Jinja2 Documentation

Release 2.0

Armin Ronacher

April 29, 2008

CONTENTS

Contents:

CHAPTER

ONE

Introduction

This is the documentation for the Jinja2 general purpose templating language. Jinja2 is a library for Python 2.4 and onwards that is designed to be flexible, fast and secure.

If you have any exposure to other text-based template languages, such as Smarty or Django, you should feel right at home with Jinja2. It's both designer and developer friendly by sticking to Python's principles and adding functionality useful for templating environments.

The key-features are...

- ... **configurable syntax**. If you are generating LaTeX or other formats with Jinja you can change the delimiters to something that integrates better into the LaTeX markup.
- ... **fast**. While performance is not the primarily target of Jinja2 it's surprisingly fast. The overhead compared to regular Python code was reduced to the very minimum.
- ... **easy to debug**. Jinja2 integrates directly into the python traceback system which allows you to debug Jinja templates with regular python debugging helpers.
- ... **secure**. It's possible to evaluate untrusted template code if the optional sandbox is enabled. This allows Jinja2 to be used as templating language for applications where users may modify the template design.

1.1 Prerequisites

Jinja2 needs at least **Python 2.4** to run. Additionally a working C-compiler that can create python extensions should be installed for the debugger. If no C-compiler is available the ctypes module should be installed.

1.2 Basic API Usage

This section gives you a brief introduction to the Python API for Jinja templates.

The most basic way to create a template and render it is through `Template`. This however is not the recommended way to work with it, but the easiest

```
>>> from jinja2 import Template
>>> template = Template('Hello {{ name }}!')
>>> template.render(name='John Doe')
u'Hello John Doe!'
```

By creating an instance of `Template` you get back a new template object that provides a method called `render()` which when called with a dict or keyword arguments expands the template. The dict or keywords arguments passed to the template are the so-called "context" of the template.

CHAPTER

TWO

API

This document describes the API to Jinja2 and not the template language. It will be most useful as reference to those implementing the template interface to the application and not those who are creating Jinja2 templates.

2.1 Basics

Jinja2 uses a central object called the template `Environment`. Instances of this class are used to store the configuration, global objects and are used to load templates from the file system or other locations. Even if you are creating templates from string by using the constructor of `Template` class, an environment is created automatically for you.

Most applications will create one `Environment` object on application initialization and use that to load templates. In some cases it's however useful to have multiple environments side by side, if different configurations are in use.

The simplest way to configure Jinja2 to load templates for your application looks roughly like this:

```
from jinja2 import Environment, PackageLoader
env = Environment(loader=PackageLoader('yourapplication', 'templates'))
```

This will create a template environment with the default settings and a loader that looks up the templates in the *templates* folder inside the *yourapplication* python package. Different loaders are available and you can also write your own if you want to load templates from a database or other resources.

To load a template from this environment you just have to call the `get_template()` method which then returns the loaded `Template`:

```
template = env.get_template('mytemplate.html')
```

To render it with some variables, just call the `render()` method:

```
print template.render(the='variables', go='here')
```

2.2 High Level API

class Environment(*block_start_string='{%'*, *block_end_string='%}'*, *variable_start_string='{{'*, *variable_end_string='}}'*, *comment_start_string='{#'*, *comment_end_string='#}'*, *line_statement_prefix=None*, *trim_blocks=False*, *extensions=()*, *optimized=True*, *undefined=<class 'jinja2.runtime.Undefined'>*, *finalize=None*, *autoescape=False*, *loader=None*)

The core component of Jinja is the *Environment*. It contains important shared variables like configuration, filters, tests, globals and others. Instances of this class may be modified if they are

not shared and if no template was loaded so far. Modifications on environments after the first template was loaded will lead to surprising effects and undefined behavior.

Here the possible initialization parameters:

block_start_string The string marking the begin of a block. Defaults to `'{%'`.

block_end_string The string marking the end of a block. Defaults to `'%}'`.

variable_start_string The string marking the begin of a print statement. Defaults to `'{{'`.

comment_start_string The string marking the begin of a comment. Defaults to `'{#'`.

comment_end_string The string marking the end of a comment. Defaults to `'#}'`.

line_statement_prefix If given and a string, this will be used as prefix for line based statements.

trim_blocks If this is set to `True` the first newline after a block is removed (block, not variable tag!). Defaults to *False*.

extensions List of Jinja extensions to use. This can either be import paths as strings or extension classes.

optimized should the optimizer be enabled? Default is *True*.

undefined `Undefined` or a subclass of it that is used to represent undefined values in the template.

finalize A callable that finalizes the variable. Per default no finalizing is applied.

autoescape If set to true the XML/HTML autoescaping feature is enabled.

loader The template loader for this environment.

