Built-in template tags and filters¶

docs.djangoproject.com/en/4.0/ref/templates/builtins

This document describes Django's built-in template tags and filters. It is recommended that you use the automatic documentation, if available, as this will also include documentation for any custom tags or filters installed.

Built-in tag reference¶

`autoescape` ¶

Controls the current auto-escaping behavior. This tag takes either `on` or `off` as an argument and that determines whether auto-escaping is in effect inside the block. The block is closed with an `endautoescape` ending tag.

When auto-escaping is in effect, all variable content has HTML escaping applied to it before placing the result into the output (but after any filters have been applied). This is equivalent to manually applying the `escape` filter to each variable.

The only exceptions are variables that are already marked as "safe" from escaping, either by the code that populated the variable, or because it has had the `safe` or `escape` filters applied.

Sample usage:

```
{% autoescape on %}
    {{ body }}
{% endautoescape %}
```

`block` ¶

Defines a block that can be overridden by child templates. See Template inheritance for more information.

`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.

`csrf_token` ¶

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

`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, 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 %}
```

You can use the `resetcycle` tag to make a `{% cycle %}` tag restart from its first value when it's next encountered.

`debug` ¶

Outputs a whole load of debugging information, including the current context and imported modules. `{% debug %}` outputs nothing when the `DEBUG` setting is `False`.

Changed in Django 2.2.27:
In older versions, debugging information was displayed when the `DEBUG` setting was `False`.

`extends` ¶

Signals that this template extends a parent template.

This tag can be used in two ways:

- `{% extends "base.html" %}` (with quotes) uses the literal value `"base.html"` as the name of the parent template to extend.
- `{% extends variable %}` uses the value of `variable`. If the variable evaluates to a string, Django will use that string as the name of the parent template. If the variable evaluates to a `Template` object, Django will use that object as the parent template.

See Template inheritance for more information.

Normally the template name is relative to the template loader's root directory. A string argument may also be a relative path starting with `./` or `../`. For example, assume the following directory structure:

```
dir1/
    template.html
    base2.html
    my/
        base3.html
base1.html
```

In `template.html`, the following paths would be valid:

```
{% extends "./base2.html" %}
{% extends "../base1.html" %}
{% extends "./my/base3.html" %}
```

`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.

`firstof` ¶

Outputs the first argument variable that is not “false” (i.e. exists, is not empty, is not a false boolean value, and is not a zero numeric value). Outputs nothing if all the passed variables are “false”.

Sample usage:

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

This is equivalent to:

```
{% if var1 %}
    {{ var1 }}
{% elif var2 %}
    {{ var2 }}
{% elif var3 %}
    {{ var3 }}
{% 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 %}
```

You can use the syntax `{% firstof var1 var2 var3 as value %}` to store the output inside a variable.

`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 %}
```

Keep in mind that for the dot operator, dictionary key lookup takes precedence over method lookup. Therefore if the `data` dictionary contains a key named `'items'`, `data.items` will return `data['items']` instead of `data.items()`. Avoid adding keys that are named like dictionary methods if you want to use those methods in a template (`items`, `values`, `keys`, etc.). Read more about the lookup order of the dot operator in the documentation of template variables.

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)

Variable	Description
`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

`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>
```

`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.

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.
{% 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 `==`, `!=`, `<`, `>`, `<=`, `>=`, `in`, `not in`, `is`, and `is not` which work as follows:

`==` operator¶

Equality. Example:

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

`!=` 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 %}
```

`<` operator¶

Less than. Example:

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

`>` operator¶

Greater than. Example:

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

`<=` operator¶

Less than or equal to. Example:

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

`>=` operator¶

Greater than or equal to. Example:

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

`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 %}
```

`not in` operator¶

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

`is` operator¶

Object identity. Tests if two values are the same object. Example:

```
{% if somevar is True %}
  This appears if and only if somevar is True.
{% endif %}

