# Jinja Documentation (3.2.x)

*Release 3.2.0.dev0*

**Pallets**

**Oct 03, 2022**

# CONTENTS:

Jinja is a fast, expressive, extensible templating engine. Special placeholders in the template allow writing code similar to Python syntax. Then the template is passed data to render the final document.

# INTRODUCTION

Jinja is a fast, expressive, extensible templating engine. Special placeholders in the template allow writing code similar to Python syntax. Then the template is passed data to render the final document.

It includes:

- Template inheritance and inclusion.

- Define and import macros within templates.

- HTML templates can use autoescaping to prevent XSS from untrusted user input.

- A sandboxed environment can safely render untrusted templates.

- Async support for generating templates that automatically handle sync and async functions without extra syntax.

- I18N support with Babel.

- Templates are compiled to optimized Python code just-in-time and cached, or can be compiled ahead-of-time.

- Exceptions point to the correct line in templates to make debugging easier.

- Extensible filters, tests, functions, and even syntax.

Jinja's philosophy is that while application logic belongs in Python if possible, it shouldn't make the template designer's job difficult by restricting functionality too much.

## 1.1 Installation

We recommend using the latest version of Python. Jinja supports Python 3.7 and newer. We also recommend using a virtual environment in order to isolate your project dependencies from other projects and the system.

Install the most recent Jinja version using pip:

```
$ pip install Jinja2
```

### 1.1.1 Dependencies

These will be installed automatically when installing Jinja.

- MarkupSafe escapes untrusted input when rendering templates to avoid injection attacks.

### 1.1.2 Optional Dependencies

These distributions will not be installed automatically.

- Babel provides translation support in templates.

# TWO

# API

This document describes the API to Jinja and not the template language (for that, see *Template Designer Documentation*). It will be most useful as reference to those implementing the template interface to the application and not those who are creating Jinja templates.

## 2.1 Basics

Jinja 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 application initialization and use that to load templates. In some cases however, it's useful to have multiple environments side by side, if different configurations are in use.

The simplest way to configure Jinja to load templates for your application is to use *PackageLoader*.

```python
from jinja2 import Environment, PackageLoader, select_autoescape
env = Environment(
    loader=PackageLoader("yourapp"),
    autoescape=select_autoescape()
)
```

This will create a template environment with a loader that looks up templates in the `templates` folder inside the `yourapp` Python package (or next to the `yourapp.py` Python module). It also enables autoescaping for HTML files. This loader only requires that `yourapp` is importable, it figures out the absolute path to the folder for you.

Different loaders are available to load templates in other ways or from other locations. They're listed in the *Loaders* section below. You can also write your own if you want to load templates from a source that's more specialized to your project.

To load a template from this environment, call the `get_template()` method, which returns the loaded *Template*.

```python
template = env.get_template("mytemplate.html")
```

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

```python
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 being a lot easier to use it also enables template inheritance.

**Notes on Autoescaping**

In future versions of Jinja we might enable autoescaping by default for security reasons. As such you are encouraged to explicitly configure autoescaping now instead of relying on the default.

## 2.2 High Level API

The high-level API is the API you will use in the application to load and render Jinja templates. The *Low Level API* on the other side is only useful if you want to dig deeper into Jinja or *develop extensions*.

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

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

Here 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 based statements. See also *Line Statements*.
>
> *line_comment_prefix*
>> If given and a string, this will be used as prefix for line based comments. See also *Line Statements*.
>>
>> New in version 2.2.
>
> *trim_blocks*
>> If this is set to `True` the first newline after a block is removed (block, not variable tag!). Defaults to *False*.
>
> *lstrip_blocks*
>> If this is set to `True` leading spaces and tabs are stripped from 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 a useful default for Linux and OS X systems as well as web applications.
>
> *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 paths as strings or extension classes. For more information have a look at *the extensions documentation*.

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

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

*finalize*
> A callable that can be used to process the result of a variable expression 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 by default. For more details about autoescaping see `Markup`. As of Jinja 2.4 this can also be a callable that is passed the template name and has to return `True` or `False` depending on autoescape should be enabled 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 means that if more than 400 templates are loaded the loader will clean out the least recently used template. If the cache size is set to `0` templates are recompiled all the time, if the cache size is `-1` the cache will not be cleaned.
>
> 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 template sources may change (ie: file system or database). If `auto_reload` is set to `True` (default) every time a template is requested the loader checks if the source changed and if yes, it will reload the template. For higher performance it's possible to disable that.

*bytecode_cache*
> If set to a bytecode cache object, this object will provide a cache for the internal Jinja bytecode so that templates don't have 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 allows using async functions and generators.

**Parameters**

- **block_start_string** (*str*) –
- **block_end_string** (*str*) –
- **variable_start_string** (*str*) –
- **variable_end_string** (*str*) –
- **comment_start_string** (*str*) –
- **comment_end_string** (*str*) –
- **line_statement_prefix** (*Optional[str]*) –
- **line_comment_prefix** (*Optional[str]*) –

- **trim_blocks** (*bool*) –
- **lstrip_blocks** (*bool*) –
- **newline_sequence** (*te.Literal['\n', '\r\n', '\r']*) –
- **keep_trailing_newline** (*bool*) –
- **extensions** (*Sequence[Union[str, Type[Extension]]]*) –
- **optimized** (*bool*) –
- **undefined** (*Type[Undefined]*) –
- **finalize** (*Optional[Callable[[...], Any]]*) –
- **autoescape** (*Union[bool, Callable[[Optional[str]], bool]]*) –
- **loader** (*Optional[BaseLoader]*) –
- **cache_size** (*int*) –
- **auto_reload** (*bool*) –
- **bytecode_cache** (*Optional[BytecodeCache]*) –
- **enable_async** (*bool*) –

**shared**

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

**sandboxed**

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

**filters**

> A dict of filters for this environment. As long as no template was loaded it's safe to add new filters or remove old. For custom filters see *Custom Filters*. For valid filter names have a look at *Notes on Identifiers*.

**tests**

> A dict of test functions for this environment. As long as no template was loaded it's safe to modify this dict. For custom tests see *Custom Tests*. For valid test names have a look at *Notes on Identifiers*.

**globals**

> A dict of variables that are available in every template loaded by the environment. As long as no template was loaded it's safe to modify this. For more details see *The Global Namespace*. For valid object names see *Notes on Identifiers*.

**policies**

> A dictionary with *Policies*. These can be reconfigured to change the runtime behavior or certain template features. Usually these are security related.

**code_generator_class**

> The class used for code generation. This should not be changed in most cases, unless you need to modify the Python code a template compiles to.

**context_class**

> The context used for templates. This should not be changed in most cases, unless you need to modify internals of how template variables are handled. For details, see *Context*.

**overlay**([*options*])

> Create a new overlay environment that shares all the data with the current environment except for cache and the overridden attributes. Extensions cannot be removed for an overlayed environment. An overlayed environment automatically gets all the extensions of the environment it is linked to plus optional extra extensions.
>
> Creating overlays should happen after the initial environment was set up completely. Not all attributes are truly linked, some are just copied over so modifications on the original environment may not shine through.
>
> Changed in version 3.1.2: Added the `newline_sequence,`, `keep_trailing_newline`, and `enable_async` parameters to match `__init__`.

> **Parameters**
>
> - **block_start_string** (*str*) –
> - **block_end_string** (*str*) –
> - **variable_start_string** (*str*) –
> - **variable_end_string** (*str*) –
> - **comment_start_string** (*str*) –
> - **comment_end_string** (*str*) –
> - **line_statement_prefix** (*Optional[str]*) –
> - **line_comment_prefix** (*Optional[str]*) –
> - **trim_blocks** (*bool*) –
> - **lstrip_blocks** (*bool*) –
> - **newline_sequence** (*te.Literal['\n', '\r\n', '\r']*) –
> - **keep_trailing_newline** (*bool*) –
> - **extensions** (*Sequence[Union[str, Type[Extension]]]*) –
> - **optimized** (*bool*) –
> - **undefined** (*Type[Undefined]*) –
> - **finalize** (*Optional[Callable[[...], Any]]*) –
> - **autoescape** (*Union[bool, Callable[[Optional[str]], bool]]*) –
> - **loader** (*Optional[BaseLoader]*) –
> - **cache_size** (*int*) –
> - **auto_reload** (*bool*) –
> - **bytecode_cache** (*Optional[BytecodeCache]*) –
> - **enable_async** (*bool*) –
>
> **Return type**
> > *Environment*

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

> Creates a new *Undefined* object for *name*. This is useful for filters or functions that may return undefined objects for some operations. All parameters except of *hint* should be provided as keyword parameters for better readability. The *hint* is used as error message for the exception if provided, otherwise the error message will be generated from *obj* and *name* automatically. The exception provided as *exc* is raised if

something with the generated undefined object is done that the undefined object does not allow. The default exception is *UndefinedError*. If a *hint* is provided the *name* may be omitted.

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