`shared`
If a template was created by using the `Template` constructor an environment is created automatically. These environments are created as shared environments which means that multiple templates may have the same anonymous environment. For all shared environments this attribute is *True*, else *False*.

`sandboxed`
If the environment is sandboxed this attribute is *True*. For the sandbox mode have a look at the documentation for the `SandboxedEnvironment`.

`filters`
A dict of filters for this environment. As long as no template was loaded it's safe to add new filters or remove old.

`tests`
A dict of test funcitons for this environment. As long as no template way loaded it's safe to modify this dict.

`globals`
A dict of global variables. These variables are always available in a template and (if the optimizer is enabled) may not be override by templates. As long as no template was loaded it's safe to modify this dict.

`from_string`(*source, globals=None, template_class=None*)
Load a template from a string. This parses the source given and returns a `Template` object.

`get_template`(*name, parent=None, globals=None*)
Load a template from the loader. If a loader is configured this method ask the loader for the template and returns a `Template`. If the *parent* parameter is not *None*, `join_path()` is called to get the real template name before loading.

The *globals* parameter can be used to provide compile-time globals. In the future this will allow the optimizer to render parts of the templates at compile-time.

If the template does not exist a `TemplateNotFound` exception is raised.

`join_path`(*template, parent*)
Join a template with the parent. By default all the lookups are relative to the loader root so this method returns the *template* parameter unchanged, but if the paths should be relative to the parent template, this function can be used to calculate the real template name.

Subclasses may override this method and implement template path joining here.

class **Template**()
The central template object. This class represents a compiled template and is used to evaluate it.

Normally the template object is generated from an `Environment` but it also has a constructor that makes it possible to create a template instance directly using the constructor. It takes the same arguments as the environment constructor but it's not possible to specify a loader.

Every template object has a few methods and members that are guaranteed to exist. However it's important that a template object should be considered immutable. Modifications on the object are not supported.

Template objects created from the constructor rather than an environment do have an *environment* attribute that points to a temporary environment that is probably shared with other templates created with the constructor and compatible settings.

```
>>> template = Template('Hello {{ name }}!')
>>> template.render(name='John Doe')
u'Hello John Doe!'
```

```
>>> stream = template.stream(name='John Doe')
>>> stream.next()
u'Hello John Doe!'
>>> stream.next()
Traceback (most recent call last):
    ...
StopIteration
```

render(**args*, ***kwargs*)
This method accepts the same arguments as the *dict* constructor: A dict, a dict subclass or some keyword arguments. If no arguments are given the context will be empty. These two calls do the same:

```
template.render(knights='that say nih')
template.render({'knights': 'that say nih'})
```

This will return the rendered template as unicode string.

stream(**args*, ***kwargs*)
Works exactly like `generate()` but returns a `TemplateStream`.

generate(**args*, ***kwargs*)
For very large templates it can be useful to not render the whole template at once but evaluate each statement after another and yield piece for piece. This method basically does exactly that and returns a generator that yields one item after another as unicode strings.

It accepts the same arguments as `render()`.

module
The template as module. This is used for imports in the template runtime but is also useful if one wants to access exported template variables from the Python layer:

```
>>> t = Template('{% macro foo() %}42{% endmacro %}23')
>>> unicode(t.module)
u'23'
>>> t.module.foo()
u'42'
```

class **TemplateStream**(*gen*)
A template stream works pretty much like an ordinary python generator but it can buffer multiple items to reduce the number of total iterations. Per default the output is unbuffered which means that for every unbuffered instruction in the template one unicode string is yielded.

If buffering is enabled with a buffer size of 5, five items are combined into a new unicode string. This is mainly useful if you are streaming big templates to a client via WSGI which flushes after each iteration.

disable_buffering()
Disable the output buffering.

enable_buffering(*size=5*)
Enable buffering. Buffer *size* items before yielding them.

2.3 Undefined Types

These classes can be used as undefined types. The `Environment` constructor takes an *undefined* parameter that can be one of those classes or a custom subclass of `Undefined`. Whenever the template engine is unable to look up a name or access an attribute one of those objects is created and returned. Some operations on undefined values are then allowed, others fail.

