Template Designer Documentation

This document describes the syntax and semantics of the template engine and will be most useful as reference to those creating Jinja templates. As the template engine is very flexible, the configuration from the application can be slightly different from the code presented here in terms of delimiters and behavior of undefined values.

Synopsis

A Jinja template is simply a text file. Jinja can generate any text-based format (HTML, XML, CSV, LaTeX, etc.). A Jinja template doesn't need to have a specific extension: `.html`, `.xml`, or any other extension is just fine.

A template contains **variables** and/or **expressions**, which get replaced with values when a template is *rendered*; and **tags**, which control the logic of the template. The template syntax is heavily inspired by Django and Python.

Below is a minimal template that illustrates a few basics using the default Jinja configuration. We will cover the details later in this document:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>My Webpage</title>
</head>
<body>
    <ul id="navigation">
    {% for item in navigation %}
        <li><a href="{{ item.href }}">{{ item.caption }}</a></li>
    {% endfor %}
    </ul>

    <h1>My Webpage</h1>
    {{ a_variable }}

    {# a comment #}
</body>
</html>
```

The following example shows the default configuration settings. An application developer can change the syntax configuration from `{% foo %}` to `<% foo %>`, or something similar.

There are a few kinds of delimiters. The default Jinja delimiters are configured as follows:

- `{% ... %}` for Statements
- `{{ ... }}` for Expressions to print to the template output
- `{# ... #}` for Comments not included in the template output
- `#  ... ##` for Line Statements

Variables

Template variables are defined by the context dictionary passed to the template.

You can mess around with the variables in templates provided they are passed in by the application. Variables may have attributes or elements on them you can access too. What attributes a variable has depends heavily on the application providing that variable.

You can use a dot (`.`) to access attributes of a variable in addition to the standard Python `__getitem__` "subscript" syntax (`[]`).

The following lines do the same thing:

```
{{ foo.bar }}
{{ foo['bar'] }}
```

It's important to know that the outer double-curly braces are *not* part of the variable, but the print statement. If you access variables inside tags don't put the braces around them.

If a variable or attribute does not exist, you will get back an undefined value. What you can do with that kind of value depends on the application configuration: the default behavior is to evaluate to an empty string if printed or iterated over, and to fail for every other operation.

Implementation:

For the sake of convenience, `foo.bar` in Jinja2 does the following things on the Python layer:

- check for an attribute called *bar* on *foo* (`getattr(foo, 'bar')`)
- if there is not, check for an item `'bar'` in *foo* (`foo.__getitem__('bar')`)
- if there is not, return an undefined object.

`foo['bar']` works mostly the same with a small difference in sequence:

- check for an item `'bar'` in *foo*. (`foo.__getitem__('bar')`)
- if there is not, check for an attribute called *bar* on *foo*. (`getattr(foo, 'bar')`)
- if there is not, return an undefined object.

This is important if an object has an item and attribute with the same name. Additionally, the **`attr()`** filter only looks up attributes.

Filters

Variables can be modified by **filters**. Filters are separated from the variable by a pipe symbol (`|`) and may have optional arguments in parentheses. Multiple filters can be chained. The output of one filter is applied to the next.

For example, `{{ name|striptags|title }}` will remove all HTML Tags from variable *name* and title-case the output (`title(striptags(name))`).

Filters that accept arguments have parentheses around the arguments, just like a function call. For example: `{{ listx|join(', ') }}` will join a list with commas (`str.join(', ', listx)`).

The List of Builtin Filters below describes all the builtin filters.

Tests

Beside filters, there are also so-called "tests" available. Tests can be used to test a variable against a common expression. To test a variable or expression, you add *is* plus the name of the test after the variable. For example, to find out if a variable is defined, you can do `name is defined`, which will then return true or false depending on whether *name* is defined in the current template context.

Tests can accept arguments, too. If the test only takes one argument, you can leave out the parentheses. For example, the following two expressions do the same thing:

```
{% if loop.index is divisibleby 3 %}
{% if loop.index is divisibleby(3) %}
```

The List of Builtin Tests below describes all the builtin tests.

Comments

To comment-out part of a line in a template, use the comment syntax which is by default set to `{# ... #}`. This is useful to comment out parts of the template for debugging or to add information for other template designers or yourself:

```
{# note: commented-out template because we no longer use this
    {% for user in users %}
        ...
    {% endfor %}
#}
```

Whitespace Control

In the default configuration:

- a single trailing newline is stripped if present
- other whitespace (spaces, tabs, newlines etc.) is returned unchanged

If an application configures Jinja to *trim_blocks*, the first newline after a template tag is removed automatically (like in PHP). The *lstrip_blocks* option can also be set to strip tabs and spaces from the beginning of a line to the start of a block. (Nothing will be stripped if there are other characters before the start of the block.)

With both *trim_blocks* and *lstrip_blocks* enabled, you can put block tags on their own lines, and the entire block line will be removed when rendered, preserving the whitespace of the contents. For example, without the *trim_blocks* and *lstrip_blocks* options, this template:

```
<div>
    {% if True %}
        yay
    {% endif %}
</div>
```

gets rendered with blank lines inside the div:

```
<div>

        yay

</div>
```

But with both *trim_blocks* and *lstrip_blocks* enabled, the template block lines are removed and other whitespace is preserved:

```
<div>
        yay
</div>
```

You can manually disable the *lstrip_blocks* behavior by putting a plus sign (`+`) at the start of a block:

```
<div>
        {%+ if something %}yay{% endif %}
</div>
```

You can also strip whitespace in templates by hand. If you add a minus sign (`-`) to the start or end of a block (e.g. a For tag), a comment, or a variable expression, the whitespaces before or after that block will be removed:

```
{% for item in seq -%}
    {{ item }}
{%- endfor %}
```

This will yield all elements without whitespace between them. If *seq* was a list of numbers from `1` to `9`, the output would be `123456789`.

If Line Statements are enabled, they strip leading whitespace automatically up to the beginning of the line.

By default, Jinja2 also removes trailing newlines. To keep single trailing newlines, configure Jinja to *keep_trailing_newline*.

Note:

You must not add whitespace between the tag and the minus sign.

valid:

```
{%- if foo -%}...{% endif %}
```

invalid:

```
{% - if foo - %}...{% endif %}
```

Escaping

It is sometimes desirable – even necessary – to have Jinja ignore parts it would otherwise handle as variables or blocks. For example, if, with the default syntax, you want to use `{{` as a raw string in a template and not start a variable, you have to use a trick.

The easiest way to output a literal variable delimiter (`{{`) is by using a variable expression:

```
{{ '{{' }}
```

For bigger sections, it makes sense to mark a block *raw*. For example, to include example Jinja syntax in a template, you can use this snippet:

```
{% raw %}
    <ul>
    {% for item in seq %}
        <li>{{ item }}</li>
    {% endfor %}
    </ul>
{% endraw %}
```

Line Statements

If line statements are enabled by the application, it's possible to mark a line as a statement. For example, if the line statement prefix is configured to `#`, the following two examples are equivalent:

```
<ul>
# for item in seq
    <li>{{ item }}</li>
# endfor
</ul>

<ul>
{% for item in seq %}
    <li>{{ item }}</li>
{% endfor %}
</ul>
```

The line statement prefix can appear anywhere on the line as long as no text precedes it. For better readability, statements that start a block (such as *for*, *if*, *elif* etc.) may end with a colon:

