Jinja2英文文档

书栈(BookStack.CN)

目　录

致谢

当前文档 《Jinja2英文文档》 由 进击的皇虫 使用 书栈(BookStack.CN) 进行构建，生成于 2018-07-08。

书栈(BookStack.CN) 仅提供文档编写、整理、归类等功能，以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理，书栈(BookStack.CN) 难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候，发现文档内容有不恰当的地方，请向我们反馈，让我们共同携手，将知识准确、高效且有效地传递给每一个人。

同时，如果您在日常工作、生活和学习中遇到有价值有营养的知识文档，欢迎分享到 书栈(BookStack.CN) ，为知识的传承献上您的一份力量！

如果当前文档生成时间太久，请到 书栈(BookStack.CN) 获取最新的文档，以跟上知识更新换代的步伐。

文档地址：http://www.bookstack.cn/books/jinja2-en

书栈官网：http://www.bookstack.cn

书栈开源：https://github.com/TruthHun

分享，让知识传承更久远！ 感谢知识的创造者，感谢知识的分享者，也感谢每一位阅读到此处的读者，因为我们都将成为知识的传承者。

README

Welcome to Jinja2¶

Jinja2 is a modern and designer-friendly templating language for Python,modelled after Django’s templates. It is fast, widely used and securewith the optional sandboxed template execution environment:

```
<title>{% block title %}{% endblock %}</title>
<ul>
{% for user in users %}
  <li><a href="{{ user.url }}">{{ user.username }}</a></li>
{% endfor %}
</ul>
```

Features:

- sandboxed execution
- powerful automatic HTML escaping system for XSS prevention
- template inheritance
- compiles down to the optimal python code just in time
- optional ahead-of-time template compilation
- easy to debug. Line numbers of exceptions directly point tothe correct line in the template.
- configurable syntax

Jinja2 Documentation¶

- Introduction
- API
- Sandbox
- Native Python Types
- Template Designer Documentation
- Extensions
- Integration
- Switching from other Template Engines
- Tips and Tricks

Additional Information¶

- Frequently Asked Questions
- Jinja Changelog

If you can’t find the information you’re looking for, have a look at theindex or try to find it using the search function:

原文：

> *http://jinja.pocoo.org/docs/2.10/*

Introduction

Introduction¶

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

If you have any exposure to other text-based template languages, such as Smarty orDjango, you should feel right at home with Jinja2. It’s both designer anddeveloper friendly by sticking to Python’s principles and adding functionalityuseful for templating environments.

Prerequisites¶

Jinja2 works with Python 2.6.x, 2.7.x and >= 3.3. If you are using Python3.2 you can use an older release of Jinja2 (2.6) as support for Python 3.2was dropped in Jinja2 version 2.7.

If you wish to use the `PackageLoader` class, you will alsoneed setuptools or distribute installed at runtime.

Installation¶

You have multiple ways to install Jinja2. If you are unsure what to do, gowith the Python egg or tarball.

As a Python egg (via easy_install)¶

You can install the most recent Jinja2 version using easy_install or pip:

```
1. easy_install Jinja2
2. pip install Jinja2
```

This will install a Jinja2 egg in your Python installation’s site-packagesdirectory.

From the tarball release¶

- Download the most recent tarball from the download page
- Unpack the tarball
- python setup.py install

Note that you either have to have *setuptools* or *distribute* installed;the latter is

preferred.
This will install Jinja2 into your Python installation’s site-packages directory.

Installing the development version¶

- Install git
- git clone git://github.com/pallets/jinja.git
- cd jinja2
- ln -s jinja2 /usr/lib/python2.X/site-packages

As an alternative to steps 4 you can also do `python setup.py develop` which will install the package via *distribute* in development mode. This alsohas the advantage that the C extensions are compiled.

MarkupSafe Dependency¶

As of version 2.7 Jinja2 depends on the MarkupSafe module. If youinstall Jinja2 via *pip* or *easy_install* it will be installedautomatically for you.

Basic API Usage¶

This section gives you a brief introduction to the Python API for Jinja2templates.

The most basic way to create a template and render it is through `Template` . This however is not the recommended way towork with it if your templates are not loaded from strings but the filesystem or another data source:

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

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

What you can see here is that Jinja2 is using unicode internally and thereturn value is an unicode string. So make sure that your application isindeed using unicode internally.

Experimental Python 3 Support¶

Jinja 2.7 brings experimental support for Python >=3.3. It means that allunittests pass on the new version, but there might still be small bugs inthere and behavior

might be inconsistent. If you notice any bugs, pleaseprovide feedback in the Jinja bug tracker.

Also please keep in mind that the documentation is written with Python 2in mind, so you will have to adapt the shown code examples to Python 3 syntaxfor yourself.

原文:

> *http://jinja.pocoo.org/docs/2.10/intro/*

API

API¶

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

Basics¶

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

Most applications will create one `Environment` object on applicationinitialization and use that to load templates. In some cases however, it’suseful to have multiple environments side by side, if different configurationsare in use.

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

```
1. from jinja2 import Environment, PackageLoader, select_autoescape
2. env = Environment(
3.     loader=PackageLoader('yourapplication', 'templates'),
4.     autoescape=select_autoescape(['html', 'xml'])
5. )
```

This will create a template environment with the default settings and aloader that looks up the templates in the *templates* folder inside the*yourapplication* python package. Different loaders are availableand you can also write your own if you want to load templates from adatabase or other resources. This also enables autoescaping for HTML andXML files.

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

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

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

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

Using a template loader rather than passing strings to `Template` or `Environment.from_string()` has multiple advantages. Besides beinga lot easier to use it also enables template inheritance.

Notes on Autoescaping

In future versions of Jinja2 we might enable autoescaping by defaultfor security reasons. As such you are encouraged to explicitlyconfigure autoescaping now instead of relying on the default.

Unicode¶

Jinja2 is using Unicode internally which means that you have to pass Unicodeobjects to the render function or bytestrings that only consist of ASCIIcharacters. Additionally newlines are normalized to one end of linesequence which is per default UNIX style (`\n`).

Python 2.x supports two ways of representing string objects. One is the*str* type and the other is the *unicode* type, both of which extend a typecalled *basestring*. Unfortunately the default is *str* which should notbe used to store text based information unless only ASCII characters areused. With Python 2.6 it is possible to make *unicode* the default on a permodule level and with Python 3 it will be the default.

To explicitly use a Unicode string you have to prefix the string literalwith a *u*: `u'Hänsel und Gretel sagen Hallo'` . That way Python willstore the string as Unicode by decoding the string with the characterencoding from the current Python module. If no encoding is specified thisdefaults to 'ASCII' which means that you can't use any non ASCII identifier.

To set a better module encoding add the following comment to the first orsecond line of the Python module using the Unicode literal:

```
1. # -*- coding: utf-8 -*-
```

We recommend utf-8 as Encoding for Python modules and templates as it'spossible to represent every Unicode character in utf-8 and because it'sbackwards compatible to ASCII. For Jinja2 the default encoding of templatesis assumed to be utf-8.

It is not possible to use Jinja2 to process non-Unicode data. The reasonfor this is that Jinja2 uses Unicode already on the language level. Forexample Jinja2 treats the non-breaking space as valid whitespace insideexpressions which requires knowledge of the encoding or operating on anUnicode string.

For more details about Unicode in Python have a look at the excellentUnicode documentation.

Another important thing is how Jinja2 is handling string literals intemplates. A naive

implementation would be using Unicode strings forall string literals but it turned out in the past that this is problematicas some libraries are typechecking against *str* explicitly. For example*datetime.strftime* does not accept Unicode arguments. To not break itcompletely Jinja2 is returning *str* for strings that fit into ASCII andfor everything else *unicode*:

```
>>> m = Template(u"{% set a, b = 'foo', 'föö' %}").module
>>> m.a
'foo'
>>> m.b
u'f\xf6\xf6'
```

High Level API¶

The high-level API is the API you will use in the application to load andrender Jinja2 templates. The Low Level API on the other side is onlyuseful if you want to dig deeper into Jinja2 or develop extensions.

class `jinja2.` `Environment` ***([options])¶***

The core component of Jinja is the *Environment*. It containsimportant shared variables like configuration, filters, tests,globals and others. Instances of this class may be modified ifthey are not shared and if no template was loaded so far.Modifications on environments after the first template was loadedwill lead to surprising effects and undefined behavior.

Here are the possible initialization parameters:

> ***block_start_string***
> *The string marking the beginning 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 beginning of a print statement.Defaults to* `'{{'`.
>
> ***variable_end_string***
> *The string marking the end of a print statement. Defaults to* `'}}'`.
>
> ***comment_start_string***
> *The string marking the beginning 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 basedstatements. See also Line Statements.*
>
> ***line_comment_prefix***

If given and a string, this will be used as prefix for line basedcomments. See also Line Statements.New in version 2.2.

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

lstrip_blocks
If this is set to `True` leading spaces and tabs are strippedfrom the start of a line to a block. Defaults to False.

newline_sequence
The sequence that starts a newline. Must be one of `'\r'`, `'\n'` or `'\r\n'`. The default is `'\n'` which is auseful default for Linux and OS X systems as well as webapplications.

keep_trailing_newline
Preserve the trailing newline when rendering templates.The default is `False`, which causes a single newline,if present, to be stripped from the end of the template.New in version 2.7.

extensions
List of Jinja extensions to use. This can either be import pathsas strings or extension classes. For more information have alook at the extensions documentation.

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

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

finalize
A callable that can be used to process the result of a variableexpression before it is output. For example one can convert `None` implicitly into an empty string here.

autoescape
If set to `True` the XML/HTML autoescaping feature is enabled bydefault. For more details about autoescaping see `Markup`. As of Jinja 2.4 this can alsobe a callable that is passed the template name and has toreturn `True` or `False` depending on autoescape should beenabled by default.Changed in version 2.4: autoescape can now be a function

loader
The template loader for this environment.

cache_size
The size of the cache. Per default this is `400` which meansthat if more than 400 templates are loaded the loader will cleanout 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.Changed in version 2.8: The cache size was increased to 400 from a low 50.

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

_bytecode_cache
If set to a bytecode cache object, this object will provide acache for the internal Jinja bytecode so that templates don'thave to be parsed if they were not changed.See Bytecode Cache for more information.

enable_async
If set to true this enables async template execution which allowsyou to take advantage of newer Python features. This requiresPython 3.6 or later.

`shared` ¶

If a template was created by using the `Template` constructoran environment is created automatically. These environments arecreated as shared environments which means that multiple templatesmay have the same anonymous environment. For all shared environmentsthis attribute is *True*, else *False*.

`sandboxed` ¶

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

`filters` ¶

A dict of filters for this environment. As long as no template wasloaded it's safe to add new filters or remove old. For custom filterssee Custom Filters. For valid filter names have a look atNotes on Identifiers.

`tests` ¶

A dict of test functions for this environment. As long as notemplate was loaded it's safe to modify this dict. For custom testssee Custom Tests. For valid test names have a look atNotes on Identifiers.

`globals` ¶

A dict of global variables. These variables are always availablein a template. As long as no template was loaded it's safeto modify this dict. For more details see The Global Namespace.For valid object names have a look at Notes on Identifiers.

`policies` ¶

A dictionary with Policies. These can be reconfigured tochange the runtime behavior or certain template features. Usuallythese are security related.

`codegenerator_class` ¶

The class used for code generation. This should not be changedin most cases, unless you need to modify the Python code atemplate compiles to.

`context_class` ¶

The context used for templates. This should not be changedin most cases, unless you need to modify internals of howtemplate variables are handled. For details, see`Context`.

`overlay` *([_options])*¶

Create a new overlay environment that shares all the data with thecurrent environment except for cache and the overridden attributes.Extensions cannot be removed for an overlayed environment. An overlayedenvironment automatically gets all the extensions of the environment itis linked to plus optional extra extensions.

Creating overlays should happen after the initial environment was setup completely. Not all attributes are truly linked, some are justcopied over so modifications on the original environment may not shinethrough.

`undefined` *([hint, obj, name, exc])*¶

Creates a new `Undefined` object for *name*. This is usefulfor filters or functions that may return undefined objects forsome operations. All parameters except of *hint* should be providedas keyword parameters for better readability. The *hint* is used aserror message for the exception if provided, otherwise the errormessage will be generated from *obj* and *name* automatically. The exceptionprovided as *exc* is raised if something with the generated undefinedobject is done that the undefined object does not allow. The defaultexception is `UndefinedError`. If a *hint* is provided the*name* may be omitted.

The most common way to create an undefined object is by providinga name only:

```
1. return environment.undefined(name='somename')
```

This means that the name _some_name is not defined. If the namewas from an attribute of an object it makes sense to tell theundefined object the holder object to improve the error message:

```
1. if not hasattr(obj, 'attr'):
       return environment.undefined(obj=obj, name='attr')
```

For a more complex example you can provide a hint. For examplethe `first()` filter creates an undefined object that way:

```
1. return environment.undefined('no first item, sequence was empty')
```

If it the *name* or *obj* is known (for example because an attributewas accessed) it should be passed to the undefined object, even ifa custom *hint* is provided. This gives undefined objects thepossibility to enhance the error message.

`addextension`*(_extension)*¶

Adds an extension after the environment was created.

New in version 2.5.

`compileexpression`*(_source, undefined_to_none=True)*¶

A handy helper method that returns a callable that accepts keywordarguments that appear as variables in the expression. If called itreturns the result of the expression.

This is useful if applications want to use the same rules as Jinjain template “configuration files” or similar situations.

Example usage:

```
1. >>> env = Environment()
   >>> expr = env.compileexpression('foo == 42')
   >>> expr(foo=23)
   False
   >>> expr(foo=42)
   True
```

Per default the return value is converted to _None if theexpression returns an undefined value. This can be changedby setting *undefined_to_none* to *False*.

```
>>> env.compileexpression('var')() is None
True
>>> env.compile_expression('var', undefined_to_none=False)()
Undefined
```

New in version 2.1.

`compile_templates`***(_target, extensions=None, filter_func=None, zip='deflated', log_function=None, ignore_errors=True, py_compile=False)¶***

Finds all the templates the loader can find, compiles themand stores them in *target*. If *zip* is *None*, instead of in azipfile, the templates will be stored in a directory.By default a deflate zip algorithm is used. To switch tothe stored algorithm, *zip* can be set to `'stored'`.

extensions and *filter_func* are passed to `list_templates()`.Each template returned will be compiled to the target folder orzipfile.

By default template compilation errors are ignored. In case alog function is provided, errors are logged. If you want templatesyntax errors to abort the compilation you can set *ignore_errors_to _False* and you will get an exception on syntax errors.

If *py_compile* is set to *True* .pyc files will be written to thetarget instead of standard .py files. This flag does not do anythingon pypy and Python 3 where pyc files are not picked up by itself anddon’t give much benefit.

New in version 2.4.

`extend`***(**attributes)¶***

Add the items to the instance of the environment if they do not existyet. This is used by extensions to registercallbacks and configuration values without

breaking inheritance.

`fromstring` ***(_source, globals=None, template_class=None)¶***

Load a template from a string. This parses the source given andreturns a `Template` object.

`getor_select_template` ***(_template_name_or_list, parent=None, globals=None)¶***

Does a typecheck and dispatches to `select_template()` if an iterable of template names is given, otherwise to `get_template()`.

New in version 2.3.

`gettemplate` ***(_name, parent=None, globals=None)¶***

Load a template from the loader. If a loader is configured thismethod asks the loader for the template and returns a `Template`.If the *parent* parameter is not *None*, `join_path()` is calledto get the real template name before loading.

The *globals* parameter can be used to provide template wide globals.These variables are available in the context at render time.

If the template does not exist a `TemplateNotFound` exception israised.

Changed in version 2.4: If *name* is a `Template` object it is returned from thefunction unchanged.

`joinpath` ***(_template, parent)¶***

Join a template with the parent. By default all the lookups arerelative to the loader root so this method returns the *template_parameter unchanged, but if the paths should be relative to theparent template, this function can be used to calculate the realtemplate name.*

Subclasses may override this method and implement template pathjoining here.

`list_templates` ***(_extensions=None, filter_func=None)¶***

Returns a list of templates for this environment. This requiresthat the loader supports the loader's `listtemplates()` *method.*

If there are other files in the template folder besides theactual templates, the returned list can be filtered. There are twoways: either _extensions is set

to a list of file extensions fortemplates, or a *filter_func* can be provided which is a callable thatis passed a template name and should return *True* if it should end upin the result list.

If the loader does not support that, a `TypeError` is raised.

New in version 2.4.

`selecttemplate`*(_names, parent=None, globals=None)*¶

Works like `get_template()` but tries a number of templatesbefore it fails. If it cannot find any of the templates, it willraise a `TemplatesNotFound` exception.

New in version 2.3.

Changed in version 2.4: If *names* contains a `Template` object it is returnedfrom the function unchanged.

class `jinja2.` `Template`¶

The central template object. This class represents a compiled templateand is used to evaluate it.

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

Every template object has a few methods and members that are guaranteedto exist. However it’s important that a template object should beconsidered immutable. Modifications on the object are not supported.

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

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

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

`globals` ¶

The dict with the globals of that template. It’s unsafe to modifythis dict as it may be shared with other templates or the environmentthat loaded the template.

`name` ¶

The loading name of the template. If the template was loaded from astring this is *None*.

`filename` ¶

The filename of the template on the file system if it was loaded fromthere. Otherwise this is *None*.

`render` ***([context])*** ¶

This method accepts the same arguments as the *dict* constructor:A dict, a dict subclass or some keyword arguments. If no argumentsare 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.

`generate` ***([context])*** ¶

For very large templates it can be useful to not render the wholetemplate at once but evaluate each statement after another and yieldpiece for piece. This method basically does exactly that and returnsa generator that yields one item

after another as unicode strings.