The closest to regular Python behavior is the *StrictUndefined* which disallows all operations beside testing if it's an undefined object.

class Undefined(*hint=None, obj=None, name=None*)
The default undefined type. This undefined type can be printed and iterated over, but every other access will raise an `UndefinedError`:

```
>>> foo = Undefined(name='foo')
>>> str(foo)
''
>>> not foo
True
>>> foo + 42
Traceback (most recent call last):
...
jinja2.exceptions.UndefinedError: 'foo' is undefined
```

class DebugUndefined(*hint=None, obj=None, name=None*)
An undefined that returns the debug info when printed.

```
>>> foo = DebugUndefined(name='foo')
>>> str(foo)
'{{ foo }}'
>>> not foo
True
>>> foo + 42
Traceback (most recent call last):
...
jinja2.exceptions.UndefinedError: 'foo' is undefined
```

class StrictUndefined(*hint=None, obj=None, name=None*)
An undefined that barks on print and iteration as well as boolean tests and all kinds of comparisons. In other words: you can do nothing with it except checking if it's defined using the *defined* test.

```
>>> foo = StrictUndefined(name='foo')
>>> str(foo)
Traceback (most recent call last):
...
jinja2.exceptions.UndefinedError: 'foo' is undefined
>>> not foo
Traceback (most recent call last):
...
jinja2.exceptions.UndefinedError: 'foo' is undefined
>>> foo + 42
Traceback (most recent call last):
...
jinja2.exceptions.UndefinedError: 'foo' is undefined
```

2.4 Loaders

Loaders are responsible for loading templates from a resource such as the file system and for keeping the compiled modules in memory. These work like Python's *sys.modules* which keeps the imported templates in memory. Unlike *sys.modules* however this cache is limited in size by default and templates are automatically reloaded. Each loader that extends `BaseLoader` supports this caching and accepts two parameters to configure it:

cache_size The size of the cache. Per default this is `50` which means that if more than 50 templates are loaded the loader will clean out the least recently used template. If the cache size is set to `0` templates are recompiled all the time, if the cache size is `-1` the cache will not be cleaned.

auto_reload Some loaders load templates from locations where the template sources may change (ie: file system or database). If *auto_reload* is set to *True* (default) every time a template is requested the loader checks if the source changed and if yes, it will reload the template. For higher performance it's possible to disable that.

class `FileSystemLoader` (*searchpath, encoding='utf-8', cache_size=50, auto_reload=True*)
Loads templates from the file system. This loader can find templates in folders on the file system and is the preferred way to load them.

The loader takes the path to the templates as string, or if multiple locations are wanted a list of them which is then looked up in the given order:

```
>>> loader = FileSystemLoader('/path/to/templates')
>>> loader = FileSystemLoader(['/path/to/templates', '/other/path'])
```

Per default the template encoding is `'utf-8'` which can be changed by setting the *encoding* parameter to something else.

class `PackageLoader` (*package_name, package_path='templates', encoding='utf-8', cache_size=50, auto_reload=True*)
Load templates from python eggs or packages. It is constructed with the name of the python package and the path to the templates in that package:

```
>>> loader = PackageLoader('mypackage', 'views')
```

If the package path is not given, `'templates'` is assumed.

Per default the template encoding is `'utf-8'` which can be changed by setting the *encoding* parameter to something else. Due to the nature of eggs it's only possible to reload templates if the package was loaded from the file system and not a zip file.

class `DictLoader` (*mapping, cache_size=50, auto_reload=False*)
Loads a template from a python dict. It's passed a dict of unicode strings bound to template names. This loader is useful for unittesting:

```
>>> loader = DictLoader({'index.html': 'source here'})
```

Because auto reloading is rarely useful this is disabled per default.

class `FunctionLoader` (*load_func, cache_size=50, auto_reload=True*)
A loader that is passed a function which does the loading. The function becomes the name of the template passed and has to return either an unicode string with the template source, a tuple in the form `'(source, filename, uptodatefunc)'` or *None* if the template does not exist.

```
>>> def load_template(name):
...     if name == 'index.html'
...         return '...'
...
>>> loader = FunctionLoader(load_template)
```

The *uptodatefunc* is a function that is called if autoreload is enabled and has to return *True* if the template is still up to date. For more details have a look at `BaseLoader.get_source()` which has the same return value.