```
# for item in seq:
    ...
# endfor
```

Note:

Line statements can span multiple lines if there are open parentheses, braces or brackets:

```
<ul>
# for href, caption in [('index.html', 'Index'),
                        ('about.html', 'About')]:
    <li><a href="{{ href }}">{{ caption }}</a></li>
# endfor
</ul>
```

Since Jinja 2.2, line-based comments are available as well. For example, if the line-comment prefix is configured to be `##`, everything from `##` to the end of the line is ignored (excluding the newline sign):

```
# for item in seq:
    <li>{{ item }}</li>     ## this comment is ignored
# endfor
```

Template Inheritance

The most powerful part of Jinja 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.

Sounds complicated but is very basic. It's easiest to understand it by starting with an example.

Base Template

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:

```
<!DOCTYPE html>
<html lang="en">
<head>
    {% block head %}
    <link rel="stylesheet" href="style.css" />
    <title>{% block title %}{% endblock %} - My Webpage</title>
    {% endblock %}
</head>
<body>
    <div id="content">{% block content %}{% endblock %}</div>
    <div id="footer">
        {% block footer %}
        &copy; Copyright 2008 by <a href="http://domain.invalid/">you</a>.
        {% endblock %}
    </div>
</body>
</html>
```

In this example, the `{% block %}` tags define four blocks that child templates can fill in. All the *block* tag does is tell the template engine that a child template may override those placeholders in the template.

Child Template

A child template might look like this:

```
{% extends "base.html" %}
{% block title %}Index{% endblock %}
{% block head %}
    {{ super() }}
    <style type="text/css">
        .important { color: #336699; }
    </style>
{% endblock %}
{% block content %}
    <h1>Index</h1>
    <p class="important">
      Welcome to my awesome homepage.
    </p>
{% 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, it first locates the parent. The extends tag should be the first tag in the template. Everything before it is printed out normally and may cause confusion. For details about this behavior and how to take advantage of it, see Null-Master Fallback.

The filename of the template depends on the template loader. For example, the **`FileSystemLoader`** allows you to access other templates by giving the filename. You can access templates in subdirectories with a slash:

```
{% extends "layout/default.html" %}
```

But this behavior can depend on the application embedding Jinja. Note that since the child template doesn't define the `footer` block, the value from the parent template is used instead.

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 placeholder to fill - it also defines the content that fills the placeholder 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.

If you want to print a block multiple times, you can, however, use the special *self* variable and call the block with that name:

```
<title>{% block title %}{% endblock %}</title>
<h1>{{ self.title() }}</h1>
{% block body %}{% endblock %}
```

Super Blocks

It's possible to render the contents of the parent block by calling *super*. This gives back the results of the parent block:

```
{% block sidebar %}
    <h3>Table Of Contents</h3>
    ...
    {{ super() }}
{% endblock %}
```

Named Block End-Tags

Jinja2 allows you to put the name of the block after the end tag for better readability:

```
{% block sidebar %}
    {% block inner_sidebar %}
        ...
    {% endblock inner_sidebar %}
{% endblock sidebar %}
```

However, the name after the *endblock* word must match the block name.

Block Nesting and Scope

Blocks can be nested for more complex layouts. However, per default blocks may not access variables from outer scopes:

```
{% for item in seq %}
    <li>{% block loop_item %}{{ item }}{% endblock %}</li>
{% endfor %}
```

This example would output empty `<li>` items because *item* is unavailable inside the block. The reason for this is that if the block is replaced by a child template, a variable would appear that was not defined in the block or passed to the context.

Starting with Jinja 2.2, you can explicitly specify that variables are available in a block by setting the block to "scoped" by adding the *scoped* modifier to a block declaration:

```
{% for item in seq %}
    <li>{% block loop_item scoped %}{{ item }}{% endblock %}</li>
{% endfor %}
```

When overriding a block, the *scoped* modifier does not have to be provided.

Template Objects

Changed in version 2.4.

If a template object was passed in the template context, you can extend from that object as well. Assuming the calling code passes a layout template as *layout_template* to the environment, this code works:

```
{% extends layout_template %}
```

Previously, the *layout_template* variable had to be a string with the layout template's filename for this to work.

HTML Escaping

When generating HTML from templates, there's always a risk that a variable will include characters that affect the resulting HTML. There are two approaches:

a. manually escaping each variable; or
b. automatically escaping everything by default.

Jinja supports both. What is used depends on the application configuration. The default configuration is no automatic escaping; for various reasons:

- Escaping everything except for safe values will also mean that Jinja is escaping variables known to not include HTML (e.g. numbers, booleans) which can be a huge performance hit.
- The information about the safety of a variable is very fragile. It could happen that by coercing safe and unsafe values, the return value is double-escaped HTML.

Working with Manual Escaping

If manual escaping is enabled, it's **your** responsibility to escape variables if needed. What to escape? If you have a variable that *may* include any of the following chars (`>`, `<`, `&`, or `"`) you **SHOULD** escape it unless the variable contains well-formed and trusted HTML. Escaping works by piping the variable through the `|e` filter:

```
{{ user.username|e }}
```

Working with Automatic Escaping

When automatic escaping is enabled, everything is escaped by default except for values explicitly marked as safe. Variables and expressions can be marked as safe either in:

a. the context dictionary by the application with *MarkupSafe.Markup*, or
b. the template, with the *|safe* filter

The main problem with this approach is that Python itself doesn't have the concept of tainted values; so whether a value is safe or unsafe can get lost.

If a value is not marked safe, auto-escaping will take place; which means that you could end up with double-escaped contents. Double-escaping is easy to avoid, however: just rely on the tools Jinja2 provides and *don't use builtin Python constructs such as str.format or the string modulo operator (%)*.

Jinja2 functions (macros, *super*, *self.BLOCKNAME*) always return template data that is marked as safe.

String literals in templates with automatic escaping are considered unsafe because native Python strings (`str`, `unicode`, `basestring`) are not *MarkupSafe.Markup* strings with an `__html__` attribute.

List of Control Structures

A control structure refers to all those things that control the flow of a program - conditionals (i.e. if/elif/else), for-loops, as well as things like macros and blocks. With the default syntax, control structures appear inside `{% ... %}` blocks.

For

Loop over each item in a sequence. For example, to display a list of users provided in a variable called *users*:

```
<h1>Members</h1>
<ul>
{% for user in users %}
  <li>{{ user.username|e }}</li>
{% endfor %}
</ul>
```

As variables in templates retain their object properties, it is possible to iterate over containers like *dict*:

```
<dl>
{% for key, value in my_dict.iteritems() %}
    <dt>{{ key|e }}</dt>
    <dd>{{ value|e }}</dd>
{% endfor %}
</dl>
```

Note, however, that **Python dicts are not ordered**; so you might want to either pass a sorted `list` of `tuple` s – or a `collections.OrderedDict` – to the template, or use the *dictsort* filter.