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

`stream` *([context])*¶

Works exactly like `generate()` but returns a `TemplateStream`.

`renderasync` *([_context])*¶

This works similar to `render()` but returns a coroutinethat when awaited returns the entire rendered template string. Thisrequires the async feature to be enabled.

Example usage:

```
1. await template.renderasync(knights='that say nih; asynchronously')
```

`generate_async` *([_context])*¶

An async version of `generate()`. Works very similarly butreturns an async iterator instead.

`makemodule` *(_vars=None, shared=False, locals=None)*¶

This method works like the `module` attribute when calledwithout arguments but it will evaluate the template on every callrather than caching it. It’s also possible to providea dict which is then used as context. The arguments are the sameas for the `new_context()` method.

`module`¶

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

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

```
    True
```

This attribute is not available if async mode is enabled.

class `jinja2.environment.` `TemplateStream` ¶

A template stream works pretty much like an ordinary python generatorbut it can buffer multiple items to reduce the number of total iterations.Per default the output is unbuffered which means that for every unbufferedinstruction in the template one unicode string is yielded.

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

`disablebuffering` ***()*** ¶

Disable the output buffering.

`dump` ***(_fp, encoding=None, errors='strict')*** ¶

Dump the complete stream into a file or file-like object.Per default unicode strings are written, if you want to encodebefore writing specify an *encoding*.

Example usage:

```
1. Template('Hello {{ name }}!').stream(name='foo').dump('hello.html')
```

`enablebuffering` ***(_size=5)*** ¶

Enable buffering. Buffer *size* items before yielding them.

Autoescaping¶

Changed in version 2.4.

Jinja2 now comes with autoescaping support. As of Jinja 2.9 theautoescape extension is

removed and built-in. However autoescaping isnot yet enabled by default though this will most likely change in thefuture. It’s recommended to configure a sensible default forautoescaping. This makes it possible to enable and disable autoescapingon a per-template basis (HTML versus text for instance).

`jinja2.` `selectautoescape` ***(_enabled_extensions=('html', 'htm', 'xml'), disabled_extensions=(), default_for_string=True, default=False)¶***

Intelligently sets the initial value of autoescaping based on thefilename of the template. This is the recommended way to configureautoescaping if you do not want to write a custom function yourself.

If you want to enable it for all templates created from strings orfor all templates with *.html* and *.xml* extensions:

```
from jinja2 import Environment, selectautoescape
env = Environment(autoescape=select_autoescape(
    enabled_extensions=('html', 'xml'),
    default_for_string=True,
))
```

Example configuration to turn it on at all times except if the templateends with .txt*:*

```
from jinja2 import Environment, select_autoescape
env = Environment(autoescape=select_autoescape(
    disabled_extensions=('txt',),
    default_for_string=True,
    default=True,
))
```

The _enabled_extensions is an iterable of all the extensions thatautoescaping should be enabled for. Likewise *disabled_extensions* isa list of all templates it should be disabled for. If a template isloaded from a string then the default from *default_for_string* is used.If nothing matches then the initial value of autoescaping is set to thevalue of *default*.

For security reasons this function operates case insensitive.

New in version 2.9.

Here a recommended setup that enables autoescaping for templates endingin `'.html'`, `'.htm'` and `'.xml'` and disabling it by defaultfor all other extensions. You can use the `select_autoescape()` function for this:

```
1. from jinja2 import Environment, selectautoescape
   env = Environment(autoescape=select_autoescape(['html', 'htm', 'xml']),
                     loader=PackageLoader('mypackage'))
```

The `select_autoescape()` function returns a function thatworks rougly like this:

```
1. def autoescape(template_name):
       if template_name is None:
           return False
       if template_name.endswith(('.html', '.htm', '.xml'))
```

When implementing a guessing autoescape function, make sure you alsoaccept _None as valid template name. This will be passed when generatingtemplates from strings. You should always configure autoescaping asdefaults in the future might change.

Inside the templates the behaviour can be temporarily changed by usingthe *autoescape* block (see Autoescape Overrides).

Notes on Identifiers¶

Jinja2 uses the regular Python 2.x naming rules. Valid identifiers have tomatch `[a-zA-`

`Z][a-zA-Z0-9]`. *As a matter of fact non ASCII charactersare currently not allowed. This limitation will probably go away as soon asunicode identifiers are fully specified for Python 3.*

Filters and tests are looked up in separate namespaces and have slightlymodified identifier syntax. Filters and tests may contain dots to groupfilters and tests by topic. For example it's perfectly valid to add afunction into the filter dict and call it to.unicode. The regularexpression for filter and test identifiers is `[a-zA-Z][a-zA-Z0-9]` `(.[a-zA-Z][a-zA-Z0-9])`` .

Undefined Types¶

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

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

class `jinja2.` `Undefined` ¶

The default undefined type. This undefined type can be printed anditerated over, but every other access will raise an `jinja2.exceptions.UndefinedError`:

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

`undefined_hint` ¶

Either _None or an unicode string with the error message forthe undefined object.

`undefined_obj` ¶

Either _None or the owner object that caused the undefined objectto be created (for example because an attribute does not exist).

`undefined_name` ¶

The name for the undefined variable / attribute or just _None_if no such information exists.

`_undefined_exception` ¶

The exception that the undefined object wants to raise. Thisis usually one of `UndefinedError` *or* `SecurityError` *.*

`_fail_with_undefined_error`***(args, kwargs)***¶

When called with any arguments this method raises `_undefined_exception` ***with an error message generatedfrom the undefined hints stored on the undefined object.***

_class `jinja2.` `DebugUndefined` ¶

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 `jinja2.` `StrictUndefined` ¶

An undefined that barks on print and iteration as well as booleantests and all kinds of comparisons. In other words: you can do nothingwith 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
```

There is also a factory function that can decorate undefined objects toimplement logging on failures:

`jinja2.` `makelogging_undefined` *(_logger=None, base=None)¶*

Given a logger object this returns a new undefined class that willlog certain failures. It will log iterations and printing. If nologger is given a default logger is created.

Example:

```
1. logger = logging.getLogger(name)
   LoggingUndefined = make_logging_undefined(
       logger=logger,
       base=Undefined
   )
```

New in version 2.8.

|Parameters:
|——-
|
- logger – the logger to use. If not provided, a default loggeris created.
- base – the base class to add logging functionality to. Thisdefaults to Undefined.

Undefined objects are created by calling `undefined`*.*

Implementation

`Undefined` *objects are implemented by overriding the specialunderscore methods. For example the default* `Undefined` *class implements unicode in a way that it returns an emptystring, however int and others still fail with an exception. Toallow conversion to int by returning* `0` *you can implement your own:*

```
class NullUndefined(Undefined):
    def int(self):
        return 0
    def float(self):
        return 0.0
```

To disallow a method, just override it and raise `_undefined_exception`*. Because this is a very commonidom in undefined objects there is the helper method* `_fail_with_undefined_error()` *that does the error raisingautomatically. Here a class that works like the regular* `Undefined` *but chokes on iteration:*

```
class NonIterableUndefined(Undefined):
    iter = Undefined.fail_with_undefined_error
```

The Context¶

_class `jinja2.runtime.` `Context` *¶*

> *The template context holds the variables of a template. It stores thevalues passed to the template and also the names the template exports.Creating instances is neither supported nor useful as it's createdautomatically at various stages of the*

template evaluation and should notbe created by hand.

The context is immutable. Modifications on `parent` *must not**happen and modifications on* `vars` *are allowed from generatedtemplate code only. Template filters and global functions marked as* `contextfunction()` *s get the active context passed as first argumentand are allowed to access the context read-only.*

The template context supports read only dict operations (get,keys, values, items, iterkeys, itervalues, iteritems, ***getitem****,* ***contains****). Additionally there is a* `resolve()` *method that doesn't fail with a KeyError but returns an* `Undefined` *object for missing variables.*

`parent` ¶

A dict of read only, global variables the template looks up. Thesecan either come from another `Context`*, from the* `Environment.globals` *or* `Template.globals` *or pointsto a dict created by combining the globals with the variablespassed to the render function. It must not be altered.*

`vars` ¶

The template local variables. This list contains environment andcontext functions from the `parent` *scope as well as localmodifications and exported variables from the template. The templatewill modify this dict during template evaluation but filters andcontext functions are not allowed to modify it.*

`environment` ¶

The environment that loaded the template.

`exportedvars` ¶

This set contains all the names the template exports. The values forthe names are in the `vars` *dict. In order to get a copy of theexported variables as dict,* `get_exported()` *can be used.*

`name` ¶

The load name of the template owning this context.

`blocks` ¶

A dict with the current mapping of blocks in the template. The keysin this dict are the names of the blocks, and the values a list ofblocks registered. The last item in each list is the current activeblock (latest in the inheritance chain).

`eval_ctx` ¶

The current Evaluation Context.

`call`**(_callable, _args, **kwargs)¶**

Call the callable with the arguments and keyword argumentsprovided but inject the active context or environment as firstargument if the callable is a `contextfunction()` *or* `environmentfunction()`*.*

`get_all`**()¶**

Return the complete context as dict including the exportedvariables. For optimizations reasons this might not return anactual copy so be careful with using it.

`get_exported`**()¶**

Get a new dict with the exported variables.

`resolve`**(_key)¶**

Looks up a variable like ***getitem*** or *get* but returns an `Undefined` object with the name of the name looked up.

Implementation

Context is immutable for the same reason Python's frame locals areimmutable inside functions. Both Jinja2 and Python are not using thecontext / frame locals as data storage for variables but only as primarydata source.

When a template accesses a variable the template does not define, Jinja2looks up the variable in the context, after that the variable is treatedas if it was defined in the template.

Loaders¶

Loaders are responsible for loading templates from a resource such as thefile system. The environment will keep the compiled modules in memory likePython's *sys.modules*. Unlike *sys.modules* however this cache is limited insize by default and templates are automatically reloaded.All loaders are subclasses of `BaseLoader`. If you want to create yourown loader, subclass `BaseLoader` and override *get_source*.

class `jinja2.` `BaseLoader` **¶**

Baseclass for all loaders. Subclass this and override *get_source* toimplement 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 filesystem could look like this:

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

class MyLoader(BaseLoader):

    def init(self, path):
        self.path = path

    def getsource(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 atuple 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 thetemplate as unicode string or a ASCII bytestring. The filename shouldbe the name of the file on the filesystem if it was loaded from there,otherwise *None*. The filename is used by python for the tracebacksif no loader extension is used.

The last item in the tuple is the *uptodate* function. If autoreloading is enabled it’s always called to check if the templatechanged. No arguments are passed so the function must store theold 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 cacheor loads one by calling `get_source()`. Subclasses should notoverride this method as loaders working on collections of otherloaders (such as `PrefixLoader` or `ChoiceLoader`)will not call this method but *get_source* directly.

Here a list of the builtin loaders Jinja2 provides:

class `jinja2.` `FileSystemLoader` ***(searchpath, encoding='utf-8', followlinks=False)***¶

Loads templates from the file system. This loader can find templatesin 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 multiplelocations are wanted a list of them which is then looked up in thegiven order:

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

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

To follow symbolic links, set the *followlinks* parameter to `True`:

```
>>> loader = FileSystemLoader('/path/to/templates', followlinks=True)
```

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

class `jinja2.` `PackageLoader` ***(package_name, package_path='templates', encoding='utf-8')***¶

Load templates from python eggs or packages. It is constructed withthe name of the python package and the path to the templates in thatpackage:

```
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 changedby setting the *encoding* parameter to something else. Due to the natureof eggs it's only possible to reload templates if the package was loadedfrom the file system and not a zip file.

class `jinja2.` `DictLoader` *(mapping)*¶

Loads a template from a python dict. It's passed a dict of unicodestrings bound to template names. This loader is useful for unittesting:

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

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

class `jinja2.` `FunctionLoader` *(load_func)*¶

A loader that is passed a function which does the loading. Thefunction receives the name of the template and has to return eitheran unicode string with the template source, a tuple in the form `(source, filename, uptodatefunc)` or *None* if the template does not exist.

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

The *_uptodatefunc* is a function that is called if autoreload is enabledand has to return *True* if the template is still up to date. For moredetails have a look at `BaseLoader.get_source()` which has the samereturn value.

class jinja2.PrefixLoader(*mapping*, *delimiter='/'*)¶

A loader that is passed a dict of loaders where each loader is boundto a prefix. The prefix is delimited from the template by a slash perdefault, which can be changed by setting the *delimiter* argument tosomething else:

```
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 jinja2.ChoiceLoader(*loaders*)¶

This loader works like the *PrefixLoader* just that no prefix isspecified. If a template could not be found by one loader the next oneis tried.

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

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

class jinja2.ModuleLoader(*path*)¶

This loader loads templates from precompiled templates.

Example usage:

```
>>> loader = ChoiceLoader([
```

```
…     ModuleLoader('/path/to/compiled/templates'),
…     FileSystemLoader('/path/to/templates')
… ])
```

Templates can be precompiled with `Environment.compile_templates()`.

Bytecode Cache¶

Jinja 2.1 and higher support external bytecode caching. Bytecode caches makeit possible to store the generated bytecode on the file system or a differentlocation to avoid parsing the templates on first use.

This is especially useful if you have a web application that is initialized onthe first request and Jinja compiles many templates at once which slows downthe application.

To use a bytecode cache, instantiate it and pass it to the `Environment`.

class `jinja2.` `BytecodeCache` ¶

To implement your own bytecode cache you have to subclass this classand override `load_bytecode()` and `dump_bytecode()`. Both ofthese methods are passed a `Bucket`.

A very basic bytecode cache that saves the bytecode on the file system:

```
from os import path

class MyCache(BytecodeCache):

    def init(self, directory):
        self.directory = directory

    def loadbytecode(self, bucket):
        filename = path.join(self.directory, bucket.key)
        if path.exists(filename):
            with open(filename, 'rb') as f:
                bucket.load_bytecode(f)

    def dump_bytecode(self, bucket):
        filename = path.join(self.directory, bucket.key)
        with open(filename, 'wb') as f:
            bucket.write_bytecode(f)
```

A more advanced version of a filesystem based bytecode cache is part ofJinja2.

`clear`*()*¶

Clears the cache. This method is not used by Jinja2 but should beimplemented to allow applications to clear the bytecode cache usedby a particular environment.

`dump_bytecode`*(_bucket)*¶

Subclasses have to override this method to write the bytecodefrom a bucket back to the cache. If it unable to do so it must notfail silently but raise an exception.

`loadbytecode`*(_bucket)*¶

Subclasses have to override this method to load bytecode into abucket. If they are not able to find code in the cache for thebucket, it must not do anything.

class `jinja2.bccache.` `Bucket`***(environment, key, checksum)***¶

Buckets are used to store the bytecode for one template. It’s createdand initialized by the bytecode cache and passed to the loading functions.

The buckets get an internal checksum from the cache assigned and use thisto automatically reject outdated cache material. Individual bytecodecache subclasses don’t have to care about cache invalidation.

`environment`¶

The `Environment` that created the bucket.

`key`¶

The unique cache key for this bucket

`code`¶

The bytecode if it’s loaded, otherwise *None*.

`bytecodefrom_string`*(_string)*¶

Load bytecode from a string.

`bytecodeto_string`*()*¶

Return the bytecode as string.

`load_bytecode`*(_f)*¶

Loads bytecode from a file or file like object.

`reset`*()¶*

Resets the bucket (unloads the bytecode).

`writebytecode`*(_f)¶*

Dump the bytecode into the file or file like object passed.

Builtin bytecode caches:

class `jinja2.` `FileSystemBytecodeCache`*(directory=None, pattern='__jinja2%s.cache')¶*

A bytecode cache that stores bytecode on the filesystem. It acceptstwo arguments: The directory where the cache items are stored and apattern string that is used to build the filename.

If no directory is specified a default cache directory is selected. OnWindows the user's temp directory is used, on UNIX systems a directoryis created for the user in the system temp directory.

The pattern can be used to have multiple separate caches operate on thesame directory. The default pattern is `'__jinja2` `%s.cache'` . `%s` is replaced with the cache key.

