian" %}
    ...
{% endifequal %}
```

An alternative to the `ifequal` tag is to use the `if` tag and the == operator.

5.2.25 ifnotequal

Just like `ifequal`, except it tests that the two arguments are not equal.

An alternative to the `ifnotequal` tag is to use the `if` tag and the `!=` operator.

5.2.26 include

Loads a template and renders it with the current context. This is a way of "including" other templates within a template.

The template name can either be a variable or a hard-coded (quoted) string, in either single or double quotes.

This example includes the contents of the template `"foo/bar.html"`:

```
{% include "foo/bar.html" %}
```

This example includes the contents of the template whose name is contained in the variable `template_name`:

```
{% include template_name %}
```

The variable may also be any object with a `render()` method that accepts a context. This allows you to reference a compiled `Template` in your context.

An included template is rendered within the context of the template that includes it. This example produces the output `"Hello, John"`:

- Context: variable `person` is set to `"john"`.
- Template:

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

- The `name_snippet.html` template:

  ```
  {{ greeting }}, {{ person|default:"friend" }}!
  ```

You can pass additional context to the template using keyword arguments:

```
{% include "name_snippet.html" with person="Jane" greeting="Hello" %}
```

If you want to render the context only with the variables provided (or even no variables at all), use the `only` option. No other variables are available to the included template:

```
{% include "name_snippet.html" with greeting="Hi" only %}
```

Note: The `include` tag should be considered as an implementation of "render this subtemplate and include the HTML", not as "parse this subtemplate and include its contents as if it were part of the parent". This means that there is no shared state between included templates – each include is a completely independent rendering process.

See also: `{% ssi %}`.

5.3 Custom tags

CHAPTER

SIX

COMMENTS

To comment-out part of a line in a template, use the comment syntax: `{# #}`.

For example, this template would render as `'hello'`:

```
{# greeting #}hello
```

A comment can contain any template code, invalid or not. For example:

```
{# {% if foo %}bar{% else %} #}
```

This syntax can only be used for single-line comments (no newlines are permitted between the `{#` and `#}` delimiters). If you need to comment out a multiline portion of the template, see the `comment` tag.

CHAPTER

SEVEN

FILTERS

7.1 Overview

You can modify variables for display by using **filters**.

Filters look like this: `{{ name|lower }}`. This displays the value of the `{{ name }}` variable after being filtered through the `lower` filter, which converts text to lowercase. Use a pipe (`|`) to apply a filter.

Filters can be "chained." The output of one filter is applied to the next. `{{ text|escape|linebreaks }}` is a common idiom for escaping text contents, then converting line breaks to `<p>` tags.

Some filters take arguments. A filter argument looks like this: `{{ bio|truncatewords:30 }}`. This will display the first 30 words of the `bio` variable.

Filter arguments that contain spaces must be quoted; for example, to join a list with commas and spaced you'd use `{{ list|join:", " }}`.

Djula provides about thirty built-in template filters. You can read all about them in the *built-in filter reference*. To give you a taste of what's available, here are some of the more commonly used template filters:

7.2 List of filters

Filters

- add
- addslashes
- capfirst
- center
- cut
- date
- default
- default_if_none
- dictsort
- dictsortreversed
- divisibleby
- escape
- escapejs
- filesizeformat
- first
- floatformat
- force_escape
- get_digit
- iriencode
- join
- last
- length
- length_is
- linebreaks
- linebreaksbr
- linenumbers
- ljust
- lower
- make_list
- phone2numeric
- pluralize
- pprint
- random
- removetags
- rjust
- safe
- safeseq
- slice
- slugify
- stringformat
- striptags
- time
- timesince
- timeuntil
- title
- truncatechars
- truncatechars_html
- truncatewords
- truncatewords_html
- unordered_list
- upper
- urlencode
- urlize
- urlizetrunc
- wordcount
- wordwrap
- yesno

7.2.1 add

Adds the argument to the value.

For example:

```
{{ value|add:"2" }}
```

If `value` is `4`, then the output will be `6`.

This filter will first try to coerce both values to integers. If this fails, it'll attempt to add the values together anyway. This will work on some data types (strings, list, etc.) and fail on others. If it fails, the result will be an empty string.

For example, if we have:

```
{{ first|add:second }}
```

and `first` is `[1, 2, 3]` and `second` is `[4, 5, 6]`, then the output will be `[1, 2, 3, 4, 5, 6]`.

Warning: Strings that can be coerced to integers will be **summed**, not concatenated, as in the first example above.

7.2.2 addslashes

Adds slashes before quotes. Useful for escaping strings in CSV, for example.

For example:

```
{{ value|addslashes }}
```

If `value` is `"I'm using Django"`, the output will be `"I\'m using Django"`.

7.2.3 capfirst

Capitalizes the first character of the value. If the first character is not a letter, this filter has no effect.

For example:

```
{{ value|capfirst }}
```

If `value` is `"django"`, the output will be `"Django"`.

7.2.4 center

Centers the value in a field of a given width.

For example:

```
"{{ value|center:"15" }}"
```

If `value` is `"Django"`, the output will be `"     Django    "`.

7.2.5 cut

Removes all values of arg from the given string.

For example:

```
{{ value|cut:" " }}
```

If `value` is `"String with spaces"`, the output will be `"Stringwithspaces"`.

7.2.6 date

Formats a date according to the given format.

Uses a similar format as PHP's `date()` function (http://php.net/date) with some differences.

Note: These format characters are not used in Django outside of templates. They were designed to be compatible with PHP to ease transitioning for designers.

Available format strings:

Format character	Description
a	`'a.m.'` or `'p.m.'` (Note that this is slightly different than PHP's output, because this includes periods to ma
A	`'AM'` or `'PM'`.
b	Month, textual, 3 letters, lowercase.
B	Not implemented.
c	ISO 8601 format. (Note: unlike others formatters, such as "Z", "O" or "r", the "c" formatter will not add timez
d	Day of the month, 2 digits with leading zeros.
D	Day of the week, textual, 3 letters.
e	Timezone name. Could be in any format, or might return an empty string, depending on the datetime.
E	Month, locale specific alternative representation usually used for long date representation.
f	Time, in 12-hour hours and minutes, with minutes left off if they're zero. Proprietary extension.
F	Month, textual, long.
g	Hour, 12-hour format without leading zeros.
G	Hour, 24-hour format without leading zeros.
h	Hour, 12-hour format.
H	Hour, 24-hour format.
i	Minutes.
I	Daylight Savings Time, whether it's in effect or not.
j	Day of the month without leading zeros.
l	Day of the week, textual, long.
L	Boolean for whether it's a leap year.
m	Month, 2 digits with leading zeros.
M	Month, textual, 3 letters.
n	Month without leading zeros.
N	Month abbreviation in Associated Press style. Proprietary extension.
o	ISO-8601 week-numbering year, corresponding to the ISO-8601 week number (W)
O	Difference to Greenwich time in hours.
P	Time, in 12-hour hours, minutes and 'a.m.'/'p.m.', with minutes left off if they're zero and the special-case strin
r	RFC 2822 formatted date.
s	Seconds, 2 digits with leading zeros.
S	English ordinal suffix for day of the month, 2 characters.
t	Number of days in the given month.

Format character	Description
T	Time zone of this machine.
u	Microseconds.
U	Seconds since the Unix Epoch (January 1 1970 00:00:00 UTC).
w	Day of the week, digits without leading zeros.
W	ISO-8601 week number of year, with weeks starting on Monday.
y	Year, 2 digits.
Y	Year, 4 digits.
z	Day of the year.
Z	Time zone offset in seconds. The offset for timezones west of UTC is always negative, and for those east of UT

For example:

```
{{ value|date:"D d M Y" }}
```

If `value` is a `datetime` object (e.g., the result of `datetime.datetime.now()`), the output will be the string `'Wed 09 Jan 2008'`.

The format passed can be one of the predefined ones `DATE_FORMAT`, `DATETIME_FORMAT`, `SHORT_DATE_FORMAT` or `SHORT_DATETIME_FORMAT`, or a custom format that uses the format specifiers shown in the table above. Note that predefined formats may vary depending on the current locale.

Assuming that `USE_L10N` is `True` and `LANGUAGE_CODE` is, for example, `"es"`, then for:

```
{{ value|date:"SHORT_DATE_FORMAT" }}
```

the output would be the string `"09/01/2008"` (the `"SHORT_DATE_FORMAT"` format specifier for the `es` locale as shipped with Django is `"d/m/Y"`).

When used without a format string:

```
{{ value|date }}
```

...the formatting string defined in the `DATE_FORMAT` setting will be used, without applying any localization.

You can combine `date` with the `time` filter to render a full representation of a `datetime` value. E.g.:

```
{{ value|date:"D d M Y" }} {{ value|time:"H:i" }}
```

7.2.7 default

If value evaluates to `False`, uses the given default. Otherwise, uses the value.

For example:

```
{{ value|default:"nothing" }}
```

If `value` is `""` (the empty string), the output will be `nothing`.

7.2.8 default_if_none

If (and only if) value is `None`, uses the given default. Otherwise, uses the value.

Note that if an empty string is given, the default value will *not* be used. Use the `default` filter if you want to fallback for empty strings.

For example:

```
{{ value|default_if_none:"nothing" }}
```

If `value` is `None`, the output will be the string `"nothing"`.

7.2.9 dictsort

Takes a list of dictionaries and returns that list sorted by the key given in the argument.

For example:

```
{{ value|dictsort:"name" }}
```

If `value` is:

```
[
    {'name': 'zed', 'age': 19},
    {'name': 'amy', 'age': 22},
    {'name': 'joe', 'age': 31},
]
```

then the output would be:

```
[
    {'name': 'amy', 'age': 22},
    {'name': 'joe', 'age': 31},
    {'name': 'zed', 'age': 19},
]
```

You can also do more complicated things like:

```
{% for book in books|dictsort:"author.age" %}
    * {{ book.title }} ({{ book.author.name }})
{% endfor %}
```

If `books` is:

```
[
    {'title': '1984', 'author': {'name': 'George', 'age': 45}},
    {'title': 'Timequake', 'author': {'name': 'Kurt', 'age': 75}},
    {'title': 'Alice', 'author': {'name': 'Lewis', 'age': 33}},
]
```

then the output would be:

```
* Alice (Lewis)
* 1984 (George)
* Timequake (Kurt)
```

7.2.10 dictsortreversed

Takes a list of dictionaries and returns that list sorted in reverse order by the key given in the argument. This works exactly the same as the above filter, but the returned value will be in reverse order.

7.2.11 divisibleby

Returns `True` if the value is divisible by the argument.

For example:

```
{{ value|divisibleby:"3" }}
```

If `value` is `21`, the output would be `True`.

7.2.12 escape

Escapes a string's HTML. Specifically, it makes these replacements:

- < is converted to `&lt;`
- > is converted to `&gt;`
- `'` (single quote) is converted to `&#39;`
- `"` (double quote) is converted to `&quot;`
- `&` is converted to `&amp;`

The escaping is only applied when the string is output, so it does not matter where in a chained sequence of filters you put `escape`: it will always be applied as though it were the last filter. If you want escaping to be applied immediately, use the `force_escape` filter.

Applying `escape` to a variable that would normally have auto-escaping applied to the result will only result in one round of escaping being done. So it is safe to use this function even in auto-escaping environments. If you want multiple escaping passes to be applied, use the `force_escape` filter.

For example, you can apply `escape` to fields when `autoescape` is off:

```
{% autoescape off %}
    {{ title|escape }}
{% endautoescape %}
```

7.2.13 escapejs

Escapes characters for use in JavaScript strings. This does *not* make the string safe for use in HTML, but does protect you from syntax errors when using templates to generate JavaScript/JSON.

For example:

```
{{ value|escapejs }}
```

If `value` is `"testing\r\njavascript \'string" <b>escaping</b>"`, the output will be `"testing\\u000D\\u000Ajavascript \\u0027string\\u0022 \\u003Cb\\u003Eescaping\\u003C/b\\u003E"`.

7.2.14 filesizeformat

Formats the value like a 'human-readable' file size (i.e. `'13 KB'`, `'4.1 MB'`, `'102 bytes'`, etc).

For example:

```
{{ value|filesizeformat }}
```

If `value` is 123456789, the output would be `117.7 MB`.

File sizes and SI units

Strictly speaking, `filesizeformat` does not conform to the International System of Units which recommends using KiB, MiB, GiB, etc. when byte sizes are calculated in powers of 1024 (which is the case here). Instead, Django uses traditional unit names (KB, MB, GB, etc.) corresponding to names that are more commonly used.

7.2.15 first

Returns the first item in a list.

For example:

```
{{ value|first }}
```

If `value` is the list `['a', 'b', 'c']`, the output will be `'a'`.

7.2.16 floatformat

When used without an argument, rounds a floating-point number to one decimal place – but only if there's a decimal part to be displayed. For example:

`value`	Template	Output
`34.23234`	`{{ value|floatformat }}`	`34.2`
`34.00000`	`{{ value|floatformat }}`	`34`
`34.26000`	`{{ value|floatformat }}`	`34.3`

If used with a numeric integer argument, `floatformat` rounds a number to that many decimal places. For example:

`value`	Template	Output
`34.23234`	`{{ value|floatformat:3 }}`	`34.232`
`34.00000`	`{{ value|floatformat:3 }}`	`34.000`
`34.26000`	`{{ value|floatformat:3 }}`	`34.260`

Particularly useful is passing 0 (zero) as the argument which will round the float to the nearest integer.

`value`	Template	Output
`34.23234`	`{{ value|floatformat:"0" }}`	`34`
`34.00000`	`{{ value|floatformat:"0" }}`	`34`
`39.56000`	`{{ value|floatformat:"0" }}`	`40`

If the argument passed to `floatformat` is negative, it will round a number to that many decimal places – but only if there's a decimal part to be displayed. For example:

`value`	Template	Output
`34.23234`	`{{ value|floatformat:"-3" }}`	`34.232`
`34.00000`	`{{ value|floatformat:"-3" }}`	`34`
`34.26000`	`{{ value|floatformat:"-3" }}`	`34.260`

Using `floatformat` with no argument is equivalent to using `floatformat` with an argument of `-1`.

7.2.17 force_escape

Applies HTML escaping to a string (see the `escape` filter for details). This filter is applied *immediately* and returns a new, escaped string. This is useful in the rare cases where you need multiple escaping or want to apply other filters to the escaped results. Normally, you want to use the `escape` filter.

For example, if you want to catch the `<p>` HTML elements created by the `linebreaks` filter:

```
{% autoescape off %}
    {{ body|linebreaks|force_escape }}
{% endautoescape %}
```

7.2.18 get_digit

Given a whole number, returns the requested digit, where 1 is the right-most digit, 2 is the second-right-most digit, etc. Returns the original value for invalid input (if input or argument is not an integer, or if argument is less than 1). Otherwise, output is always an integer.

For example:

```
{{ value|get_digit:"2" }}
```

If `value` is `123456789`, the output will be `8`.

7.2.19 iriencode

Converts an IRI (Internationalized Resource Identifier) to a string that is suitable for including in a URL. This is necessary if you're trying to use strings containing non-ASCII characters in a URL.

It's safe to use this filter on a string that has already gone through the `urlencode` filter.

For example:

```
{{ value|iriencode }}
```

If `value` is `"?test=1&me=2"`, the output will be `"?test=1&amp;me=2"`.

7.2.20 join

Joins a list with a string, like Python's `str.join(list)`

For example:

```
{{ value|join:" // " }}
```

If `value` is the list `['a', 'b', 'c']`, the output will be the string `"a // b // c"`.

7.2.21 last

Returns the last item in a list.

For example:

```
{{ value|last }}
```

If `value` is the list `['a', 'b', 'c', 'd']`, the output will be the string `"d"`.

7.2.22 length

Returns the length of the value. This works for both strings and lists.

For example:

```
{{ value|length }}
```

If `value` is `['a', 'b', 'c', 'd']` or `"abcd"`, the output will be `4`.

The filter returns `0` for an undefined variable. Previously, it returned an empty string.

7.2.23 length_is

Returns `True` if the value's length is the argument, or `False` otherwise.

For example:

```
{{ value|length_is:"4" }}
```

If `value` is `['a', 'b', 'c', 'd']` or `"abcd"`, the output will be `True`.

7.2.24 linebreaks

Replaces line breaks in plain text with appropriate HTML; a single newline becomes an HTML line break (`<br />`) and a new line followed by a blank line becomes a paragraph break (`</p>`).

For example:

```
{{ value|linebreaks }}
```

If `value` is `Joel\nis a slug`, the output will be `<p>Joel<br />is a slug</p>`.

7.2.25 linebreaksbr

Converts all newlines in a piece of plain text to HTML line breaks (`<br />`).

For example:

```
{{ value|linebreaksbr }}
```

If `value` is `Joel\nis a slug`, the output will be `Joel<br />is a slug`.

7.2.26 linenumbers

Displays text with line numbers.

For example:

```
{{ value|linenumbers }}
```

If `value` is:

```
one
two
three
```

the output will be:

```
1. one
2. two
3. three
```

7.2.27 ljust

Left-aligns the value in a field of a given width.

Argument: field size

For example:

```
"{{ value|ljust:"10" }}"
```

If `value` is `Django`, the output will be `"Django    "`.

7.2.28 lower

Converts a string into all lowercase.

For example:

```
{{ value|lower }}
```

If `value` is `Still MAD At Yoko`, the output will be `still mad at yoko`.

7.2.29 make_list

Returns the value turned into a list. For a string, it's a list of characters. For an integer, the argument is cast into an unicode string before creating a list.

For example:

```
{{ value|make_list }}
```

If `value` is the string `"Joel"`, the output would be the list `['J', 'o', 'e', 'l']`. If `value` is `123`, the output will be the list `['1', '2', '3']`.

7.2.30 phone2numeric

Converts a phone number (possibly containing letters) to its numerical equivalent.

The input doesn't have to be a valid phone number. This will happily convert any string.

For example:

```
{{ value|phone2numeric }}
```

If `value` is `800-COLLECT`, the output will be `800-2655328`.

7.2.31 pluralize

Returns a plural suffix if the value is not 1. By default, this suffix is `'s'`.

Example:

```
You have {{ num_messages }} message{{ num_messages|pluralize }}.
```

If `num_messages` is `1`, the output will be `You have 1 message.` If `num_messages` is `2` the output will be `You have 2 messages.`

For words that require a suffix other than `'s'`, you can provide an alternate suffix as a parameter to the filter.

Example:

```
You have {{ num_walruses }} walrus{{ num_walruses|pluralize:"es" }}.
```

For words that don't pluralize by simple suffix, you can specify both a singular and plural suffix, separated by a comma.

Example:

```
You have {{ num_cherries }} cherr{{ num_cherries|pluralize:"y,ies" }}.
```

Note: Use `blocktrans` to pluralize translated strings.

7.2.32 pprint

A wrapper around `pprint.pprint()` – for debugging, really.

7.2.33 random

Returns a random item from the given list.

For example:

```
{{ value|random }}
```

If `value` is the list `['a', 'b', 'c', 'd']`, the output could be `"b"`.

7.2.34 removetags

Removes a space-separated list of [X]HTML tags from the output.

For example:

```
{{ value|removetags:"b span"|safe }}
```

If `value` is `"<b>Joel</b> <button>is</button> a <span>slug</span>"` the output will be `"Joel <button>is</button> a slug"`.

Note that this filter is case-sensitive.

If `value` is `"<B>Joel</B> <button>is</button> a <span>slug</span>"` the output will be `"<B>Joel</B> <button>is</button> a slug"`.

7.2.35 rjust

Right-aligns the value in a field of a given width.

Argument: field size

For example:

```
"{{ value|rjust:"10" }}"
```

If `value` is `Django`, the output will be `"    Django"`.

7.2.36 safe

Marks a string as not requiring further HTML escaping prior to output. When autoescaping is off, this filter has no effect.

Note: If you are chaining filters, a filter applied after `safe` can make the contents unsafe again. For example, the following code prints the variable as is, unescaped:

```
{{ var|safe|escape }}
```

7.2.37 safeseq

Applies the `safe` filter to each element of a sequence. Useful in conjunction with other filters that operate on sequences, such as `join`. For example:

```
{{ some_list|safeseq|join:", " }}
```

You couldn't use the `safe` filter directly in this case, as it would first convert the variable into a string, rather than working with the individual elements of the sequence.

7.2.38 slice

Returns a slice of the list.

Uses the same syntax as Python's list slicing. See http://www.diveintopython3.net/native-datatypes.html#slicinglists for an introduction.

Example:

```
{{ some_list|slice:":2" }}
```

If `some_list` is `['a', 'b', 'c']`, the output will be `['a', 'b']`.

7.2.39 slugify

Converts to lowercase, removes non-word characters (alphanumerics and underscores) and converts spaces to hyphens. Also strips leading and trailing whitespace.

For example:

```
{{ value|slugify }}
```

If `value` is `"Joel is a slug"`, the output will be `"joel-is-a-slug"`.

7.2.40 stringformat

Formats the variable according to the argument, a string formatting specifier. This specifier uses Python string formatting syntax, with the exception that the leading "%" is dropped.

See http://docs.python.org/library/stdtypes.html#string-formatting-operations for documentation of Python string formatting

For example:

```
{{ value|stringformat:"E" }}
```

If `value` is `10`, the output will be `1.000000E+01`.

7.2.41 striptags

Makes all possible efforts to strip all [X]HTML tags.

For example:

```
{{ value|striptags }}
```

If `value` is `"<b>Joel</b> <button>is</button> a <span>slug</span>"`, the output will be `"Joel is a slug"`.

No safety guarantee

Note that `striptags` doesn't give any guarantee about its output being entirely HTML safe, particularly with non valid HTML input. So **NEVER** apply the `safe` filter to a `striptags` output. If you are looking for something more robust, you can use the `bleach` Python library, notably its clean method.

7.2.42 time

Formats a time according to the given format.

Given format can be the predefined one `TIME_FORMAT`, or a custom format, same as the `date` filter. Note that the predefined format is locale-dependent.

For example:

```
{{ value|time:"H:i" }}
```

If `value` is equivalent to `datetime.datetime.now()`, the output will be the string `"01:23"`.

Another example:

Assuming that `USE_L10N` is `True` and `LANGUAGE_CODE` is, for example, `"de"`, then for:

```
{{ value|time:"TIME_FORMAT" }}
```

the output will be the string `"01:23:00"` (The `"TIME_FORMAT"` format specifier for the `de` locale as shipped with Django is `"H:i:s"`).

The `time` filter will only accept parameters in the format string that relate to the time of day, not the date (for obvious reasons). If you need to format a `date` value, use the `date` filter instead (or along `time` if you need to render a full `datetime` value).

There is one exception the above rule: When passed a `datetime` value with attached timezone information (a *time-zone-aware* `datetime` instance) the `time` filter will accept the timezone-related *format specifiers* `'e'`, `'O'` , `'T'` and `'Z'`.

When used without a format string:

```
{{ value|time }}
```

...the formatting string defined in the `TIME_FORMAT` setting will be used, without applying any localization.

The ability to receive and act on values with attached timezone information was added in Django 1.7.

7.2.43 timesince

Formats a date as the time since that date (e.g., “4 days, 6 hours”).

Takes an optional argument that is a variable containing the date to use as the comparison point (without the argument, the comparison point is *now*). For example, if `blog_date` is a date instance representing midnight on 1 June 2006, and `comment_date` is a date instance for 08:00 on 1 June 2006, then the following would return “8 hours”:

```
{{ blog_date|timesince:comment_date }}
```

Comparing offset-naive and offset-aware datetimes will return an empty string.

Minutes is the smallest unit used, and “0 minutes” will be returned for any date that is in the future relative to the comparison point.

7.2.44 timeuntil

Similar to `timesince`, except that it measures the time from now until the given date or datetime. For example, if today is 1 June 2006 and `conference_date` is a date instance holding 29 June 2006, then `{{ conference_date|timeuntil }}` will return “4 weeks”.

Takes an optional argument that is a variable containing the date to use as the comparison point (instead of *now*). If `from_date` contains 22 June 2006, then the following will return “1 week”:

```
{{ conference_date|timeuntil:from_date }}
```

Comparing offset-naive and offset-aware datetimes will return an empty string.

Minutes is the smallest unit used, and “0 minutes” will be returned for any date that is in the past relative to the comparison point.

7.2.45 title

Converts a string into titlecase by making words start with an uppercase character and the remaining characters lowercase. This tag makes no effort to keep “trivial words” in lowercase.

For example:

```
{{ value|title }}
```

If `value` is `"my FIRST post"`, the output will be `"My First Post"`.

7.2.46 truncatechars

Truncates a string if it is longer than the specified number of characters. Truncated strings will end with a translatable ellipsis sequence ("...").

Argument: Number of characters to truncate to

For example:

```
{{ value|truncatechars:9 }}
```

If `value` is `"Joel is a slug"`, the output will be `"Joel i..."`.

7.2.47 truncatechars_html

Similar to `truncatechars`, except that it is aware of HTML tags. Any tags that are opened in the string and not closed before the truncation point are closed immediately after the truncation.

For example:

```
{{ value|truncatechars_html:9 }}
```

If `value` is `"<p>Joel is a slug</p>"`, the output will be `"<p>Joel i...</p>"`.

Newlines in the HTML content will be preserved.

7.2.48 truncatewords

Truncates a string after a certain number of words.

Argument: Number of words to truncate after

For example:

```
{{ value|truncatewords:2 }}
```

If `value` is `"Joel is a slug"`, the output will be `"Joel is ..."`.

Newlines within the string will be removed.

7.2.49 truncatewords_html

Similar to `truncatewords`, except that it is aware of HTML tags. Any tags that are opened in the string and not closed before the truncation point, are closed immediately after the truncation.

This is less efficient than `truncatewords`, so should only be used when it is being passed HTML text.

For example:

```
{{ value|truncatewords_html:2 }}
```

If `value` is `"<p>Joel is a slug</p>"`, the output will be `"<p>Joel is ...</p>"`.

Newlines in the HTML content will be preserved.

7.2.50 unordered_list

Recursively takes a self-nested list and returns an HTML unordered list – WITHOUT opening and closing <ul> tags.

The list is assumed to be in the proper format. For example, if `var` contains `['States', ['Kansas', ['Lawrence', 'Topeka'], 'Illinois']]`, then `{{ var|unordered_list }}` would return:

```
<li>States
<ul>
        <li>Kansas
        <ul>
                <li>Lawrence</li>
                <li>Topeka</li>
        </ul>
        </li>
        <li>Illinois</li>
</ul>
</li>
```

Note: An older, more restrictive and verbose input format is also supported: `['States', [['Kansas', [['Lawrence', []], ['Topeka', []]]], ['Illinois', []]]],`

7.2.51 upper

Converts a string into all uppercase.

For example:

```
{{ value|upper }}
```

If `value` is `"Joel is a slug"`, the output will be `"JOEL IS A SLUG"`.

7.2.52 urlencode

Escapes a value for use in a URL.

For example:

```
{{ value|urlencode }}
```

If `value` is `"http://www.example.org/foo?a=b&c=d"`, the output will be `"http%3A//www.example.org/foo%3Fa%3Db%26c%3Dd"`.

An optional argument containing the characters which should not be escaped can be provided.

If not provided, the '/' character is assumed safe. An empty string can be provided when *all* characters should be escaped. For example:

```
{{ value|urlencode:"" }}
```

If `value` is `"http://www.example.org/"`, the output will be `"http%3A%2F%2Fwww.example.org%2F"`.

7.2.53 urlize

Converts URLs and email addresses in text into clickable links.

This template tag works on links prefixed with `http://`, `https://`, or `www.`. For example, `http://goo.gl/aia1t` will get converted but `goo.gl/aia1t` won't.

It also supports domain-only links ending in one of the original top level domains (`.com`, `.edu`, `.gov`, `.int`, `.mil`, `.net`, and `.org`). For example, `djangoproject.com` gets converted.

Support for domain-only links that include characters after the top-level domain (e.g. `djangoproject.com/` and `djangoproject.com/download/`) was added.

Links can have trailing punctuation (periods, commas, close-parens) and leading punctuation (opening parens), and `urlize` will still do the right thing.

Links generated by `urlize` have a `rel="nofollow"` attribute added to them.

For example:

```
{{ value|urlize }}
```

If `value` is `"Check out www.djangoproject.com"`, the output will be `"Check out <a href="http://www.djangoproject.com" rel="nofollow">www.djangoproject.com</a>"`.

In addition to web links, `urlize` also converts email addresses into `mailto:` links. If `value` is `"Send questions to foo@example.com"`, the output will be `"Send questions to <a href="mailto:foo@example.com">foo@example</a>"`.

The `urlize` filter also takes an optional parameter `autoescape`. If `autoescape` is `True`, the link text and URLs will be escaped using Django's built-in `escape` filter. The default value for `autoescape` is `True`.

Note: If `urlize` is applied to text that already contains HTML markup, things won't work as expected. Apply this filter only to plain text.

7.2.54 urlizetrunc

Converts URLs and email addresses into clickable links just like urlize, but truncates URLs longer than the given character limit.

Argument: Number of characters that link text should be truncated to, including the ellipsis that's added if truncation is necessary.

For example:

```
{{ value|urlizetrunc:15 }}
```

If `value` is `"Check out www.djangoproject.com"`, the output would be `'Check out <a href="http://www.djangoproject.com" rel="nofollow">www.djangopr...</a>'`.

As with urlize, this filter should only be applied to plain text.

7.2.55 wordcount

Returns the number of words.

For example:

```
{{ value|wordcount }}
```

If `value` is `"Joel is a slug"`, the output will be `4`.

7.2.56 wordwrap

Wraps words at specified line length.

Argument: number of characters at which to wrap the text

For example:

```
{{ value|wordwrap:5 }}
```

If `value` is `Joel is a slug`, the output would be:

```
Joel
is a
slug
```

7.2.57 yesno

Maps values for true, false and (optionally) None, to the strings "yes", "no", "maybe", or a custom mapping passed as a comma-separated list, and returns one of those strings according to the value:

For example:

```
{{ value|yesno:"yeah,no,maybe" }}
```

Value	Argument	Outputs
`True`		`yes`
`True`	`"yeah,no,maybe"`	`yeah`
`False`	`"yeah,no,maybe"`	`no`
`None`	`"yeah,no,maybe"`	`maybe`
`None`	`"yeah,no"`	`"no"` (converts None to False if no mapping for None is given)

7.3 Internationalization tags and filters

Django provides template tags and filters to control each aspect of `internationalization` in templates. They allow for granular control of translations, formatting, and time zone conversions.

7.3.1 i18n

This library allows specifying translatable text in templates. To enable it, set `USE_I18N` to `True`, then load it with `{% load i18n %}`.

See *specifying-translation-strings-in-template-code*.

7.3.2 l10n

This library provides control over the localization of values in templates. You only need to load the library using `{% load l10n %}`, but you'll often set `USE_L10N` to `True` so that localization is active by default.

See *topic-l10n-templates*.

7.3.3 tz

This library provides control over time zone conversions in templates. Like `l10n`, you only need to load the library using `{% load tz %}`, but you'll usually also set `USE_TZ` to `True` so that conversion to local time happens by default.

See *time-zones-in-templates*.

CHAPTER

EIGHT

TEMPLATE INHERITANCE

The most powerful – and thus the most complex – part of Djula's template engine is template inheritance. Template inheritance allows you to build a base "skeleton" template that contains all the common elements of your site and defines **blocks** that child templates can override.

It's easiest to understand template inheritance by starting with an example:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <link rel="stylesheet" href="style.css" />
    <title>{% block title %}My amazing site{% endblock %}</title>
</head>

<body>
    <div id="sidebar">
        {% block sidebar %}
        <ul>
            <li><a href="/">Home</a></li>
            <li><a href="/blog/">Blog</a></li>
        </ul>
        {% endblock %}
    </div>

    <div id="content">
        {% block content %}{% endblock %}
    </div>
</body>
</html>
```

This template, which we'll call `base.html`, defines a simple HTML skeleton document that you might use for a simple two-column page. It's the job of "child" templates to fill the empty blocks with content.

In this example, the `block` tag defines three blocks that child templates can fill in. All the `block` tag does is to tell the template engine that a child template may override those portions of the template.

A child template might look like this:

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

{% block title %}My amazing blog{% endblock %}

{% block content %}
{% for entry in blog_entries %}
    <h2>{{ entry.title }}</h2>
    <p>{{ entry.body }}</p>
```

```
{% endfor %}
{% endblock %}
```

The `extends` tag is the key here. It tells the template engine that this template "extends" another template. When the template system evaluates this template, first it locates the parent – in this case, "base.html".

At that point, the template engine will notice the three `block` tags in `base.html` and replace those blocks with the contents of the child template. Depending on the value of `blog_entries`, the output might look like:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <link rel="stylesheet" href="style.css" />
    <title>My amazing blog</title>
</head>

<body>
    <div id="sidebar">
        <ul>
            <li><a href="/">Home</a></li>
            <li><a href="/blog/">Blog</a></li>
        </ul>
    </div>

    <div id="content">
        <h2>Entry one</h2>
        <p>This is my first entry.</p>

        <h2>Entry two</h2>
        <p>This is my second entry.</p>
    </div>
</body>
</html>
```

Note that since the child template didn't define the `sidebar` block, the value from the parent template is used instead. Content within a `{% block %}` tag in a parent template is always used as a fallback.

You can use as many levels of inheritance as needed. One common way of using inheritance is the following three-level approach:

- Create a `base.html` template that holds the main look-and-feel of your site.
- Create a `base_SECTIONNAME.html` template for each "section" of your site. For example, `base_news.html`, `base_sports.html`. These templates all extend `base.html` and include section-specific styles/design.
- Create individual templates for each type of page, such as a news article or blog entry. These templates extend the appropriate section template.

This approach maximizes code reuse and makes it easy to add items to shared content areas, such as section-wide navigation.

Here are some tips for working with inheritance:

- If you use `{% extends %}` in a template, it must be the first template tag in that template. Template inheritance won't work, otherwise.
- More `{% block %}` tags in your base templates are better. Remember, child templates don't have to define all parent blocks, so you can fill in reasonable defaults in a number of blocks, then only define the ones you need later. It's better to have more hooks than fewer hooks.

- If you find yourself duplicating content in a number of templates, it probably means you should move that content to a `{% block %}` in a parent template.
- If you need to get the content of the block from the parent template, the `{{ block.super }}` variable will do the trick. This is useful if you want to add to the contents of a parent block instead of completely overriding it. Data inserted using `{{ block.super }}` will not be automatically escaped (see the **'next section'_**), since it was already escaped, if necessary, in the parent template.
- For extra readability, you can optionally give a *name* to your `{% endblock %}` tag. For example:

  ```
  {% block content %}
  ...
  {% endblock content %}
  ```

  In larger templates, this technique helps you see which `{% block %}` tags are being closed.

Finally, note that you can't define multiple `block` tags with the same name in the same template. This limitation exists because a block tag works in "both" directions. That is, a block tag doesn't just provide a hole to fill – it also defines the content that fills the hole in the *parent*. If there were two similarly-named `block` tags in a template, that template's parent wouldn't know which one of the blocks' content to use.

CHAPTER

NINE

AUTOMATIC HTML ESCAPING

When generating HTML from templates, there's always a risk that a variable will include characters that affect the resulting HTML. For example, consider this template fragment:

```
Hello, {{ name }}.
```

At first, this seems like a harmless way to display a user's name, but consider what would happen if the user entered their name as this:

```
<script>alert('hello')</script>
```

With this name value, the template would be rendered as:

```
Hello, <script>alert('hello')</script>
```

...which means the browser would pop-up a JavaScript alert box!

Similarly, what if the name contained a `'<'` symbol, like this?

```
<b>username
```

That would result in a rendered template like this:

```
Hello, <b>username
```

...which, in turn, would result in the remainder of the Web page being bolded!

Clearly, user-submitted data shouldn't be trusted blindly and inserted directly into your Web pages, because a malicious user could use this kind of hole to do potentially bad things. This type of security exploit is called a Cross Site Scripting (XSS) attack.