Inside of a for-loop block, you can access some special variables:

Variable	Description
loop.index	The current iteration of the loop. (1 indexed)
loop.indexo	The current iteration of the loop. (0 indexed)
loop.revindex	The number of iterations from the end of the loop (1 indexed)
loop.revindexo	The number of iterations from the end of the loop (0 indexed)
loop.first	True if first iteration.
loop.last	True if last iteration.
loop.length	The number of items in the sequence.
loop.cycle	A helper function to cycle between a list of sequences. See the explanation below.
loop.depth	Indicates how deep in a recursive loop the rendering currently is. Starts at level 1
loop.depth0	Indicates how deep in a recursive loop the rendering currently is. Starts at level 0
loop.previtem	The item from the previous iteration of the loop. Undefined during the first iteration.
loop.nextitem	The item from the following iteration of the loop. Undefined during the last iteration.
*loop.changed(*val)*	True if previously called with a different value (or not called at all).

Within a for-loop, it's possible to cycle among a list of strings/variables each time through the loop by using the special *loop.cycle* helper:

```
{% for row in rows %}
    <li class="{{ loop.cycle('odd', 'even') }}">{{ row }}</li>
{% endfor %}
```

Since Jinja 2.1, an extra *cycle* helper exists that allows loop-unbound cycling. For more information, have a look at the List of Global Functions.

Unlike in Python, it's not possible to *break* or *continue* in a loop. You can, however, filter the sequence during iteration, which allows you to skip items. The following example skips all the users which are hidden:

```
{% for user in users if not user.hidden %}
    <li>{{ user.username|e }}</li>
{% endfor %}
```

The advantage is that the special *loop* variable will count correctly; thus not counting the users not iterated over.

If no iteration took place because the sequence was empty or the filtering removed all the items from the sequence, you can render a default block by using *else*:

```
<ul>
{% for user in users %}
    <li>{{ user.username|e }}</li>
{% else %}
    <li><em>no users found</em></li>
{% endfor %}
</ul>
```

Note that, in Python, *else* blocks are executed whenever the corresponding loop **did not** *break*. Since Jinja loops cannot *break* anyway, a slightly different behavior of the *else* keyword was chosen.

It is also possible to use loops recursively. This is useful if you are dealing with recursive data such as sitemaps or RDFa. To use loops recursively, you basically have to add the *recursive* modifier to the loop definition and call the *loop* variable with the new iterable where you want to recurse.

The following example implements a sitemap with recursive loops:

```
<ul class="sitemap">
{%- for item in sitemap recursive %}
    <li><a href="{{ item.href|e }}">{{ item.title }}</a>
    {%- if item.children -%}
        <ul class="submenu">{{ loop(item.children) }}</ul>
    {%- endif %}</li>
{%- endfor %}
</ul>
```

The *loop* variable always refers to the closest (innermost) loop. If we have more than one level of loops, we can rebind the variable *loop* by writing *{% set outer_loop = loop %}* after the loop that we want to use recursively. Then, we can call it using *{{ outer_loop(...) }}*

Please note that assignments in loops will be cleared at the end of the iteration and cannot outlive the loop scope. Older versions of Jinja2 had a bug where in some circumstances it appeared that assignments would work. This is not supported. See Assignments for more information about how to deal with this.

If all you want to do is check whether some value has changed since the last iteration or will change in the next iteration, you can use *previtem* and *nextitem*:

```
{% for value in values %}
    {% if loop.previtem is defined and value > loop.previtem %}
        The value just increased!
    {% endif %}
    {{ value }}
    {% if loop.nextitem is defined and loop.nextitem > value %}
        The value will increase even more!
    {% endif %}
{% endfor %}
```

If you only care whether the value changed at all, using *changed* is even easier:

```
{% for entry in entries %}
    {% if loop.changed(entry.category) %}
        <h2>{{ entry.category }}</h2>
    {% endif %}
    <p>{{ entry.message }}</p>
{% endfor %}
```

If

The *if* statement in Jinja is comparable with the Python if statement. In the simplest form, you can use it to test if a variable is defined, not empty and not false:

```
{% if users %}
<ul>
{% for user in users %}
    <li>{{ user.username|e }}</li>
{% endfor %}
</ul>
{% endif %}
```

For multiple branches, *elif* and *else* can be used like in Python. You can use more complex Expressions there, too:

```
{% if kenny.sick %}
    Kenny is sick.
{% elif kenny.dead %}
    You killed Kenny!  You bastard!!!
{% else %}
    Kenny looks okay --- so far
{% endif %}
```

If can also be used as an inline expression and for loop filtering.

Macros

Macros are comparable with functions in regular programming languages. They are useful to put often used idioms into reusable functions to not repeat yourself (“DRY”).

Here’s a small example of a macro that renders a form element:

```
{% macro input(name, value='', type='text', size=20) -%}
    <input type="{{ type }}" name="{{ name }}" value="{{
        value|e }}" size="{{ size }}">
{%- endmacro %}
```

The macro can then be called like a function in the namespace:

```
<p>{{ input('username') }}</p>
<p>{{ input('password', type='password') }}</p>
```

If the macro was defined in a different template, you have to import it first.

Inside macros, you have access to three special variables:

varargs
: If more positional arguments are passed to the macro than accepted by the macro, they end up in the special *varargs* variable as a list of values.

kwargs
: Like *varargs* but for keyword arguments. All unconsumed keyword arguments are stored in this special variable.

caller
: If the macro was called from a call tag, the caller is stored in this variable as a callable macro.

Macros also expose some of their internal details. The following attributes are available on a macro object:

name
: The name of the macro. `{{ input.name }}` will print `input`.

arguments
: A tuple of the names of arguments the macro accepts.

defaults
: A tuple of default values.

catch_kwargs
: This is *true* if the macro accepts extra keyword arguments (i.e.: accesses the special *kwargs* variable).

catch_varargs
: This is *true* if the macro accepts extra positional arguments (i.e.: accesses the special *varargs* variable).

caller
: This is *true* if the macro accesses the special *caller* variable and may be called from a call tag.

If a macro name starts with an underscore, it's not exported and can't be imported.

Call

In some cases it can be useful to pass a macro to another macro. For this purpose, you can use the special *call* block. The following example shows a macro that takes advantage of the call functionality and how it can be used:

```
{% macro render_dialog(title, class='dialog') -%}
    <div class="{{ class }}">
        <h2>{{ title }}</h2>
        <div class="contents">
            {{ caller() }}
        </div>
    </div>
{%- endmacro %}

{% call render_dialog('Hello World') %}
    This is a simple dialog rendered by using a macro and
    a call block.
{% endcall %}
```

It's also possible to pass arguments back to the call block. This makes it useful as a replacement for loops. Generally speaking, a call block works exactly like a macro without a name.

Here's an example of how a call block can be used with arguments:

```
{% macro dump_users(users) -%}
    <ul>
    {%- for user in users %}
        <li><p>{{ user.username|e }}</p>{{ caller(user) }}</li>
    {%- endfor %}
    </ul>
{%- endmacro %}

{% call(user) dump_users(list_of_user) %}
    <dl>
        <dl>Realname</dl>
        <dd>{{ user.realname|e }}</dd>
        <dl>Description</dl>
        <dd>{{ user.description }}</dd>
    </dl>
{% endcall %}
```

Filters

Filter sections allow you to apply regular Jinja2 filters on a block of template data. Just wrap the code in the special *filter* section:

```
{% filter upper %}
    This text becomes uppercase
{% endfilter %}
```

Assignments

Inside code blocks, you can also assign values to variables. Assignments at top level (outside of blocks, macros or loops) are exported from the template like top level macros and can be imported by other templates.