```
return environment.undefined(name='some_name')
```

This means that the name *some_name* is not defined. If the name was from an attribute of an object it makes sense to tell the undefined object the holder object to improve the error message:

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

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

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

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

**add_extension**(*extension*)

Adds an extension after the environment was created.

New in version 2.5.

> **Parameters**
> > **extension** (*Union[str, Type[Extension]]*) –
>
> **Return type**
> > None

**compile_expression**(*source*, *undefined_to_none=True*)

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

This is useful if applications want to use the same rules as Jinja in template "configuration files" or similar situations.

Example usage:

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

Per default the return value is converted to *None* if the expression returns an undefined value. This can be changed by setting *undefined_to_none* to *False*.

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

New in version 2.1.

**Parameters**

- **source** (*str*) –

- **undefined_to_none** (*bool*) –

**Return type**
*TemplateExpression*

**compile_templates**(*target*, *extensions=None*, *filter_func=None*, *zip='deflated'*, *log_function=None*,
                *ignore_errors=True*)

Finds all the templates the loader can find, compiles them and stores them in *target*. If *zip* is *None*, instead
of in a zipfile, the templates will be stored in a directory. By default a deflate zip algorithm is used. To
switch to the 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 or zipfile.

By default template compilation errors are ignored. In case a log function is provided, errors are logged.
If you want template syntax errors to abort the compilation you can set *ignore_errors* to *False* and you will
get an exception on syntax errors.

New in version 2.4.

**Parameters**

- **target** (*Union[str, PathLike]*) –

- **extensions** (*Optional[Collection[str]]*) –

- **filter_func** (*Optional[Callable[[str], bool]]*) –

- **zip** (*Optional[str]*) –

- **log_function** (*Optional[Callable[[str], None]]*) –

- **ignore_errors** (*bool*) –

**Return type**
None

**extend**(*\*\*attributes*)

Add the items to the instance of the environment if they do not exist yet. This is used by *extensions* to
register callbacks and configuration values without breaking inheritance.

**Parameters**
**attributes** (*Any*) –

**Return type**
None

**from_string**(*source*, *globals=None*, *template_class=None*)

Load a template from a source string without using `loader`.

**Parameters**

- **source** (*Union[str, Template]*) – Jinja source to compile into a template.

- **globals** (*Optional[MutableMapping[str, Any]]*) – Extend the environment
*globals* with these extra variables available for all renders of this template. If the template
has already been loaded and cached, its globals are updated with any new items.

- **template_class** (*Optional[Type[Template]]*) – Return an instance of this
*Template* class.

> **Return type**
>> Template

**get_or_select_template**(*template_name_or_list*, *parent=None*, *globals=None*)

> Use *select_template()* if an iterable of template names is given, or *get_template()* if one name is given.

> New in version 2.3.

>> **Parameters**

>>> • **template_name_or_list** (*Union[str, Template, List[Union[str, Template]]]*) –

>>> • **parent** (*Optional[str]*) –

>>> • **globals** (*Optional[MutableMapping[str, Any]]*) –

>> **Return type**
>>> Template

**get_template**(*name*, *parent=None*, *globals=None*)

> Load a template by name with `loader` and return a *Template*. If the template does not exist a *TemplateNotFound* exception is raised.

>> **Parameters**

>>> • **name** (*Union[str, Template]*) – Name of the template to load. When loading templates from the filesystem, "/" is used as the path separator, even on Windows.

>>> • **parent** (*Optional[str]*) – The name of the parent template importing this template. *join_path()* can be used to implement name transformations with this.

>>> • **globals** (*Optional[MutableMapping[str, Any]]*) – Extend the environment *globals* with these extra variables available for all renders of this template. If the template has already been loaded and cached, its globals are updated with any new items.

>> **Return type**
>>> Template

> Changed in version 3.0: If a template is loaded from cache, `globals` will update the template's globals instead of ignoring the new values.

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

**join_path**(*template*, *parent*)

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

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

>> **Parameters**

>>> • **template** (*str*) –

>>> • **parent** (*str*) –

>> **Return type**
>>> str

**list_templates**(*extensions=None*, *filter_func=None*)

> Returns a list of templates for this environment. This requires that the loader supports the loader's `list_templates()` method.
>
> If there are other files in the template folder besides the actual templates, the returned list can be filtered. There are two ways: either *extensions* is set to a list of file extensions for templates, or a *filter_func* can be provided which is a callable that is passed a template name and should return *True* if it should end up in the result list.
>
> If the loader does not support that, a `TypeError` is raised.
>
> New in version 2.4.
>
> > **Parameters**
> >
> > - **extensions** (`Optional[Collection[str]]`) –
> >
> > - **filter_func** (`Optional[Callable[[str], bool]]`) –
> >
> > **Return type**
> > > *List*[str]

**select_template**(*names*, *parent=None*, *globals=None*)

> Like *get_template()*, but tries loading multiple names. If none of the names can be loaded a *TemplatesNotFound* exception is raised.
>
> > **Parameters**
> >
> > - **names** (`Iterable[Union[str, Template]]`) – List of template names to try loading in order.
> >
> > - **parent** (`Optional[str]`) – The name of the parent template importing this template. *join_path()* can be used to implement name transformations with this.
> >
> > - **globals** (`Optional[MutableMapping[str, Any]]`) – Extend the environment *globals* with these extra variables available for all renders of this template. If the template has already been loaded and cached, its globals are updated with any new items.
> >
> > **Return type**
> > > Template
>
> Changed in version 3.0: If a template is loaded from cache, `globals` will update the template's globals instead of ignoring the new values.
>
> Changed in version 2.11: If `names` is *Undefined*, an *UndefinedError* is raised instead. If no templates were found and `names` contains *Undefined*, the message is more helpful.
>
> Changed in version 2.4: If `names` contains a *Template* object it is returned unchanged.
>
> New in version 2.3.

**class** jinja2.**Template**(*source*, *block_start_string='{%'*, *block_end_string='%}'*, *variable_start_string='{{'*, *variable_end_string='}}'*, *comment_start_string='{#'*, *comment_end_string='#}'*, *line_statement_prefix=None*, *line_comment_prefix=None*, *trim_blocks=False*, *lstrip_blocks=False*, *newline_sequence='\n'*, *keep_trailing_newline=False*, *extensions=()*, *optimized=True*, *undefined=<class 'jinja2.runtime.Undefined'>*, *finalize=None*, *autoescape=False*, *enable_async=False*)

> A compiled template that can be rendered.
>
> Use the methods on *Environment* to create or load templates. The environment is used to configure how templates are compiled and behave.

---

It is also possible to create a template object directly. This is not usually recommended. The constructor takes most of the same arguments as *Environment*. All templates created with the same environment arguments share the same ephemeral `Environment` instance behind the scenes.

A template object should be considered immutable. Modifications on the object are not supported.

> **Parameters**
> - **source** (*Union[str, Template]*) –
> - **block_start_string** (*str*) –
> - **block_end_string** (*str*) –
> - **variable_start_string** (*str*) –
> - **variable_end_string** (*str*) –
> - **comment_start_string** (*str*) –
> - **comment_end_string** (*str*) –
> - **line_statement_prefix** (*Optional[str]*) –
> - **line_comment_prefix** (*Optional[str]*) –
> - **trim_blocks** (*bool*) –
> - **lstrip_blocks** (*bool*) –
> - **newline_sequence** (*te.Literal['\n', '\r\n', '\r']*) –
> - **keep_trailing_newline** (*bool*) –
> - **extensions** (*Sequence[Union[str, Type[Extension]]]*) –
> - **optimized** (*bool*) –
> - **undefined** (*Type[Undefined]*) –
> - **finalize** (*Optional[Callable[[...], Any]]*) –
> - **autoescape** (*Union[bool, Callable[[Optional[str]], bool]]*) –
> - **enable_async** (*bool*) –
>
> **Return type**
> > *Any*

**globals**

> A dict of variables that are available every time the template is rendered, without needing to pass them during render. This should not be modified, as depending on how the template was loaded it may be shared with the environment and other templates.
>
> Defaults to *Environment.globals* unless extra values are passed to *Environment.get_template()*.
>
> Globals are only intended for data that is common to every render of the template. Specific data should be passed to *render()*.
>
> See *The Global Namespace*.

**name**

> The loading name of the template. If the template was loaded from a string this is *None*.

**filename**

> The filename of the template on the file system if it was loaded from there. 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 arguments are given the context will be empty. These two calls do the same:

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

    This will return the rendered template as a string.

        **Parameters**

            • **args** (*Any*) –

            • **kwargs** (*Any*) –

        **Return type**

            str

**generate**(\[*context*\])

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

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

        **Parameters**

            • **args** (*Any*) –

            • **kwargs** (*Any*) –

        **Return type**

            *Iterator*[str]

**stream**(\[*context*\])

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

        **Parameters**

            • **args** (*Any*) –

            • **kwargs** (*Any*) –

        **Return type**

            TemplateStream

**async render_async**(\[*context*\])

    This works similar to *render()* but returns a coroutine that when awaited returns the entire rendered template string. This requires the async feature to be enabled.

    Example usage:

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

        **Parameters**

            • **args** (*Any*) –

            • **kwargs** (*Any*) –

        **Return type**

            str

async generate_async([*context*])

> An async version of *generate()*. Works very similarly but returns an async iterator instead.
>
> > **Parameters**
> >
> > > - **args** (*Any*) –
> > >
> > > - **kwargs** (*Any*) –
> >
> > **Return type**
> > > *AsyncIterator*[str]

make_module(*vars=None*, *shared=False*, *locals=None*)

> This method works like the *module* attribute when called without arguments but it will evaluate the template on every call rather than caching it. It's also possible to provide a dict which is then used as context. The arguments are the same as for the *new_context()* method.
>
> > **Parameters**
> >
> > > - **vars** (*Optional[Dict[str, Any]]*) –
> > >
> > > - **shared** (*bool*) –
> > >
> > > - **locals** (*Optional[Mapping[str, Any]]*) –
> >
> > **Return type**
> > > *TemplateModule*

property module: TemplateModule

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