class **PrefixLoader**(*mapping*, *delimiter='/'*, *cache_size=50*, *auto_reload=True*)

A loader that is passed a dict of loaders where each loader is bound to a prefix. The caching is independent of the actual loaders so the per loader cache settings are ignored. The prefix is delimited from the template by a slash:

```
>>> loader = PrefixLoader({
...     'app1':     PackageLoader('mypackage.app1'),
...     'app2':     PackageLoader('mypackage.app2')
... })
```

By loading `'app1/index.html'` the file from the app1 package is loaded, by loading `'app2/index.html'` the file from the second.

class **ChoiceLoader**(*loaders*, *cache_size=50*, *auto_reload=True*)

This loader works like the *PrefixLoader* just that no prefix is specified. If a template could not be found by one loader the next one is tried. Like for the *PrefixLoader* the cache settings of the actual loaders don't matter as the choice loader does the caching.

```
>>> loader = ChoiceLoader([
...     FileSystemLoader('/path/to/user/templates'),
...     PackageLoader('myapplication')
])
```

This is useful if you want to allow users to override builtin templates from a different location.

All loaders are subclasses of `BaseLoader`. If you want to create your own loader, subclass `BaseLoader` and override *get_source*.

class **BaseLoader**(*cache_size=50*, *auto_reload=True*)

Baseclass for all loaders. Subclass this and override *get_source* to implement a custom loading mechanism. The environment provides a *get_template* method that calls the loader's *load* method to get the `Template` object.

A very basic example for a loader that looks up templates on the file system could look like this:

```
from jinja2 import BaseLoader, TemplateNotFound
from os.path import join, exists, getmtime

class MyLoader(BaseLoader):

    def __init__(self, path, cache_size=50, auto_reload=True):
        BaseLoader.__init__(self, cache_size, auto_reload)
        self.path = path

    def get_source(self, environment, template):
        path = join(self.path, template)
        if not exists(path):
            raise TemplateNotFound(template)
        mtime = getmtime(path)
        with file(path) as f:
            source = f.read().decode('utf-8')
        return source, path, lambda: mtime != getmtime(path)
```

get_source(*environment*, *template*)

Get the template source, filename and reload helper for a template. It's passed the environment and template name and has to return a tuple in the form `'(source, filename, uptodate)'` or raise a *TemplateNotFound* error if it can't locate the template.

The source part of the returned tuple must be the source of the template as unicode string or a ASCII bytestring. The filename should be the name of the file on the filesystem if it was

loaded from there, otherwise *None*. The filename is used by python for the tracebacks if no loader extension is used.

The last item in the tuple is the *uptodate* function. If auto reloading is enabled it's always called to check if the template changed. No arguments are passed so the function must store the old state somewhere (for example in a closure). If it returns *False* the template will be reloaded.

load(*environment*, *name*, *globals=None*)
Loads a template. This method looks up the template in the cache or loads one by calling `get_source()`. Subclasses should not override this method as loaders working on collections of other loaders (such as `PrefixLoader` or `ChoiceLoader`) will not call this method but *get_source* directly.

2.5 Utilities

These helper functions and classes are useful if you add custom filters or functions to a Jinja2 environment.

environmentfilter(*f*)
Decorator for marking evironment dependent filters. The environment used for the template is passed to the filter as first argument.

contextfilter(*f*)
Decorator for marking context dependent filters. The current context argument will be passed as first argument.

environmentfunction(*f*)
This decorator can be used to mark a callable as environment callable. A environment callable is passed the current environment as first argument if it was directly stored in the context.

contextfunction(*f*)
This decorator can be used to mark a callable as context callable. A context callable is passed the active context as first argument if it was directly stored in the context.

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. This function will not escaped objects that do have an HTML representation such as already escaped data.

class **Markup**()
Marks a string as being safe for inclusion in HTML/XML output without needing to be escaped. This implements the *__html__* interface a couple of frameworks and web applications use.

The *escape* function returns markup objects so that double escaping can't happen. If you want to use autoescaping in Jinja just set the finalizer of the environment to *escape*.

2.6 Exceptions

class **TemplateError**()
Baseclass for all template errors.

class **UndefinedError**()
Raised if a template tries to operate on `Undefined`.

class **TemplateNotFound**(*name*)
Raised if a template does not exist.

class **TemplateSyntaxError**(*message*, *lineno*, *name*)
Raised to tell the user that there is a problem with the template.

class **TemplateAssertionError**(*message*, *lineno*, *name*)

Like a template syntax error, but covers cases where something in the template caused an error at compile time that wasn't necessarily caused by a syntax error.