{% if somevar is None %}
  This appears if somevar is None, or if somevar is not found in the context.
{% endif %}
```

`is not` operator¶

Negated object identity. Tests if two values are not the same object. This is the negation of the `is` operator. Example:

```
{% if somevar is not True %}
  This appears if somevar is not True, or if somevar is not found in the
  context.
{% endif %}

{% if somevar is not None %}
  This appears if and only if somevar is not None.
{% endif %}
```

Filters¶

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

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

Complex expressions¶

All of the above can be combined to form complex expressions. For such expressions, it can be important to know how the operators are grouped when the expression is evaluated - that is, the precedence rules. The precedence of the operators, from lowest to highest, is as follows:

- `or`
- `and`
- `not`
- `in`
- `==`, `!=`, `<`, `>`, `<=`, `>=`

(This follows Python exactly). So, for example, the following complex `if` tag:

```
{% if a == b or c == d and e %}
```

...will be interpreted as:

```
(a == b) or ((c == d) and e)
```

If you need different precedence, you will need to use nested `if` tags. Sometimes that is better for clarity anyway, for the sake of those who do not know the precedence rules.

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 %}
```

`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 %}
```

`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" %}
```

Normally the template name is relative to the template loader's root directory. A string argument may also be a relative path starting with `./` or `../` as described in the `extends` tag.

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.

Additionally, the variable may be an iterable of template names, in which case the first that can be loaded will be used, as per `select_template()`.

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"` and variable `greeting` is set to `"Hello"`.
- 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.

Blocks are evaluated *before* they are included. This means that a template that includes blocks from another will contain blocks that have *already been evaluated and rendered* - not blocks that can be overridden by, for example, an extending template.

`load` ¶

Loads a custom template tag set.

For example, the following template would load all the tags and filters registered in `somelibrary` and `otherlibrary` located in package `package`:

```
{% load somelibrary package.otherlibrary %}
```

You can also selectively load individual filters or tags from a library, using the `from` argument. In this example, the template tags/filters named `foo` and `bar` will be loaded from `somelibrary`:

```
{% load foo bar from somelibrary %}
```

See Custom tag and filter libraries for more information.

`lorem` ¶

Displays random “lorem ipsum” Latin text. This is useful for providing sample data in templates.

Usage:

```
{% lorem [count] [method] [random] %}
```

The `{% lorem %}` tag can be used with zero, one, two or three arguments. The arguments are:

Argument	Description
`count`	A number (or variable) containing the number of paragraphs or words to generate (default is 1).
`method`	Either `w` for words, `p` for HTML paragraphs or `b` for plain-text paragraph blocks (default is `b`).
`random`	The word `random`, which if given, does not use the common paragraph (“Lorem ipsum dolor sit amet…”) when generating text.

Examples:

- `{% lorem %}` will output the common “lorem ipsum” paragraph.
- `{% lorem 3 p %}` will output the common “lorem ipsum” paragraph and two random paragraphs each wrapped in HTML `<p>` tags.
- `{% lorem 2 w random %}` will output two random Latin words.

`now` ¶

Displays the current date and/or time, using a format according to the given string. Such string can contain format specifiers characters as described in the `date` filter section.

Example:

```
It is {% now "jS F Y H:i" %}
```

Note that you can backslash-escape a format string if you want to use the “raw” value. In this example, both “o” and “f” are backslash-escaped, because otherwise each is a format string that displays the year and the time, respectively:

```
It is the {% now "jS \o\f F" %}
```

This would display as “It is the 4th of September”.

Note

The format passed can also be one of the predefined ones `DATE_FORMAT`, `DATETIME_FORMAT`, `SHORT_DATE_FORMAT` or `SHORT_DATETIME_FORMAT`. The predefined formats may vary depending on the current locale and if Format localization is enabled, e.g.:

```
It is {% now "SHORT_DATETIME_FORMAT" %}
```

You can also use the syntax `{% now "Y" as current_year %}` to store the output (as a string) inside a variable. This is useful if you want to use `{% now %}` inside a template tag like `blocktranslate` for example:

```
{% now "Y" as current_year %}
{% blocktranslate %}Copyright {{ current_year }}{% endblocktranslate %}
```

`regroup` ¶

Regroups a list of alike objects by a common attribute.