```
>>> t = Template('{% macro foo() %}42{% endmacro %}23')
>>> 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 generator but it can buffer multiple items to reduce the number of total iterations. Per default the output is unbuffered which means that for every unbuffered instruction in the template one string is yielded.
>
> If buffering is enabled with a buffer size of 5, five items are combined into a new string. This is mainly useful if you are streaming big templates to a client via WSGI which flushes after each iteration.
>
> > **Parameters**
> > > **gen** (*Iterator[str]*) –

disable_buffering()

> Disable the output buffering.
>
> > **Return type**
> > > None

dump(*fp*, *encoding=None*, *errors='strict'*)

> Dump the complete stream into a file or file-like object. Per default strings are written, if you want to encode before writing specify an *encoding*.
>
> Example usage:

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

> **Parameters**
>
> - **fp** (*Union[str, IO]*) –
>
> - **encoding** (*Optional[str]*) –
>
> - **errors** (*Optional[str]*) –
>
> **Return type**
> None

**enable_buffering**(*size=5*)

> Enable buffering. Buffer *size* items before yielding them.
>
> **Parameters**
> **size** (*int*) –
>
> **Return type**
> None

## 2.3 Autoescaping

Changed in version 2.4.

Jinja now comes with autoescaping support. As of Jinja 2.9 the autoescape extension is removed and built-in. However autoescaping is not yet enabled by default though this will most likely change in the future. It's recommended to configure a sensible default for autoescaping. This makes it possible to enable and disable autoescaping on a per-template basis (HTML versus text for instance).

jinja2.**select_autoescape**(*enabled_extensions=('html', 'htm', 'xml')*, *disabled_extensions=()*, *default_for_string=True*, *default=False*)

> Intelligently sets the initial value of autoescaping based on the filename of the template. This is the recommended way to configure autoescaping if you do not want to write a custom function yourself.
>
> If you want to enable it for all templates created from strings or for all templates with *.html* and *.xml* extensions:

```
from jinja2 import Environment, select_autoescape
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 template ends 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 that autoescaping should be enabled for. Likewise *disabled_extensions* is a list of all templates it should be disabled for. If a template is loaded from a string then

the default from *default_for_string* is used. If nothing matches then the initial value of autoescaping is set to the value of *default*.

For security reasons this function operates case insensitive.

New in version 2.9.

> **Parameters**
>
> > - **enabled_extensions** (*Collection[str]*) –
> >
> > - **disabled_extensions** (*Collection[str]*) –
> >
> > - **default_for_string** (*bool*) –
> >
> > - **default** (*bool*) –
>
> **Return type**
> > *Callable*[[*Optional*[str]], bool]

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

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

The `select_autoescape()` function returns a function that works roughly like this:

```python
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 also accept *None* as valid template name. This will be passed when generating templates from strings. You should always configure autoescaping as defaults in the future might change.

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

## 2.4 Notes on Identifiers

Jinja uses Python naming rules. Valid identifiers can be any combination of characters accepted by Python.

Filters and tests are looked up in separate namespaces and have slightly modified identifier syntax. Filters and tests may contain dots to group filters and tests by topic. For example it's perfectly valid to add a function into the filter dict and call it *to.str*. The regular expression for filter and test identifiers is `[a-zA-Z_][a-zA-Z0-9_]*(\.[a-zA-Z_][a-zA-Z0-9_]*)*`.

## 2.5 Undefined Types

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

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

**class** jinja2.**Undefined**

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

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

**Parameters**

- **hint** (*Optional[str]*) –

- **obj** (*Any*) –

- **name** (*Optional[str]*) –

- **exc** (*Type[TemplateRuntimeError]*) –

**_undefined_hint**

Either *None* or a string with the error message for the undefined object.

**_undefined_obj**

Either *None* or the owner object that caused the undefined object to 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. This is 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 generated from the undefined hints stored on the undefined object.

**class** jinja2.**ChainableUndefined**

An undefined that is chainable, where both `__getattr__` and `__getitem__` return itself rather than raising an `UndefinedError`.

```
>>> foo = ChainableUndefined(name='foo')
>>> str(foo.bar['baz'])
''
>>> foo.bar['baz'] + 42
Traceback (most recent call last):
  ...
jinja2.exceptions.UndefinedError: 'foo' is undefined
```

New in version 2.11.0.

> **Parameters**
>
> > - **hint** (*Optional[str]*) –
> >
> > - **obj** (*Any*) –
> >
> > - **name** (*Optional[str]*) –
> >
> > - **exc** (*Type[TemplateRuntimeError]*) –

**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
```

> **Parameters**
>
> > - **hint** (*Optional[str]*) –
> >
> > - **obj** (*Any*) –
> >
> > - **name** (*Optional[str]*) –
> >
> > - **exc** (*Type[TemplateRuntimeError]*) –

**class** jinja2.**StrictUndefined**

> An undefined that barks on print and iteration as well as boolean tests and all kinds of comparisons. In other words: you can do nothing with it except checking if it's defined using the *defined* test.

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

```
Traceback (most recent call last):
  ...
jinja2.exceptions.UndefinedError: 'foo' is undefined
```

> **Parameters**
>
> > - **hint** (*Optional[str]*) –
> >
> > - **obj** (*Any*) –
> >
> > - **name** (*Optional[str]*) –
> >
> > - **exc** (*Type[TemplateRuntimeError]*) –

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

jinja2.**make_logging_undefined**(*logger=None*, *base=<class 'jinja2.runtime.Undefined'>*)

> Given a logger object this returns a new undefined class that will log certain failures. It will log iterations and printing. If no logger is given a default logger is created.
>
> Example:

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

> New in version 2.8.
>
> > **Parameters**
> >
> > > - **logger** (*Optional[logging.Logger]*) – the logger to use. If not provided, a default logger is created.
> > >
> > > - **base** (*Type[Undefined]*) – the base class to add logging functionality to. This defaults to *Undefined*.
> >
> > **Return type**
> >
> > > *Type*[Undefined]

Undefined objects are created by calling `undefined`.

---

**Implementation**

*Undefined* is implemented by overriding the special `__underscore__` methods. For example the default *Undefined* class implements `__str__` to returns an empty string, while `__int__` and others fail with an exception. To allow conversion to int by returning `0` you can implement your own subclass.

```python
class NullUndefined(Undefined):
    def __int__(self):
        return 0

    def __float__(self):
        return 0.0
```

To disallow a method, override it and raise _undefined_exception. Because this is very common there is the helper method _fail_with_undefined_error() that raises the error with the correct information. Here's a class that works like the regular *Undefined* but fails on iteration:

```python
class NonIterableUndefined(Undefined):
    def __iter__(self):
        self._fail_with_undefined_error()
```

## 2.6 The Context

**class** jinja2.runtime.**Context**

The template context holds the variables of a template. It stores the values passed to the template and also the names the template exports. Creating instances is neither supported nor useful as it's created automatically at various stages of the template evaluation and should not be created by hand.

The context is immutable. Modifications on *parent* **must not** happen and modifications on *vars* are allowed from generated template code only. Template filters and global functions marked as pass_context() get the active context passed as first argument and 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.

**Parameters**

- **environment** (*Environment*) –

- **parent** (*Dict[str, Any]*) –

- **name** (*Optional[str]*) –

- **blocks** (*Dict[str, Callable[[Context], Iterator[str]]]*) –

- **globals** (*Optional[MutableMapping[str, Any]]*) –

**parent**

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

**vars**

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

**environment**

The environment that loaded the template.

**exported_vars**

This set contains all the names the template exports. The values for the names are in the *vars* dict. In order to get a copy of the exported 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 keys in this dict are the names of the blocks, and the values a list of blocks registered. The last item in each list is the current active block (latest in the inheritance chain).

**eval_ctx**

The current *Evaluation Context*.

**call**(*callable*, *\*args*, *\*\*kwargs*)

Call the callable with the arguments and keyword arguments provided but inject the active context or environment as first argument if the callable has `pass_context()` or `pass_environment()`.

> **Parameters**
>
> - **_Context__obj** (`Callable`) –
>
> - **args** (*Any*) –
>
> - **kwargs** (*Any*) –
>
> **Return type**
> *Union*[*Any*, Undefined]

**get**(*key*, *default=None*)

Look up a variable by name, or return a default if the key is not found.

> **Parameters**
>
> - **key** (`str`) – The variable name to look up.
>
> - **default** (`Optional[Any]`) – The value to return if the key is not found.
>
> **Return type**
> *Any*

**get_all**()

Return the complete context as dict including the exported variables. For optimizations reasons this might not return an actual copy so be careful with using it.

> **Return type**
> *Dict*[str, *Any*]

**get_exported**()

Get a new dict with the exported variables.

> **Return type**
> *Dict*[str, *Any*]

**resolve**(*key*)

Look up a variable by name, or return an `Undefined` object if the key is not found.

If you need to add custom behavior, override `resolve_or_missing()`, not this method. The various lookup functions use that method, not this one.

> **Parameters**
> **key** (`str`) – The variable name to look up.
>
> **Return type**
> *Union*[*Any*, Undefined]

**resolve_or_missing**(*key*)

> Look up a variable by name, or return a `missing` sentinel if the key is not found.
>
> Override this method to add custom lookup behavior. *resolve()*, *get()*, and **__getitem__**() use this method. Don't call this method directly.
>
> > **Parameters**
> > > **key** (*str*) – The variable name to look up.
> >
> > **Return type**
> > > *Any*

The context is immutable, it prevents modifications, and if it is modified somehow despite that those changes may not show up. For performance, Jinja does not use the context as data storage for, only as a primary data source. Variables that the template does not define are looked up in the context, but variables the template does define are stored locally.

Instead of modifying the context directly, a function should return a value that can be assigned to a variable within the template itself.