To avoid this problem, you have two options:

- One, you can make sure to run each untrusted variable through the `escape` filter (documented below), which converts potentially harmful HTML characters to unharmful ones. This was the default solution in Django for its first few years, but the problem is that it puts the onus on *you*, the developer / template author, to ensure you're escaping everything. It's easy to forget to escape data.
- Two, you can take advantage of Django's automatic HTML escaping. The remainder of this section describes how auto-escaping works.

By default in Django, every template automatically escapes the output of every variable tag. Specifically, these five characters are escaped:

- < is converted to `&lt;`
- > is converted to `&gt;`
- `'` (single quote) is converted to `&#39;`

- `"` (double quote) is converted to `&quot;`
- `&` is converted to `&amp;`

Again, we stress that this behavior is on by default. If you're using Django's template system, you're protected.

9.1 How to turn it off

If you don't want data to be auto-escaped, on a per-site, per-template level or per-variable level, you can turn it off in several ways.

Why would you want to turn it off? Because sometimes, template variables contain data that you *intend* to be rendered as raw HTML, in which case you don't want their contents to be escaped. For example, you might store a blob of HTML in your database and want to embed that directly into your template. Or, you might be using Django's template system to produce text that is *not* HTML – like an email message, for instance.

9.1.1 For individual variables

To disable auto-escaping for an individual variable, use the `safe` filter:

```
This will be escaped: {{ data }}
This will not be escaped: {{ data|safe }}
```

Think of *safe* as shorthand for *safe from further escaping* or *can be safely interpreted as HTML*. In this example, if `data` contains `'<b>'`, the output will be:

```
This will be escaped: &lt;b&gt;
This will not be escaped: <b>
```

9.1.2 For template blocks

To control auto-escaping for a template, wrap the template (or just a particular section of the template) in the `autoescape` tag, like so:

```
{% autoescape off %}
    Hello {{ name }}
{% endautoescape %}
```

The `autoescape` tag takes either `on` or `off` as its argument. At times, you might want to force auto-escaping when it would otherwise be disabled. Here is an example template:

```
Auto-escaping is on by default. Hello {{ name }}

{% autoescape off %}
    This will not be auto-escaped: {{ data }}.

    Nor this: {{ other_data }}
    {% autoescape on %}
        Auto-escaping applies again: {{ name }}
    {% endautoescape %}
{% endautoescape %}
```

The auto-escaping tag passes its effect onto templates that extend the current one as well as templates included via the `include` tag, just like all block tags. For example:

```
# base.html

{% autoescape off %}
<h1>{% block title %}{% endblock %}</h1>
{% block content %}
{% endblock %}
{% endautoescape %}


# child.html

{% extends "base.html" %}
{% block title %}This & that{% endblock %}
{% block content %}{{ greeting }}{% endblock %}
```

Because auto-escaping is turned off in the base template, it will also be turned off in the child template, resulting in the following rendered HTML when the `greeting` variable contains the string `<b>Hello!</b>`:

```
<h1>This & that</h1>
<b>Hello!</b>
```

9.2 Notes

Generally, template authors don't need to worry about auto-escaping very much. Developers on the Python side (people writing views and custom filters) need to think about the cases in which data shouldn't be escaped, and mark data appropriately, so things Just Work in the template.

If you're creating a template that might be used in situations where you're not sure whether auto-escaping is enabled, then add an `escape` filter to any variable that needs escaping. When auto-escaping is on, there's no danger of the `escape` filter *double-escaping* data – the `escape` filter does not affect auto-escaped variables.

9.3 String literals and automatic escaping

As we mentioned earlier, filter arguments can be strings:

```
{{ data|default:"This is a string literal." }}
```

All string literals are inserted **without** any automatic escaping into the template – they act as if they were all passed through the `safe` filter. The reasoning behind this is that the template author is in control of what goes into the string literal, so they can make sure the text is correctly escaped when the template is written.

This means you would write

```
{{ data|default:"3 &lt; 2" }}
```

...rather than

```
{{ data|default:"3 < 2" }}  <-- Bad! Don't do this.
```

This doesn't affect what happens to data coming from the variable itself. The variable's contents are still automatically escaped, if necessary, because they're beyond the control of the template author.

CHAPTER

TEN

CUSTOM TAG AND FILTER LIBRARIES

Certain applications provide custom tag and filter libraries. To access them in a template, use the `load` tag:

```
{% load comments %}

{% comment_form for blogs.entries entry.id with is_public yes %}
```

In the above, the `load` tag loads the `comments` tag library, which then makes the `comment_form` tag available for use. Consult the documentation area in your admin to find the list of custom libraries in your installation.

The `load` tag can take multiple library names, separated by spaces. Example:

```
{% load comments i18n %}
```

See `/howto/custom-template-tags` for information on writing your own custom template libraries.

10.1 Custom libraries and template inheritance

When you load a custom tag or filter library, the tags/filters are only made available to the current template – not any parent or child templates along the template-inheritance path.

For example, if a template `foo.html` has `{% load comments %}`, a child template (e.g., one that has `{% extends "foo.html" %}`) will *not* have access to the comments template tags and filters. The child template is responsible for its own `{% load comments %}`.

This is a feature for the sake of maintainability and sanity.

CHAPTER

ELEVEN

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

INDEX