Assignments use the *set* tag and can have multiple targets:

```
{% set navigation = [('index.html', 'Index'), ('about.html', 'About')] %}
{% set key, value = call_something() %}
```

Scoping Behavior:

Please keep in mind that it is not possible to set variables inside a block and have them show up outside of it. This also applies to loops. The only exception to that rule are if statements which do not introduce a scope. As a result the following template is not going to do what you might expect:

```
{% set iterated = false %}
{% for item in seq %}
    {{ item }}
    {% set iterated = true %}
{% endfor %}
{% if not iterated %} did not iterate {% endif %}
```

It is not possible with Jinja syntax to do this. Instead use alternative constructs like the loop else block or the special *loop* variable:

```
{% for item in seq %}
    {{ item }}
{% else %}
```

```
    did not iterate
{% endfor %}
```

As of version 2.10 more complex use cases can be handled using namespace objects which allow propagating of changes across scopes:

```
{% set ns = namespace(found=false) %}
{% for item in items %}
    {% if item.check_something() %}
        {% set ns.found = true %}
    {% endif %}
    * {{ item.title }}
{% endfor %}
Found item having something: {{ ns.found }}
```

Note hat the `obj.attr` notation in the *set* tag is only allowed for namespace objects; attempting to assign an attribute on any other object will raise an exception.

New in version 2.10: Added support for namespace objects

Block Assignments

New in version 2.8.

Starting with Jinja 2.8, it's possible to also use block assignments to capture the contents of a block into a variable name. This can be useful in some situations as an alternative for macros. In that case, instead of using an equals sign and a value, you just write the variable name and then everything until `{% endset %}` is captured.

Example:

```
{% set navigation %}
    <li><a href="/">Index</a>
    <li><a href="/downloads">Downloads</a>
{% endset %}
```

The *navigation* variable then contains the navigation HTML source.

Changed in version 2.10.

Starting with Jinja 2.10, the block assignment supports filters.

Example:

```
{% set reply | wordwrap %}
    You wrote:
    {{ message }}
{% endset %}
```

Extends

The *extends* tag can be used to extend one template from another. You can have multiple *extends* tags in a file, but only one of them may be executed at a time.

See the section about Template Inheritance above.

Blocks

Blocks are used for inheritance and act as both placeholders and replacements at the same time. They are documented in detail in the Template Inheritance section.

Include

The *include* statement is useful to include a template and return the rendered contents of that file into the current namespace:

```
{% include 'header.html' %}
    Body
{% include 'footer.html' %}
```

Included templates have access to the variables of the active context by default. For more details about context behavior of imports and includes, see Import Context Behavior.

From Jinja 2.2 onwards, you can mark an include with `ignore missing`; in which case Jinja will ignore the statement if the template to be included does not exist. When combined with `with` or `without context`, it must be placed *before* the context visibility statement. Here are some valid examples:

```
{% include "sidebar.html" ignore missing %}
{% include "sidebar.html" ignore missing with context %}
{% include "sidebar.html" ignore missing without context %}
```

New in version 2.2.

You can also provide a list of templates that are checked for existence before inclusion. The first template that exists will be included. If *ignore missing* is given, it will fall back to rendering nothing if none of the templates exist, otherwise it will raise an exception.

Example:

```
{% include ['page_detailed.html', 'page.html'] %}
{% include ['special_sidebar.html', 'sidebar.html'] ignore missing %}
```

Changed in version 2.4: If a template object was passed to the template context, you can include that object using *include*.

Import

Jinja2 supports putting often used code into macros. These macros can go into different templates and get imported from there. This works similarly to the import statements in Python. It's important to know that imports are cached and imported templates don't have access to the current template variables, just the globals by default. For more details about context behavior of imports and includes, see Import Context Behavior.

There are two ways to import templates. You can import a complete template into a variable or request specific macros / exported variables from it.

Imagine we have a helper module that renders forms (called *forms.html*):

```
{% macro input(name, value='', type='text') -%}
    <input type="{{ type }}" value="{{ value|e }}" name="{{ name }}">
{%- endmacro %}
```

```
{%- macro textarea(name, value='', rows=10, cols=40) -%}
    <textarea name="{{ name }}" rows="{{ rows }}" cols="{{ cols
        }}">{{ value|e }}</textarea>
{%- endmacro %}
```

The easiest and most flexible way to access a template's variables and macros is to import the whole template module into a variable. That way, you can access the attributes:

```
{% import 'forms.html' as forms %}
<dl>
    <dt>Username</dt>
    <dd>{{ forms.input('username') }}</dd>
    <dt>Password</dt>
    <dd>{{ forms.input('password', type='password') }}</dd>
</dl>
<p>{{ forms.textarea('comment') }}</p>
```

Alternatively, you can import specific names from a template into the current namespace:

```
{% from 'forms.html' import input as input_field, textarea %}
<dl>
    <dt>Username</dt>
    <dd>{{ input_field('username') }}</dd>
    <dt>Password</dt>
    <dd>{{ input_field('password', type='password') }}</dd>
</dl>
<p>{{ textarea('comment') }}</p>
```

Macros and variables starting with one or more underscores are private and cannot be imported.

Changed in version 2.4: If a template object was passed to the template context, you can import from that object.

Import Context Behavior

By default, included templates are passed the current context and imported templates are not. The reason for this is that imports, unlike includes, are cached; as imports are often used just as a module that holds macros.

This behavior can be changed explicitly: by adding *with context* or *without context* to the import/include directive, the current context can be passed to the template and caching is disabled automatically.

Here are two examples:

```
{% from 'forms.html' import input with context %}
{% include 'header.html' without context %}
```

Note:

In Jinja 2.0, the context that was passed to the included template did not include variables defined in the template. As a matter of fact, this did not work:

```
{% for box in boxes %}
    {% include "render_box.html" %}
{% endfor %}
```

The included template `render_box.html` is *not* able to access *box* in Jinja 2.0. As of Jinja 2.1, `render_box.html` *is* able to do so.

Expressions

Jinja allows basic expressions everywhere. These work very similarly to regular Python; even if you're not working with Python you should feel comfortable with it.

Literals

The simplest form of expressions are literals. Literals are representations for Python objects such as strings and numbers. The following literals exist:

"Hello World":
: Everything between two double or single quotes is a string. They are useful whenever you need a string in the template (e.g. as arguments to function calls and filters, or just to extend or include a template).

42 / 42.23:
: Integers and floating point numbers are created by just writing the number down. If a dot is present, the number is a float, otherwise an integer. Keep in mind that, in Python, `42` and `42.0` are different (`int` and `float`, respectively).

['list', 'of', 'objects']:
: Everything between two brackets is a list. Lists are useful for storing sequential data to be iterated over. For example, you can easily create a list of links using lists and tuples for (and with) a for loop:

```
<ul>
{% for href, caption in [('index.html', 'Index'), ('about.html', 'About'),
                         ('downloads.html', 'Downloads')] %}
    <li><a href="{{ href }}">{{ caption }}</a></li>
{% endfor %}
</ul>
```

('tuple', 'of', 'values'):
: Tuples are like lists that cannot be modified ("immutable"). If a tuple only has one item, it must be followed by a comma (`('1-tuple',)`). Tuples are usually used to represent items of two or more elements. See the list example above for more details.