```
1. >>> bcc = FileSystemBytecodeCache('/tmp/jinjacache', '%s.cache')
```

This bytecode cache supports clearing of the cache using the clear method.

_class `jinja2.` `MemcachedBytecodeCache`*(client, prefix='jinja2/bytecode/', timeout=None, ignore_memcache_errors=True)¶*

This class implements a bytecode cache that uses a memcache cache forstoring the information. It does not enforce a specific memcache library(tummy's memcache or cmemcache) but will accept any class that providesthe minimal interface required.

Libraries compatible with this class:

- werkzeug.contrib.cache
- python-memcached
- cmemcache

(Unfortunately the django cache interface is not compatible because itdoes not support storing binary data, only unicode. You can however passthe underlying cache client to the bytecode cache which is availableas *django.core.cache.cache._client*.)

The minimal interface for the client passed to the constructor is this:

class `MinimalClientInterface` ¶

`set` ***(key, value[, timeout])*** ¶

Stores the bytecode in the cache. *value* is a string and*timeout* the timeout of the key. If timeout is not provideda default timeout or no timeout should be assumed, if it’sprovided it’s an integer with the number of seconds the cacheitem should exist.

`get` ***(key)*** ¶

Returns the value for the cache key. If the item does notexist in the cache the return value must be *None*.

The other arguments to the constructor are the prefix for all keys thatis added before the actual cache key and the timeout for the bytecode inthe cache system. We recommend a high (or no) timeout.

This bytecode cache does not support clearing of used items in the cache.The clear method is a no-operation function.

New in version 2.7: Added support for ignoring memcache errors through the*ignore_memcache_errors* parameter.

Async Support¶

Starting with version 2.9, Jinja2 also supports the Python *async* and*await* constructs. As far as template designers go this feature isentirely opaque to them however as a developer you should be aware of howit’s implemented as it influences what type of APIs you can safely exposeto the template environment.

First you need to be aware that by default async support is disabled asenabling it will generate different template code behind the scenes whichpasses everything through the asyncio event loop. This is important tounderstand because it has some impact to

what you are doing:

- template rendering will require an event loop to be set for thecurrent thread (asyncio.get_event_loop needs to return one)
- all template generation code internally runs async generators whichmeans that you will pay a performance penalty even if the non syncmethods are used!
- The sync methods are based on async methods if the async mode isenabled which means that *render* for instance will internally invoke*render_async* and run it as part of the current event loop until theexecution finished.
  Awaitable objects can be returned from functions in templates and anyfunction call in a template will automatically await the result. Thismeans that you can let provide a method that asynchronously loads datafrom a database if you so desire and from the template designer's point ofview this is just another function they can call. This means that the `await` you would normally issue in Python is implied. However thisonly applies to function calls. If an attribute for instance would be anavaitable object then this would not result in the expected behavior.

Likewise iterations with a *for* loop support async iterators.

Policies¶

Starting with Jinja 2.9 policies can be configured on the environmentwhich can slightly influence how filters and other template constructsbehave. They can be configured with the `policies` attribute.

Example:

```
1. env.policies['urlize.rel'] = 'nofollow noopener'
```

`compiler.asciistr`:
This boolean controls on Python 2 if Jinja2 should store ASCII onlyliterals as bytestring instead of unicode strings. This used to bealways enabled for Jinja versions below 2.9 and now can be changed.Traditionally it was done this way since some APIs in Python 2 failedbadly for unicode strings (for instance the datetime strftime API).Now however sometimes the inverse is true (for instance str.format).If this is set to False then all strings are stored as unicodeinternally.

`truncate.leeway`:
Configures the leeway default for the _truncate filter. Leeway asintroduced in 2.9 but to restore compatibility with older templatesit can be configured to *0* to get the old behavior back. The defaultis *5*.

`urlize.rel`:
A string that defines the items for the *rel* attribute of generatedlinks with the *urlize* filter. These items are always added. Thedefault is *noopener*.

`urlize.target`:

The default target that is issued for links from the *urlize* filterif no other target is defined by the call explicitly.

`json.dumpsfunction`:

If this is set to a value other than _None then the *tojson* filterwill dump with this function instead of the default one. Note thatthis function should accept arbitrary extra arguments which might bepassed in the future from the filter. Currently the only argumentthat might be passed is *indent*. The default dump function is `json.dumps`.

`json.dumpskwargs`:

Keyword arguments to be passed to the dump function. The default is `{'sort_keys': True}`.

`ext.i18n.trimmed`:

If this is set to _True, `{% trans %}` blocks of thei18n Extension will always unify linebreaks and surroundingwhitespace as if the *trimmed* modifier was used.

Utilities¶

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

`jinja2.` `environmentfilter`*(f)*¶

Decorator for marking environment dependent filters. The current `Environment` is passed to the filter as first argument.

`jinja2.` `contextfilter`*(f)*¶

Decorator for marking context dependent filters. The current `Context` will be passed as first argument.

`jinja2.` `evalcontextfilter`*(f)*¶

Decorator for marking eval-context dependent filters. An evalcontext object is passed as first argument. For more informationabout the eval context, see Evaluation Context.

New in version 2.4.

`jinja2.` `environmentfunction`*(f)*¶

This decorator can be used to mark a function or method as environmentcallable. This decorator works exactly like the `contextfunction()` decorator just that the first

argument is the active `Environment` and not context.

jinja2.contextfunction(f)¶

This decorator can be used to mark a function or method context callable.A context callable is passed the active `Context` as first argument whencalled from the template. This is useful if a function wants to get accessto the context or functions provided on the context object. For examplea function that returns a sorted list of template variables the currenttemplate exports could look like this:

```
@contextfunction
def getexported_names(context):
    return sorted(context.exported_vars)
```

jinja2.evalcontextfunction(_f)¶

This decorator can be used to mark a function or method as an evalcontext callable. This is similar to the `contextfunction()` but instead of passing the context, an evaluation context object ispassed. For more information about the eval context, seeEvaluation Context.

New in version 2.4.

jinja2.escape(s)¶

Convert the characters `&`, `<`, `>`, `'`, and `"` in string *s_to HTML-safe sequences. Use this if you need to display text that mightcontain such characters in HTML. This function will not escaped objectsthat do have an HTML representation such as already escaped data.*

The return value is a `Markup` string.

jinja2.clear_caches()¶

Jinja2 keeps internal caches for environments and lexers. These areused so that Jinja2 doesn't have to recreate environments and lexers allthe time. Normally you don't have to care about that but if you aremeasuring memory consumption you may want to clean the caches.

`jinja2.` `is_undefined` *(_obj)*¶

Check if the object passed is undefined. This does nothing more thanperforming an instance check against `Undefined` but looks nicer.This can be used for custom filters or tests that want to react toundefined variables. For example a custom default filter can look likethis:

```
def default(var, default=''):
    if isundefined(var):
        return default
    return var
```

_class `jinja2.` `Markup` *([string])*¶

Marks a string as being safe for inclusion in HTML/XML output withoutneeding to be escaped. This implements the ***html*** interface a coupleof frameworks and web applications use. `Markup` is a directsubclass of *unicode* and provides all the methods of *unicode* just thatit escapes arguments passed and always returns *Markup*.

The *escape* function returns markup objects so that double escaping can’thappen.

The constructor of the `Markup` class can be used for threedifferent things: When passed an unicode object it’s assumed to be safe,when passed an object with an HTML representation (has an ***html***method) that representation is used, otherwise the object passed isconverted into a unicode string and then assumed to be safe:

```
>>> Markup("Hello <em>World</em>!")
Markup(u'Hello <em>World</em>!')
>>> class Foo(object):
…  def html(self):
…   return '<a href="#">foo</a>'
…
>>> Markup(Foo())
Markup(u'<a href="#">foo</a>')
```

If you want object passed being always treated as unsafe you can use the`escape()` classmethod to create a `Markup` object:

```
>>> Markup.escape("Hello <em>World</em>!")
Markup(u'Hello &lt;em&gt;World&lt;/em&gt;!')
```

Operations on a markup string are markup aware which means that allarguments are passed through the `escape()` function:

```
>>> em = Markup("<em>%s</em>")
>>> em % "foo & bar"
Markup(u'<em>foo &amp; bar</em>')
>>> strong = Markup("<strong>%(text)s</strong>")
>>> strong % {'text': '<blink>hacker here</blink>'}
Markup(u'<strong>&lt;blink&gt;hacker here&lt;/blink&gt;</strong>')
>>> Markup("<em>Hello</em> ") + "<foo>"
Markup(u'<em>Hello</em> &lt;foo&gt;')
```

classmethod `escape` ***(s)¶***

Escape the string. Works like `escape()` with the differencethat for subclasses of `Markup` this function would return thecorrect subclass.

`striptags` ***()¶***

Unescape markup into an text_type string and strip all tags. Thisalso resolves known HTML4 and XHTML entities. Whitespace isnormalized to one:

```
>>> Markup("Main &raquo;  <em>About</em>").striptags()
u'Main \xbb About'
```

`unescape` ***()¶***

Unescape markup again into an text_type string. This also resolvesknown HTML4 and XHTML entities:

```
>>> Markup("Main &raquo; <em>About</em>").unescape()
u'Main \xbb <em>About</em>'
```

Note

The Jinja2 `Markup` class is compatible with at least Pylons andGenshi. It’s expected that more template engines and framework will pickup the ***html*** concept soon.

Exceptions¶

exception `jinja2.` `TemplateError` ***(message=None)¶***

Baseclass for all template errors.

exception `jinja2.` `UndefinedError` ***(message=None)¶***

Raised if a template tries to operate on `Undefined`.

exception `jinja2.` `TemplateNotFound` ***(name, message=None)¶***

Raised if a template does not exist.

exception `jinja2.` `TemplatesNotFound` ***(names=(), message=None)¶***

Like `TemplateNotFound` but raised if multiple templatesare selected. This is a subclass of `TemplateNotFound` exception, so just catching the base exception will catch both.

New in version 2.2.

exception `jinja2.` `TemplateSyntaxError` ***(message, lineno, name=None, filename=None)¶***

Raised to tell the user that there is a problem with the template.

`message` ***¶***

The error message as utf-8 bytestring.

`lineno` ¶

The line number where the error occurred

`name` ¶

The load name for the template as unicode string.

`filename` ¶

The filename that loaded the template as bytestring in the encodingof the file system (most likely utf-8 or mbcs on Windows systems).

The reason why the filename and error message are bytestrings and notunicode strings is that Python 2.x is not using unicode for exceptionsand tracebacks as well as the compiler. This will change with Python 3.

exception `jinja2.` `TemplateRuntimeError` ***(message=None)*** ¶

A generic runtime error in the template engine. Under some situationsJinja may raise this exception.

exception `jinja2.` `TemplateAssertionError` ***(message, lineno, name=None, filename=None)*** ¶

Like a template syntax error, but covers cases where something in thetemplate caused an error at compile time that wasn't necessarily causedby a syntax error. However it's a direct subclass of `TemplateSyntaxError` and has the same attributes.

Custom Filters¶

Custom filters are just regular Python functions that take the left side ofthe filter as first argument and the arguments passed to the filter asextra arguments or keyword arguments.

For example in the filter `{{ 42|myfilter(23) }}` the function would becalled with `myfilter(42, 23)` . Here for example a simple filter that canbe applied to datetime objects to format them:

```
1. def datetimeformat(value, format='%H:%M / %d-%m-%Y'):
2.     return value.strftime(format)
```

You can register it on the template environment by updating the `filters` dict on the environment:

```
1. environment.filters['datetimeformat'] = datetimeformat
```

Inside the template it can then be used as follows:

```
written on: {{ article.pub_date|datetimeformat }}
publication date: {{ article.pub_date|datetimeformat('%d-%m-%Y') }}
```

Filters can also be passed the current template context or environment. Thisis useful if a filter wants to return an undefined value or check the current `autoescape` setting. For this purpose three decoratorsexist: `environmentfilter()` , `contextfilter()` and `evalcontextfilter()` .

Here a small example filter that breaks a text into HTML line breaks andparagraphs and marks the return value as safe HTML string if autoescaping isenabled:

```
import re
from jinja2 import evalcontextfilter, Markup, escape

_paragraph_re = re.compile(r'(?:\r\n|\r|\n){2,}')

@evalcontextfilter
def nl2br(eval_ctx, value):
    result = u'\n\n'.join(u'<p>%s</p>' % p.replace('\n', Markup('<br>\n'))
                          for p in _paragraph_re.split(escape(value)))
    if eval_ctx.autoescape:
        result = Markup(result)
    return result
```

Context filters work the same just that the first argument is the currentactive `Context` rather than the environment.

Evaluation Context¶

The evaluation context (short eval context or eval ctx) is a new objectintroduced in Jinja 2.4 that makes it possible to activate and deactivatecompiled features at runtime.

Currently it is only used to enable and disable the automatic escaping butcan be used for extensions as well.

In previous Jinja versions filters and functions were marked asenvironment callables in order to check for the autoescape status from theenvironment. In new versions it’s encouraged to check the setting from theevaluation context instead.

Previous versions:

```
@environmentfilter
def filter(env, value):
    result = do_something(value)
    if env.autoescape:
        result = Markup(result)
    return result
```

In new versions you can either use a `contextfilter()` and access theevaluation context from the actual context, or use a `evalcontextfilter()` which directly passes the evaluation context tothe function:

```
1. @contextfilter
   def filter(context, value):
       result = do_something(value)
       if context.eval_ctx.autoescape:
           result = Markup(result)
       return result

2. @evalcontextfilter
   def filter(eval_ctx, value):
       result = do_something(value)
       if eval_ctx.autoescape:
           result = Markup(result)
       return result
```

The evaluation context must not be modified at runtime. Modificationsmust only happen with a `nodes.EvalContextModifier` and `nodes.ScopedEvalContextModifier` from an extension, not on theeval context object itself.

class `jinja2.nodes.` `EvalContext` ***(environment, template_name=None)***¶

Holds evaluation time information. Custom attributes can be attachedto it in extensions.

`autoescape`¶

True or *False* depending on if autoescaping is active or not.

`volatile`¶

True if the compiler cannot evaluate some expressions at compiletime. At runtime this should always be *False*.

Custom Tests¶

Tests work like filters just that there is no way for a test to get accessto the environment or context and that they can't be chained. The returnvalue of a test should be *True* or *False*. The purpose of a test is togive the template designers the possibility to perform type and conformabilitychecks.

Here a simple test that checks if a variable is a prime number:

```
1. import math
2.
3. def is_prime(n):
4.     if n == 2:
```

```
        return True
    for i in xrange(2, int(math.ceil(math.sqrt(n))) + 1):
        if n % i == 0:
            return False
    return True
```

You can register it on the template environment by updating the `tests` dict on the environment:

```
environment.tests['prime'] = is_prime
```

A template designer can then use the test like this:

```
{% if 42 is prime %}
    42 is a prime number
{% else %}
    42 is not a prime number
{% endif %}
```

The Global Namespace¶

Variables stored in the `Environment.globals` dict are special as theyare available for imported templates too, even if they are imported withoutcontext. This is the place where you can put variables and functionsthat should be available all the time. Additionally `Template.globals` exist that are variables available to a specific template that are availableto all `render()` calls.

Low Level API¶

The low level API exposes functionality that can be useful to understand someimplementation details, debugging purposes or advanced extension techniques. Unless you know exactly what you are doing wedon't recommend using any of those.

Environment. ***lex*** ***(source, name=None, filename=None)¶***

Lex the given sourcecode and return a generator that yieldstokens as tuples in the form `(lineno, tokentype, value)`.This can be useful for extension developmentand debugging templates.

This does not perform preprocessing. If you want the preprocessingof the extensions to be applied you have to filter source throughthe `preprocess()` method.

Environment. ***parse*** ***(_source, name=None, filename=None)¶***

Parse the sourcecode and return the abstract syntax tree. Thistree of nodes is used by the compiler to convert the template intoexecutable source- or bytecode.

This is useful for debugging or toextract information from templates.

If you are developing Jinja2 extensionsthis gives you a good overview of the node tree generated.

`Environment.` `preprocess` ***(source, name=None, filename=None)¶***

Preprocesses the source with all extensions. This is automaticallycalled for all parsing and compiling methods but *not* for `lex()` because there you usually only want the actual source tokenized.

`Template.` `newcontext` ***(_vars=None, shared=False, locals=None)¶***

Create a new `Context` for this template. The varsprovided will be passed to the template. Per default the globalsare added to the context. If shared is set to *True* the datais passed as it to the context without adding the globals.

locals can be a dict of local variables for internal usage.

`Template.` `rootrender_func` ***(_context)¶***

This is the low level render function. It's passed a `Context` that has to be created by `new_context()` of the same template ora compatible template. This render function is generated by thecompiler from the template code and returns a generator that yieldsunicode strings.

If an exception in the template code happens the template engine willnot rewrite the exception but pass through the original one. As amatter of fact this function should only be called from within a `render()` / `generate()` / `stream()` call.

`Template.` `blocks` ***¶***

A dict of block render functions. Each of these functions works exactlylike the `root_render_func()` with the same limitations.

`Template.` `isup_to_date` ***¶***

This attribute is _False if there is a newer version of the templateavailable, otherwise *True*.

Note

The low-level API is fragile. Future Jinja2 versions will try not tochange it in a backwards incompatible way but modifications in the Jinja2core may shine through. For example if Jinja2 introduces a new AST nodein later versions that may be returned by `parse()` .

The Meta API¶

New in version 2.2.

The meta API returns some information about abstract syntax trees thatcould help applications to implement more advanced template concepts. Allthe functions of the meta API operate on an abstract syntax tree asreturned by the `Environment.parse()` method.

jinja2.meta. *findundeclared_variables*(*_ast*)¶

Returns a set of all variables in the AST that will be looked up fromthe context at runtime. Because at compile time it's not known whichvariables will be used depending on the path the execution takes atruntime, all variables are returned.

```
>>> from jinja2 import Environment, meta
>>> env = Environment()
>>> ast = env.parse('{% set foo = 42 %}{{ bar + foo }}')
>>> meta.findundeclared_variables(ast) == set(['bar'])
True
```

Implementation

Internally the code generator is used for finding undeclared variables.This is good to know because the code generator might raise a `TemplateAssertionError` *during compilation and as a matter offact this function can currently raise that exception as well.*

jinja2.meta. *find_referenced_templates*(*_ast*)¶

Finds all the referenced templates from the AST. This will return aniterator over all the hardcoded template extensions, inclusions andimports. If dynamic inheritance or inclusion is used, *None* will beyielded.