```
{% set comments = get_latest_comments() %}
```

## 2.7 Loaders

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

**class** jinja2.**BaseLoader**

> Baseclass for all loaders. Subclass this and override *get_source* to implement a custom loading mechanism. The environment provides a *get_template* method that calls the loader's *load* method to get the *Template* object.
>
> A very basic example for a loader that looks up templates on the file system could look like this:

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

class MyLoader(BaseLoader):

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

    def get_source(self, environment, template):
        path = join(self.path, template)
        if not exists(path):
            raise TemplateNotFound(template)
        mtime = getmtime(path)
        with open(path) as f:
            source = f.read()
        return source, path, lambda: mtime == getmtime(path)
```

**get_source**(*environment*, *template*)

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

The source part of the returned tuple must be the source of the template as a string. The filename should be the name of the file on the filesystem if it was loaded from there, otherwise None. The filename is used by Python for the tracebacks if no loader extension is used.

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

> **Parameters**
>
> - **environment** (*Environment*) –
> - **template** (*str*) –
>
> **Return type**
> *Tuple*[str, *Optional*[str], *Optional*[*Callable*[[], bool]]]

**load**(*environment*, *name*, *globals=None*)

Loads a template. This method looks up the template in the cache or loads one by calling `get_source()`. Subclasses should not override this method as loaders working on collections of other loaders (such as *PrefixLoader* or *ChoiceLoader*) will not call this method but *get_source* directly.

> **Parameters**
>
> - **environment** (*Environment*) –
> - **name** (*str*) –
> - **globals** (*Optional[MutableMapping[str, Any]]*) –
>
> **Return type**
> *Template*

Here a list of the builtin loaders Jinja provides:

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

Load templates from a directory in the file system.

The path can be relative or absolute. Relative paths are relative to the current working directory.

```
loader = FileSystemLoader("templates")
```

A list of paths can be given. The directories will be searched in order, stopping at the first matching template.

```
loader = FileSystemLoader(["/override/templates", "/default/templates"])
```

> **Parameters**
>
> - **searchpath** (*Union[str, PathLike, Sequence[Union[str, PathLike]]]*) – A path, or list of paths, to the directory that contains the templates.
> - **encoding** (*str*) – Use this encoding to read the text from template files.
> - **followlinks** (*bool*) – Follow symbolic links in the path.

Changed in version 2.8: Added the followlinks parameter.

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

Load templates from a directory in a Python package.

> **Parameters**
>
> - **package_name** (*str*) – Import name of the package that contains the template directory.

- **package_path** (`str`) – Directory within the imported package that contains the templates.
- **encoding** (`str`) – Encoding of template files.

The following example looks up templates in the `pages` directory within the `project.ui` package.

```
loader = PackageLoader("project.ui", "pages")
```

Only packages installed as directories (standard pip behavior) or zip/egg files (less common) are supported. The Python API for introspecting data in packages is too limited to support other installation methods the way this loader requires.

There is limited support for **PEP 420** namespace packages. The template directory is assumed to only be in one namespace contributor. Zip files contributing to a namespace are not supported.

Changed in version 3.0: No longer uses `setuptools` as a dependency.

Changed in version 3.0: Limited PEP 420 namespace package support.

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

Loads a template from a Python dict mapping template names to template source. This loader is useful for unittesting:

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

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

> **Parameters**
> **mapping** (`Mapping[str, str]`) –

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

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

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

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

> **Parameters**
> **load_func** (`Callable[[str], Optional[Union[str, Tuple[str, Optional[str], Optional[Callable[[], bool]]]]]]`) –

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

A loader that is passed a dict of loaders where each loader is bound to a prefix. The prefix is delimited from the template by a slash per default, which can be changed by setting the *delimiter* argument to something 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.

> **Parameters**
>
> - **mapping** (*Mapping[str, BaseLoader]*) –
> - **delimiter** (*str*) –

**class** jinja2.**ChoiceLoader**(*loaders*)

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

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

> This is useful if you want to allow users to override builtin templates from a different location.
>
> **Parameters**
> > **loaders** (*Sequence[BaseLoader]*) –

**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()*.
>
> **Parameters**
> > **path** (*Union[str, PathLike, Sequence[Union[str, PathLike]]]*) –

## 2.8 Bytecode Cache

Jinja 2.1 and higher support external bytecode caching. Bytecode caches make it possible to store the generated byte-code on the file system or a different location to avoid parsing the templates on first use.

This is especially useful if you have a web application that is initialized on the first request and Jinja compiles many templates at once which slows down the 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 class and override *load_bytecode()* and *dump_bytecode()*. Both of these 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 load_bytecode(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 of Jinja.

**clear**()

> Clears the cache. This method is not used by Jinja but should be implemented to allow applications to clear the bytecode cache used by a particular environment.
>
> > **Return type**
> > None

**dump_bytecode**(*bucket*)

> Subclasses have to override this method to write the bytecode from a bucket back to the cache. If it unable to do so it must not fail silently but raise an exception.
>
> > **Parameters**
> > **bucket** (Bucket) –
> >
> > **Return type**
> > None

**load_bytecode**(*bucket*)

> Subclasses have to override this method to load bytecode into a bucket. If they are not able to find code in the cache for the bucket, it must not do anything.
>
> > **Parameters**
> > **bucket** (Bucket) –
> >
> > **Return type**
> > None

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

> Buckets are used to store the bytecode for one template. It's created and initialized by the bytecode cache and passed to the loading functions.
>
> The buckets get an internal checksum from the cache assigned and use this to automatically reject outdated cache material. Individual bytecode cache subclasses don't have to care about cache invalidation.
>
> > **Parameters**
> >
> > - **environment** (Environment) –
> >
> > - **key** (str) –
> >
> > - **checksum** (str) –

**environment**

> The Environment that created the bucket.

---

**key**

> The unique cache key for this bucket

**code**

> The bytecode if it's loaded, otherwise *None*.

**bytecode_from_string**(*string*)

> Load bytecode from bytes.
>
> > **Parameters**
> > > **string** (*bytes*) –
> >
> > **Return type**
> > > None

**bytecode_to_string**()

> Return the bytecode as bytes.
>
> > **Return type**
> > > bytes

**load_bytecode**(*f*)

> Loads bytecode from a file or file like object.
>
> > **Parameters**
> > > **f** (*BinaryIO*) –
> >
> > **Return type**
> > > None

**reset**()

> Resets the bucket (unloads the bytecode).
>
> > **Return type**
> > > None

**write_bytecode**(*f*)

> Dump the bytecode into the file or file like object passed.
>
> > **Parameters**
> > > **f** (*IO[bytes]*) –
> >
> > **Return type**
> > > None

Builtin bytecode caches:

**class** jinja2.**FileSystemBytecodeCache**(*directory=None*, *pattern='__jinja2_%s.cache'*)

> A bytecode cache that stores bytecode on the filesystem. It accepts two arguments: The directory where the cache items are stored and a pattern string that is used to build the filename.
>
> If no directory is specified a default cache directory is selected. On Windows the user's temp directory is used, on UNIX systems a directory is created for the user in the system temp directory.
>
> The pattern can be used to have multiple separate caches operate on the same directory. The default pattern is `'__jinja2_%s.cache'`. `%s` is replaced with the cache key.
>
> ```
> >>> bcc = FileSystemBytecodeCache('/tmp/jinja_cache', '%s.cache')
> ```
>
> This bytecode cache supports clearing of the cache using the clear method.
>
> > **Parameters**

- **directory** (*Optional[str]*) –

- **pattern** (*str*) –

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

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

Libraries compatible with this class:

- cachelib

- python-memcached

(Unfortunately the django cache interface is not compatible because it does not support storing binary data, only text. You can however pass the underlying cache client to the bytecode cache which is available as *django.core.cache.cache._client*.)

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

> **Parameters**
>
>> - **client** (*_MemcachedClient*) –
>>
>> - **prefix** (*str*) –
>>
>> - **timeout** (*Optional[int]*) –
>>
>> - **ignore_memcache_errors** (*bool*) –

**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 provided a default timeout or no timeout should be assumed, if it's provided it's an integer with the number of seconds the cache item should exist.
>
> **get**(*key*)
>
>> Returns the value for the cache key. If the item does not exist in the cache the return value must be *None*.