This complex tag is best illustrated by way of an example: say that `cities` is a list of cities represented by dictionaries containing `"name"`, `"population"`, and `"country"` keys:

```
cities = [
    {'name': 'Mumbai', 'population': '19,000,000', 'country': 'India'},
    {'name': 'Calcutta', 'population': '15,000,000', 'country': 'India'},
    {'name': 'New York', 'population': '20,000,000', 'country': 'USA'},
    {'name': 'Chicago', 'population': '7,000,000', 'country': 'USA'},
    {'name': 'Tokyo', 'population': '33,000,000', 'country': 'Japan'},
]
```

...and you'd like to display a hierarchical list that is ordered by country, like this:

- India
  - Mumbai: 19,000,000
  - Calcutta: 15,000,000
- USA
  - New York: 20,000,000
  - Chicago: 7,000,000
- Japan
    Tokyo: 33,000,000

You can use the `{% regroup %}` tag to group the list of cities by country. The following snippet of template code would accomplish this:

```
{% regroup cities by country as country_list %}

<ul>
{% for country in country_list %}
    <li>{{ country.grouper }}
    <ul>
        {% for city in country.list %}
          <li>{{ city.name }}: {{ city.population }}</li>
        {% endfor %}
    </ul>
    </li>
{% endfor %}
</ul>
```

Let's walk through this example. `{% regroup %}` takes three arguments: the list you want to regroup, the attribute to group by, and the name of the resulting list. Here, we're regrouping the `cities` list by the `country` attribute and calling the result `country_list`.

`{% regroup %}` produces a list (in this case, `country_list`) of **group objects**. Group objects are instances of `namedtuple()` with two fields:

- `grouper` – the item that was grouped by (e.g., the string "India" or "Japan").
- `list` – a list of all items in this group (e.g., a list of all cities with country='India').

Because `{% regroup %}` produces `namedtuple()` objects, you can also write the previous example as:

```
{% regroup cities by country as country_list %}

<ul>
{% for country, local_cities in country_list %}
    <li>{{ country }}
    <ul>
        {% for city in local_cities %}
          <li>{{ city.name }}: {{ city.population }}</li>
        {% endfor %}
    </ul>
    </li>
{% endfor %}
</ul>
```

Note that `{% regroup %}` does not order its input! Our example relies on the fact that the `cities` list was ordered by `country` in the first place. If the `cities` list did *not* order its members by `country`, the regrouping would naively display more than one group for a single country. For example, say the `cities` list was set to this (note that the countries are not grouped together):

```
cities = [
    {'name': 'Mumbai', 'population': '19,000,000', 'country': 'India'},
    {'name': 'New York', 'population': '20,000,000', 'country': 'USA'},
    {'name': 'Calcutta', 'population': '15,000,000', 'country': 'India'},
    {'name': 'Chicago', 'population': '7,000,000', 'country': 'USA'},
    {'name': 'Tokyo', 'population': '33,000,000', 'country': 'Japan'},
]
```

With this input for `cities`, the example `{% regroup %}` template code above would result in the following output:

- India
  Mumbai: 19,000,000
- USA
  New York: 20,000,000
- India
  Calcutta: 15,000,000
- USA
  Chicago: 7,000,000
- Japan
  Tokyo: 33,000,000

The easiest solution to this gotcha is to make sure in your view code that the data is ordered according to how you want to display it.

Another solution is to sort the data in the template using the `dictsort` filter, if your data is in a list of dictionaries:

```
{% regroup cities|dictsort:"country" by country as country_list %}
```

Grouping on other properties¶

Any valid template lookup is a legal grouping attribute for the regroup tag, including methods, attributes, dictionary keys and list items. For example, if the “country” field is a foreign key to a class with an attribute “description,” you could use:

```
{% regroup cities by country.description as country_list %}
```

Or, if `country` is a field with `choices`, it will have a `get_FOO_display()` method available as an attribute, allowing you to group on the display string rather than the `choices` key:

```
{% regroup cities by get_country_display as country_list %}
```

`{{ country.grouper }}` will now display the value fields from the `choices` set rather than the keys.