CHAPTER

THREE

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 might be slightly different from here in terms of delimiters and behavior of undefined values.

3.1 Synopsis

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

A template contains **variables** or **expressions**, which get replaced with values when the template is evaluated, 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. We will cover the details later in that document:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<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 }}
</body>
</html>
```

This covers the default settings. The application developer might have changed the syntax from '`{% foo %}`' to '`<% foo %>`' or something similar.

There are two kinds of delimiers. '`{% ... %}`' and '`{{ ... }}`'. The first one is used to execute statements such as for-loops or assign values, the latter prints the result of the expression to the template.

3.2 Variables

The application passes variables to the templates you can mess around in the template. Variables may have attributes or elements on them you can access too. How a variable looks like, heavily depends on the application providing those.

You can use a dot (`.`) to access attributes of a variable, alternative the so-called "subscribe" syntax (`[]`) can be used. The following lines do the same:

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

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

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 that it evaluates to an empty string if printed and that you can iterate over it, but every other operation fails.

3.3 Filters

Variables can by 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.

'`{{ name|striptags|title }}`' for example will remove all HTML Tags from the *name* and title-cases it. Filters that accept arguments have parentheses around the arguments, like a function call. This example will join a list by spaces: '`{{ list|join(', ') }}`'.

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

3.4 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 depening on if *name* is defined.

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

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

The *List of Builtin Tests* below descibes all the builtin tests.

3.5 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: disabled template because we no longer user this
    {% for user in users %}
        ...
```

```
    {% endfor %}
#}
```

3.6 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.

3.6.1 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 PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html lang="en">
<html xmlns="http://www.w3.org/1999/xhtml">
<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>
```

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

3.6.2 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 on my awsome 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, first it 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.

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 an 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 hole to fill - it also defines the content that fills the hole in the *parent*. If there were two similarly-named '`{% block %}`' tags in a template, that template's parent wouldn't know which one of the blocks' content to use.

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 %}
```

Unlike Python Jinja does not support multiple inheritance. So you can only have one extends tag called per rendering.

3.6.3 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 %}
```

3.7 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: manually escaping each variable or automatically escaping everything by default.

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

- escaping everything except of safe values will also mean that Jinja is escaping variables known to not include HTML such as numbers which is 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.

3.7.1 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 **have to** escape it unless the variable contains well-formed and trusted HTML. Escaping works by piping the variable through the `|e` filter: '`{{ user.username|e }}`'.

3.7.2 Working with Automatic Escaping

When automatic escaping is enabled everything is escaped by default except for values explicitly marked as safe. Those can either be marked by the application or in the template by using the *|safe* filter. The main problem with this approach is that Python itself doesn't have the concept of tainted values so the information if a value is safe or unsafe can get lost. If the information is lost escaping will take place which means that you could end up with double escaped contents.

Double escaping is easy to avoid however, just relay on the tools Jinja2 provides and don't use builtin Python constructs such as the string modulo operator.

Functions returning template data (macros, *super*, *self.BLOCKNAME*) return safe markup always.

String literals in templates with automatic escaping are considered unsafe too. The reason for this is that the safe string is an extension to Python and not every library will work properly with it.

3.8 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. Control structures appear inside '`{% ... %}`' blocks in the default syntax.

3.8.1 For

Loop over each item in a sequece. 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>
```

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.index0	The current iteration of the loop. (0 indexed)
loop.revindex	The number of iterations from the end of the loop (1 indexed)
loop.revindex0	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.

Within a for-loop, it's psosible 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 %}
```

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 replacement block by using *else*:

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

3.8.2 If

The *if* statement in Jinja is comparable with the if statements of Python. In the simplest form you can use it to test if a variable is defined, not empty or 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 *inline expression* and for *loop filtering*.

3.8.3 Macros

Macros comparable with functions in regular programming languages. They are useful to put often used idoms into reusable functions to not repeat yourself.

Macros can be defined in helper templates with then are *imported* or directly in the template where they are used. There is one big difference between those two possibilities. A macro that is defined in the template where it's also used has access to the context passed to the template. A macro defined in another template and then imported can only access variables defined there or in the global context.

Here 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 defiend 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 then accepted by the macro they end up in the special *varargs* variable as 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 macro which can be called.

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 (ie: accesses the special *kwargs* variable).

catch_varargs This is *true* if the macro accepts extra positional arguments (ie: 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.

3.8.4 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 replacement for loops. It is however not possible to call a call block with another call block.

Here 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 %}
```

3.8.5 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 are just written in code blocks like any other statement just without explicit keyword:

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

3.8.6 Extends

The *extends* tag can be used to extend a template from another one. You can have multiple of them in a file but only one of them may be executed at the time. There is no support for multiple inheritance. See the section about *Template Inheritance* above.