{'dict': 'of', 'key': 'and', 'value': 'pairs'}:
: A dict in Python is a structure that combines keys and values. Keys must be unique and always have exactly one value. Dicts are rarely used in templates; they are useful in some rare cases such as the **`xmlattr()`** filter.

true / false:
: true is always true and false is always false.

Note:

The special constants *true*, *false*, and *none* are indeed lowercase. Because that caused confusion in the past, (*True* used to expand to an undefined variable that was considered false), all three can now also be written in title case (*True*, *False*, and *None*). However, for consistency, (all Jinja identifiers are lowercase) you should use the lowercase versions.

Math

Jinja allows you to calculate with values. This is rarely useful in templates but exists for completeness' sake. The following operators are supported:

+

Adds two objects together. Usually the objects are numbers, but if both are strings or lists, you can concatenate them this way. This, however, is not the preferred way to concatenate strings! For string concatenation, have a look-see at the ~ operator. `{{ 1 + 1 }}` is `2`.

-
: Subtract the second number from the first one. `{{ 3 - 2 }}` is `1`.

/
: Divide two numbers. The return value will be a floating point number. `{{ 1 / 2 }}` is `{{ 0.5 }}`. (Just like `from __future__ import division`.)

//
: Divide two numbers and return the truncated integer result. `{{ 20 // 7 }}` is `2`.

%
: Calculate the remainder of an integer division. `{{ 11 % 7 }}` is `4`.

*
: Multiply the left operand with the right one. `{{ 2 * 2 }}` would return `4`. This can also be used to repeat a string multiple times. `{{ '=' * 80 }}` would print a bar of 80 equal signs.

**
: Raise the left operand to the power of the right operand. `{{ 2**3 }}` would return `8`.

Comparisons

==
: Compares two objects for equality.

!=
: Compares two objects for inequality.

>
: *true* if the left hand side is greater than the right hand side.

>=
: *true* if the left hand side is greater or equal to the right hand side.

<
: *true* if the left hand side is lower than the right hand side.

<=
: *true* if the left hand side is lower or equal to the right hand side.

Logic

For *if* statements, *for* filtering, and *if* expressions, it can be useful to combine multiple expressions:

and
: Return true if the left and the right operand are true.

or
: Return true if the left or the right operand are true.

not
: negate a statement (see below).

(expr)
: group an expression.

Note:

The `is` and `in` operators support negation using an infix notation, too: `foo is not bar` and `foo not in bar` instead of `not foo is bar` and `not foo in bar`. All other expressions require a prefix notation: `not (foo and bar).`

Other Operators

The following operators are very useful but don't fit into any of the other two categories:

in
: Perform a sequence / mapping containment test. Returns true if the left operand is contained in the right. `{{ 1 in [1, 2, 3] }}` would, for example, return true.

is
: Performs a test.

|
: Applies a filter.

~
: Converts all operands into strings and concatenates them.

  `{{ "Hello " ~ name ~ "!" }}` would return (assuming *name* is set to `'John'`) `Hello John!`.

()
: Call a callable: `{{ post.render() }}`. Inside of the parentheses you can use positional arguments and keyword arguments like in Python:

  `{{ post.render(user, full=true) }}`.

. / []
: Get an attribute of an object. (See Variables)

If Expression

It is also possible to use inline *if* expressions. These are useful in some situations. For example, you can use this to extend from one template if a variable is defined, otherwise from the default layout template:

```
{% extends layout_template if layout_template is defined else 'master.html' %}
```

The general syntax is `<do something> if <something is true> else <do something else>`.

The *else* part is optional. If not provided, the else block implicitly evaluates into an undefined object:

```
.. sourcecode:: jinja

    {{ '[%s]' % page.title if page.title }}
```

List of Builtin Filters

abs(*number*)
: Return the absolute value of the argument.

attr(*obj*, *name*)

Get an attribute of an object. `foo|attr("bar")` works like `foo.bar` just that always an attribute is returned and items are not looked up.

See Notes on subscriptions for more details.

batch(*value*, *linecount*, *fill_with=None*)

A filter that batches items. It works pretty much like *slice* just the other way round. It returns a list of lists with the given number of items. If you provide a second parameter this is used to fill up missing items. See this example:

```
<table>
{%- for row in items|batch(3, ' ') %}
  <tr>
  {%- for column in row %}
    <td>{{ column }}</td>
  {%- endfor %}
  </tr>
{%- endfor %}
</table>
```

capitalize(*s*)

Capitalize a value. The first character will be uppercase, all others lowercase.

center(*value*, *width=80*)

Centers the value in a field of a given width.

default(*value*, *default_value=u''*, *boolean=False*)

If the value is undefined it will return the passed default value, otherwise the value of the variable:

```
{{ my_variable|default('my_variable is not defined') }}
```

This will output the value of `my_variable` if the variable was defined, otherwise `'my_variable is not defined'`. If you want to use default with variables that evaluate to false you have to set the second parameter to *true*:

```
{{ ''|default('the string was empty', true) }}
```

Aliases: `d`

dictsort(*value*, *case_sensitive=False*, *by='key'*, *reverse=False*)

Sort a dict and yield (key, value) pairs. Because python dicts are unsorted you may want to use this function to order them by either key or value:

```
{% for item in mydict|dictsort %}
    sort the dict by key, case insensitive

{% for item in mydict|dictsort(reverse=true) %}
    sort the dict by key, case insensitive, reverse order

{% for item in mydict|dictsort(true) %}
    sort the dict by key, case sensitive

{% for item in mydict|dictsort(false, 'value') %}
    sort the dict by value, case insensitive
```

escape(*s*)

Convert the characters &, <, >, ‘, and ” in string s to HTML-safe sequences. Use this if you need to display text that might contain such characters in HTML. Marks return value as markup string.

Aliases: `e`

`filesizeformat`(*value*, *binary=False*)
Format the value like a 'human-readable' file size (i.e. 13 kB, 4.1 MB, 102 Bytes, etc). Per default decimal prefixes are used (Mega, Giga, etc.), if the second parameter is set to *True* the binary prefixes are used (Mebi, Gibi).

`first`(*seq*)
Return the first item of a sequence.

`float`(*value*, *default=0.0*)
Convert the value into a floating point number. If the conversion doesn't work it will return `0.0`. You can override this default using the first parameter.

`forceescape`(*value*)
Enforce HTML escaping. This will probably double escape variables.

`format`(*value*, **args*, ***kwargs*)
Apply python string formatting on an object:

```
{{ "%s - %s"|format("Hello?", "Foo!") }}
    -> Hello? - Foo!
```

`groupby`(*value*, *attribute*)
Group a sequence of objects by a common attribute.

If you for example have a list of dicts or objects that represent persons with *gender*, *first_name* and *last_name* attributes and you want to group all users by genders you can do something like the following snippet:

```
<ul>
{% for group in persons|groupby('gender') %}
    <li>{{ group.grouper }}<ul>
    {% for person in group.list %}
        <li>{{ person.first_name }} {{ person.last_name }}</li>
    {% endfor %}</ul></li>
{% endfor %}
</ul>
```

Additionally it's possible to use tuple unpacking for the grouper and list:

```
<ul>
{% for grouper, list in persons|groupby('gender') %}
    ...
{% endfor %}
</ul>
```

As you can see the item we're grouping by is stored in the *grouper* attribute and the *list* contains all the objects that have this grouper in common.