`resetcycle`¶

Resets a previous cycle so that it restarts from its first item at its next encounter. Without arguments, `{% resetcycle %}` will reset the last `{% cycle %}` defined in the template.

Example usage:

```
{% for coach in coach_list %}
    <h1>{{ coach.name }}</h1>
    {% for athlete in coach.athlete_set.all %}
        <p class="{% cycle 'odd' 'even' %}">{{ athlete.name }}</p>
    {% endfor %}
    {% resetcycle %}
{% endfor %}
```

This example would return this HTML:

```
<h1>José Mourinho</h1>
<p class="odd">Thibaut Courtois</p>
<p class="even">John Terry</p>
<p class="odd">Eden Hazard</p>

<h1>Carlo Ancelotti</h1>
<p class="odd">Manuel Neuer</p>
<p class="even">Thomas Müller</p>
```

Notice how the first block ends with `class="odd"` and the new one starts with `class="odd"`. Without the `{% resetcycle %}` tag, the second block would start with `class="even"`.

You can also reset named cycle tags:

```
{% for item in list %}
    <p class="{% cycle 'odd' 'even' as stripe %} {% cycle 'major' 'minor' 'minor'
'minor' 'minor' as tick %}">
        {{ item.data }}
    </p>
    {% ifchanged item.category %}
        <h1>{{ item.category }}</h1>
        {% if not forloop.first %}{% resetcycle tick %}{% endif %}
    {% endifchanged %}
{% endfor %}
```

In this example, we have both the alternating odd/even rows and a “major” row every fifth row. Only the five-row cycle is reset when a category changes.

`spaceless` ¶

Removes whitespace between HTML tags. This includes tab characters and newlines.

Example usage:

```
{% spaceless %}
    <p>
        <a href="foo/">Foo</a>
    </p>
{% endspaceless %}
```

This example would return this HTML:

```
<p><a href="foo/">Foo</a></p>
```

Only space between *tags* is removed – not space between tags and text. In this example, the space around `Hello` won't be stripped:

```
{% spaceless %}
    <strong>
        Hello
    </strong>
{% endspaceless %}
```

`templatetag` ¶

Outputs one of the syntax characters used to compose template tags.

The template system has no concept of "escaping" individual characters. However, you can use the `{% templatetag %}` tag to display one of the template tag character combinations.

The argument tells which template bit to output:

Argument	Outputs
`openblock`	`{%`
`closeblock`	`%}`
`openvariable`	`{{`
`closevariable`	`}}`
`openbrace`	`{`
`closebrace`	`}`
`opencomment`	`{#`
`closecomment`	`#}`

Sample usage:

```
The {% templatetag openblock %} characters open a block.
```

See also the `verbatim` tag for another way of including these characters.

`url` ¶

Returns an absolute path reference (a URL without the domain name) matching a given view and optional parameters. Any special characters in the resulting path will be encoded using `iri_to_uri()`.

This is a way to output links without violating the DRY principle by having to hard-code URLs in your templates:

```
{% url 'some-url-name' v1 v2 %}
```

The first argument is a URL pattern name. It can be a quoted literal or any other context variable. Additional arguments are optional and should be space-separated values that will be used as arguments in the URL. The example above shows passing positional arguments. Alternatively you may use keyword syntax:

```
{% url 'some-url-name' arg1=v1 arg2=v2 %}
```

Do not mix both positional and keyword syntax in a single call. All arguments required by the URLconf should be present.

For example, suppose you have a view, `app_views.client`, whose URLconf takes a client ID (here, `client()` is a method inside the views file `app_views.py`). The URLconf line might look like this:

```
path('client/<int:id>/', app_views.client, name='app-views-client')
```

If this app's URLconf is included into the project's URLconf under a path such as this:

```
path('clients/', include('project_name.app_name.urls'))
```

...then, in a template, you can create a link to this view like this:

```
{% url 'app-views-client' client.id %}
```

The template tag will output the string `/clients/client/123/`.