The other arguments to the constructor are the prefix for all keys that is added before the actual cache key and the timeout for the bytecode in the 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.

## 2.9 Async Support

New in version 2.9.

Jinja supports the Python `async` and `await` syntax. For the template designer, this support (when enabled) is entirely transparent, templates continue to look exactly the same. However, developers should be aware of the implementation as it affects what types of APIs you can use.

By default, async support is disabled. Enabling it will cause the environment to compile different code behind the scenes in order to handle async and sync code in an asyncio event loop. This has the following implications:

- Template rendering requires an event loop to be available to the current thread. `asyncio.get_running_loop()` must return an event loop.

- The compiled code uses `await` for functions and attributes, and uses `async for` loops. In order to support using both async and sync functions in this context, a small wrapper is placed around all calls and access, which adds overhead compared to purely async code.

- Sync methods and filters become wrappers around their corresponding async implementations where needed. For example, `render` invokes `async_render`, and `|map` supports async iterables.

Awaitable objects can be returned from functions in templates and any function call in a template will automatically await the result. The `await` you would normally add in Python is implied. For example, you can provide a method that asynchronously loads data from a database, and from the template designer's point of view it can be called like any other function.

## 2.10 Policies

Starting with Jinja 2.9 policies can be configured on the environment which can slightly influence how filters and other template constructs behave. They can be configured with the *policies* attribute.

Example:

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

**`truncate.leeway:`**
> Configures the leeway default for the *truncate* filter. Leeway as introduced in 2.9 but to restore compatibility with older templates it can be configured to *0* to get the old behavior back. The default is *5*.

**`urlize.rel:`**
> A string that defines the items for the *rel* attribute of generated links with the *urlize* filter. These items are always added. The default is *noopener*.

**`urlize.target:`**
> The default target that is issued for links from the *urlize* filter if no other target is defined by the call explicitly.

**`urlize.extra_schemes:`**
> Recognize URLs that start with these schemes in addition to the default `http://`, `https://`, and `mailto:`.

**`json.dumps_function:`**
> If this is set to a value other than *None* then the *tojson* filter will dump with this function instead of the default one. Note that this function should accept arbitrary extra arguments which might be passed in the future from the filter. Currently the only argument that might be passed is *indent*. The default dump function is `json.dumps`.

**`json.dumps_kwargs:`**
> 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 the *i18n Extension* will always unify linebreaks and surrounding whitespace as if the *trimmed* modifier was used.

# 2.11 Utilities

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

jinja2.**pass_context**(*f*)

> Pass the *Context* as the first argument to the decorated function when called while rendering a template.
>
> Can be used on functions, filters, and tests.
>
> If only Context.eval_context is needed, use *pass_eval_context()*. If only Context.environment is needed, use *pass_environment()*.
>
> New in version 3.0.0: Replaces contextfunction and contextfilter.
>
> > **Parameters**
> > > **f** (*F*) –
> >
> > **Return type**
> > > *F*

jinja2.**pass_eval_context**(*f*)

> Pass the *EvalContext* as the first argument to the decorated function when called while rendering a template. See *Evaluation Context*.
>
> Can be used on functions, filters, and tests.
>
> If only EvalContext.environment is needed, use *pass_environment()*.
>
> New in version 3.0.0: Replaces evalcontextfunction and evalcontextfilter.
>
> > **Parameters**
> > > **f** (*F*) –
> >
> > **Return type**
> > > *F*

jinja2.**pass_environment**(*f*)

> Pass the *Environment* as the first argument to the decorated function when called while rendering a template.
>
> Can be used on functions, filters, and tests.
>
> New in version 3.0.0: Replaces environmentfunction and environmentfilter.
>
> > **Parameters**
> > > **f** (*F*) –
> >
> > **Return type**
> > > *F*

jinja2.**clear_caches**()

> Jinja keeps internal caches for environments and lexers. These are used so that Jinja doesn't have to recreate environments and lexers all the time. Normally you don't have to care about that but if you are measuring memory consumption you may want to clean the caches.
>
> > **Return type**
> > > None

jinja2.**is_undefined**(*obj*)

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

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

> **Parameters**
>> **obj** (*Any*) –
>
> **Return type**
>> bool

## 2.12 Exceptions

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

> Baseclass for all template errors.
>
>> **Parameters**
>>> **message** (*Optional[str]*) –
>>
>> **Return type**
>>> None

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

> Raised if a template tries to operate on *Undefined*.
>
>> **Parameters**
>>> **message** (*Optional[str]*) –
>>
>> **Return type**
>>> None

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

> Raised if a template does not exist.
>
> Changed in version 2.11: If the given name is *Undefined* and no message was provided, an *UndefinedError* is raised.
>
>> **Parameters**
>>
>>> - **name** (*Optional[Union[str, Undefined]]*) –
>>>
>>> - **message** (*Optional[str]*) –
>>
>> **Return type**
>>> None

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

> Like *TemplateNotFound* but raised if multiple templates are selected. This is a subclass of *TemplateNotFound* exception, so just catching the base exception will catch both.
>
> Changed in version 2.11: If a name in the list of names is *Undefined*, a message about it being undefined is shown rather than the empty string.

New in version 2.2.

> **Parameters**
>
> - **names** (*Sequence[Union[str, Undefined]]*) –
>
> - **message** (*Optional[str]*) –
>
> **Return type**
> None

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

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

> **Parameters**
>
> - **message** (*str*) –
>
> - **lineno** (*int*) –
>
> - **name** (*Optional[str]*) –
>
> - **filename** (*Optional[str]*) –
>
> **Return type**
> None

**message**

> The error message.

**lineno**

> The line number where the error occurred.

**name**

> The load name for the template.

**filename**

> The filename that loaded the template in the encoding of the file system (most likely utf-8, or mbcs on Windows systems).

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

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

> **Parameters**
> **message** (*Optional[str]*) –
>
> **Return type**
> None

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

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

> **Parameters**
>
> - **message** (*str*) –
>
> - **lineno** (*int*) –
>
> - **name** (*Optional[str]*) –
>
> - **filename** (*Optional[str]*) –

> **Return type**
>> None

## 2.13 Custom Filters

Filters are Python functions that take the value to the left of the filter as the first argument and produce a new value. Arguments passed to the filter are passed after the value.

For example, the filter {{ 42|myfilter(23) }} is called behind the scenes as `myfilter(42, 23)`.

Jinja comes with some *built-in filters*. To use a custom filter, write a function that takes at least a `value` argument, then register it in `Environment.filters`.

Here's a filter that formats datetime objects:

```python
def datetime_format(value, format="%H:%M %d-%m-%y"):
    return value.strftime(format)

environment.filters["datetime_format"] = datetime_format
```

Now it can be used in templates:

```
{{ article.pub_date|datetimeformat }}
{{ article.pub_date|datetimeformat("%B %Y") }}
```

Some decorators are available to tell Jinja to pass extra information to the filter. The object is passed as the first argument, making the value being filtered the second argument.

- *pass_environment()* passes the *Environment*.
- *pass_eval_context()* passes the *Evaluation Context*.
- *pass_context()* passes the current *Context*.

Here's a filter that converts line breaks into HTML <br> and <p> tags. It uses the eval context to check if autoescape is currently enabled before escaping the input and marking the output safe.

```python
import re
from jinja2 import pass_eval_context
from markupsafe import Markup, escape

@pass_eval_context
def nl2br(eval_ctx, value):
    br = "<br>\n"

    if eval_ctx.autoescape:
        value = escape(value)
        br = Markup(br)

    result = "\n\n".join(
        f"<p>{br.join(p.splitlines())}<p>"
        for p in re.split(r"(?:\r\n|\r(?!\n)|\n){2,}", value)
    )
    return Markup(result) if autoescape else result
```

## 2.14 Custom Tests

Test are Python functions that take the value to the left of the test as the first argument, and return `True` or `False`. Arguments passed to the test are passed after the value.

For example, the test `{{ 42 is even }}` is called behind the scenes as `is_even(42)`.

Jinja comes with some *built-in tests*. To use a custom tests, write a function that takes at least a `value` argument, then register it in `Environment.tests`.