Changed in version 2.6: It's now possible to use dotted notation to group by the child attribute of another attribute.

`indent`(*s*, *width=4*, *first=False*, *blank=False*, *indentfirst=None*)
Return a copy of the string with each line indented by 4 spaces. The first line and blank lines are not indented by default.

Parameters:
- **width** – Number of spaces to indent by.
- **first** – Don't skip indenting the first line.

- **blank** – Don't skip indenting empty lines.

Changed in version 2.10: Blank lines are not indented by default.

Rename the `indentfirst` argument to `first`.

int(*value*, *default=0*, *base=10*)
Convert the value into an integer. If the conversion doesn't work it will return `0`. You can override this default using the first parameter. You can also override the default base (10) in the second parameter, which handles input with prefixes such as 0b, 0o and 0x for bases 2, 8 and 16 respectively. The base is ignored for decimal numbers and non-string values.

join(*value*, *d=u''*, *attribute=None*)
Return a string which is the concatenation of the strings in the sequence. The separator between elements is an empty string per default, you can define it with the optional parameter:

```
{{ [1, 2, 3]|join('|') }}
    -> 1|2|3

{{ [1, 2, 3]|join }}
    -> 123
```

It is also possible to join certain attributes of an object:

```
{{ users|join(', ', attribute='username') }}
```

New in version 2.6: The *attribute* parameter was added.

last(*seq*)
Return the last item of a sequence.

length(*object*)
Return the number of items of a sequence or mapping.

Aliases: `count`

list(*value*)
Convert the value into a list. If it was a string the returned list will be a list of characters.

lower(*s*)
Convert a value to lowercase.

map()
Applies a filter on a sequence of objects or looks up an attribute. This is useful when dealing with lists of objects but you are really only interested in a certain value of it.

The basic usage is mapping on an attribute. Imagine you have a list of users but you are only interested in a list of usernames:

```
Users on this page: {{ users|map(attribute='username')|join(', ') }}
```

Alternatively you can let it invoke a filter by passing the name of the filter and the arguments afterwards. A good example would be applying a text conversion filter on a sequence:

```
Users on this page: {{ titles|map('lower')|join(', ') }}
```

New in version 2.7.

max(*value*, *case_sensitive=False*, *attribute=None*)
Return the largest item from the sequence.

```
{{ [1, 2, 3]|max }}
    -> 3
```

Parameters:
- **case_sensitive** – Treat upper and lower case strings as distinct.
- **attribute** – Get the object with the max value of this attribute.

min(*value*, *case_sensitive=False*, *attribute=None*)
Return the smallest item from the sequence.

```
{{ [1, 2, 3]|min }}
    -> 1
```

Parameters:
- **case_sensitive** – Treat upper and lower case strings as distinct.
- **attribute** – Get the object with the max value of this attribute.

pprint(*value*, *verbose=False*)
Pretty print a variable. Useful for debugging.

With Jinja 1.2 onwards you can pass it a parameter. If this parameter is truthy the output will be more verbose (this requires *pretty*)

random(*seq*)
Return a random item from the sequence.

reject()
Filters a sequence of objects by applying a test to each object, and rejecting the objects with the test succeeding.

If no test is specified, each object will be evaluated as a boolean.

Example usage:

```
{{ numbers|reject("odd") }}
```

New in version 2.7.

rejectattr()
Filters a sequence of objects by applying a test to the specified attribute of each object, and rejecting the objects with the test succeeding.

If no test is specified, the attribute's value will be evaluated as a boolean.

```
{{ users|rejectattr("is_active") }}
{{ users|rejectattr("email", "none") }}
```

New in version 2.7.

replace(*s*, *old*, *new*, *count=None*)
Return a copy of the value with all occurrences of a substring replaced with a new one. The first argument is the substring that should be replaced, the second is the replacement string. If the optional third argument `count` is given, only the first `count` occurrences are replaced:

```
{{ "Hello World"|replace("Hello", "Goodbye") }}
    -> Goodbye World

{{ "aaaaargh"|replace("a", "d'oh, ", 2) }}
    -> d'oh, d'oh, aaargh
```

reverse(*value*)
Reverse the object or return an iterator that iterates over it the other way round.

round(*value*, *precision=0*, *method='common'*)
Round the number to a given precision. The first parameter specifies the precision (default is `0`), the second the rounding method:

- `'common'` rounds either up or down
- `'ceil'` always rounds up
- `'floor'` always rounds down

If you don't specify a method `'common'` is used.

```
{{ 42.55|round }}
    -> 43.0
{{ 42.55|round(1, 'floor') }}
    -> 42.5
```

Note that even if rounded to 0 precision, a float is returned. If you need a real integer, pipe it through *int*:

```
{{ 42.55|round|int }}
    -> 43
```

safe(*value*)
Mark the value as safe which means that in an environment with automatic escaping enabled this variable will not be escaped.

select()
Filters a sequence of objects by applying a test to each object, and only selecting the objects with the test succeeding.

If no test is specified, each object will be evaluated as a boolean.

Example usage:

```
{{ numbers|select("odd") }}
{{ numbers|select("odd") }}
{{ numbers|select("divisibleby", 3) }}
{{ numbers|select("lessthan", 42) }}
{{ strings|select("equalto", "mystring") }}
```

New in version 2.7.

selectattr()
Filters a sequence of objects by applying a test to the specified attribute of each object, and only selecting the objects with the test succeeding.

If no test is specified, the attribute's value will be evaluated as a boolean.

Example usage:

```
{{ users|selectattr("is_active") }}
{{ users|selectattr("email", "none") }}
```

New in version 2.7.

slice(*value*, *slices*, *fill_with=None*)

Slice an iterator and return a list of lists containing those items. Useful if you want to create a div containing three ul tags that represent columns:

```
<div class="columwrapper">
  {%- for column in items|slice(3) %}
    <ul class="column-{{ loop.index }}">
    {%- for item in column %}
      <li>{{ item }}</li>
    {%- endfor %}
    </ul>
  {%- endfor %}
</div>
```

If you pass it a second argument it's used to fill missing values on the last iteration.

sort(*value*, *reverse=False*, *case_sensitive=False*, *attribute=None*)

Sort an iterable. Per default it sorts ascending, if you pass it true as first argument it will reverse the sorting.

If the iterable is made of strings the third parameter can be used to control the case sensitiveness of the comparison which is disabled by default.

```
{% for item in iterable|sort %}
    ...
{% endfor %}
```

It is also possible to sort by an attribute (for example to sort by the date of an object) by specifying the *attribute* parameter:

```
{% for item in iterable|sort(attribute='date') %}
    ...
{% endfor %}
```

Changed in version 2.6: The *attribute* parameter was added.

string(*object*)

Make a string unicode if it isn't already. That way a markup string is not converted back to unicode.

striptags(*value*)

Strip SGML/XML tags and replace adjacent whitespace by one space.

sum(*iterable*, *attribute=None*, *start=0*)

Returns the sum of a sequence of numbers plus the value of parameter 'start' (which defaults to 0). When the sequence is empty it returns start.