```
>>> from jinja2 import Environment, meta
>>> env = Environment()
>>> ast = env.parse('{% extends "layout.html" %}{% include helper %}')
>>> list(meta.find_referenced_templates(ast))
['layout.html', None]
```

> This function is useful for dependency tracking. For example if you wantto rebuild parts of the website after a layout template has changed.

原文：

> *http://jinja.pocoo.org/docs/2.10/api/*

Sandbox

Sandbox¶

The Jinja2 sandbox can be used to evaluate untrusted code. Access to unsafeattributes and methods is prohibited.

Assuming *env* is a `SandboxedEnvironment` in the default configurationthe following piece of code shows how it works:

```
>>> env.from_string("{{ func.func_code }}").render(func=lambda:None)
u''
>>> env.from_string("{{ func.func_code.do_something }}").render(func=lambda:None)
Traceback (most recent call last):
  ...
SecurityError: access to attribute 'func_code' of 'function' object is unsafe.
```

API¶

class `jinja2.sandbox.` `SandboxedEnvironment` ***([options])¶***

The sandboxed environment. It works like the regular environment buttells the compiler to generate sandboxed code. Additionally subclasses ofthis environment may override the methods that tell the runtime whatattributes or functions are safe to access.

If the template tries to access insecure code a `SecurityError` israised. However also other exceptions may occur during the rendering sothe caller has to ensure that all exceptions are caught.

`callbinop` ***(_context, operator, left, right)¶***

For intercepted binary operator calls (`intercepted_binops()`)this function is executed instead of the builtin operator. This canbe used to fine tune the behavior of certain operators.

New in version 2.6.

`callunop` ***(_context, operator, arg)¶***

For intercepted unary operator calls (`intercepted_unops()`)this function is executed instead of the builtin operator. This canbe used to fine tune the

behavior of certain operators.

New in version 2.6.

`defaultbinop_table` **= {'%': <built-in function mod>, '': <built-in function mul>, '**': <built-in function pow>, '+': <built-in function add>, '-': <built-in function sub>, '/': <built-in function truediv>, '//': <built-in function floordiv>}¶**

default callback table for the binary operators. A copy of this isavailable on each instance of a sandboxed environment as `binop_table`

`default_unop_table` **= {'+': <built-in function pos>, '-': <built-in function neg>}¶**

default callback table for the unary operators. A copy of this isavailable on each instance of a sandboxed environment as `unop_table`

`intercepted_binops` **= frozenset([])¶**

a set of binary operators that should be intercepted. Each operatorthat is added to this set (empty by default) is delegated to the `call_binop()` *method that will perform the operator. The defaultoperator callback is specified by* `binop_table`*.*

The following binary operators are interceptable: `//`, `%`, `+`, ``, `-`, `/`, and `*`

The default operation form the operator table corresponds to thebuiltin function. Intercepted calls are always slower than the nativeoperator call, so make sure only to intercept the ones you areinterested in.

New in version 2.6.

`interceptedunops` **= frozenset([])¶**

a set of unary operators that should be intercepted. Each operatorthat is added to this set (empty by default) is delegated to the `call_unop()` *method that will perform the operator. The defaultoperator callback is specified by* `unop_table`*.*

The following unary operators are interceptable: `+`, `-`

The default operation form the operator table corresponds to thebuiltin function. Intercepted calls are always slower than the nativeoperator call, so make sure only to intercept the ones you areinterested in.

New in version 2.6.

`is_safe_attribute`**(_obj, attr, value)¶**

The sandboxed environment will call this method to check if theattribute of an object is safe to access. Per default all attributesstarting with an underscore are considered private as well as thespecial attributes of internal python objects as returned by the `is_internal_attribute()` *function.*

`issafe_callable`**(_obj)¶**

Check if an object is safely callable. Per default a function isconsidered safe unless the unsafe_callable attribute exists and isTrue. Override this method to alter the behavior, but this won't affect the unsafe decorator from this module.

class `jinja2.sandbox.` `ImmutableSandboxedEnvironment`**([options])¶**

Works exactly like the regular SandboxedEnvironment but does notpermit modifications on the builtin mutable objects list, set, anddict by using the `modifies_known_mutable()` *function.*

exception `jinja2.sandbox.` `SecurityError`**(message=None)¶**

Raised if a template tries to do something insecure if thesandbox is enabled.

`jinja2.sandbox.` `unsafe`**(f)¶**

Marks a function or method as unsafe.

```
@unsafe
def delete(self):
    pass
```

`jinja2.sandbox.` `isinternal_attribute`**(_obj, attr)¶**

Test if the attribute given is an internal python attribute. Forexample this function returns True for the func_code attribute ofpython objects. This is useful if the environment method `is_safe_attribute()` *is overridden.*

```
>>> from jinja2.sandbox import isinternal_attribute
>>> is_internal_attribute(str, "mro")
True
>>> is_internal_attribute(str, "upper")
False
```

jinja2.sandbox.modifies_known_mutable(_obj, attr)¶

This function checks if an attribute on a builtin mutable object(list, dict, set or deque) would modify it if called. It also supportsthe "user"-versions of the objects (sets.Set, _UserDict. etc.) andwith Python 2.6 onwards the abstract base classes _MutableSet,MutableMapping, and *MutableSequence*.

```
>>> modifiesknown_mutable({}, "clear")
True
>>> modifies_known_mutable({}, "keys")
False
>>> modifies_known_mutable([], "append")
True
>>> modifies_known_mutable([], "index")
False
```

If called with an unsupported object (such as unicode) _False isreturned.

```
>>> modifies_known_mutable("foo", "upper")
False
```

Note

The Jinja2 sandbox alone is no solution for perfect security. Especiallyfor web applications you have to keep in mind that users may createtemplates with arbitrary HTML in so it's crucial to ensure that (if youare running multiple users on the same

server) they can’t harm each othervia JavaScript insertions and much more.

Also the sandbox is only as good as the configuration. We stronglyrecommend only passing non-shared resources to the template and usesome sort of whitelisting for attributes.

Also keep in mind that templates may raise runtime or compile time errors,so make sure to catch them.

Operator Intercepting¶

New in version 2.6.

For maximum performance Jinja2 will let operators call directly the typespecific callback methods. This means that it’s not possible to have thisintercepted by overriding `Environment.call()` . Furthermore aconversion from operator to special method is not always directly possibledue to how operators work. For instance for divisions more than onespecial method exist.

With Jinja 2.6 there is now support for explicit operator intercepting.This can be used to customize specific operators as necessary. In orderto intercept an operator one has to override the `SandboxedEnvironment.intercepted_binops` attribute. Once theoperator that needs to be intercepted is added to that set Jinja2 willgenerate bytecode that calls the `SandboxedEnvironment.call_binop()` function. For unary operators the *unary* attributes and methods have tobe used instead.

The default implementation of `SandboxedEnvironment.call_binop` will use the `SandboxedEnvironment.binop_table` to translateoperator symbols into callbacks performing the default operator behavior.

This example shows how the power (`**`) operator can be disabled inJinja2:

```
from jinja2.sandbox import SandboxedEnvironment


class MyEnvironment(SandboxedEnvironment):
    intercepted_binops = frozenset(['**'])

    def call_binop(self, context, operator, left, right):
        if operator == '**':
            return self.undefined('the power operator is unavailable')
        return SandboxedEnvironment.call_binop(self, context,
                                               operator, left, right)
```

Make sure to always call into the super method, even if you are notintercepting the call. Jinja2 might internally call the method toevaluate expressions.

原文:

> *http://jinja.pocoo.org/docs/2.10/sandbox/*

Native Python Types

Native Python Types¶

The default `Environment` renders templates to strings. With `NativeEnvironment`, rendering a template produces a native Python type.This is useful if you are using Jinja outside the context of creating textfiles. For example, your code may have an intermediate step where users may usetemplates to define values that will then be passed to a traditional stringenvironment.

Examples¶

Adding two values results in an integer, not a string with a number:

```
>>> env = NativeEnvironment()
>>> t = env.from_string('{{ x + y }}')
>>> result = t.render(x=4, y=2)
>>> print(result)
6
>>> print(type(result))
int
```

Rendering list syntax produces a list:

```
>>> t = env.from_string('[{% for item in data %}{{ item + 1 }},{% endfor %}]')
>>> result = t.render(data=range(5))
>>> print(result)
[1, 2, 3, 4, 5]
>>> print(type(result))
list
```

Rendering something that doesn’t look like a Python literal produces a string:

```
>>> t = env.from_string('{{ x }} * {{ y }}')
>>> result = t.render(x=4, y=2)
>>> print(result)
4 * 2
>>> print(type(result))
str
```

Rendering a Python object produces that object as long as it is the only node:

```
>>> class Foo:
...     def __init__(self, value):
...         self.value = value
```

```
...
>>> result = env.from_string('{{ x }}').render(x=Foo(15))
>>> print(type(result).__name__)
Foo
>>> print(result.value)
15
```

API¶

class `jinja2.nativetypes.` `NativeEnvironment` ***([options])¶***

An environment that renders templates to native Python types.

class `jinja2.nativetypes.` `NativeTemplate` ***([options])¶***

`render` ***(*args, **kwargs)¶***

Render the template to produce a native Python type. If the resultis a single node, its value is returned. Otherwise, the nodes areconcatenated as strings. If the result can be parsed with `ast.literal_eval()`, the parsed value is returned. Otherwise, thestring is returned.

原文:

> *http://jinja.pocoo.org/docs/2.10/nativetypes/*

Template Designer Documentation

Template Designer Documentation¶

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

Synopsis¶

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

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

Below is a minimal template that illustrates a few basics using the defaultJinja 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 applicationdeveloper can change the syntax configuration from `{% foo %}` to `<% foo %>` , or something similar.

There are a few kinds of delimiters. The default Jinja delimiters areconfigured 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 thetemplate.

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

You can use a dot (`.`) to access attributes of a variable in additionto the standard Python `getitem` “subscript” syntax (`[]`).

The following lines do the same thing:

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

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

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

Implementation

For the sake of convenience, `foo.bar` in Jinja2 does the followingthings 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 samename. Additionally, the `attr()` filter only looks up attributes.

Filters¶

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

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

Filters that accept arguments have parentheses around the arguments, just likea function call. For example: `{{ listx|join(', ') }}` will join a list withcommas (`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 usedto test a variable against a common expression. To test a variable orexpression, you add *is* plus the name of the test after the variable. Forexample, 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 definedin the current template context.

Tests can accept arguments, too. If the test only takes one argument, you canleave out the parentheses. For example, the following twoexpressions 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 isby default set to `{# … #}` . This is useful to comment out parts of thetemplate for debugging or to add information for other template designers oryourself:

```
{# 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 atemplate tag is removed automatically (like in PHP). The _lstrip_blocks_option can also be set to strip tabs and spaces from the beginning of aline to the start of a block. (Nothing will be stripped if there areother characters before the start of the block.)

With both *trim_blocks* and *lstrip_blocks* enabled, you can put block tagson their own lines, and the entire block line will be removed whenrendered, 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 blocklines are removed and other whitespace is preserved:

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

You can manually disable the *lstrip_blocks* behavior by putting aplus 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 minussign (`-`) to the start or end of a block (e.g. a `For` tag), acomment, or a variable expression, the

whitespaces before or afterthat block will be removed:

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

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

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

By default, Jinja2 also removes trailing newlines. To keep singletrailing 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 partsit would otherwise handle as variables or blocks. For example, if, withthe default syntax, you want to use `{{` as a raw string in a template andnot start a variable, you have to use a trick.

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

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

For bigger sections, it makes sense to mark a block *raw*. For example, toinclude 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 aline as a statement. For example, if the line statement prefix is configuredto `#`, 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 textprecedes 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, ifthe line-comment prefix is configured to be `##`, everything from `##` tothe 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 inheritanceallows you to build a base “skeleton” template that contains all the commonelements of your site and defines **blocks** that child templates can override.

Sounds complicated but is very basic. It’s easiest to understand it by startingwith an example.

Base Template¶

This template, which we’ll call `base.html` , defines a simple HTML skeletondocument 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 templatescan fill in. All the *block* tag does is tell the template engine that achild 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 thatthis template “extends” another template. When the template system evaluatesthis template, it first locates the parent. The extends tag should be thefirst tag in the template. Everything before it is printed out normally andmay cause confusion. For details about this behavior and how to takeadvantage 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 thefilename. You can access templates in subdirectories with a slash:

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

But this behavior can depend on the application embedding Jinja. Note thatsince the child template doesn’t define the `footer` block, the value fromthe parent template is used instead.

You can’t define multiple `{% block %}` tags with the same name in thesame 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 betterreadability:

```
{% 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 blocksmay 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 unavailableinside the block. The reason for this is that if the block is replaced bya child template, a variable would appear that was not defined in the block orpassed to the context.

Starting with Jinja 2.2, you can explicitly specify that variables areavailable 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 canextend from that object as well. Assuming the calling code passesa layout template as *layout_template* to the environment, thiscode works:

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

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

HTML Escaping¶

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

- manually escaping each variable; or
- 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 isescaping 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 couldhappen that by coercing safe and unsafe values, the return value isdouble-escaped HTML.

Working with Manual Escaping¶

If manual escaping is enabled, it's **your** responsibility to escapevariables 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 trustedHTML. Escaping works by piping the variable through the `|e` filter:

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

Working with Automatic Escaping¶

When automatic escaping is enabled, everything is escaped by default exceptfor values explicitly marked as safe. Variables and expressionscan be marked as safe either in:

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

The main problem with this approach is that Python itself doesn't have theconcept 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 thatyou could end up with double-escaped contents. Double-escaping is easy toavoid, however: just rely on the tools Jinja2 provides and *don't use builtinPython constructs such as str.format or the string modulo operator (%)*.

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

String literals in templates with automatic escaping are considered unsafebecause 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 aprogram - conditionals (i.e. if/elif/else), for-loops, as well as things likemacros 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 usersprovided in a variable called *users*:

```
1. <h1>Members</h1>
2. <ul>
3. {% for user in users %}
4.   <li>{{ user.username|e }}</li>
5. {% endfor %}
6. </ul>
```

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

```
1. <dl>
2. {% for key, value in my_dict.iteritems() %}
3.     <dt>{{ key|e }}</dt>
4.     <dd>{{ value|e }}</dd>
5. {% endfor %}
6. </dl>
```

Note, however, that **Python dicts are not ordered**; so you might want toeither 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.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 ofsequences. See the explanation below.
loop.depth	Indicates how deep in a recursive loopthe rendering currently is. Starts at level 1
loop.depth0	Indicates how deep in a recursive loopthe 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/variableseach 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-unboundcycling. 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. Youcan, however, filter the sequence during iteration, which allows you to skipitems. 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; thusnot counting the users not iterated over.

If no iteration took place because the sequence was empty or the filteringremoved all the items from the sequence, you can render a default blockby 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 correspondingloop **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 aredealing with recursive data such as sitemaps or RDFa.To use loops recursively, you basically have to add the *recursive* modifierto the loop definition and call the *loop* variable with the new iterablewhere 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 wehave more than one level of loops, we can rebind the variable *loop* bywriting *{% set outer_loop = loop %}* after the loop that we want touse recursively. Then, we can call it using *{{ outer_loop(…) }}*

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

If all you want to do is check whether some value has changed since thelast 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 eveneasier:

```
{% 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, notempty 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 canuse 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 forloop filtering.

Macros¶

Macros are comparable with functions in regular programming languages. Theyare useful to put often used idioms into reusable functions to not repeatyourself (“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 toimport it first.

Inside macros, you have access to three special variables:

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

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

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

Macros also expose some of their internal details. The following attributesare 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.: accessesthe 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 maybe called from a call tag.

If a macro name starts with an underscore, it’s not exported and can’tbe imported.

Call¶

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

```
{% macro renderdialog(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 ituseful as a replacement for loops. Generally speaking, a call block worksexactly 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 oftemplate 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 attop level (outside of blocks, macros or loops) are exported from the templatelike 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 = callsomething() %}
```

Scoping Behavior

Please keep in mind that it is not possible to set variables inside ablock and have them show up outside of it. This also applies toloops. The only exception to that rule are if statements which do notintroduce a scope. As a result the following template is not goingto 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 usealternative 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 namespaceobjects 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 fornamespace objects; attempting to assign an attribute on any other objectwill 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 tocapture the contents of a block into a variable name. This can be usefulin some situations as an alternative for macros. In that case, instead ofusing an equals sign and a value, you just write the variable name and theneverything 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 canhave multiple *extends* tags in a file, but only one of them may be executed ata time.

See the section about Template Inheritance above.

Blocks¶

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

Include¶

The *include* statement is useful to include a template and return therendered 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 bydefault. 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`; inwhich case Jinja will ignore the statement if the template to be includeddoes not exist. When combined with `with` or `without context`, it mustbe placed *before* the context visibility statement. Here are some validexamples:

```
{% 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 existencebefore inclusion. The first template that exists will be included. If*ignore missing* is given, it will fall back to rendering nothing ifnone of the templates exist, otherwise it will raise an exception.

Example:

```
{% include ['pagedetailed.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 caninclude that object using _include.

Import¶

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

There are two ways to import templates. You can import a complete templateinto 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 variablesand macros is to import the whole template module into a variable.That way, you can access the attributes:

```
1. {% 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 currentnamespace:

```
1. {% from 'forms.html' import input as inputfield, 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 andcannot be imported.

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

Import Context Behavior¶

By default, included templates are passed the current context and importedtemplates 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 contextcan 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 templatedid not include variables defined in the template. As a matter offact, this did not work:

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

The included template `render_box.html` *is _not* able to access*box* in Jinja 2.0. As of Jinja

2.1, `renderbox.html` *is* ableto do so.

Expressions¶

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

Literals¶

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

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

42 / 42.23:
Integers and floating point numbers are created by just writing thenumber down. If a dot is present, the number is a float, otherwise aninteger. 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 storingsequential data to be iterated over. For example, you can easilycreate a list of links using lists and tuples for (and with) a for loop:

```
1. <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 tupleonly 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 mustbe unique and always have exactly one value. Dicts are rarely used intemplates; 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 expandto 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 templatesbut exists for completeness' sake. The following operators are supported:

+
Adds two objects together. Usually the objects are numbers, but if both arestrings or lists, you can concatenate them this way. This, however, is notthe preferred way to concatenate strings! For string concatenation, havea 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 }}` wouldreturn `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 tocombine 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 othertwo categories:

in
Perform a sequence / mapping containment test. Returns true if the leftoperand is contained in the right. `{{ 1 in [1, 2, 3] }}` would, forexample, return true.

is
Performs a test.

|
Applies a filter.

~

Converts all operands into strings and concatenates them.

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

()

Call a callable: `{{ post.render() }}`. Inside of the parentheses youcan 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 somesituations. For example, you can use this to extend from one template if avariable is defined, otherwise from the default layout template:

```
1. {% 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 implicitlyevaluates into an undefined object:

```
1. .. 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 notlooked 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 thegiven number of items. If you provide a second parameter thisis 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 otherslowercase.

`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:

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

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

```
1. {{ ''|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 areunsorted you may want to use this function to order them by eitherkey or value:

```
1. {% 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-safesequences. Use this if you need to display text that might containsuch 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 binaryprefixes 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 theconversion doesn’t work it will return `0.0`. You canoverride 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 personswith *gender*, *first_name* and *last_name* attributes and you want togroup all users by genders you can do something like the followingsnippet:

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

Additionally it’s possible to use tuple unpacking for the grouper andlist:

```
<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 grouperin common.