Note that if the URL you're reversing doesn't exist, you'll get an `NoReverseMatch` exception raised, which will cause your site to display an error page.

If you'd like to retrieve a URL without displaying it, you can use a slightly different call:

```
{% url 'some-url-name' arg arg2 as the_url %}

<a href="{{ the_url }}">I'm linking to {{ the_url }}</a>
```

The scope of the variable created by the `as var` syntax is the `{% block %}` in which the `{% url %}` tag appears.

This `{% url ... as var %}` syntax will *not* cause an error if the view is missing. In practice you'll use this to link to views that are optional:

```
{% url 'some-url-name' as the_url %}
{% if the_url %}
  <a href="{{ the_url }}">Link to optional stuff</a>
{% endif %}
```

If you'd like to retrieve a namespaced URL, specify the fully qualified name:

```
{% url 'myapp:view-name' %}
```

This will follow the normal namespaced URL resolution strategy, including using any hints provided by the context as to the current application.

Warning

Don't forget to put quotes around the URL pattern `name`, otherwise the value will be interpreted as a context variable!

`verbatim` ¶

Stops the template engine from rendering the contents of this block tag.

A common use is to allow a JavaScript template layer that collides with Django's syntax. For example:

```
{% verbatim %}
    {{if dying}}Still alive.{{/if}}
{% endverbatim %}
```

You can also designate a specific closing tag, allowing the use of `{% endverbatim %}` as part of the unrendered contents:

```
{% verbatim myblock %}
    Avoid template rendering via the {% verbatim %}{% endverbatim %} block.
{% endverbatim myblock %}
```

`widthratio` ¶

For creating bar charts and such, this tag calculates the ratio of a given value to a maximum value, and then applies that ratio to a constant.

For example:

```
<img src="bar.png" alt="Bar"
     height="10" width="{% widthratio this_value max_value max_width %}">
```

If `this_value` is 175, `max_value` is 200, and `max_width` is 100, the image in the above example will be 88 pixels wide (because 175/200 = .875; .875 * 100 = 87.5 which is rounded up to 88).

In some cases you might want to capture the result of `widthratio` in a variable. It can be useful, for instance, in a `blocktranslate` like this:

```
{% widthratio this_value max_value max_width as width %}
{% blocktranslate %}The width is: {{ width }}{% endblocktranslate %}
```

`with` ¶

Caches a complex variable under a simpler name. This is useful when accessing an "expensive" method (e.g., one that hits the database) multiple times.

For example:

```
{% with total=business.employees.count %}
    {{ total }} employee{{ total|pluralize }}
{% endwith %}
```

The populated variable (in the example above, `total`) is only available between the `{% with %}` and `{% endwith %}` tags.

You can assign more than one context variable:

```
{% with alpha=1 beta=2 %}
    ...
{% endwith %}
```

Note

The previous more verbose format is still supported: `{% with business.employees.count as total %}`

Built-in filter reference¶

`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.

`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"`.

`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"`.

`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     "`.

`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"`.

`date` ¶

Formats a date according to the given format.

Uses a similar format to PHP's date() function 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	Example output
Day		
`d`	Day of the month, 2 digits with leading zeros.	`'01'` to `'31'`
`j`	Day of the month without leading zeros.	`'1'` to `'31'`
`D`	Day of the week, textual, 3 letters.	`'Fri'`
`l`	Day of the week, textual, long.	`'Friday'`
`S`	English ordinal suffix for day of the month, 2 characters.	`'st'`, `'nd'`, `'rd'` or `'th'`
`w`	Day of the week, digits without leading zeros.	`'0'` (Sunday) to `'6'` (Saturday)
`z`	Day of the year.	`1` to `366`
Week		