It is also possible to sum up only certain attributes:

```
Total: {{ items|sum(attribute='price') }}
```

Changed in version 2.6: The *attribute* parameter was added to allow suming up over attributes. Also the *start* parameter was moved on to the right.

title(*s*)

Return a titlecased version of the value. I.e. words will start with uppercase letters, all remaining characters are lowercase.

tojson(*value*, *indent=None*)
Dumps a structure to JSON so that it's safe to use in `<script>` tags. It accepts the same arguments and returns a JSON string. Note that this is available in templates through the `|tojson` filter which will also mark the result as safe. Due to how this function escapes certain characters this is safe even if used outside of `<script>` tags.

The following characters are escaped in strings:

- `<`
- `>`
- `&`
- `&#39;`

This makes it safe to embed such strings in any place in HTML with the notable exception of double quoted attributes. In that case single quote your attributes or HTML escape it in addition.

The indent parameter can be used to enable pretty printing. Set it to the number of spaces that the structures should be indented with.

Note that this filter is for use in HTML contexts only.

New in version 2.9.

trim(*value*)
Strip leading and trailing whitespace.

truncate(*s*, *length=255*, *killwords=False*, *end='...'*, *leeway=None*)
Return a truncated copy of the string. The length is specified with the first parameter which defaults to `255`. If the second parameter is `true` the filter will cut the text at length. Otherwise it will discard the last word. If the text was in fact truncated it will append an ellipsis sign (`"..."`). If you want a different ellipsis sign than `"..."` you can specify it using the third parameter. Strings that only exceed the length by the tolerance margin given in the fourth parameter will not be truncated.

```
{{ "foo bar baz qux"|truncate(9) }}
    -> "foo..."
{{ "foo bar baz qux"|truncate(9, True) }}
    -> "foo ba..."
{{ "foo bar baz qux"|truncate(11) }}
    -> "foo bar baz qux"
{{ "foo bar baz qux"|truncate(11, False, '...', 0) }}
    -> "foo bar..."
```

The default leeway on newer Jinja2 versions is 5 and was 0 before but can be reconfigured globally.

unique(*value*, *case_sensitive=False*, *attribute=None*)
Returns a list of unique items from the the given iterable.

```
{{ ['foo', 'bar', 'foobar', 'FooBar']|unique }}
    -> ['foo', 'bar', 'foobar']
```

The unique items are yielded in the same order as their first occurrence in the iterable passed to the filter.

Parameters:
- **case_sensitive** – Treat upper and lower case strings as distinct.
- **attribute** – Filter objects with unique values for this attribute.

upper(*s*)
Convert a value to uppercase.

urlencode(*value*)
Escape strings for use in URLs (uses UTF-8 encoding). It accepts both dictionaries and regular strings as well as pairwise iterables.

New in version 2.7.

urlize(*value*, *trim_url_limit=None*, *nofollow=False*, *target=None*, *rel=None*)
Converts URLs in plain text into clickable links.

If you pass the filter an additional integer it will shorten the urls to that number. Also a third argument exists that makes the urls "nofollow":

```
{{ mytext|urlize(40, true) }}
    links are shortened to 40 chars and defined with rel="nofollow"
```

If *target* is specified, the `target` attribute will be added to the `<a>` tag:

```
{{ mytext|urlize(40, target='_blank') }}
```

Changed in version 2.8+: The *target* parameter was added.

wordcount(*s*)
Count the words in that string.

wordwrap(*s*, *width=79*, *break_long_words=True*, *wrapstring=None*)
Return a copy of the string passed to the filter wrapped after `79` characters. You can override this default using the first parameter. If you set the second parameter to *false* Jinja will not split words apart if they are longer than *width*. By default, the newlines will be the default newlines for the environment, but this can be changed using the wrapstring keyword argument.

New in version 2.7: Added support for the *wrapstring* parameter.

xmlattr(*d*, *autospace=True*)
Create an SGML/XML attribute string based on the items in a dict. All values that are neither *none* nor *undefined* are automatically escaped:

```
<ul{{ {'class': 'my_list', 'missing': none,
        'id': 'list-%d'|format(variable)}|xmlattr }}>
...
</ul>
```

Results in something like this:

```
<ul class="my_list" id="list-42">
...
</ul>
```

As you can see it automatically prepends a space in front of the item if the filter returned something unless the second parameter is false.

List of Builtin Tests

callable(*object*)

Return whether the object is callable (i.e., some kind of function). Note that classes are callable, as are instances with a __call__() method.

defined(*value*)
Return true if the variable is defined:

```
{% if variable is defined %}
    value of variable: {{ variable }}
{% else %}
    variable is not defined
{% endif %}
```

See the **default()** filter for a simple way to set undefined variables.

divisibleby(*value*, *num*)
Check if a variable is divisible by a number.

eq(*a*, *b*)
Aliases: `==`, `equalto`

escaped(*value*)
Check if the value is escaped.

even(*value*)
Return true if the variable is even.

ge(*a*, *b*)
Aliases: `>=`

gt(*a*, *b*)
Aliases: `>`, `greaterthan`

in(*value*, *seq*)
Check if value is in seq.

New in version 2.10.

iterable(*value*)
Check if it's possible to iterate over an object.

le(*a*, *b*)
Aliases: `<=`

lower(*value*)
Return true if the variable is lowercased.

lt(*a*, *b*)
Aliases: `<`, `lessthan`

mapping(*value*)
Return true if the object is a mapping (dict etc.).

New in version 2.6.

ne(*a*, *b*)
Aliases: `!=`

none(*value*)
Return true if the variable is none.

number(*value*)
Return true if the variable is a number.

odd(*value*)
Return true if the variable is odd.

sameas(*value*, *other*)
Check if an object points to the same memory address than another object:

```
{% if foo.attribute is sameas false %}
    the foo attribute really is the `False` singleton
{% endif %}
```

sequence(*value*)
Return true if the variable is a sequence. Sequences are variables that are iterable.

string(*value*)
Return true if the object is a string.

undefined(*value*)
Like **defined()** but the other way round.

upper(*value*)
Return true if the variable is uppercased.

List of Global Functions

The following functions are available in the global scope by default:

range([*start*,]*stop*[, *step*])
Return a list containing an arithmetic progression of integers. `range(i, j)` returns `[i, i+1, i+2, ..., j-1]`; start (!) defaults to `0`. When step is given, it specifies the increment (or decrement). For example, `range(4)` and `range(0, 4, 1)` return `[0, 1, 2, 3]`. The end point is omitted! These are exactly the valid indices for a list of 4 elements.

This is useful to repeat a template block multiple times, e.g. to fill a list. Imagine you have 7 users in the list but you want to render three empty items to enforce a height with CSS:

```
<ul>
{% for user in users %}
    <li>{{ user.username }}</li>
{% endfor %}
{% for number in range(10 - users|count) %}
    <li class="empty"><span>...</span></li>
{% endfor %}
</ul>
```

lipsum(*n=5*, *html=True*, *min=20*, *max=100*)
Generates some lorem ipsum for the template. By default, five paragraphs of HTML are generated with each paragraph between 20 and 100 words. If html is False, regular text is returned. This is useful to generate simple contents for layout testing.

dict(***items*)
A convenient alternative to dict literals. `{'foo': 'bar'}` is the same as `dict(foo='bar')`.

class **cycler**(**items*)

The cycler allows you to cycle among values similar to how *loop.cycle* works. Unlike *loop.cycle*, you can use this cycler outside of loops or over multiple loops.

This can be very useful if you want to show a list of folders and files with the folders on top but both in the same list with alternating row colors.