3.8.7 Block

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

3.8.8 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 current template variables minus local modifications.

3.8.9 Import

Jinja2 supports putting often used code into macros. These macros can go into different templates and get imported from there. This works similar 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.

There are two ways to import templates. You can import the 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 is importing the whole 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 names from the 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>
```

3.9 Expressions

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

3.9.1 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 (for example as arguments to function calls, 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 for Python `42` and `42.0` is something different.

['list', 'of', 'objects']: Everything between two brackets is a list. Lists are useful to store sequential data in or to iterate over them. For example you can easily create a list of links using lists and tuples 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, just that you can't modify them. If the tuple only has one item you have to end it with a comma. Tuples are usually used to represent items of two or more elements. See the example above for more details.

{'dict': 'of', 'keys': '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. Keep in mind that those literals are lowercase!

3.9.2 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 with each other. Usually numbers but if both objects 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 at the ~ operator. `'{{ 1 + 1 }}'` is `2`.

- Substract two numbers from each other. `'{{ 3 - 2 }}'` is `1`.

/ Divide two numbers. The return value will be a floating point number. `'{{ 1 / 2 }}'` is `'{{ 0.5 }}'`.

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

% Calculate the remainder of an integer division between the left and right operand. `'{{ 11 % 7 }}'` is `4`.

***** Multiply the left operand with the right one. `'{{ 2 * 2 }}'` would return `4`. This can also be used to repeat 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`.

3.9.3 Logic

For *if* statements / *for* filtering or *if* expressions it can be useful to combine group multiple expressions:

and Return true if the left and the right operand is true.

or Return true if the left or the right operand is true.

not negate a statement (see below).

(expr) group an expression.

Note that there is no support for any bit operations or something similar.

- special note regarding `not`: 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).'`

3.9.4 Other Operators

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

in Perform 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 `'John'`) `'Hello John!'`.

() Call a callable: `'{{ post.render() }}'`. Inside of the parentheses you can use arguments and keyword arguments like in python: `'{{ post.render(user, full=true) }}'`.

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

3.9.5 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>'`.

3.10 List of Builtin Filters

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

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 missing items. See this example:

```
<table>
{%- for row in items|batch(3, ' ') %}
  <tr>
  {%- for column in row %}
    <tr>{{ 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'*)
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|dicsort(true) %}
    sort the dict by key, case sensitive

{% for item in mydict|dictsort(false, 'value') %}
    sort the dict by key, case insensitive, sorted
    normally and ordered by value.
```

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.

aliases `e`

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

first(*seq*)
Return the frist 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.

indent (*s, width=4, indentfirst=False*)
Return a copy of the passed string, each line indented by 4 spaces. The first line is not indented. If you want to change the number of spaces or indent the first line too you can pass additional parameters to the filter:

```
{{ mytext|indent(2, True) }}
    indent by two spaces and indent the first line too.
```

int (*value, default=0*)
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.

join (*value, d=u''*)
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 ith with the optional parameter:

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

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

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.

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.

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

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

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 div 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*)
Sort a sequence. Per default it sorts ascending, if you pass it true as first argument it will reverse the sorting.

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 (*sequence, start=0*)
Returns the sum of a sequence of numbers (NOT strings) plus the value of parameter 'start'. When the sequence is empty, returns start.

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

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

truncate (*s, length=255, killwords=False, end='...'*)
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 try to save 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.

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

urlize (*value, trim_url_limit=None, nofollow=False*)
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"
```

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

wordwrap (*s, pos=79, hard=False*)
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 *true* Jinja will also split words apart (usually a bad idea because it makes reading hard).

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.

3.11 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.

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

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

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

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

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.

CHAPTER

FOUR

Indices and tables

- *Index*
- *Module Index*
- *Search Page*

INDEX

J

INDEX

S

T

U

W

X