Format character	Description	Example output
`W`	ISO-8601 week number of year, with weeks starting on Monday.	`1`, `53`
Month		
`m`	Month, 2 digits with leading zeros.	`'01'` to `'12'`
`n`	Month without leading zeros.	`'1'` to `'12'`
`M`	Month, textual, 3 letters.	`'Jan'`
`b`	Month, textual, 3 letters, lowercase.	`'jan'`
`E`	Month, locale specific alternative representation usually used for long date representation.	`'listopada'` (for Polish locale, as opposed to `'Listopad'`)
`F`	Month, textual, long.	`'January'`
`N`	Month abbreviation in Associated Press style. Proprietary extension.	`'Jan.'`, `'Feb.'`, `'March'`, `'May'`
`t`	Number of days in the given month.	`28` to `31`
Year		
`y`	Year, 2 digits with leading zeros.	`'00'` to `'99'`
`Y`	Year, 4 digits with leading zeros.	`'0001'`, …, `'1999'`, …, `'9999'`
`L`	Boolean for whether it's a leap year.	`True` or `False`

Format character	Description	Example output
`o`	ISO-8601 week-numbering year, corresponding to the ISO-8601 week number (W) which uses leap weeks. See Y for the more common year format.	`'1999'`
Time		
`g`	Hour, 12-hour format without leading zeros.	`'1'` to `'12'`
`G`	Hour, 24-hour format without leading zeros.	`'0'` to `'23'`
`h`	Hour, 12-hour format.	`'01'` to `'12'`
`H`	Hour, 24-hour format.	`'00'` to `'23'`
`i`	Minutes.	`'00'` to `'59'`
`s`	Seconds, 2 digits with leading zeros.	`'00'` to `'59'`
`u`	Microseconds.	`000000` to `999999`
`a`	`'a.m.'` or `'p.m.'` (Note that this is slightly different than PHP's output, because this includes periods to match Associated Press style.)	`'a.m.'`
`A`	`'AM'` or `'PM'`.	`'AM'`
`f`	Time, in 12-hour hours and minutes, with minutes left off if they're zero. Proprietary extension.	`'1'`, `'1:30'`

Format character	Description	Example output
`P`	Time, in 12-hour hours, minutes and ‘a.m.’/’p.m.’, with minutes left off if they’re zero and the special-case strings ‘midnight’ and ‘noon’ if appropriate. Proprietary extension.	`'1 a.m.'`, `'1:30 p.m.'`, `'midnight'`, `'noon'`, `'12:30 p.m.'`
Timezone		
`e`	Timezone name. Could be in any format, or might return an empty string, depending on the datetime.	`''`, `'GMT'`, `'-500'`, `'US/Eastern'`, etc.
`I`	Daylight saving time, whether it’s in effect or not.	`'1'` or `'0'`
`O`	Difference to Greenwich time in hours.	`'+0200'`
`T`	Time zone of this machine.	`'EST'`, `'MDT'`
`Z`	Time zone offset in seconds. The offset for timezones west of UTC is always negative, and for those east of UTC is always positive.	`-43200` to `43200`
Date/Time		
`c`	ISO 8601 format. (Note: unlike other formatters, such as “Z”, “O” or “r”, the “c” formatter will not add timezone offset if value is a naive datetime (see `datetime.tzinfo`).	`2008-01-02T10:30:00.000123+02:00`, or `2008-01-02T10:30:00.000123` if the datetime is naive
`r`	**RFC 5322** formatted date.	`'Thu, 21 Dec 2000 16:01:07 +0200'`

Format character	Description	Example output
`U`	Seconds since the Unix Epoch (January 1 1970 00:00:00 UTC).	

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, the `DATE_FORMAT` format specifier is used. Assuming the same settings as the previous example:

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

outputs `9 de Enero de 2008` (the `DATE_FORMAT` format specifier for the `es` locale is `r'j \d\e F \d\e Y'`). Both “d” and “e” are backslash-escaped, because otherwise each is a format string that displays the day and the timezone name, respectively.

