docs.jinkan.org

扩展 — Jinja2 2.7 documentation

86-108 分钟

Jinja2 支持扩展来添加过滤器、测试、全局变量或者甚至是处理器。扩展的主要动力是把诸如添加国际化支持的常用代码迁移到一个可重用的类。

添加扩展¶

扩展在 Jinja2 环境创建时被添加。一旦环境被创建，就不能添加额外的扩展。要添加一个扩展，传递一个扩展类或导入路径的列表到 `Environment` 构造函数的 *environment* 参数。下面的例子创建了一个加载了 i18n 扩展的 Jinja2 环境:

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

i18n 扩展¶

Import name: *jinja2.ext.i18n*

Jinja2 当前只附带一个扩展，就是 i18n 扩展。它可以与 gettext 或 babel 联合使用。如果启用了 i18n 扩展， Jinja2 提供了 *trans* 语句来标记被其包裹的字符串为可翻译的，并调用 *gettext* 。

在启用虚拟的 _ 函数后，之后的 *gettext* 调用会被添加到环境的全局变量。那么一个国际化的应用应该不仅在全局，以及在每次渲染中在命名空间中提供至少一个 *gettext* 或可选的 *ngettext* 函数。

环境方法¶

在启用这个扩展后，环境提供下面的额外方法:

`jinja2.Environment.install_gettext_translations`(*translations*, *newstyle=False*)¶

在该环境中全局安装翻译。提供的翻译对象要至少实现 *uggettext* 和 *ungettext* 。

gettext.NullTranslations 和 *gettext.GNUTranslations* 类和 Babel‘s 的 *Translations* 类也被支持。

Changed in version 2.5: 添加了新样式的 gettext

`jinja2.Environment.install_null_translations`(*newstyle=False*)¶

安装虚拟的 gettext 函数。这在你想使应用为国际化做准备但还不想实现完整的国际化系统时很有用。

Changed in version 2.5: 添加了新样式的 gettext

`jinja2.Environment.install_gettext_callables`(*gettext*, *ngettext*, *newstyle=False*)¶

在环境中把给出的 *gettext* 和 *ngettext* 可调用量安装为全局变量。它们应该表现得几乎与标准库中的 `gettext.ugettext()` 和 `gettext.ungettext()` 函数相同。

如果激活了 *新样式*，可调用量被包装为新样式的可调用量一样工作。更多信息见 *新样式 Gettext* 。

New in version 2.5.

`jinja2.Environment.uninstall_gettext_translations`()¶

再次卸载翻译。

从给定的模板或源中提取本地化字符串。

对找到的每一个字符串，这个函数生产一个 `(lineno, function, message )` 元组，在这里:

- *lineno* 是这个字符串所在行的行号。
- *function* 是 *gettext* 函数使用的名称（如果字符串是从内嵌的 Python 代码中抽取的）。
- *message* 是字符串本身（一个 *unicode* 对象，在函数有多个字符串参数时是一个 *unicode* 对象的元组）。

如果安装了 Babel， *Babel 集成* 可以用来为 babel 抽取字符串。

对于一个对多种语言可用而对所有用户给出同一种的语言的 web 应用（例如一个法国社区安全了一个多种语言的论坛软件）可能会一次性加载翻译并且在环境生成时把翻译方法添加到环境上:

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

get_get_translations 函数会返回当前配置的翻译器。（比如使用 *gettext.find* ）

模板设计者的 *i18n* 扩展使用在 *模板文档* 中有描述。

新样式 **Gettext**¶

New in version 2.5.