Changed in version 2.6: It’s now possible to use dotted notation to group by the childattribute 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. Thefirst 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 theconversion doesn't work it will return `0`. You canoverride this default using the first parameter. Youcan also override the default base (10) in the secondparameter, which handles input with prefixes such as0b, 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 thesequence. The separator between elements is an empty string perdefault, 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 listwill 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 reallyonly interested in a certain value of it.

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

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

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

```
1. 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 parameteris 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 specifiedattribute of each object, and rejecting the objects with the testsucceeding.

If no test is specified, the attribute’s value will be evaluated asa boolean.

```
{{ users|rejectattr("isactive") }}
{{ 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 substringreplaced with a new one. The first argument is the substringthat 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 otherway round.

`round` *(value, precision=0, method='common')*¶

Round the number to a given precision. The firstparameter specifies the precision (default is `0`), thesecond 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. Ifyou 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 automaticescaping 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 specifiedattribute of each object, and only selecting the objects with thetest succeeding.

If no test is specified, the attribute’s value will be evaluated asa boolean.

Example usage:

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

New in version 2.7.

`slice` *(_value, slices, fill_with=None)*¶

Slice an iterator and return a list of lists containingthose items. Useful if you want to create a div containingthree 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 missingvalues on the last iteration.

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

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

If the iterable is made of strings the third parameter can be used tocontrol the case sensitiveness of the comparison which is disabled bydefault.

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

It is also possible to sort by an attribute (for example to sortby 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 markupstring 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 returnsstart.

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 overattributes. Also the *start* parameter was moved on to the right.

`title` *(s)*¶

Return a titlecased version of the value. I.e. words will start withuppercase 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 thatthis is available in templates through the `|tojson` filter which willalso mark the result as safe. Due to how this function escapes certaincharacters this is safe even if used outside of `<script>` tags.

The following characters are escaped in strings:

- <
- >
- &
- '

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

The indent parameter can be used to enable pretty printing. Set it tothe 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 specifiedwith the first parameter which defaults to `255`. If the secondparameter is `true` the filter will cut the text at length. Otherwiseit will discard the last word. If the text was in facttruncated it will append an ellipsis sign (`"…"`). If you want adifferent ellipsis sign than `"…"` you can specify it using thethird parameter. Strings that only exceed the length by the tolerancemargin given in the fourth parameter will not be truncated.

```
1. {{ "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 butcan be reconfigured globally.

`unique`*(value, case_sensitive=False, attribute=None)*¶

Returns a list of unique items from the the given iterable.

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

The unique items are yielded in the same order as their first occurrence inthe 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 bothdictionaries 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 urlsto that number. Also a third argument exists that makes the urls“nofollow”:

```
1. {{ 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:

```
1. {{ 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 firstparameter. If you set the second parameter to *false* Jinja will notsplit words apart if they are longer than *width*. By default, the newlineswill be the default newlines for the environment, but this can be changedusing 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 automaticallyescaped:

```
<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 itemif 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 undefinedvariables.

`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 anotherobject:

```
{% 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 variablesthat 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 torender 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 paragraphsof HTML are generated with each paragraph between 20 and 100 words.If html is False, regular text is returned. This is useful to generate simplecontents for layout testing.

`dict` *(**items)*¶

A convenient alternative to dict literals. `{'foo': 'bar'}` is the sameas `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 ofloops or over multiple loops.

This can be very useful if you want to show a list of folders andfiles with the folders on top but both in the same list with alternatingrow 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 ispassed a string and will return that string every time it’s called, exceptthe first time (in which case it returns an empty string). You canuse 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 loopbody to an outer scope. Initial values can be provided as a dict, askeyword 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 beenabled by an application. An application could also provide furtherextensions not covered by this documentation; in which case there shouldbe a separate document explaining said extensions.

i18n¶

If the i18n extension is enabled, it's possible to mark parts in the templateas 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 justaccessing an attribute of an object — you need to bind the expression to aname 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, separatethe pieces with a comma (`,`):

```
{% trans book_title=book.title, author=author.name %}
```

```
2. This is {{ book_title }} by {{ author }}
3. {% 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*:

```
1. {% trans count=list|length %}
2. There is {{ count }} {{ name }} object.
3. {% pluralize %}
4. There are {{ count }} {{ name }} objects.
5. {% endtrans %}
```

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

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

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

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

If trimming is enabled globally, the *notrimmed* modifier can be used todisable 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:

```
1. {{ _('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), usingplaceholders 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 receivesthe 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 availablethat works exactly like the regular variable expression (`{{ … }}`); exceptit 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 touse *break* and *continue* in loops. When *break* is reached, the loop isterminated; if *continue* is reached, the processing is stopped and continueswith 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 examplesare equivalent:

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

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

An important note on scoping here. In Jinja versions before 2.9 thebehavior of referencing one variable to another had some unintendedconsequences. In particular one variable could refer to another definedin the same with block's opening statement. This caused issues with thecleaned up scoping behavior and has since been improved. In particularin 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 ofthe first attribute. If you depend on this behavior you can rewrite it touse the `set` tag:

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

Extension

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

原文:

> *http://jinja.pocoo.org/docs/2.10/templates/*

Extensions

Extensions¶

Jinja2 supports extensions that can add extra filters, tests, globals or evenextend the parser. The main motivation of extensions is to move often usedcode into a reusable class like adding support for internationalization.

Adding Extensions¶

Extensions are added to the Jinja2 environment at creation time. Once theenvironment is created additional extensions cannot be added. To add anextension pass a list of extension classes or import paths to the*extensions* parameter of the `Environment` constructor. The followingexample creates a Jinja2 environment with the i18n extension loaded:

```
1. jinja_env = Environment(extensions=['jinja2.ext.i18n'])
```

i18n Extension¶

Import name:*jinja2.ext.i18n*

The i18n extension can be used in combination with gettext or babel. Ifthe i18n extension is enabled Jinja2 provides a *trans* statement that marksthe wrapped string as translatable and calls *gettext*.

After enabling, dummy *__ function that forwards calls to _gettext* is addedto the environment globals. An internationalized application then has toprovide a *gettext* function and optionally an *ngettext* function into thenamespace, either globally or for each rendering.

Environment Methods¶

After enabling the extension, the environment provides the followingadditional methods:

jinja2.Environment. ***installgettext_translations*** ***(_translations, newstyle=False)¶***

> Installs a translation globally for that environment. The translationsobject provided must implement at least *ugettext* and *ungettext*.The *gettext.NullTranslations* and *gettext.GNUTranslations* classesas well as Babels *Translations* class are supported.

Changed in version 2.5: newstyle gettext added

`jinja2.Environment.` `installnull_translations` *(_newstyle=False)*¶

Install dummy gettext functions. This is useful if you want to preparethe application for internationalization but don't want to implement thefull internationalization system yet.

Changed in version 2.5: newstyle gettext added

`jinja2.Environment.` `installgettext_callables` *(_gettext, ngettext, newstyle=False)*¶
Installs the given *gettext* and *ngettext* callables into theenvironment as globals. They are supposed to behave exactly like thestandard library's `gettext.ugettext()` and `gettext.ungettext()` functions.

If *newstyle* is activated, the callables are wrapped to work likenewstyle callables. See Whitespace Trimming for more information.

New in version 2.5.

`jinja2.Environment.` `uninstallgettext_translations` *()*¶

Uninstall the translations again.

`jinja2.Environment.` `extract_translations` *(_source)*¶

Extract localizable strings from the given template node or source.

For every string found this function yields a `(lineno, function, message)` tuple, where:

- *lineno* is the number of the line on which the string was found,
- *function* is the name of the *gettext* function used (if thestring was extracted from embedded Python code), and
- *message* is the string itself (a *unicode* object, or a tupleof *unicode* objects for functions with multiple string arguments).

If Babel is installed, the babel integrationcan be used to extract strings for babel.

For a web application that is available in multiple languages but gives allthe users the same language (for example a multilingual forum softwareinstalled for a French

community) may load the translations once and add thetranslation methods to the environment at environment generation time:

```
translations = getgettext_translations()
env = Environment(extensions=['jinja2.ext.i18n'])
env.install_gettext_translations(translations)
```

The _get_gettext_translations function would return the translator for thecurrent configuration. (For example by using *gettext.find*)

The usage of the *i18n* extension for template designers is covered as partof the template documentation.

Whitespace Trimming¶

New in version 2.10.

Linebreaks and surrounding whitespace can be automatically trimmed by enablingthe `ext.i18n.trimmed` policy.

Newstyle Gettext¶

New in version 2.5.

Starting with version 2.5 you can use newstyle gettext calls. These areinspired by trac's internal gettext functions and are fully supported bythe babel extraction tool. They might not work as expected by otherextraction tools in case you are not using Babel's.

What's the big difference between standard and newstyle gettext calls? Ingeneral they are less to type and less error prone. Also if they are usedin an autoescaping environment they better support automatic escaping.Here are some common differences

between old and new calls:

standard gettext:

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

newstyle gettext looks like this instead:

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

The advantages of newstyle gettext are that you have less to type and thatnamed placeholders become mandatory. The latter sounds like adisadvantage but solves a lot of troubles translators are often facingwhen they are unable to switch the positions of two placeholder. Withnewstyle gettext, all format strings look the same.

Furthermore with newstyle gettext, string formatting is also used if noplaceholders are used which makes all strings behave exactly the same.Last but not least are newstyle gettext calls able to properly markstrings for autoescaping which solves lots of escaping related issues manytemplates are experiencing over time when using autoescaping.

Expression Statement¶

Import name:*jinja2.ext.do*

The “do” aka expression-statement extension adds a simple *do* tag to thetemplate engine

that works like a variable expression but ignores thereturn value.

Loop Controls¶

Import name:*jinja2.ext.loopcontrols*

This extension adds support for *break* and *continue* in loops. Afterenabling, Jinja2 provides those two keywords which work exactly like inPython.

With Statement¶

Import name:*jinja2.ext.with__*

Changed in version 2.9.

This extension is now built-in and no longer does anything.

Autoescape Extension¶

***Import name:**_jinja2.ext.autoescape*

Changed in version 2.9.

This extension was removed and is now built-in. Enabling the extensionno longer does anything.

Writing Extensions¶

By writing extensions you can add custom tags to Jinja2. This is a non-trivialtask and usually not needed as the default tags and expressions cover allcommon use cases. The i18n extension is a good example of why extensions areuseful. Another one would be fragment caching.

When writing extensions you have to keep in mind that you are working with theJinja2

template compiler which does not validate the node tree you are passingto it. If the AST is malformed you will get all kinds of compiler or runtimeerrors that are horrible to debug. Always make sure you are using the nodesyou create correctly. The API documentation below shows which nodes exist andhow to use them.

Example Extension¶

The following example implements a *cache* tag for Jinja2 by using theWerkzeug caching contrib module:

```
from jinja2 import nodes
from jinja2.ext import Extension


class FragmentCacheExtension(Extension):
    # a set of names that trigger the extension.
    tags = set(['cache'])

    def init(self, environment):
        super(FragmentCacheExtension, self).init(environment)

        # add the defaults to the environment
        environment.extend(
            fragmentcache_prefix='',
            fragment_cache=None
        )

    def parse(self, parser):
        # the first token is the token that started the tag.  In our case
        # we only listen to &#39;cache&#39; so this will be a name token with
        # cache as value.  We get the line number so that we can give
        # that line number to the nodes we create by hand.
        lineno = next(parser.stream).lineno

        # now we parse a single expression that is used as cache key.
        args = [parser.parse_expression()]

        # if there is a comma, the user provided a timeout.  If not use
        # None as second parameter.
        if parser.stream.skip_if('comma'):
            args.append(parser.parse_expression())
        else:
            args.append(nodes.Const(None))

        # now we parse the body of the cache block up to endcache and
        # drop the needle (which would always be endcache in that case)
        body = parser.parse_statements(['name:endcache'], drop_needle=True)

        # now return a CallBlock node that calls our _cache_support
        # helper method on this extension.
        return nodes.CallBlock(self.call_method('_cache_support', args),
                               [], [], body).set_lineno(lineno)

    def _cache_support(self, name, timeout, caller):
        """Helper callback."""
        key = self.environment.fragment_cache_prefix + name
```

```
            # try to load the block from the cache
            # if there is no fragment in the cache, render it and store
            # it in the cache.
            rv = self.environment.fragment_cache.get(key)
            if rv is not None:
                return rv
            rv = caller()
            self.environment.fragment_cache.add(key, rv, timeout)
            return rv
```

And here is how you use it in an environment:

```
from jinja2 import Environment
from werkzeug.contrib.cache import SimpleCache

env = Environment(extensions=[FragmentCacheExtension])
env.fragment_cache = SimpleCache()
```

Inside the template it's then possible to mark blocks as cacheable. Thefollowing example caches a sidebar for 300 seconds:

```
{% cache 'sidebar', 300 %}
<div class="sidebar">
    …
</div>
{% endcache %}
```

Extension API¶

Extensions always have to extend the `jinja2.ext.Extension` *class:*

_class `jinja2.ext.` `Extension` ***(environment)¶***

Extensions can be used to add extra functionality to the Jinja templatesystem at the parser level. Custom extensions are bound to an environmentbut may not store environment specific data on *self*. The reason forthis is that an extension can be bound to another environment (foroverlays) by creating a copy and reassigning the *environment* attribute.

As extensions are created by the environment they cannot accept anyarguments for configuration. One may want to work around that by usinga factory function, but that is not possible as extensions are identifiedby their import name. The correct way to configure the extension isstoring the configuration values on the environment. Because this way theenvironment ends up acting as central configuration storage theattributes may clash which is why extensions have to ensure that the namesthey choose for configuration are not too generic. `prefix` for exampleis a terrible name, `fragmentcache_prefix` *on the other hand is a goodname as includes the name of the extension (fragment cache).*

`identifier` ¶

The identifier of the extension. This is always the true import nameof the extension class and must not be changed.

`tags` ¶

If the extension implements custom tags this is a set of tag namesthe extension is listening for.

`attr` ***(_name, lineno=None)*** ¶

Return an attribute node for the current extension. This is usefulto pass constants on extensions to generated template code.