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" }}
```

`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`.

`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 `nothing`.

`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)
```

`dictsort` can also order a list of lists (or any other object implementing `__getitem__()`) by elements at specified index. For example:

```
{{ value|dictsort:0 }}
```

If `value` is:

```
[
    ('a', '42'),
    ('c', 'string'),
    ('b', 'foo'),
]
```

then the output would be:

```
[
    ('a', '42'),
    ('b', 'foo'),
    ('c', 'string'),
]
```

You must pass the index as an integer rather than a string. The following produce empty output:

```
{{ values|dictsort:"0" }}
```

Ordering by elements at specified index is not supported on dictionaries.

Changed in Django 2.2.26:
In older versions, ordering elements at specified index was supported on dictionaries.

`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.

`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`.

`escape` ¶

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

- `<` is converted to `<`
- `>` is converted to `>`
- `'` (single quote) is converted to `&#x27;`
- `"` (double quote) is converted to `&quot;`
- `&` is converted to `&`

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 %}
```

`escapejs` ¶

Escapes characters for use in JavaScript strings. This does *not* make the string safe for use in HTML or JavaScript template literals, 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"`.

`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.

`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'`.

`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`

value	Template	Output
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

If the argument passed to `floatformat` has the `g` suffix, it will force grouping by the `THOUSAND_SEPARATOR` for the active locale. For example, when the active locale is `en` (English):

value	Template	Output
34232.34	{{ value|floatformat:"2g" }}	34,232.34
34232.06	{{ value|floatformat:"g" }}	34,232.1
34232.00	{{ value|floatformat:"-3g" }}	34,232

Output is always localized (independently of the `{% localize off %}` tag) unless the argument passed to `floatformat` has the `u` suffix, which will force disabling localization. For example, when the active locale is `pl` (Polish):

value	Template	Output
34.23234	{{ value|floatformat:"3" }}	34,232
34.23234	{{ value|floatformat:"3u" }}	34.232

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

Changed in Django 3.2:
The `g` suffix to force grouping by thousand separators was added.

Changed in Django 4.0:
`floatformat` template filter no longer depends on the `USE_L10N` setting and always returns localized output.

The `u` suffix to force disabling localization was added.

`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 %}
```

`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`.

`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&me=2"`.

`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"`.

`json_script` ¶

Safely outputs a Python object as JSON, wrapped in a `<script>` tag, ready for use with JavaScript.

Argument: HTML "id" of the `<script>` tag.

For example:

```
{{ value|json_script:"hello-data" }}
```

If `value` is the dictionary `{'hello': 'world'}`, the output will be:

```
<script id="hello-data" type="application/json">{"hello": "world"}</script>
```

The resulting data can be accessed in JavaScript like this:

```
const value = JSON.parse(document.getElementById('hello-data').textContent);
```

XSS attacks are mitigated by escaping the characters "<", ">" and "&". For example if `value` is `{'hello': 'world</script>&'}`, the output is:

```
<script id="hello-data" type="application/json">{"hello":
"world\\u003C/script\\u003E\\u0026amp;"}</script>
```

This is compatible with a strict Content Security Policy that prohibits in-page script execution. It also maintains a clean separation between passive data and executable code.

`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"`.

`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.

`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`.

`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>`.

`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`.

`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
```

`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    "`.

`lower` ¶

Converts a string into all lowercase.

For example:

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

If `value` is `Totally LOVING this Album!`, the output will be `totally loving this album!`.

`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 to a 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']`.

`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`.

`pluralize` ¶

Returns a plural suffix if the value is not `1`, `'1'`, or an object of length 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 `blocktranslate` to pluralize translated strings.

`pprint` ¶

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

`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"`.

`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"`.

`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 }}
```

`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.

`slice` ¶

Returns a slice of the list.

Uses the same syntax as Python's list slicing. See https://diveinto.org/python3/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']`.

`slugify` ¶

Converts to ASCII. Converts spaces to hyphens. Removes characters that aren't alphanumerics, underscores, or hyphens. Converts to lowercase. 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"`.