The following example shows how *cycler* can be used:

```
{% set row_class = cycler('odd', 'even') %}
<ul class="browser">
{% for folder in folders %}
  <li class="folder {{ row_class.next() }}">{{ folder|e }}</li>
{% endfor %}
{% for filename in files %}
  <li class="file {{ row_class.next() }}">{{ filename|e }}</li>
{% endfor %}
</ul>
```

A cycler has the following attributes and methods:

reset()

Resets the cycle to the first item.

next()

Goes one item ahead and returns the then-current item.

current

Returns the current item.

New in version 2.1.

class **joiner**(*sep=', '*)

A tiny helper that can be used to “join” multiple sections. A joiner is passed a string and will return that string every time it’s called, except the first time (in which case it returns an empty string). You can use this to join things:

```
{% set pipe = joiner("|") %}
{% if categories %} {{ pipe() }}
    Categories: {{ categories|join(", ") }}
{% endif %}
{% if author %} {{ pipe() }}
    Author: {{ author() }}
{% endif %}
{% if can_edit %} {{ pipe() }}
    <a href="?action=edit">Edit</a>
{% endif %}
```

New in version 2.1.

class **namespace**(...)

Creates a new container that allows attribute assignment using the `{% set %}` tag:

```
{% set ns = namespace() %}
{% set ns.foo = 'bar' %}
```

The main purpose of this is to allow carrying a value from within a loop body to an outer scope. Initial values can be provided as a dict, as keyword arguments, or both (same behavior as Python’s *dict* constructor):

```
{% set ns = namespace(found=false) %}
{% for item in items %}
    {% if item.check_something() %}
        {% set ns.found = true %}
    {% endif %}
    * {{ item.title }}
{% endfor %}
Found item having something: {{ ns.found }}
```

New in version 2.10.

Extensions

The following sections cover the built-in Jinja2 extensions that may be enabled by an application. An application could also provide further extensions not covered by this documentation; in which case there should be a separate document explaining said extensions.

i18n

If the i18n extension is enabled, it's possible to mark parts in the template as translatable. To mark a section as translatable, you can use *trans*:

```
<p>{% trans %}Hello {{ user }}!{% endtrans %}</p>
```

To translate a template expression — say, using template filters, or by just accessing an attribute of an object — you need to bind the expression to a name for use within the translation block:

```
<p>{% trans user=user.username %}Hello {{ user }}!{% endtrans %}</p>
```

If you need to bind more than one expression inside a *trans* tag, separate the pieces with a comma (`,`):

```
{% trans book_title=book.title, author=author.name %}
This is {{ book_title }} by {{ author }}
{% endtrans %}
```

Inside trans tags no statements are allowed, only variable tags are.

To pluralize, specify both the singular and plural forms with the *pluralize* tag, which appears between *trans* and *endtrans*:

```
{% trans count=list|length %}
There is {{ count }} {{ name }} object.
{% pluralize %}
There are {{ count }} {{ name }} objects.
{% endtrans %}
```

By default, the first variable in a block is used to determine the correct singular or plural form. If that doesn't work out, you can specify the name which should be used for pluralizing by adding it as parameter to *pluralize*:

```
{% trans ..., user_count=users|length %}...
{% pluralize user_count %}...{% endtrans %}
```

When translating longer blocks of text, whitespace and linebreaks result in rather ugly and error-prone translation strings. To avoid this, a trans block can be marked as trimmed which will replace all linebreaks and the whitespace surrounding them with a single space and remove leading/trailing whitespace:

```
{% trans trimmed book_title=book.title %}
    This is {{ book_title }}.
    You should read it!
{% endtrans %}
```

If trimming is enabled globally, the *notrimmed* modifier can be used to disable it for a *trans* block.

New in version 2.10: The *trimmed* and *notrimmed* modifiers have been added.

It's also possible to translate strings in expressions. For that purpose, three functions exist:

- *gettext*: translate a single string
- *ngettext*: translate a pluralizable string
- _: alias for *gettext*

For example, you can easily print a translated string like this:

```
{{ _('Hello World!') }}
```

To use placeholders, use the *format* filter:

```
{{ _('Hello %(user)s!')|format(user=user.username) }}
```

For multiple placeholders, always use keyword arguments to *format*, as other languages may not use the words in the same order.

Changed in version 2.5.

If newstyle gettext calls are activated (Whitespace Trimming), using placeholders is a lot easier:

```
{{ gettext('Hello World!') }}
{{ gettext('Hello %(name)s!', name='World') }}
{{ ngettext('%(num)d apple', '%(num)d apples', apples|count) }}
```

Note that the *ngettext* function's format string automatically receives the count as a *num* parameter in addition to the regular parameters.

Expression Statement

If the expression-statement extension is loaded, a tag called *do* is available that works exactly like the regular variable expression (`{{ ... }}`); except it doesn't print anything. This can be used to modify lists:

```
{% do navigation.append('a string') %}
```

Loop Controls

If the application enables the Loop Controls, it's possible to use *break* and *continue* in loops. When *break* is reached, the loop is terminated; if *continue* is reached, the processing is stopped and continues with the next iteration.

Here's a loop that skips every second item:

```
{% for user in users %}
    {%- if loop.index is even %}{% continue %}{% endif %}
    ...
{% endfor %}
```

Likewise, a loop that stops processing after the 10th iteration:

```
{% for user in users %}
    {%- if loop.index >= 10 %}{% break %}{% endif %}
{%- endfor %}
```

Note that `loop.index` starts with 1, and `loop.index0` starts with 0 (See: For).

With Statement

New in version 2.3.

The with statement makes it possible to create a new inner scope. Variables set within this scope are not visible outside of the scope.

With in a nutshell:

```
{% with %}
    {% set foo = 42 %}
    {{ foo }}           foo is 42 here
{% endwith %}
foo is not visible here any longer
```

Because it is common to set variables at the beginning of the scope, you can do that within the *with* statement. The following two examples are equivalent:

```
{% with foo = 42 %}
    {{ foo }}
{% endwith %}

{% with %}
    {% set foo = 42 %}
    {{ foo }}
{% endwith %}
```

An important note on scoping here. In Jinja versions before 2.9 the behavior of referencing one variable to another had some unintended consequences. In particular one variable could refer to another defined in the same with block's opening statement. This caused issues with the cleaned up scoping behavior and has since been improved. In particular in newer Jinja2 versions the following code always refers to the variable *a* from outside the *with* block:

```
{% with a={}, b=a.attribute %}...{% endwith %}
```

In earlier Jinja versions the *b* attribute would refer to the results of the first attribute. If you depend on this behavior you can rewrite it to use the `set` tag:

```
{% with a={} %}
    {% set b = a.attribute %}
{% endwith %}
```

Extension:

In older versions of Jinja (before 2.9) it was required to enable this feature with an extension. It's now enabled by default.

Autoescape Overrides

New in version 2.4.

If you want you can activate and deactivate the autoescaping from within the templates.

Example:

```
{% autoescape true %}
    Autoescaping is active within this block
{% endautoescape %}

{% autoescape false %}
    Autoescaping is inactive within this block
{% endautoescape %}
```

After an *endautoescape* the behavior is reverted to what it was before.

Extension:

In older versions of Jinja (before 2.9) it was required to enable this feature with an extension. It's now enabled by default.