从版本 2.5 开始你可以使用新样式的 gettext 调用。这些的启发源于 trac 的内部 gettext 函数并且完全被 babel 抽取工具支持。如果你不使用 Babel 的抽取工具，它可能不会像其它抽取工具预期的那样工作。

标准 gettext 调用和新样式的 gettext 调用有什么区别？通常，它们要输入的东西更少，出错率更低。并且如果在自动转义环境中使用它们，它们也能更好地支持自动转义。这里是一些新老样式调

用的差异:

标准 gettext:

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

新样式看起来是这样:

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

新样式 gettext 的优势是你需要输入的更少，并且命名占位符是强制的。后者看起来似乎是缺陷，但解决了当翻译者不能切换两个占位符的位置时经常勉励的一大堆麻烦。使用新样式的 gettext，所有的格式化字符串看起来都一样。

除此之外，在新样式 gettext 中，如果没有使用占位符，字符串格式化也会被使用，这使得所有的字符串表现一致。最后，不仅是新样式的 gettext 调用可以妥善地为解决了许多转义相关问题的自动转义标记字符串，许多模板也在使用自动转义时体验了多次。

表达式语句¶

Import name: *jinja2.ext.do*

“do”又叫做表达式语句扩展，向模板引擎添加了一个简单的 *do* 标签，其工作如同一个变量表达式，只是忽略返回值。

循环控制¶

Import name: *jinja2.ext.loopcontrols*

这个扩展添加了循环中的 *break* 和 *continue* 支持。在启用它之后， Jinja2 提供的这两个关键字如同 Python 中那样工作。

With 语句¶

Import name: *jinja2.ext.with_*

New in version 2.3.

这个扩展添加了 with 关键字支持。使用这个关键字可以在模板中强制一块嵌套的作用域。变量可以在 with 语句的块头中直接声明，或直接在里面使用标准的 *set* 语句。

自动转义扩展¶

Import name: *jinja2.ext.autoescape*

New in version 2.4.

自动转义扩展允许你在模板内开关自动转义特性。如果环境的 `autoescape` 设定为 *False*，它可以被激活。如果是 *True* 可以被关闭。这个设定的覆盖是有作用域的。

编写扩展¶

你可以编写扩展来向 Jinja2 中添加自定义标签。这是一个不平凡的任务，而且通常不需要，因为默认的标签和表达式涵盖了所有常用情况。如 i18n 扩展是一个扩展有用的好例子，而另一个会是碎片缓存。

当你编写扩展时，你需要记住你在与 Jinja2 模板编译器一同工作，而它并不验证你传递到它的节点树。如果 AST 是畸形的，你会得到各种各样的编译器或运行时错误，这调试起来极其可怕。始终确保你在使用创建正确的节点。下面的 API 文档展示了有什么节点和如何使用它们。

示例扩展¶

下面的例子用 Werkzeug 的缓存 contrib 模块为 Jinja2 实现了一个 *cache* 标签:

```
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(
            fragment_cache_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 ``'cache'`` 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 = parser.stream.next().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
```

而这是你在环境中使用它的方式:

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

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

之后，在模板中可以标记块为可缓存的。下面的例子缓存一个边栏 300 秒:

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

扩展 **API**¶

扩展总是继承 jinja2.ext.Extension 类:

class jinja2.ext.Extension(*environment*)¶

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

As extensions are created by the environment they cannot accept any arguments for configuration. One may want to work around that by using a factory function, but that is not possible as extensions are identified by their import name. The correct way to configure the extension is storing the configuration values on the environment. Because this way the environment ends up acting as central configuration storage the attributes may clash which is why extensions have to ensure that the names they choose for configuration are not too generic. `prefix` for example is a terrible name, `fragment_cache_prefix` on the other hand is a good name as includes the name of the extension (fragment cache).