`stringformat` ¶

Formats the variable according to the argument, a string formatting specifier. This specifier uses the printf-style String Formatting syntax, with the exception that the leading "%" is dropped.

For example:

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

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

`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 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.

`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"`.

Note that you can backslash-escape a format string if you want to use the “raw” value. In this example, both “h” and “m” are backslash-escaped, because otherwise each is a format string that displays the hour and the month, respectively:

```
{{ value|time:"H\h i\m" }}
```

This would display as “01h 23m”.

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"` (The `"TIME_FORMAT"` format specifier for the `de` locale as shipped with Django is `"H:i"`).

The `time` filter will only accept parameters in the format string that relate to the time of day, not the date. If you need to format a `date` value, use the `date` filter instead (or along with `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, the `TIME_FORMAT` format specifier is used:

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

is the same as:

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

`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.

`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.

`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"`.

`truncatechars` ¶

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

Argument: Number of characters to truncate to

For example:

```
{{ value|truncatechars:7 }}
```

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

`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:7 }}
```

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.

`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.

`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.

`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>
```

`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"`.

`urlencode` ¶

Escapes a value for use in a URL.

For example:

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

If `value` is `"https://www.example.org/foo?a=b&c=d"`, the output will be `"https%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 `"https://www.example.org/"`, the output will be `"https%3A%2F%2Fwww.example.org%2F"`.

`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, `https://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.

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.com</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, or to email addresses that contain single quotes (`'`), things won't work as expected. Apply this filter only to plain text.

`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.djangoproj…</a>'` .

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

`wordcount` ¶

Returns the number of words.

For example:

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

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

`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
```

`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)

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.

`i18n` ¶

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

See Internationalization: in template code.

`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 Controlling localization in templates.

`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 zone aware output in templates.

Other tags and filters libraries¶

Django comes with a couple of other template-tag libraries that you have to enable explicitly in your `INSTALLED_APPS` setting and enable in your template with the `{% load %}` tag.

`django.contrib.humanize` ¶

A set of Django template filters useful for adding a "human touch" to data. See django.contrib.humanize.

`static` ¶

`static` ¶

To link to static files that are saved in `STATIC_ROOT` Django ships with a `static` template tag. If the `django.contrib.staticfiles` app is installed, the tag will serve files using `url()` method of the storage specified by `STATICFILES_STORAGE`. For example:

```
{% load static %}
<img src="{% static 'images/hi.jpg' %}" alt="Hi!">
```

It is also able to consume standard context variables, e.g. assuming a `user_stylesheet` variable is passed to the template:

```
{% load static %}
<link rel="stylesheet" href="{% static user_stylesheet %}" type="text/css"
media="screen">
```

If you'd like to retrieve a static URL without displaying it, you can use a slightly different call:

```
{% load static %}
{% static "images/hi.jpg" as myphoto %}
<img src="{{ myphoto }}">
```

Using Jinja2 templates?

See Jinja2 for information on using the `static` tag with Jinja2.

`get_static_prefix` ¶

You should prefer the `static` template tag, but if you need more control over exactly where and how `STATIC_URL` is injected into the template, you can use the `get_static_prefix` template tag:

```
{% load static %}
<img src="{% get_static_prefix %}images/hi.jpg" alt="Hi!">
```

There's also a second form you can use to avoid extra processing if you need the value multiple times:

```
{% load static %}
{% get_static_prefix as STATIC_PREFIX %}

<img src="{{ STATIC_PREFIX }}images/hi.jpg" alt="Hi!">
<img src="{{ STATIC_PREFIX }}images/hi2.jpg" alt="Hello!">
```

`get_media_prefix` ¶

Similar to the `get_static_prefix`, `get_media_prefix` populates a template variable with the media prefix `MEDIA_URL`, e.g.:

```
{% load static %}
<body data-media-url="{% get_media_prefix %}">
```

By storing the value in a data attribute, we ensure it's escaped appropriately if we want to use it in a JavaScript context.