```
1. self.attr('my_attribute', lineno=lineno)
```

`call_method` ***(_name, args=None, kwargs=None, dyn_args=None, dyn_kwargs=None, lineno=None)*** ¶

Call a method of the extension. This is a shortcut for `attr()` + `jinja2.nodes.Call`.

`filterstream` ***(_stream)*** ¶

It’s passed a `TokenStream` that can be usedto filter tokens returned. This method

has to return an iterable of `Token`s, but it doesn't have to return a `TokenStream`.

In the *ext* folder of the Jinja2 source distribution there is a filecalled *inlinegettext.py* which implements a filter that utilizes thismethod.

`parse` ***(parser)*** ¶

If any of the `tags` matched this method is called with theparser as first argument. The token the parser stream is pointing atis the name token that matched. This method has to return one or alist of multiple nodes.

`preprocess` ***(source, name, filename=None)*** ¶

This method is called before the actual lexing and can be used topreprocess the source. The *filename* is optional. The return valuemust be the preprocessed source.

Parser API¶

The parser passed to `Extension.parse()` provides ways to parseexpressions of different types. The following methods may be used byextensions:

class `jinja2.parser.` `Parser` ***(environment, source, name=None, filename=None, state=None)*** ¶

This is the central parsing class Jinja2 uses. It's passed toextensions and can be used to parse expressions or statements.

`filename` ¶

The filename of the template the parser processes. This is **not**the load name of the template. For the load name see `name`.For templates that were not loaded form the file system this is*None*.

`name` ¶

The load name of the template.

`stream` ¶

The current `TokenStream`

`fail` ***(msg, lineno=None, exc=<class 'jinja2.exceptions.TemplateSyntaxError'>)*** ¶

Convenience method that raises *exc* with the message, passedline number or last line number as well as the current name andfilename.

`freeidentifier` ***(_lineno=None)*** ¶

Return a new free identifier as `InternalName`.

`parseassign_target`***(_with_tuple=True, name_only=False, extra_end_rules=None, with_namespace=False)¶***

Parse an assignment target. As Jinja2 allows assignments totuples, this function can parse all allowed assignment targets. Perdefault assignments to tuples are parsed, that can be disable howeverby setting *with_tuple* to *False*. If only assignments to names arewanted *name_only* can be set to *True*. The *extra_end_rules_parameter is forwarded to the tuple parsing function. If_with_namespace* is enabled, a namespace assignment may be parsed.

`parseexpression`***(_with_condexpr=True)¶***

Parse an expression. Per default all expressions are parsed, ifthe optional *with_condexpr* parameter is set to *False* conditionalexpressions are not parsed.

`parsestatements`***(_end_tokens, drop_needle=False)¶***

Parse multiple statements into a list until one of the end tokensis reached. This is used to parse the body of statements as it alsoparses template data if appropriate. The parser checks first if thecurrent token is a colon and skips it if there is one. Then it checksfor the block end and parses until if one of the *end_tokens* isreached. Per default the active token in the stream at the end ofthe call is the matched end token. If this is not wanted *drop_needle_can be set to _True* and the end token is removed.

`parsetuple`***(_simplified=False, with_condexpr=True, extra_end_rules=None, explicit_parentheses=False)¶***

Works like *parse_expression* but if multiple expressions aredelimited by a comma a `Tuple` node is created.This method could also return a regular expression instead of a tupleif no commas where found.

The default parsing mode is a full tuple. If *simplified* is *True_only names and literals are parsed. The _no_condexpr* parameter isforwarded to `parse_expression()`.

Because tuples do not require delimiters and may end in a bogus commaan extra hint is needed that marks the end of a tuple. For examplefor loops support tuples between *for* and *in*. In that case the*extra_end_rules* is set to `['name:in']`.

explicit_parentheses is true if the parsing was triggered by anexpression in parentheses. This is used to figure out if an emptytuple is a valid expression or not.

class `jinja2.lexer.` `TokenStream`***(generator, name, filename)¶***

A token stream is an iterable that yields `Token`s. Theparser however does not iterate over it but calls `next()` to goone token ahead. The current active token is stored as `current`.

`current` ¶

The current `Token`.

`eos` ¶

Are we at the end of the stream?

`expect` ***(expr)*** ¶

Expect a given token type and return it. This accepts the sameargument as `jinja2.lexer.Token.test()`.

`look` ***()*** ¶

Look at the next token.

`nextif` ***(_expr)*** ¶

Perform the token test and return the token if it matched.Otherwise the return value is *None*.

`push` ***(token)*** ¶

Push a token back to the stream.

`skip` ***(n=1)*** ¶

Got n tokens ahead.

`skipif` ***(_expr)*** ¶

Like `next_if()` but only returns *True* or *False*.

class `jinja2.lexer.` `Token` ¶

Token class.

`lineno` ¶

The line number of the token

`type` ¶

The type of the token. This string is interned so you may compareit with

arbitrary strings using the *is* operator.

`value` ¶

The value of the token.

`test` ***(expr)*** ¶

Test a token against a token expression. This can either be atoken type or `'tokentype:token_value'`. *This can only testagainst string values and types.*

`test_any` ***(*iterable)*** ¶

Test against multiple token expressions.

There is also a utility function in the lexer module that can count newlinecharacters in strings:

`jinja2.lexer.` `count_newlines` ***(_value)*** ¶

Count the number of newline characters in the string. This isuseful for extensions that filter a stream.

AST¶

The AST (Abstract Syntax Tree) is used to represent a template after parsing.It's build of nodes that the compiler then converts into executable Pythoncode objects. Extensions that provide custom statements can return nodes toexecute custom Python code.

The list below describes all nodes that are currently available. The AST maychange between Jinja2 versions but will stay backwards compatible.

For more information have a look at the repr of `jinja2.Environment.parse()` .

class `jinja2.nodes.` `Node` ¶

Baseclass for all Jinja2 nodes. There are a number of nodes availableof different types. There are four major types:

- Stmt: statements
- Expr: expressions
- Helper: helper nodes
- Template: the outermost wrapper node

All nodes have fields and attributes. Fields may be other nodes, lists,or arbitrary values. Fields are passed to the constructor as regularpositional arguments, attributes as keyword arguments. Each node hastwo attributes: *lineno*

(the line number of the node) and *environment*.The *environment* attribute is set at the end of the parsing process forall nodes automatically.

`find` ***(node_type)¶***

Find the first node of a given type. If no such node exists thereturn value is *None*.

`findall` ***(_node_type)¶***

Find all the nodes of a given type. If the type is a tuple,the check is performed for any of the tuple items.

`iterchild_nodes` ***(_exclude=None, only=None)¶***

Iterates over all direct child nodes of the node. This iteratesover all fields and yields the values of they are nodes. If the valueof a field is a list all the nodes in that list are returned.

`iterfields` ***(_exclude=None, only=None)¶***

This method iterates over all fields that are defined and yields `(key, value)` tuples. Per default all fields are returned, butit's possible to limit that to some fields by providing the *only_parameter or to exclude some using the _exclude* parameter. Bothshould be sets or tuples of field names.

`setctx` ***(_ctx)¶***

Reset the context of a node and all child nodes. Per default theparser will all generate nodes that have a 'load' context as it's themost common one. This method is used in the parser to set assignmenttargets and other nodes to a store context.

`setenvironment` ***(_environment)¶***

Set the environment for all nodes.

`setlineno` ***(_lineno, override=False)¶***

Set the line numbers of the node and children.

class `jinja2.nodes.` `Expr` ***¶***

Baseclass for all expressions.

|Node type:
|——-
| `Node`

`asconst`*(_eval_ctx=None)*¶

Return the value of the expression as constant or raise `Impossible` if this was not possible.

An `EvalContext` can be provided, if none is givena default context is created which requires the nodes to havean attached environment.

Changed in version 2.4: the *eval_ctx* parameter was added.

`canassign`*()*¶

Check if it's possible to assign something to this node.

_class `jinja2.nodes.` `BinExpr`*(left, right)*¶

Baseclass for all binary expressions.

|Node type:
|——-
| `Expr`

class `jinja2.nodes.` `Add`*(left, right)*¶

Add the left to the right node.

|Node type:
|——-
| `BinExpr`

class `jinja2.nodes.` `And`*(left, right)*¶

Short circuited AND.

|Node type:
|——-
| `BinExpr`

class `jinja2.nodes.` `Div`*(left, right)*¶

Divides the left by the right node.

|Node type:
|——-
| `BinExpr`

class `jinja2.nodes.` `FloorDiv` ***(left, right)¶***

Divides the left by the right node and truncates conver theresult into an integer by truncating.

|Node type:
|——-
| `BinExpr`

class `jinja2.nodes.` `Mod` ***(left, right)¶***

Left modulo right.

|Node type:
|——-
| `BinExpr`

class `jinja2.nodes.` `Mul` ***(left, right)¶***

Multiplies the left with the right node.

|Node type:
|——-
| `BinExpr`

class `jinja2.nodes.` `Or` ***(left, right)¶***

Short circuited OR.

|Node type:
|——-
| `BinExpr`

class `jinja2.nodes.` `Pow` ***(left, right)¶***

Left to the power of right.

|Node type:
|——-

| `BinExpr`

class `jinja2.nodes.` `Sub` ***(left, right)¶***

Subtract the right from the left node.

|Node type:
|——-
| `BinExpr`

class `jinja2.nodes.` `Call` ***(node, args, kwargs, dyn_args, dyn_kwargs)¶***

Calls an expression. *args* is a list of arguments, *kwargs* a listof keyword arguments (list of `Keyword` nodes), and *dyn_args_and _dyn_kwargs* has to be either *None* or a node that is used asnode for dynamic positional (`args`) *or keyword* (`kwargs`) ***arguments.***

|Node type:
|——-
| `Expr`

class `jinja2.nodes.` `Compare` ***(expr, ops)¶***

Compares an expression with some other expressions. ops must be alist of `Operand` ***s.***

|Node type:
|——-
| `Expr`

class `jinja2.nodes.` `Concat` ***(nodes)¶***

Concatenates the list of expressions provided after converting them tounicode.

|Node type:
|——-
| `Expr`

class `jinja2.nodes.` `CondExpr` ***(test, expr1, expr2)¶***

A conditional expression (inline if expression). (`{{ foo if bar else baz }}` ***)***

|Node type:

|——-
| `Expr`

class `jinja2.nodes.` `ContextReference` ¶

Returns the current template context. It can be used like a `Name` *node, with a* `'load'` *ctx and will return thecurrent* `Context` *object.*

Here an example that assigns the current template name to avariable named foo:

```
1. Assign(Name('foo', ctx='store'),
          Getattr(ContextReference(), 'name'))
```

|Node type:
|——-
| `Expr`

class `jinja2.nodes.` `EnvironmentAttribute` (name) ¶

Loads an attribute from the environment object. This is useful forextensions that want to call a callback stored on the environment.

|Node type:
|——-
| `Expr`

class `jinja2.nodes.` `ExtensionAttribute` (identifier, name) ¶

Returns the attribute of an extension bound to the environment.The identifier is the identifier of the `Extension` *.*

This node is usually constructed by calling the `attr()` *method on an extension.*

|Node type:
|——-
| `Expr`

class `jinja2.nodes.` `Filter`(node, name, args, kwargs, dyn_args, dyn_kwargs)¶

This node applies a filter on an expression. name is the name ofthe filter, the rest of the fields are the same as for `Call`.

If the node of a filter is None the contents of the last buffer arefiltered. Buffers are created by macros and filter blocks.

|Node type:
|——-
| `Expr`

class `jinja2.nodes.` `Getattr`(node, attr, ctx)¶

Get an attribute or item from an expression that is a ascii-onlybytestring and prefer the attribute.

|Node type:
|——-
| `Expr`

class `jinja2.nodes.` `Getitem`(node, arg, ctx)¶

Get an attribute or item from an expression and prefer the item.

|Node type:
|——-
| `Expr`

class `jinja2.nodes.` `ImportedName`(importname)¶

If created with an import name the import name is returned on nodeaccess. For example `ImportedName('cgi.escape')` returns the escape_function from the cgi module on evaluation. Imports are optimized by thecompiler so there is no need to assign them to local variables.

|Node type:
|——-
| `Expr`

_class `jinja2.nodes.` `InternalName`(name)¶

An internal name in the compiler. You cannot create these nodesyourself but the

parser provides a `free_identifier()` method that createsa new identifier for you. This identifier is not available from thetemplate and is not threated specially by the compiler.

|Node type:
|——-
| `Expr`

class `jinja2.nodes.` `Literal` ¶

Baseclass for literals.

|Node type:
|——-
| `Expr`

class `jinja2.nodes.` `Const` (value) ¶

All constant values. The parser will return this node for simpleconstants such as `42` or `"foo"` but it can be used to store morecomplex values such as lists too. Only constants with a saferepresentation (objects where `eval(repr(x)) == x` is true).

|Node type:
|——-
| `Literal`

class `jinja2.nodes.` `Dict` (items) ¶

Any dict literal such as `{1: 2, 3: 4}`. The items must be a list of `Pair` nodes.

|Node type:
|——-
| `Literal`

class `jinja2.nodes.` `List` (items) ¶

Any list literal such as `[1, 2, 3]`

|Node type:
|——-
| `Literal`

class `jinja2.nodes.` `TemplateData` (data) ¶

A constant template string.

|Node type:
|——-
| `Literal`

class `jinja2.nodes.` `Tuple` ***(items, ctx)¶***

For loop unpacking and some other things like multiple argumentsfor subscripts. Like for `Name` *ctx specifies if the tupleis used for loading the names or storing.*

|Node type:
|——-
| `Literal`

class `jinja2.nodes.` `MarkSafe` ***(expr)¶***

Mark the wrapped expression as safe (wrap it as Markup).

|Node type:
|——-
| `Expr`

class `jinja2.nodes.` `MarkSafeIfAutoescape` ***(expr)¶***

Mark the wrapped expression as safe (wrap it as Markup) butonly if autoescaping is active.

New in version 2.5.

|Node type:
|——-
| `Expr`

class `jinja2.nodes.` `Name` ***(name, ctx)¶***

Looks up a name or stores a value in a name.The ctx of the node can be one of the following values:

- store: store a value in the name

- load: load that name
- param: like store but if the name was defined as function parameter.

|Node type:
|——-
| `Expr`

class `jinja2.nodes.` `NSRef`(name, attr)¶

Reference to a namespace value assignment

|Node type:
|——-
| `Expr`

class `jinja2.nodes.` `Slice`(start, stop, step)¶

Represents a slice object. This must only be used as argument for `Subscript`.

|Node type:
|——-
| `Expr`

class `jinja2.nodes.` `Test`(node, name, args, kwargs, dyn_args, dyn_kwargs)¶

Applies a test on an expression. name is the name of the test, therest of the fields are the same as for `Call`.

|Node type:
|——-
| `Expr`

class `jinja2.nodes.` `UnaryExpr`(node)¶

Baseclass for all unary expressions.

|Node type:
|——-
| `Expr`

class `jinja2.nodes.` `Neg`(node)¶

Make the expression negative.

|Node type:
|——-
| `UnaryExpr`

class `jinja2.nodes.` `Not`(node)¶

Negate the expression.

|Node type:
|——-
| `UnaryExpr`

class `jinja2.nodes.` `Pos`(node)¶

Make the expression positive (noop for most expressions)

|Node type:
|——-
| `UnaryExpr`

class `jinja2.nodes.` `Helper`¶

Nodes that exist in a specific context only.

|Node type:
|——-
| `Node`

class `jinja2.nodes.` `Keyword`(key, value)¶

A key, value pair for keyword arguments where key is a string.

|Node type:
|——-
| `Helper`

class `jinja2.nodes.` `Operand`(op, expr)¶

Holds an operator and an expression.The following operators are available: `%`, ``, ``, `+`, `-`, `//`, `/`, `eq`, `gt`, `gteq`, `in`, `lt`, `lteq`, `ne`, `not`, `notin`

|Node type:
|——-

| `Helper`

class `jinja2.nodes.` `Pair` ***(key, value)¶***

A key, value pair for dicts.

|Node type:
|——-
| `Helper`

class `jinja2.nodes.` `Stmt` ***¶***

Base node for all statements.

|Node type:
|——-
| `Node`

class `jinja2.nodes.` `Assign` ***(target, node)¶***

Assigns an expression to a target.

|Node type:
|——-
| `Stmt`

class `jinja2.nodes.` `AssignBlock` ***(target, filter, body)¶***

Assigns a block to a target.

|Node type:
|——-
| `Stmt`

class `jinja2.nodes.` `Block` ***(name, body, scoped)¶***

A node that represents a block.

|Node type:
|——-
| `Stmt`

class `jinja2.nodes.` `Break` ¶

Break a loop.

|Node type:
|——-
| `Stmt`

class `jinja2.nodes.` `CallBlock` *(call, args, defaults, body)* ¶

Like a macro without a name but a call instead. *call* is called withthe unnamed macro as *caller* argument this node holds.

|Node type:
|——-
| `Stmt`

class `jinja2.nodes.` `Continue` ¶

Continue a loop.

|Node type:
|——-
| `Stmt`

class `jinja2.nodes.` `EvalContextModifier` *(options)* ¶

Modifies the eval context. For each option that should be modified,a `Keyword` has to be added to the `options` list.

Example to change the *autoescape* setting:

```
1. EvalContextModifier(options=[Keyword('autoescape', Const(True))])
```

|Node type:
|——-
| `Stmt`

class `jinja2.nodes.` `ScopedEvalContextModifier`***(options, body)***¶

Modifies the eval context and reverts it later. Works exactly like `EvalContextModifier` but will only modify the `EvalContext` for nodes in the `body`.

|Node type:
|——-
| `EvalContextModifier`

class `jinja2.nodes.` `ExprStmt`***(node)***¶

A statement that evaluates an expression and discards the result.

|Node type:
|——-
| `Stmt`

class `jinja2.nodes.` `Extends`***(template)***¶

Represents an extends statement.

|Node type:
|——-
| `Stmt`

class `jinja2.nodes.` `FilterBlock`***(body, filter)***¶

Node for filter sections.

|Node type:
|——-
| `Stmt`

class `jinja2.nodes.` `For`***(target, iter, body, else__, _test, recursive)***¶

The for loop. *target* is the target for the iteration (usually a `Name` or `Tuple`), *iter* the iterable. *body* is a listof nodes that are used as loop-body, and *else__ a list of nodes for the_else* block. If no else node exists it has to be an empty list.

For filtered nodes an expression can be stored as *test*, otherwise *None*.

|Node type:
|——-
| `Stmt`

class `jinja2.nodes.` `FromImport` ***(template, names, with_context)¶***

A node that represents the from import tag. It’s important to notpass unsafe names to the name attribute. The compiler translates theattribute lookups directly into getattr calls and does *not* use thesubscript callback of the interface. As exported variables may notstart with double underscores (which the parser asserts) this is not aproblem for regular Jinja code, but if this node is used in an extensionextra care must be taken.

The list of names may contain tuples if aliases are wanted.

|Node type:
|——-
| `Stmt`

class `jinja2.nodes.` `If` ***(test, body, elif, _else)¶***

If _test is true, *body* is rendered, else *else__.*

|*Node type:*
|——-
| `Stmt`

_class `jinja2.nodes.` `Import` ***(template, target, with_context)¶***

A node that represents the import tag.

|Node type:
|——-
| `Stmt`

class `jinja2.nodes.` `Include` ***(template, with_context, ignore_missing)¶***

A node that represents the include tag.

|Node type:
|——-
| `Stmt`

class jinja2.nodes.Macro(name, args, defaults, body)¶

A macro definition. *name* is the name of the macro, *args* a list ofarguments and *defaults* a list of defaults if there are any. *body* isa list of nodes for the macro body.

|Node type:
|——-
| Stmt

class jinja2.nodes.Output(nodes)¶

A node that holds multiple expressions which are then printed out.This is used both for the *print* statement and the regular template data.

|Node type:
|——-
| Stmt

class jinja2.nodes.OverlayScope(context, body)¶

An overlay scope for extensions. This is a largely unoptimized scopethat however can be used to introduce completely arbitrary variables intoa sub scope from a dictionary or dictionary like object. The *context_field has to evaluate to a dictionary object.*

Example usage:

```
1. OverlayScope(context=self.call_method('get_context'),
                body=[…])
```

New in version 2.10.

|*Node type:*
|——-
| Stmt

_class jinja2.nodes.Scope(body)¶

An artificial scope.

|Node type:
|——-
|Stmt

class jinja2.nodes.With(targets, values, body)¶

Specific node for with statements. In older versions of Jinja thewith statement was implemented on the base of the *Scope* node instead.

New in version 2.9.3.

|Node type:
|——-
|Stmt

class jinja2.nodes.Template(body)¶

Node that represents a template. This must be the outermost node thatis passed to the compiler.

|Node type:
|——-
|Node

exception jinja2.nodes.Impossible¶

Raised if the node could not perform a requested action.

原文:

> http://jinja.pocoo.org/docs/2.10/extensions/

Integration

Integration¶

Jinja2 provides some code for integration into other tools such as frameworks,the Babel library or your favourite editor for fancy code highlighting.This is a brief description of whats included.

Files to help integration are availablehere.

Babel Integration¶

Jinja provides support for extracting gettext messages from templates via aBabel extractor entry point called *jinja2.ext.babel_extract*. The Babelsupport is implemented as part of the i18n Extension extension.

Gettext messages extracted from both *trans* tags and code expressions.

To extract gettext messages from templates, the project needs a Jinja2 sectionin its Babel extraction method mapping file:

```
1. [jinja2: **/templates/**.html]
2. encoding = utf-8
```

The syntax related options of the `Environment` are also available asconfiguration values in the mapping file. For example to tell the extractionthat templates use `%` as *line_statement_prefix* you can use this code:

```
1. [jinja2: **/templates/**.html]
2. encoding = utf-8
3. line_statement_prefix = %
```

Extensions may also be defined by passing a comma separated listof import paths as *extensions* value. The i18n extension is addedautomatically.

Changed in version 2.7: Until 2.7 template syntax errors were always ignored. This was donesince many people are dropping non template html files into thetemplates folder and it would randomly fail. The assumption was thattestsuites will catch syntax errors in templates anyways. If you don’twant that behavior you can add `silent=false` to the settings andexceptions are propagated.

Pylons¶

With Pylons 0.9.7 onwards it’s incredible easy to integrate Jinja into aPylons powered

application.

The template engine is configured in *config/environment.py*. The configurationfor Jinja2 looks something like that:

```
1. from jinja2 import Environment, PackageLoader
2. config['pylons.app_globals'].jinja_env = Environment(
3.     loader=PackageLoader('yourapplication', 'templates')
4. )
```

After that you can render Jinja templates by using the *render_jinja* functionfrom the *pylons.templating* module.

Additionally it's a good idea to set the Pylons' *c* object into strict mode.Per default any attribute to not existing attributes on the *c* object returnan empty string and not an undefined object. To change this just use thissnippet and add it into your *config/environment.py*:

```
1. config['pylons.strict_c'] = True
```

TextMate¶

There is a bundle for TextMate that supports syntax highlighting for Jinja1 and Jinja2 for text basedtemplates as well as HTML. It also contains a few often used snippets.

Vim¶

A syntax plugin for Vim exists in the Vim-scripts directory as well as the*ext* folder at the root of the Jinja2 project. The script supports Jinja1 andJinja2. Once installed two file types are available *jinja* and *htmljinja*.The first one for text based templates, the latter for HTML templates.

Copy the files into your *syntax* folder.

原文:

> *http://jinja.pocoo.org/docs/2.10/integration/*

Switching from other Template Engines

Switching from other Template Engines¶

If you have used a different template engine in the past and want to switchto Jinja2 here is a small guide that shows the basic syntactic and semanticchanges between some common, similar text template engines for Python.

Jinja1¶

Jinja2 is mostly compatible with Jinja1 in terms of API usage and templatesyntax. The differences between Jinja1 and 2 are explained in the followinglist.

API¶

Loaders
Jinja2 uses a different loader API. Because the internal representationof templates changed there is no longer support for external cachingsystems such as memcached. The memory consumed by templates is comparablewith regular Python modules now and external caching doesn't give anyadvantage. If you have used a custom loader in the past have a look atthe new loader API.

Loading templates from strings
In the past it was possible to generate templates from a string with thedefault environment configuration by using *jinja.from_string*. Jinja2provides a `Template` class that can be used to do the same, butwith optional additional configuration.

Automatic unicode conversion
Jinja1 performed automatic conversion of bytestrings in a given encodinginto unicode objects. This conversion is no longer implemented as itwas inconsistent as most libraries are using the regular Python ASCIIbytestring to Unicode conversion. An application powered by Jinja2*has to* use unicode internally everywhere or make sure that Jinja2 onlygets unicode strings passed.

i18n
Jinja1 used custom translators for internationalization. i18n is nowavailable as Jinja2 extension and uses a simpler, more gettext friendlyinterface and has support for babel. For more details seei18n Extension.

Internal methods
Jinja1 exposed a few internal methods on the environment object suchas *call_function*, *get_attribute* and others. While they were markedas being an internal method it was possible to override them. Jinja2doesn't have equivalent methods.

Sandbox

Jinja1 was running sandbox mode by default. Few applications actuallyused that feature so it became optional in Jinja2. For more detailsabout the sandboxed execution see `SandboxedEnvironment`.

Context

Jinja1 had a stacked context as storage for variables passed to theenvironment. In Jinja2 a similar object exists but it doesn't allowmodifications nor is it a singleton. As inheritance is dynamic nowmultiple context objects may exist during template evaluation.

Filters and Tests

Filters and tests are regular functions now. It's no longer necessaryand allowed to use factory functions.

Templates¶

Jinja2 has mostly the same syntax as Jinja1. What's different is thatmacros require parentheses around the argument list now.

Additionally Jinja2 allows dynamic inheritance now and dynamic includes.The old helper function *rendertemplate* is gone now, *include* can be usedinstead. Includes no longer import macros and variable assignments, forthat the new *import* tag is used. This concept is explained in theImport documentation.

Another small change happened in the *for*-tag. The special loop variabledoesn't have a *parent* attribute, instead you have to alias the loopyourself. See Accessing the parent Loop for more details.

Django¶

If you have previously worked with Django templates, you should findJinja2 very familiar. In fact, most of the syntax elements look andwork the same.

However, Jinja2 provides some more syntax elements covered in thedocumentation and some work a bit different.

This section covers the template changes. As the API is fundamentallydifferent we won't cover it here.

Method Calls¶

In Django method calls work implicitly, while Jinja requires the explicitPython syntax. Thus this Django code:

```
1. {% for page in user.get_created_pages %}
2.     ...
3. {% endfor %}
```

…looks like this in Jinja:

```
{% for page in user.get_created_pages() %}
    ...
{% endfor %}
```

This allows you to pass variables to the method, which is not possible inDjango. This syntax is also used for macros.

Filter Arguments¶

Jinja2 provides more than one argument for filters. Also the syntax forargument passing is different. A template that looks like this in Django:

```
{{ items|join:", " }}
```

looks like this in Jinja2:

```
{{ items|join(', ') }}
```

It is a bit more verbose, but it allows different types of arguments -including variables - and more than one of them.

Tests¶

In addition to filters there also are tests you can perform using the isoperator. Here are some examples:

```
{% if user.user_id is odd %}
    {{ user.username|e }} is odd
{% else %}
    hmm. {{ user.username|e }} looks pretty normal
{% endif %}
```

Loops¶

For loops work very similarly to Django, but notably the Jinja2 specialvariable for the loop context is called *loop*, not *forloop* as in Django.

In addition, the Django *empty* argument is called *else* in Jinja2. Forexample, the Django template:

```
{% for item in items %}
    {{ item }}
{% empty %}
```

```
    No items!
{% endfor %}
```

…looks like this in Jinja2:

```
{% for item in items %}
    {{ item }}
{% else %}
    No items!
{% endfor %}
```

Cycle¶

The `{% cycle %}` tag does not exist in Jinja2; however, you can achieve thesame output by using the *cycle* method on the loop context special variable.

The following Django template:

```
{% for user in users %}
    <li class="{% cycle 'odd' 'even' %}">{{ user }}</li>
{% endfor %}
```

…looks like this in Jinja2:

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

There is no equivalent of `{% cycle … as variable %}` .

Mako¶

If you have used Mako so far and want to switch to Jinja2 you can configureJinja2 to look more like Mako:

```
env = Environment('<%', '%>', '- {', '}', '<%doc>', '</%doc>', '%', '##')
```

With an environment configured like that, Jinja2 should be able to interpreta small subset of Mako templates. Jinja2 does not support embedded Pythoncode, so you would have to move that out of the template. The syntax for defs(which are called macros in Jinja2) and template inheritance is different too.The following Mako template:

```
<%inherit file="layout.html" />
<%def name="title()">Page Title</%def>
<ul>
% for item in list:
```

```
    <li>- {item}</li>
% endfor
</ul>
```

Looks like this in Jinja2 with the above configuration:

```
<% extends "layout.html" %>
<% block title %>Page Title<% endblock %>
<% block body %>
<ul>
% for item in list:
    <li>- {item}</li>
% endfor
</ul>
<% endblock %>
```

原文：

> *http://jinja.pocoo.org/docs/2.10/switching/*

Tips and Tricks

Tips and Tricks¶

This part of the documentation shows some tips and tricks for Jinja2templates.

Null-Master Fallback¶

Jinja2 supports dynamic inheritance and does not distinguish between parentand child template as long as no *extends* tag is visited. While this leadsto the surprising behavior that everything before the first *extends* tagincluding whitespace is printed out instead of being ignored, it can be usedfor a neat trick.

Usually child templates extend from one template that adds a basic HTMLskeleton. However it's possible to put the *extends* tag into an *if* tag toonly extend from the layout template if the *standalone* variable evaluatesto false which it does per default if it's not defined. Additionally a verybasic skeleton is added to the file so that if it's indeed rendered with*standalone* set to *True* a very basic HTML skeleton is added:

```
{% if not standalone %}{% extends 'master.html' %}{% endif -%}
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<title>{% block title %}The Page Title{% endblock %}</title>
<link rel="stylesheet" href="style.css" type="text/css">
{% block body %}
  <p>This is the page body.</p>
{% endblock %}
```

Alternating Rows¶

If you want to have different styles for each row of a table orlist you can use the *cycle* method on the *loop* object:

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

cycle can take an unlimited amount of strings. Each time thistag is encountered the next item from the list is rendered.

Highlighting Active Menu Items¶

Often you want to have a navigation bar with an active navigationitem. This is really simple to achieve. Because assignments outsideof _block_s in child templates are global and executed before the layouttemplate is evaluated it’s possible to define the active menu item in thechild template:

```
{% extends "layout.html" %}
{% set active_page = "index" %}
```

The layout template can then access *active_page*. Additionally it makessense to define a default for that variable:

```
{% set navigation_bar = [
    ('/', 'index', 'Index'),
    ('/downloads/', 'downloads', 'Downloads'),
    ('/about/', 'about', 'About')
] -%}
{% set active_page = active_page|default('index') -%}
...
<ul id="navigation">
{% for href, id, caption in navigation_bar %}
  <li{% if id == active_page %} class="active"{% endif
  %}><a href="{{ href|e }}">{{ caption|e }}</a></li>
{% endfor %}
</ul>
...
```

Accessing the parent Loop¶

The special *loop* variable always points to the innermost loop. If it’sdesired to have access to an outer loop it’s possible to alias it:

```
<table>
{% for row in table %}
  <tr>
  {% set rowloop = loop %}
  {% for cell in row %}
    <td id="cell-{{ rowloop.index }}-{{ loop.index }}">{{ cell }}</td>
  {% endfor %}
  </tr>
{% endfor %}
</table>
```

原文：

> *http://jinja.pocoo.org/docs/2.10/tricks/*

Frequently Asked Questions

Frequently Asked Questions¶

This page answers some of the often asked questions about Jinja.

Why is it called Jinja?¶

The name Jinja was chosen because it’s the name of a Japanese temple andtemple and template share a similar pronunciation. It is not named afterthe city in Uganda.

How fast is it?¶

We really hate benchmarks especially since they don’t reflect much. Theperformance of a template depends on many factors and you would have tobenchmark different engines in different situations. The benchmarks from thetestsuite show that Jinja2 has a similar performance to Mako and is between10 and 20 times faster than Django’s template engine or Genshi. These numbersshould be taken with tons of salt as the benchmarks that took these numbersonly test a few performance related situations such as looping. Generallyspeaking the performance of a template engine doesn’t matter much as theusual bottleneck in a web application is either the database or the applicationcode.

How Compatible is Jinja2 with Django?¶

The default syntax of Jinja2 matches Django syntax in many ways. Howeverthis similarity doesn’t mean that you can use a Django template unmodifiedin Jinja2. For example filter arguments use a function call syntax ratherthan a colon to separate filter name and arguments. Additionally theextension interface in Jinja is fundamentally different from the Django onewhich means that your custom tags won’t work any longer.

Generally speaking you will use much less custom extensions as the Jinjatemplate system allows you to use a certain subset of Python expressionswhich can replace most Django extensions. For example instead of usingsomething like this:

```
{% load comments %}
{% get_latest_comments 10 as latest_comments %}
{% for comment in latest_comments %}
    ...
{% endfor %}
```

You will most likely provide an object with attributes to retrievecomments from the

database:

```
{% for comment in models.comments.latest(10) %}
    ...
{% endfor %}
```

Or directly provide the model for quick testing:

```
{% for comment in Comment.objects.order_by('-pub_date')[:10] %}
    ...
{% endfor %}
```

Please keep in mind that even though you may put such things into templatesit still isn’t a good idea. Queries should go into the view code and notthe template!

Isn’t it a terrible idea to put Logic into Templates?¶

Without a doubt you should try to remove as much logic from templates aspossible. But templates without any logic mean that you have to do allthe processing in the code which is boring and stupid. A template enginethat does that is shipped with Python and called *string.Template*. Comeswithout loops and if conditions and is by far the fastest template engineyou can get for Python.

So some amount of logic is required in templates to keep everyone happy.And Jinja leaves it pretty much to you how much logic you want to put intotemplates. There are some restrictions in what you can do and what not.

Jinja2 neither allows you to put arbitrary Python code into templates nordoes it allow all Python expressions. The operators are limited to themost common ones and more advanced expressions such as list comprehensionsand generator expressions are not supported. This keeps the template engineeasier to maintain and templates more readable.

Why is Autoescaping not the Default?¶

There are multiple reasons why automatic escaping is not the default modeand also not the recommended one. While automatic escaping of variablesmeans that you will less likely have an XSS problem it also causes a hugeamount of extra processing in the template engine which can cause seriousperformance problems. As Python doesn’t provide a way to mark strings asunsafe Jinja has to hack around that limitation by providing a customstring class (the `Markup` string) that safely interacts with safeand unsafe strings.

With explicit escaping however the template engine doesn’t have to performany safety checks on variables. Also a human knows not to escape integersor strings that may

never contain characters one has to escape or alreadyHTML markup. For example when iterating over a list over a table ofintegers and floats for a table of statistics the template designer canomit the escaping because he knows that integers or floats don’t containany unsafe parameters.

Additionally Jinja2 is a general purpose template engine and not only usedfor HTML/XML generation. For example you may generate LaTeX, emails,CSS, JavaScript, or configuration files.

Why is the Context immutable?¶

When writing a `contextfunction()` or something similar you may havenoticed that the context tries to stop you from modifying it. If you havemanaged to modify the context by using an internal context API you mayhave noticed that changes in the context don’t seem to be visible in thetemplate. The reason for this is that Jinja uses the context only asprimary data source for template variables for performance reasons.

If you want to modify the context write a function that returns a variableinstead that one can assign to a variable by using set:

```
1. {% set comments = get_latest_comments() %}
```

My tracebacks look weird. What’s happening?¶

If the debugsupport module is not compiled and you are using a Pythoninstallation without ctypes (Python 2.4 without ctypes, Jython or Google’sAppEngine) Jinja2 is unable to provide correct debugging information andthe traceback may be incomplete. There is currently no good workaroundfor Jython or the AppEngine as ctypes is unavailable there and it’s notpossible to use the debugsupport extension.

If you are working in the Google AppEngine development server you canwhitelist the ctypes module to restore the tracebacks. This however won’twork in production environments:

```
1. import os
2. if os.environ.get('SERVER_SOFTWARE', '').startswith('Dev'):
3.     from google.appengine.tools.devappserver2.python import sandbox
4.     sandbox._WHITE_LIST_C_MODULES += ['_ctypes', 'gestalt']
```

Credit for this snippet goes to Thomas Johansson

Why is there no Python 2.3/2.4/2.5/3.1/3.2 support?¶

Python 2.3 is missing a lot of features that are used heavily in Jinja2. Thisdecision

was made as with the upcoming Python 2.6 and 3.0 versions it becomesharder to maintain the code for older Python versions. If you really needPython 2.3 support you either have to use Jinja 1 or other templatingengines that still support 2.3.

Python 2.4/2.5/3.1/3.2 support was removed when we switched to supportingPython 2 and 3 by the same sourcecode (without using 2to3). It was required todrop support because only Python 2.6/2.7 and >=3.3 support byte and unicodeliterals in a way compatible to each other version. If you really need supportfor older Python 2 (or 3) versions, you can just use Jinja2 2.6.

My Macros are overridden by something¶

In some situations the Jinja scoping appears arbitrary:

layout.tmpl:

```
1. {% macro foo() %}LAYOUT{% endmacro %}
2. {% block body %}{% endblock %}
```

child.tmpl:

```
1. {% extends 'layout.tmpl' %}
2. {% macro foo() %}CHILD{% endmacro %}
3. {% block body %}{{ foo() }}{% endblock %}
```

This will print `LAYOUT` in Jinja2. This is a side effect of havingthe parent template evaluated after the child one. This allows childtemplates passing information to the parent template. To avoid thisissue rename the macro or variable in the parent template to have anuncommon prefix.

原文:

> *http://jinja.pocoo.org/docs/2.10/faq/*

Jinja Changelog

Jinja Changelog¶

Version 2.10¶

released on November 8th 2017

- Added a new extension node called OverlayScope which can be used tocreate an unoptimized scope that will look up all variables from aderived context.
- Added an in test that works like the in operator. This can be usedin combination with reject and select.
- Added previtem and nextitem to loop contexts, providing access to theprevious/next item in the loop. If such an item does not exist, the value isundefined.
- Added changed(*values) to loop contexts, providing an easy way ofchecking whether a value has changed since the last iteration (or rathersince the last call of the method)
- Added a namespace function that creates a special object which allowsattribute assignment using the set tag. This can be used to carry dataacross scopes, e.g. from a loop body to code that comes after the loop.
- Added a trimmed modifier to {% trans %} to strip linebreaks andsurrounding whitespace. Also added a new policy to enable this for alltrans blocks.
- The random filter is no longer incorrectly constant folded and willproduce a new random choice each time the template is rendered. (#478)
- Added a unique filter. (#469)
- Added min and max filters. (#475)
- Added tests for all comparison operators: eq, ne, lt, le,gt, ge. (#665)
- import statement cannot end with a trailing comma. (#617, #618)
- indent filter will not indent blank lines by default. (#685)
- Add reverse argument for dictsort filter. (#692)
- Add a NativeEnvironment that renders templates to native Python typesinstead of strings. (#708)
- Added filter support to the block set tag. (#489)
- tojson filter marks output as safe to match documented behavior.(#718)
- Resolved a bug where getting debug locals for tracebacks couldmodify template context.
- Fixed a bug where having many {% elif … %} blocks resulted in a“too many levels of indentation” error. These blocks now compile tonative elif ..: instead of else: if ..: (#759)

Version 2.9.6¶

(bugfix release, released on April 3rd 2017)

- Fixed custom context behavior in fast resolve mode (#675)

Version 2.9.5¶

(bugfix release, released on January 28th 2017)

- Restored the original repr of the internal _GroupTuple because thiscaused issues with ansible and it was an unintended change. (#654)
- Added back support for custom contexts that override the old resolvemethod since it was hard for people to spot that this could cause aregression.
- Correctly use the buffer for the else block of for loops. This causedinvalid syntax errors to be caused on 2.x and completely wrong behavioron Python 3 (#669)
- Resolve an issue where the {% extends %} tag could not be used withasync environments. (#668)
- Reduce memory footprint slightly by reducing our unicode database dumpwe use for identifier matching on Python 3 (#666)
- Fixed autoescaping not working for macros in async compilation mode. (#671)

Version 2.9.4¶

(bugfix release, released on January 10th 2017)

- Solved some warnings for string literals. (#646)
- Increment the bytecode cache version which was not done due to anoversight before.
- Corrected bad code generation and scoping for filtered loops. (#649)
- Resolved an issue where top-level output silencing after known extendblocks could generate invalid code when blocks where contained in ifstatements. (#651)
- Made the truncate.leeway default configurable to improve compatibilitywith older templates.

Version 2.9.3¶

(bugfix release, released on January 8th 2017)

- Restored the use of blocks in macros to the extend that was possiblebefore. On Python 3 it would render a generator repr instead ofthe block contents. (#645)
- Set a consistent behavior for assigning of variables in inner scopeswhen the variable is also read from an outer scope. This now sets theintended behavior in all situations however it does not restore theold behavior where limited assignments to outer scopes was possible.For more information and a discussion see #641
- Resolved an issue where block scoped would not take advantage of thenew scoping rules. In some more exotic cases a variable overriden in alocal scope would not

make it into a block.
- Change the code generation of the with statement to be in line with thenew scoping rules. This resolves some unlikely bugs in edge cases. Thisalso introduces a new internal With node that can be used by extensions.

Version 2.9.2¶

(bugfix release, released on January 8th 2017)

- Fixed a regression that caused for loops to not be able to use the samevariable for the target as well as source iterator. (#640)
- Add support for a previously unknown behavior of macros. It used to bepossible in some circumstances to explicitly provide a caller argumentto macros. While badly buggy and unintended it turns out that this is acommon case that gets copy pasted around. To not completely break backwardscompatibility with the most common cases it’s now possible to provide anexplicit keyword argument for caller if it’s given an explicit default.(#642)

Version 2.9.1¶

(bugfix release, released on January 7th 2017)

- Resolved a regression with call block scoping for macros. Nested callerblocks that used the same identifiers as outer macros could refer to thewrong variable incorrectly.

Version 2.9¶

(codename Derivation, released on January 7th 2017)

- Change cache key definition in environment. This fixes a performanceregression introduced in 2.8.
- Added support for generator_stop on supported Python versions(Python 3.5 and later)
- Corrected a long standing issue with operator precedence of math operationsnot being what was expected.
- Added support for Python 3.6 async iterators through a new async mode.
- Added policies for filter defaults and similar things.
- urlize now sets “rel noopener” by default.
- Support attribute fallback for old-style classes in 2.x.
- Support toplevel set statements in extend situations.
- Restored behavior of Cycler for Python 3 users.
- Subtraction now follows the same behavior as other operators on undefinedvalues.
- map and friends will now give better error messages if you forgot toquote the parameter.

- Depend on MarkupSafe 0.23 or higher.
- Improved the truncate filter to support better truncation in casethe string is barely truncated at all.
- Change the logic for macro autoescaping to be based on the runtimeautoescaping information at call time instead of macro define time.
- Ported a modified version of the tojson filter from Flask to Jinja2and hooked it up with the new policy framework.
- Block sets are now marked safe by default.
- On Python 2 the asciification of ASCII strings can now be disabled withthe compiler.ascii_str policy.
- Tests now no longer accept an arbitrary expression as first argument buta restricted one. This means that you can now properly use multipletests in one expression without extra parentheses. In particular you cannow write foo is divisibleby 2 or foo is divisibleby 3as you would expect.
- Greatly changed the scoping system to be more consistent with what templatedesigners and developers expect. There is now no more magic differencebetween the different include and import constructs. Context is now alwayspropagated the same way. The only remaining differences is the defaultsfor with context and without context.
- The with and autoescape tags are now built-in.
- Added the new select_autoescape function which helps configuring betterautoescaping easier.
- Fixed a runtime error in the sandbox when attributes of async generatorswere accessed.

Version 2.8.1¶

(bugfix release, released on December 29th 2016)

- Fixed the for_qs flag for urlencode.
- Fixed regression when applying int to non-string values.
- SECURITY: if the sandbox mode is used format expressions are now sandboxedwith the same rules as in Jinja. This solves various information leakageproblems that can occur with format strings.

Version 2.8¶

(codename Replacement, released on July 26th 2015)

- Added target parameter to urlize function.
- Added support for followsymlinks to the file system loader.
- The truncate filter now counts the length.
- Added equalto filter that helps with select filters.
- Changed cache keys to use absolute file names if availableinstead of load names.
- Fixed loop length calculation for some iterators.
- Changed how Jinja2 enforces strings to be native strings inPython 2 to work when

people break their default encoding.
- Added make_logging_undefined() which returns an undefinedobject that logs failures into a logger.
- If unmarshalling of cached data fails the template will bereloaded now.
- Implemented a block set tag.
- Default cache size was increased to 400 from a low 50.
- Fixed is number test to accept long integers in all Python versions.
- Changed is number to accept Decimal as a number.
- Added a check for default arguments followed by non-default arguments. Thischange makes {% macro m(x, y=1, z) %}…{% endmacro %} a syntax error.The previous behavior for this code was broken anyway (resulting in thedefault value being applied to y).
- Add ability to use custom subclasses of jinja2.compiler.CodeGenerator andjinja2.runtime.Context by adding two new attributes to the environment(code_generator_class and context_class) (pull request #404).
- added support for context/environment/evalctx decorator functions onthe finalize callback of the environment.
- escape query strings for urlencode properly. Previously slashes were notescaped in that place.
- Add ‘base’ parameter to ‘int’ filter.

Version 2.7.3¶

(bugfix release, released on June 6th 2014)

- Security issue: Corrected the security fix for the cache folder. Thisfix was provided by RedHat.

Version 2.7.2¶

(bugfix release, released on January 10th 2014)

- Prefix loader was not forwarding the locals properly toinner loaders. This is now fixed.
- Security issue: Changed the default folder for the filesystem cache to beuser specific and read and write protected on UNIX systems. SeeDebian bug 734747 for more information.

Version 2.7.1¶

(bugfix release, released on August 7th 2013)

- Fixed a bug with call_filter not working properly on environmentand context filters.
- Fixed lack of Python 3 support for bytecode caches.

- Reverted support for defining blocks in included templates as thisbroke existing templates for users.
- Fixed some warnings with hashing of undefineds and nodes if Pythonis run with warnings for Python 3.
- Added support for properly hashing undefined objects.
- Fixed a bug with the title filter not working on already uppercasestrings.

Version 2.7¶

(codename Translation, released on May 20th 2013)

- Choice and prefix loaders now dispatch source and template lookupseparately in order to work in combination with module loaders asadvertised.
- Fixed filesizeformat.
- Added a non-silent option for babel extraction.
- Added urlencode filter that automatically quotes values forURL safe usage with utf-8 as only supported encoding. If applicationswant to change this encoding they can override the filter.
- Added keep-trailing-newline configuration to environments andtemplates to optionally preserve the final trailing newline.
- Accessing last on the loop context no longer causes the iteratorto be consumed into a list.
- Python requirement changed: 2.6, 2.7 or >= 3.3 are required now,supported by same source code, using the “six” compatibility library.
- Allow contextfunction and other decorators to be applied to **call**.
- Added support for changing from newline to different signs in the wordwrapfilter.
- Added support for ignoring memcache errors silently.
- Added support for keeping the trailing newline in templates.
- Added finer grained support for stripping whitespace on the left sideof blocks.
- Added map, select, reject, selectattr and rejectattrfilters.
- Added support for loop.depth to figure out how deep inside a recursiveloop the code is.
- Disabled py_compile for pypy and python 3.

Version 2.6¶

(codename Convolution, released on July 24th 2011)

- internal attributes now raise an internal attribute error now insteadof returning an undefined. This fixes problems when passing undefinedobjects to Python semantics expecting APIs.
- traceback support now works properly for PyPy. (Tested with 1.4)
- implemented operator intercepting for sandboxed environments. Thisallows application developers to disable builtin operators for bettersecurity. (For instance limit the mathematical operators to actualintegers instead of longs)
- groupby filter now supports dotted notation for grouping by attributesof

attributes.

- scoped blocks now properly treat toplevel assignments and imports.Previously an import suddenly “disappeared” in a scoped block.
- automatically detect newer Python interpreter versions before loading codefrom bytecode caches to prevent segfaults on invalid opcodes. The segfaultin earlier Jinja2 versions here was not a Jinja2 bug but a limitation inthe underlying Python interpreter. If you notice Jinja2 segfaulting inearlier versions after an upgrade of the Python interpreter you don’t haveto upgrade, it’s enough to flush the bytecode cache. This just no longermakes this necessary, Jinja2 will automatically detect these cases now.
- the sum filter can now sum up values by attribute. This is a backwardsincompatible change. The argument to the filter previously was theoptional starting index which defaults to zero. This now became thesecond argument to the function because it’s rarely used.
- like sum, sort now also makes it possible to order items by attribute.
- like sum and sort, join now also is able to join attributes of objectsas string.
- the internal eval context now has a reference to the environment.
- added a mapping test to see if an object is a dict or an object witha similar interface.

Version 2.5.5¶

(re-release of 2.5.4 with built documentation removed for filesize.
Released on October 18th 2010)

- built documentation is no longer part of release.

Version 2.5.4¶

(bugfix release, released on October 17th 2010)

- Fixed extensions not loading properly with overlays.
- Work around a bug in cpython for the debugger that causes segfaultson 64bit big-endian architectures.

Version 2.5.3¶

(bugfix release, released on October 17th 2010)

- fixed an operator precedence error introduced in 2.5.2. Statementslike “-foo.bar” had their implicit parentheses applied around thefirst part of the expression (“(-foo).bar”) instead of the morecorrect “-(foo.bar)”.

Version 2.5.2¶

(bugfix release, released on August 18th 2010)

- improved setup.py script to better work with assumptions peoplemight still have from it (–with-speedups).
- fixed a packaging error that excluded the new debug support.

Version 2.5.1¶

(bugfix release, released on August 17th 2010)

- StopIteration exceptions raised by functions called from templatesare now intercepted and converted to undefineds. This solves alot of debugging grief. (StopIteration is used internally toabort template execution)
- improved performance of macro calls slightly.
- babel extraction can now properly extract newstyle gettext calls.
- using the variable num in newstyle gettext for something elsethan the pluralize count will no longer raise a KeyError.
- removed builtin markup class and switched to markupsafe. For backwardscompatibility the pure Python implementation still exists but ispulled from markupsafe by the Jinja2 developers. The debug supportwent into a separate feature called “debugsupport” and is disabledby default because it is only relevant for Python 2.4
- fixed an issue with unary operators having the wrong precedence.

Version 2.5¶

(codename Incoherence, released on May 29th 2010)

- improved the sort filter (should have worked like this for along time) by adding support for case insensitive searches.
- fixed a bug for getattribute constant folding.
- support for newstyle gettext translations which result in anicer in-template user interface and more consistentcatalogs. (Whitespace Trimming)
- it’s now possible to register extensions after an environmentwas created.

Version 2.4.1¶

(bugfix release, released on April 20th 2010)

- fixed an error reporting bug for undefineds.

Version 2.4¶

(codename Correlation, released on April 13th 2010)

- the environment template loading functions now transparentlypass through a template object if it was passed to it. Thismakes it possible to import or extend from a template objectthat was passed to the template.
- added a ModuleLoader that can load templates fromprecompiled sources. The environment now features a methodto compile the templates from a configured loader into a zipfile or folder.
- the _speedups C extension now supports Python 3.
- added support for autoescaping toggling sections and supportfor evaluation contexts (Evaluation Context).
- extensions have a priority now.

Version 2.3.1¶

(bugfix release, released on February 19th 2010)

- fixed an error reporting bug on all python versions
- fixed an error reporting bug on Python 2.4

Version 2.3¶

(codename 3000 Pythons, released on February 10th 2010)

- fixes issue with code generator that causes unbound variablesto be generated if set was used in if-blocks and other smallidentifier problems.
- include tags are now able to select between multiple templatesand take the first that exists, if a list of templates isgiven.
- fixed a problem with having call blocks in outer scopes thathave an argument that is also used as local variable in aninner frame (#360).
- greatly improved error message reporting (#339)
- implicit tuple expressions can no longer be totally empty.This change makes {% if %}…{% endif %} a syntax errornow. (#364)
- added support for translator comments if extracted via babel.
- added with-statement extension.
- experimental Python 3 support.

Version 2.2.1¶

(bugfix release, released on September 14th 2009)

- fixes some smaller problems for Jinja2 on Jython.

Version 2.2¶

(codename Kong, released on September 13th 2009)

- Include statements can now be marked with ignore missing to skipnon existing templates.
- Priority of not raised. It’s now possible to write *not foo in bar_as an alias to _foo not in bar* like in python. Previously the grammarrequired parentheses (*not (foo in bar)*) which was odd.
- Fixed a bug that caused syntax errors when defining macros or using the*{% call %}* tag inside loops.
- Fixed a bug in the parser that made {{ foo[1, 2] }} impossible.
- Made it possible to refer to names from outer scopes in included templatesthat were unused in the callers frame (#327)
- Fixed a bug that caused internal errors if names where used as iterationvariable and regular variable *after* the loop if that variable was unused*before* the loop. (#331)
- Added support for optional scoped modifier to blocks.
- Added support for line-comments.
- Added the meta module.
- Renamed (undocumented) attribute “overlay” to “overlayed” on theenvironment because it was clashing with a method of the same name.
- speedup extension is now disabled by default.

Version 2.1.1¶

(bugfix release, released on December 25th 2008)

- Fixed a translation error caused by looping over empty recursive loops.

Version 2.1¶

(codename Yasuzō, released on November 23rd 2008)

- fixed a bug with nested loops and the special loop variable. Before thechange an inner loop overwrote the loop variable from the outer one afteriteration.
- fixed a bug with the i18n extension that caused the explicit pluralizationblock to look up the wrong variable.
- fixed a limitation in the lexer that made {{ foo.0.0 }} impossible.
- index based subscribing of variables with a constant value returns anundefined object now instead of raising an index error. This was a bugcaused by eager optimizing.
- the i18n extension looks up foo.ugettext now followed by foo.gettextif an translations object is installed. This makes dealing with customtranslations classes easier.
- fixed a confusing behavior with conditional extending. loops were partiallyexecuted under some conditions even though they were not part of a visiblearea.
- added sort filter that works like dictsort but for arbitrary sequences.
- fixed a bug with empty statements in macros.

- implemented a bytecode cache system. (Bytecode Cache)
- the template context is now weakref-able
- inclusions and imports “with context” forward all variables now, not onlythe initial context.
- added a cycle helper called cycler.
- added a joining helper called joiner.
- added a compile_expression method to the environment that allows compilingof Jinja expressions into callable Python objects.
- fixed an escaping bug in urlize

Version 2.0¶

(codename jinjavitus, released on July 17th 2008)

- the subscribing of objects (looking up attributes and items) changed fromslightly. It’s now possible to give attributes or items a higher priorityby either using dot-notation lookup or the bracket syntax. This alsochanged the AST slightly. Subscript is gone and was replaced withGetitem and Getattr.

For more information see the implementation details.

- added support for preprocessing and token stream filtering for extensions.This would allow extensions to allow simplified gettext calls in templatedata and something similar.
- added jinja2.environment.TemplateStream.dump().
- added missing support for implicit string literal concatenation.{{ "foo" "bar" }} is equivalent to {{ "foobar" }}
- else is optional for conditional expressions. If not given it evaluatesto false.
- improved error reporting for undefined values by providing a position.
- filesizeformat filter uses decimal prefixes now per default and can beset to binary mode with the second parameter.
- fixed bug in finalizer

Version 2.0rc1¶

(no codename, released on June 9th 2008)

- first release of Jinja2

原文:

> *http://jinja.pocoo.org/docs/2.10/changelog/*