`identifier`¶

扩展的标识符。这始终是扩展类的真实导入名，不能被修改。

`tags`¶

如果扩展实现自定义标签，这是扩展监听的标签名的集合。

attr(*name*, *lineno=None*)¶

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

```
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.

filter_stream(*stream*)¶

It's passed a TokenStream that can be used to filter tokens returned. This method has to return an iterable of Tokens, but it doesn't have to return a TokenStream.

In the *ext* folder of the Jinja2 source distribution there is a file called *inlinegettext.py* which implements a filter that utilizes this method.

parse(*parser*)¶

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

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

This method is called before the actual lexing and can be used to preprocess the source. The *filename* is optional. The return value must be the preprocessed source.

解析器 API¶

传递到 Extension.parse() 的解析器提供解析不同类型表达式的方式。下面的方法可能会在扩展中使用:

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

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

filename¶

解析器处理的模板文件名。这 不是 模板的加载名。加载名见 name 。对于不是从文件系统中加载的模板，这个值为 *None* 。

name¶

模板的加载名。

stream¶

当前的 TokenStream 。

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

Convenience method that raises *exc* with the message, passed line number or last line number as well as the current name and filename.

`free_identifier`(*lineno=None*)¶

Return a new free identifier as `InternalName`.

`parse_assign_target`(*with_tuple=True*, *name_only=False*, *extra_end_rules=None*)¶

Parse an assignment target. As Jinja2 allows assignments to tuples, this function can parse all allowed assignment targets. Per default assignments to tuples are parsed, that can be disable however by setting *with_tuple* to *False*. If only assignments to names are wanted *name_only* can be set to *True*. The *extra_end_rules* parameter is forwarded to the tuple parsing function.

`parse_expression`(*with_condexpr=True*)¶

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

`parse_statements`(*end_tokens*, *drop_needle=False*)¶

Parse multiple statements into a list until one of the end tokens is reached. This is used to parse the body of statements as it also parses template data if appropriate. The parser checks first if the current token is a colon and skips it if there is one. Then it checks for the block end and parses

until if one of the *end_tokens* is reached. Per default the active token in the stream at the end of the call is the matched end token. If this is not wanted *drop_needle* can be set to *True* and the end token is removed.

`parse_tuple`(*simplified=False*, *with_condexpr=True*, *extra_end_rules=None*, *explicit_parentheses=False*)¶

Works like *parse_expression* but if multiple expressions are delimited by a comma a `Tuple` node is created. This method could also return a regular expression instead of a tuple if 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 is forwarded to `parse_expression()`.

Because tuples do not require delimiters and may end in a bogus comma an extra hint is needed that marks the end of a tuple. For example for 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 an expression in parentheses. This is used to figure out if an empty tuple is a valid expression or not.

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

A token stream is an iterable that yields `Token`s. The parser however does not iterate over it but calls

`next()` to go one token ahead. The current active token is stored as `current`.

`current`¶

当前的 `Token` 。

`eos`¶

Are we at the end of the stream?

`expect`(*expr*)¶

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

`look`()¶

Look at the next token.

`next`()¶

Go one token ahead and return the old one

`next_if`(*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.

skip_if(*expr*)¶

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

class jinja2.lexer.Token¶

Token class.

lineno¶

token 的行号。

type¶

token 的类型。这个值是被禁锢的，所以你可以用 *is* 运算符同任意字符串比较。

value¶

token 的值。

test(*expr*)¶

Test a token against a token expression. This can either be a token type or `'token_type:token_value'`. This can only test against string values and types.

test_any(**iterable*)¶

Test against multiple token expressions.

同样，在词法分析模块中也有一个实用函数可以计算字符串中的换行符数目:

```
.. autofunction:: jinja2.lexer.count_newlines
```

AST¶

AST（抽象语法树: Abstract Syntax Tree）用于表示解析后的模板。它有编译器之后转换到可执行的 Python 代码对象的节点构建。提供自定义语句的扩展可以返回执行自定义 Python 代码的节点。

下面的清单展示了所有当前可用的节点。 AST 在 Jinja2 的各个版本中有差异，但会向后兼容。

更多信息请见 `jinja2.Environment.parse()` 。

class `jinja2.nodes.Node`¶

Baseclass for all Jinja2 nodes. There are a number of nodes available of 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 regular positional arguments, attributes as keyword arguments. Each node has two attributes: *lineno* (the line number of the node) and *environment*. The *environment* attribute is set at the end of the parsing process for all nodes automatically.

  `find`(*node_type*)¶

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

  `find_all`(*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.

  `iter_child_nodes`(*exclude=None*, *only=None*)¶

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

  `iter_fields`(*exclude=None*, *only=None*)¶

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

names.

`set_ctx`(*ctx*)¶

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

`set_environment`(*environment*)¶

Set the environment for all nodes.

`set_lineno`(*lineno*, *override=False*)¶

Set the line numbers of the node and children.

class `jinja2.nodes.Expr`¶

Baseclass for all expressions.

`as_const`(*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 given a default context is created which requires the nodes to have an attached environment.

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

can_assign()¶

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

class jinja2.nodes.BinExpr(*left*, *right*)¶

Baseclass for all binary expressions.

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

Add the left to the right node.

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

Short circuited AND.

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

Divides the left by the right node.

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

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

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

Left modulo right.

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

Multiplies the left with the right node.

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

Short circuited OR.

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

Left to the power of right.

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

Substract the right from the left node.

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

Calls an expression. *args* is a list of arguments, *kwargs* a list of keyword arguments (list of Keyword nodes), and *dyn_args* and *dyn_kwargs* has to be either *None* or a node that is used as node for dynamic positional (*args) or keyword (**kwargs) arguments.

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

Compares an expression with some other expressions. *ops* must be a list of Operands.

class jinja2.nodes.Concat(*nodes*)¶

Concatenates the list of expressions provided after converting them to unicode.

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

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

class jinja2.nodes.ContextReference¶

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

Here an example that assigns the current template name to a variable named *foo*:

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

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

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

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.

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

This node applies a filter on an expression. *name* is the name of the 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 are filtered. Buffers are created by macros and filter blocks.

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

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

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

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

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

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

class `jinja2.nodes.InternalName`(*name*)¶

An internal name in the compiler. You cannot create these nodes yourself but the parser provides a `free_identifier()` method that creates a new identifier for you. This identifier is not available from the template and is not threated specially by the compiler.

class `jinja2.nodes.Literal`¶

Baseclass for literals.

class `jinja2.nodes.Const`(*value*)¶

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

class `jinja2.nodes.Dict`(*items*)¶

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

class `jinja2.nodes.List`(*items*)¶

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

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

A constant template string.

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

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

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

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

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

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

New in version 2.5.

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.

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

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

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, the rest of the fields are the same as for Call.

class jinja2.nodes.UnaryExpr(*node*)¶

Baseclass for all unary expressions.

class jinja2.nodes.Neg(*node*)¶

Make the expression negative.

class jinja2.nodes.Not(*node*)¶

Negate the expression.

class jinja2.nodes.Pos(*node*)¶

Make the expression positive (noop for most expressions)

class jinja2.nodes.Helper¶

Nodes that exist in a specific context only.

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

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

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`

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

A key, value pair for dicts.

class `jinja2.nodes.Stmt`¶

Base node for all statements.

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

Assigns an expression to a target.

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

A node that represents a block.

class `jinja2.nodes.Break`¶

Break a loop.

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

Like a macro without a name but a call instead. *call* is called with the unnamed macro as *caller*

argument this node holds.

class `jinja2.nodes.Continue`¶

Continue a loop.

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:

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

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`.

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

A statement that evaluates an expression and discards the result.

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

Represents an extends statement.

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

Node for filter sections.

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 list of 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*.

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

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

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

class jinja2.nodes.If(*test*, *body*, *else_*)¶

If *test* is true, *body* is rendered, else *else_*.

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

A node that represents the import tag.

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

A node that represents the include tag.

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

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

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.

class jinja2.nodes.Scope(*body*)¶

An artificial scope.

class jinja2.nodes.Template(*body*)¶

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

exception jinja2.nodes.Impossible¶

Raised if the node could not perform a requested action.