Here's a test that checks if a value is a prime number:

```python
import math

def is_prime(n):
    if n == 2:
        return True

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

    return True

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

Now it can be used in templates:

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

Some decorators are available to tell Jinja to pass extra information to the filter. The object is passed as the first argument, making the value being filtered the second argument.

- `pass_environment()` passes the `Environment`.
- `pass_eval_context()` passes the *Evaluation Context*.
- `pass_context()` passes the current `Context`.

## 2.15 Evaluation Context

The evaluation context (short eval context or eval ctx) makes it possible to activate and deactivate compiled features at runtime.

Currently it is only used to enable and disable automatic escaping, but it can be used by extensions as well.

The `autoescape` setting should be checked on the evaluation context, not the environment. The evaluation context will have the computed value for the current template.

Instead of `pass_environment`:

```
@pass_environment
def filter(env, value):
    result = do_something(value)

    if env.autoescape:
        result = Markup(result)

    return result
```

Use `pass_eval_context` if you only need the setting:

```
@pass_eval_context
def filter(eval_ctx, value):
    result = do_something(value)

    if eval_ctx.autoescape:
        result = Markup(result)

    return result
```

Or use `pass_context` if you need other context behavior as well:

```
@pass_context
def filter(context, value):
    result = do_something(value)

    if context.eval_ctx.autoescape:
        result = Markup(result)

    return result
```

The evaluation context must not be modified at runtime. Modifications must only happen with a *nodes. EvalContextModifier* and *nodes.ScopedEvalContextModifier* from an extension, not on the eval context object itself.

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

> Holds evaluation time information. Custom attributes can be attached to it in extensions.

> > **Parameters**
> >
> > - **environment** (*Environment*) –
> > - **template_name** (*Optional[str]*) –

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

> **volatile**
> > *True* if the compiler cannot evaluate some expressions at compile time. At runtime this should always be *False*.

## 2.16 The Global Namespace

The global namespace stores variables and functions that should be available without needing to pass them to *Template.render()*. They are also available to templates that are imported or included without context. Most applications should only use *Environment.globals*.

*Environment.globals* are intended for data that is common to all templates loaded by that environment. *Template.globals* are intended for data that is common to all renders of that template, and default to *Environment.globals* unless they're given in *Environment.get_template()*, etc. Data that is specific to a render should be passed as context to *Template.render()*.

Only one set of globals is used during any specific rendering. If templates A and B both have template globals, and B extends A, then only B's globals are used for both when using `b.render()`.

Environment globals should not be changed after loading any templates, and template globals should not be changed at any time after loading the template. Changing globals after loading a template will result in unexpected behavior as they may be shared between the environment and other templates.

## 2.17 Low Level API

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

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

Lex the given sourcecode and return a generator that yields tokens as tuples in the form (`lineno, token_type, value`). This can be useful for *extension development* and debugging templates.

This does not perform preprocessing. If you want the preprocessing of the extensions to be applied you have to filter source through the *preprocess()* method.

> **Parameters**
> - **source** (*str*) –
> - **name** (*Optional[str]*) –
> - **filename** (*Optional[str]*) –
>
> **Return type**
> *Iterator*[*Tuple*[int, str, str]]

Environment.**parse**(*source*, *name=None*, *filename=None*)

Parse the sourcecode and return the abstract syntax tree. This tree of nodes is used by the compiler to convert the template into executable source- or bytecode. This is useful for debugging or to extract information from templates.

If you are *developing Jinja extensions* this gives you a good overview of the node tree generated.

> **Parameters**
> - **source** (*str*) –
> - **name** (*Optional[str]*) –
> - **filename** (*Optional[str]*) –
>
> **Return type**
> Template

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

> Preprocesses the source with all extensions. This is automatically called for all parsing and compiling methods but *not* for `lex()` because there you usually only want the actual source tokenized.
>
> > **Parameters**
> >
> > - **source** (*str*) –
> >
> > - **name** (*Optional[str]*) –
> >
> > - **filename** (*Optional[str]*) –
> >
> > **Return type**
> >
> > > str

Template.**new_context**(*vars=None*, *shared=False*, *locals=None*)

> Create a new `Context` for this template. The vars provided will be passed to the template. Per default the globals are added to the context. If shared is set to *True* the data is passed as is to the context without adding the globals.
>
> *locals* can be a dict of local variables for internal usage.
>
> > **Parameters**
> >
> > - **vars** (*Optional[Dict[str, Any]]*) –
> >
> > - **shared** (*bool*) –
> >
> > - **locals** (*Optional[Mapping[str, Any]]*) –
> >
> > **Return type**
> >
> > > Context

Template.**root_render_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 or a compatible template. This render function is generated by the compiler from the template code and returns a generator that yields strings.
>
> If an exception in the template code happens the template engine will not rewrite the exception but pass through the original one. As a matter 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 exactly like the `root_render_func()` with the same limitations.

Template.**is_up_to_date**

> This attribute is *False* if there is a newer version of the template available, otherwise *True*.

---

**Note**

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

---

## 2.18 The Meta API

New in version 2.2.

The meta API returns some information about abstract syntax trees that could help applications to implement more advanced template concepts. All the functions of the meta API operate on an abstract syntax tree as returned by the *Environment.parse()* method.

jinja2.meta.**find_undeclared_variables**(*ast*)

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

```
>>> from jinja2 import Environment, meta
>>> env = Environment()
>>> ast = env.parse('{% set foo = 42 %}{{ bar + foo }}')
>>> meta.find_undeclared_variables(ast) == {'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 of fact this function can currently raise that exception as well.

> > **Parameters**
> >     **ast** (*Template*) –
> >
> > **Return type**
> >     *Set*[str]

jinja2.meta.**find_referenced_templates**(*ast*)

> Finds all the referenced templates from the AST. This will return an iterator over all the hardcoded template extensions, inclusions and imports. If dynamic inheritance or inclusion is used, *None* will be yielded.

```
>>> 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 want to rebuild parts of the website after a layout template has changed.
>
> **Parameters**
>     **ast** (*Template*) –
>
> **Return type**
>     *Iterator*[*Optional*[str]]

# SANDBOX

The Jinja sandbox can be used to render untrusted templates. Access to attributes, method calls, operators, mutating data structures, and string formatting can be intercepted and prohibited.

```
>>> from jinja2.sandbox import SandboxedEnvironment
>>> env = SandboxedEnvironment()
>>> func = lambda: "Hello, Sandbox!"
>>> env.from_string("{{ func() }}").render(func=func)
'Hello, Sandbox!'
>>> env.from_string("{{ func.__code__.co_code }}").render(func=func)
Traceback (most recent call last):
  ...
SecurityError: access to attribute '__code__' of 'function' object is unsafe.
```

A sandboxed environment can be useful, for example, to allow users of an internal reporting system to create custom emails. You would document what data is available in the templates, then the user would write a template using that information. Your code would generate the report data and pass it to the user's sandboxed template to render.

## 3.1 Security Considerations

The sandbox alone is not a solution for perfect security. Keep these things in mind when using the sandbox.

Templates can still raise errors when compiled or rendered. Your code should attempt to catch errors instead of crashing.

It is possible to construct a relatively small template that renders to a very large amount of output, which could correspond to a high use of CPU or memory. You should run your application with limits on resources such as CPU and memory to mitigate this.

Jinja only renders text, it does not understand, for example, JavaScript code. Depending on how the rendered template will be used, you may need to do other postprocessing to restrict the output.

Pass only the data that is relevant to the template. Avoid passing global data, or objects with methods that have side effects. By default the sandbox prevents private and internal attribute access. You can override is_safe_attribute() to further restrict attributes access. Decorate methods with unsafe() to prevent calling them from templates when passing objects as data. Use ImmutableSandboxedEnvironment to prevent modifying lists and dictionaries.

# 3.2 API

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

> The sandboxed environment. It works like the regular environment but tells the compiler to generate sandboxed code. Additionally subclasses of this environment may override the methods that tell the runtime what attributes or functions are safe to access.
>
> If the template tries to access insecure code a *SecurityError* is raised. However also other exceptions may occur during the rendering so the caller has to ensure that all exceptions are caught.
>
> > **Parameters**
> >
> > - **args** (*Any*) –
> > - **kwargs** (*Any*) –

**call_binop**(*context*, *operator*, *left*, *right*)

> For intercepted binary operator calls (*intercepted_binops()*) this function is executed instead of the builtin operator. This can be used to fine tune the behavior of certain operators.
>
> New in version 2.6.
>
> > **Parameters**
> >
> > - **context** (*Context*) –
> > - **operator** (*str*) –
> > - **left** (*Any*) –
> > - **right** (*Any*) –
> >
> > **Return type**
> > *Any*

**call_unop**(*context*, *operator*, *arg*)

> For intercepted unary operator calls (*intercepted_unops()*) this function is executed instead of the builtin operator. This can be used to fine tune the behavior of certain operators.
>
> New in version 2.6.
>
> > **Parameters**
> >
> > - **context** (*Context*) –
> > - **operator** (*str*) –
> > - **arg** (*Any*) –
> >
> > **Return type**
> > *Any*

**default_binop_table**: `Dict[str, Callable[[Any, Any], Any]]` = {'%': <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 is available on each instance of a sandboxed environment as `binop_table`

**default_unop_table**: `Dict[str, Callable[[Any], Any]]` = {'+': <built-in function pos>, '-': <built-in function neg>}

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

**intercepted_binops:** `FrozenSet[str] = frozenset({})`

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

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

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

New in version 2.6.

**intercepted_unops:** `FrozenSet[str] = frozenset({})`

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

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

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

New in version 2.6.

**is_safe_attribute**(*obj*, *attr*, *value*)

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

> **Parameters**
>
> - **obj** (*Any*) –
> - **attr** (*str*) –
> - **value** (*Any*) –
>
> **Return type**
> bool

**is_safe_callable**(*obj*)

Check if an object is safely callable. By default callables are considered safe unless decorated with `unsafe()`.

This also recognizes the Django convention of setting `func.alters_data = True`.

> **Parameters**
> **obj** (*Any*) –
>
> **Return type**
> bool

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

Works exactly like the regular *SandboxedEnvironment* but does not permit modifications on the builtin mutable objects *list*, *set*, and *dict* by using the `modifies_known_mutable()` function.

> **Parameters**
>
> - **args** (*Any*) –

> - **kwargs** (*Any*) –

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

> Raised if a template tries to do something insecure if the sandbox is enabled.
>
> > **Parameters**
> > **message** (*Optional[str]*) –
> >
> > **Return type**
> > None

jinja2.sandbox.**unsafe**(*f*)

> Marks a function or method as unsafe.
>
> > **Parameters**
> > **f** (*F*) –
> >
> > **Return type**
> > F

jinja2.sandbox.**is_internal_attribute**(*obj*, *attr*)

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

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

> > **Parameters**
> >
> > - **obj** (*Any*) –
> >
> > - **attr** (*str*) –
> >
> > **Return type**
> > [bool](#)

jinja2.sandbox.**modifies_known_mutable**(*obj*, *attr*)

> This function checks if an attribute on a builtin mutable object (list, dict, set or deque) or the corresponding ABCs would modify it if called.

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

> If called with an unsupported object, `False` is returned.

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

> > **Parameters**

- **obj** (*Any*) –
- **attr** (*str*) –

**Return type**
    bool

## 3.3 Operator Intercepting

For performance, Jinja outputs operators directly when compiling. This means it's not possible to intercept operator behavior by overriding SandboxEnvironment.call by default, because operator special methods are handled by the Python interpreter, and might not correspond with exactly one method depending on the operator's use.

The sandbox can instruct the compiler to output a function to intercept certain operators instead. Override *SandboxedEnvironment.intercepted_binops* and *SandboxedEnvironment.intercepted_unops* with the operator symbols you want to intercept. The compiler will replace the symbols with calls to *SandboxedEnvironment.call_binop()* and *SandboxedEnvironment.call_unop()* instead. The default implementation of those methods will use SandboxedEnvironment.binop_table and SandboxedEnvironment.unop_table to translate operator symbols into operator functions.

For example, the power (**) operator can be disabled:

```python
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 super().call_binop(self, context, operator, left, right)
```

# 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 text files. For example, your code may have an intermediate step where users may use templates to define values that will then be passed to a traditional string environment.

## 4.1 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):
```

(continues on next page)

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

## 4.2 API

**class** jinja2.nativetypes.**NativeEnvironment**($\big[$*options*$\big]$)

An environment that renders templates to native Python types.

> **Parameters**
>
> - **block_start_string** (*str*) –
> - **block_end_string** (*str*) –
> - **variable_start_string** (*str*) –
> - **variable_end_string** (*str*) –
> - **comment_start_string** (*str*) –
> - **comment_end_string** (*str*) –
> - **line_statement_prefix** (*Optional[str]*) –
> - **line_comment_prefix** (*Optional[str]*) –
> - **trim_blocks** (*bool*) –
> - **lstrip_blocks** (*bool*) –
> - **newline_sequence** (*te.Literal['\n', '\r\n', '\r']*) –
> - **keep_trailing_newline** (*bool*) –
> - **extensions** (*Sequence[Union[str, Type[Extension]]]*) –
> - **optimized** (*bool*) –
> - **undefined** (*Type[Undefined]*) –
> - **finalize** (*Optional[Callable[[...], Any]]*) –
> - **autoescape** (*Union[bool, Callable[[Optional[str]], bool]]*) –
> - **loader** (*Optional[BaseLoader]*) –
> - **cache_size** (*int*) –
> - **auto_reload** (*bool*) –
> - **bytecode_cache** (*Optional[BytecodeCache]*) –
> - **enable_async** (*bool*) –

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

> **Parameters**
>
> > - **source** (*Union[str, Template]*) –
> > - **block_start_string** (*str*) –
> > - **block_end_string** (*str*) –
> > - **variable_start_string** (*str*) –
> > - **variable_end_string** (*str*) –
> > - **comment_start_string** (*str*) –
> > - **comment_end_string** (*str*) –
> > - **line_statement_prefix** (*Optional[str]*) –
> > - **line_comment_prefix** (*Optional[str]*) –
> > - **trim_blocks** (*bool*) –
> > - **lstrip_blocks** (*bool*) –
> > - **newline_sequence** (*te.Literal['\n', '\r\n', '\r']*) –
> > - **keep_trailing_newline** (*bool*) –
> > - **extensions** (*Sequence[Union[str, Type[Extension]]]*) –
> > - **optimized** (*bool*) –
> > - **undefined** (*Type[Undefined]*) –
> > - **finalize** (*Optional[Callable[[...], Any]]*) –
> > - **autoescape** (*Union[bool, Callable[[Optional[str]], bool]]*) –
> > - **enable_async** (*bool*) –
>
> **Return type**
> > *Any*

**render**(*\*args*, *\*\*kwargs*)

> Render the template to produce a native Python type. If the result is a single node, its value is returned. Otherwise, the nodes are concatenated as strings. If the result can be parsed with ast.literal_eval(), the parsed value is returned. Otherwise, the string is returned.
>
> **Parameters**
>
> > - **args** (*Any*) –
> > - **kwargs** (*Any*) –
>
> **Return type**
> > *Any*

# TEMPLATE DESIGNER DOCUMENTATION

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

## 5.1 Synopsis

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

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

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

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

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

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

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

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

- {% ... %} for *Statements*

- {{ ... }} for *Expressions* to print to the template output

- {# ... #} for *Comments* not included in the template output

*Line Statements and Comments* are also possible, though they don't have default prefix characters. To use them, set `line_statement_prefix` and `line_comment_prefix` when creating the *Environment*.

### 5.1.1 Template File Extension

As stated above, any file can be loaded as a template, regardless of file extension. Adding a `.jinja` extension, like `user.html.jinja` may make it easier for some IDEs or editor plugins, but is not required. Autoescaping, introduced later, can be applied based on file extension, so you'll need to take the extra suffix into account in that case.

Another good heuristic for identifying templates is that they are in a `templates` folder, regardless of extension. This is a common layout for projects.

## 5.2 Variables

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

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

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

The following lines do the same thing:

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

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

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

---

**Implementation**

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

- check for an attribute called *bar* on *foo* (`getattr(foo, 'bar')`)

- if there is not, check for an item `'bar'` in *foo* (`foo.__getitem__('bar')`)

- if there is not, return an undefined object.

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

- check for an item `'bar'` in *foo*. (`foo.__getitem__('bar')`)

- if there is not, check for an attribute called *bar* on *foo*. (`getattr(foo, 'bar')`)

- if there is not, return an undefined object.

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

---

## 5.3 Filters

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

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

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

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

## 5.4 Tests

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

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

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

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

## 5.5 Comments

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

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

## 5.6 Whitespace Control

In the default configuration:

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

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

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

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

gets rendered with blank lines inside the div:

```
<div>


        yay

</div>
```

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

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

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

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

Similarly, you can manually disable the `trim_blocks` behavior by putting a plus sign (+) at the end of a block:

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

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

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

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

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

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

**Note**

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

**valid**:

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

**invalid**:

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

## 5.7 Escaping

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

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

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

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

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

**Note**

Minus sign at the end of {% raw -%} tag cleans all the spaces and newlines preceding the first character of your raw data.

## 5.8 Line Statements

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

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

<ul>
{% for item in seq %}
```

```
    <li>{{ item }}</li>
{% endfor %}
</ul>
```

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

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

---

**Note**

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

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

---

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

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

## 5.9 Template Inheritance

The most powerful part of Jinja is template inheritance. Template inheritance allows you to build a base "skeleton" template that contains all the common elements of your site and defines **blocks** that child templates can override.

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

### 5.9.1 Base Template

This template, which we'll call `base.html`, defines a simple HTML skeleton document that you might use for a simple two-column page. It's the job of "child" templates to fill the empty blocks with content:

```
<!DOCTYPE html>
<html lang="en">
<head>
    {% block head %}
    <link rel="stylesheet" href="style.css" />
    <title>{% block title %}{% endblock %} - My Webpage</title>
    {% endblock %}
</head>
```

```
<body>
    <div id="content">{% block content %}{% endblock %}</div>
    <div id="footer">
        {% block footer %}
        &copy; Copyright 2008 by <a href="http://domain.invalid/">you</a>.
        {% endblock %}
    </div>
</body>
</html>
```

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

block tags can be inside other blocks such as if, but they will always be executed regardless of if the if block is actually rendered.

### 5.9.2 Child Template

A child template might look like this:

```
{% extends "base.html" %}
{% block title %}Index{% endblock %}
{% block head %}
    {{ super() }}
    <style type="text/css">
        .important { color: #336699; }
    </style>
{% endblock %}
{% block content %}
    <h1>Index</h1>
    <p class="important">
      Welcome to my awesome homepage.
    </p>
{% endblock %}
```

The {% extends %} tag is the key here. It tells the template engine that this template "extends" another template. When the template system evaluates this template, it first locates the parent. The extends tag should be the first tag in the template. Everything before it is printed out normally and may cause confusion. For details about this behavior and how to take advantage of it, see *Null-Default Fallback*. Also a block will always be filled in regardless of whether the surrounding condition is evaluated to be true or false.

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

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

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

You can't define multiple {% block %} tags with the same name in the same template. This limitation exists because a block tag works in "both" directions. That is, a block tag doesn't just provide a placeholder to fill - it also defines the content that fills the placeholder in the *parent*. If there were two similarly-named {% block %} tags in a template, that template's parent wouldn't know which one of the blocks' content to use.

---

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

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

### 5.9.3 Super Blocks

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

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

### 5.9.4 Nesting extends

In the case of multiple levels of `{% extends %}`, `super` references may be chained (as in `super.super()`) to skip levels in the inheritance tree.

For example:

```
# parent.tmpl
body: {% block body %}Hi from parent.{% endblock %}

# child.tmpl
{% extends "parent.tmpl" %}
{% block body %}Hi from child. {{ super() }}{% endblock %}

# grandchild1.tmpl
{% extends "child.tmpl" %}
{% block body %}Hi from grandchild1.{% endblock %}

# grandchild2.tmpl
{% extends "child.tmpl" %}
{% block body %}Hi from grandchild2. {{ super.super() }} {% endblock %}
```

Rendering `child.tmpl` will give body:  Hi from child. Hi from parent.

Rendering `grandchild1.tmpl` will give body:  Hi from grandchild1.

Rendering `grandchild2.tmpl` will give body:  Hi from grandchild2. Hi from parent.

### 5.9.5 Named Block End-Tags

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

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

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

### 5.9.6 Block Nesting and Scope

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

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

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

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

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

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

### 5.9.7 Required Blocks

Blocks can be marked as `required`. They must be overridden at some point, but not necessarily by the direct child template. Required blocks may only contain space and comments, and they cannot be rendered directly.

Listing 1: `page.txt`

```
{% block body required %}{% endblock %}
```

Listing 2: `issue.txt`

```
{% extends "page.txt" %}
```

Listing 3: `bug_report.txt`

```
{% extends "issue.txt" %}
{% block body %}Provide steps to demonstrate the bug.{% endblock %}
```

Rendering `page.txt` or `issue.txt` will raise `TemplateRuntimeError` because they don't override the `body` block. Rendering `bug_report.txt` will succeed because it does override the block.

When combined with `scoped`, the `required` modifier must be placed *after* the scoped modifier. Here are some valid examples:

```
{% block body scoped %}{% endblock %}
{% block body required %}{% endblock %}
{% block body scoped required %}{% endblock %}
```

### 5.9.8 Template Objects

`extends`, `include`, and `import` can take a template object instead of the name of a template to load. This could be useful in some advanced situations, since you can use Python code to load a template first and pass it in to `render`.

```python
if debug_mode:
    layout = env.get_template("debug_layout.html")
else:
    layout = env.get_template("layout.html")

user_detail = env.get_template("user/detail.html")
return user_detail.render(layout=layout)
```

```
{% extends layout %}
```

Note how `extends` is passed the variable with the template object that was passed to `render`, instead of a string.

## 5.10 HTML Escaping

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

   a. manually escaping each variable; or

   b. automatically escaping everything by default.

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

- Escaping everything except for safe values will also mean that Jinja is escaping variables known to not include HTML (e.g. numbers, booleans) which can be a huge performance hit.

- The information about the safety of a variable is very fragile. It could happen that by coercing safe and unsafe values, the return value is double-escaped HTML.

### 5.10.1 Working with Manual Escaping

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

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

### 5.10.2 Working with Automatic Escaping

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

  a. The context dictionary by the application with `markupsafe.Markup`

  b. The template, with the `|safe` filter.

If a string that you marked safe is passed through other Python code that doesn't understand that mark, it may get lost. Be aware of when your data is marked safe and how it is processed before arriving at the template.

If a value has been escaped but is not marked safe, auto-escaping will still take place and result in double-escaped characters. If you know you have data that is already safe but not marked, be sure to wrap it in `Markup` or use the `|safe` filter.

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

String literals in templates with automatic escaping are considered unsafe because native Python strings are not safe.

## 5.11 List of Control Structures

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

### 5.11.1 For

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

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

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

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

Python dicts may not be in the order you want to display them in. If order matters, use the `|dictsort` filter.

```
<dl>
{% for key, value in my_dict | dictsort %}
    <dt>{{ key|e }}</dt>
    <dd>{{ value|e }}</dd>
{% endfor %}
</dl>
```

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

| Variable | Description |
|---|---|
| *loop.index* | The current iteration of the loop. (1 indexed) |
| *loop.index0* | The current iteration of the loop. (0 indexed) |
| *loop.revindex* | The number of iterations from the end of the loop (1 indexed) |
| *loop.revindex0* | The number of iterations from the end of the loop (0 indexed) |
| *loop.first* | True if first iteration. |
| *loop.last* | True if last iteration. |
| *loop.length* | The number of items in the sequence. |
| *loop.cycle* | A helper function to cycle between a list of sequences. See the explanation below. |
| *loop.depth* | Indicates how deep in a recursive loop the rendering currently is. Starts at level 1 |
| *loop.depth0* | Indicates how deep in a recursive loop the rendering currently is. Starts at level 0 |
| *loop.previtem* | The item from the previous iteration of the loop. Undefined during the first iteration. |
| *loop.nextitem* | The item from the following iteration of the loop. Undefined during the last iteration. |
| *loop.changed(\*val)* | True if previously called with a different value (or not called at all). |

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

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

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

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

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

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

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

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

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

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

The following example implements a sitemap with recursive loops:

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

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

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

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

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

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

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

## 5.11.2 If

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

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

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

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

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

## 5.11.3 Macros

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

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

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

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

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

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

Inside macros, you have access to three special variables:

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

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

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

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

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

*arguments*
> A tuple of the names of arguments the macro accepts.

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

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

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

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

Due to how scopes work in Jinja, a macro in a child template does not override a macro in a parent template. The following will output "LAYOUT", not "CHILD".

Listing 4: `layout.txt`

```
{% macro foo() %}LAYOUT{% endmacro %}
{% block body %}{% endblock %}
```

Listing 5: `child.txt`

```
{% extends 'layout.txt' %}
{% macro foo() %}CHILD{% endmacro %}
{% block body %}{{ foo() }}{% endblock %}
```

## 5.11.4 Call

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

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

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

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

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

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

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

## 5.11.5 Filters

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

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

## 5.11.6 Assignments

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

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

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

### Scoping Behavior

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

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

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

```
{% for item in seq %}
    {{ item }}
{% else %}
    did not iterate
{% endfor %}
```

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

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

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

New in version 2.10: Added support for namespace objects

### 5.11.7 Block Assignments

New in version 2.8.

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

Example:

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

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

Changed in version 2.10.

Starting with Jinja 2.10, the block assignment supports filters.

Example:

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

## 5.11.8 Extends

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

See the section about *Template Inheritance* above.

## 5.11.9 Blocks

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

## 5.11.10 Include

The `include` tag renders another template and outputs the result into the current template.

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

The included template has access to context of the current template by default. Use `without context` to use a separate context instead. `with context` is also valid, but is the default behavior. See *Import Context Behavior*.

The included template can `extend` another template and override blocks in that template. However, the current template cannot override any blocks that the included template outputs.

Use `ignore missing` to ignore the statement if the template does not exist. It must be placed *before* a context visibility statement.

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

If a list of templates is given, each will be tried in order until one is not missing. This can be used with `ignore missing` to ignore if none of the templates exist.

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

A variable, with either a template name or template object, can also be passed to the statment.

## 5.11.11 Import

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

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

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

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

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

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

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

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

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

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

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

## 5.12 Import Context Behavior

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

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

Here are two examples:

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

---

**Note**

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

---

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

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

## 5.13 Expressions

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

### 5.13.1 Literals

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

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

**`42 / 123_456`**
Integers are whole numbers without a decimal part. The '_' character can be used to separate groups for legibility.

**`42.23 / 42.1e2 / 123_456.789`**
Floating point numbers can be written using a '.' as a decimal mark. They can also be written in scientific notation with an upper or lower case 'e' to indicate the exponent part. The '_' character can be used to separate groups for legibility, but cannot be used in the exponent part.

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

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

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

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

**`true / false`**
`true` is always true and `false` is always false.

**Note**

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

## 5.13.2 Math

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

**+**

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

**-**

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

**/**

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

**//**

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

**%**

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

**\***

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

**\*\***

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

Unlike Python, chained pow is evaluated left to right. `{{ 3**3**3 }}` is evaluated as `(3**3)**3` in Jinja, but would be evaluated as `3**(3**3)` in Python. Use parentheses in Jinja to be explicit about what order you want. It is usually preferable to do extended math in Python and pass the results to `render` rather than doing it in the template.

This behavior may be changed in the future to match Python, if it's possible to introduce an upgrade path.

## 5.13.3 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.

### 5.13.4 Logic

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

**and**
>   Return true if the left and the right operand are true.

**or**
>   Return true if the left or the right operand are true.

**not**
>   negate a statement (see below).

**(expr)**
>   Parentheses 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)`.

---

### 5.13.5 Other Operators

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

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

**is**
>   Performs a *test*.

**| (pipe, vertical bar)**
>   Applies a *filter*.

**~ (tilde)**
>   Converts all operands into strings and concatenates them.
>
>   `{{ "Hello " ~ name ~ "!" }}` would return (assuming *name* is set to `'John'`) `Hello John!`.

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

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

### 5.13.6 If Expression

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

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

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

The *else* part is optional. If not provided, the else block implicitly evaluates into an *Undefined* object (regardless of what `undefined` in the environment is set to):

```
{{ "[{}]".format(page.title) if page.title }}
```

### 5.13.7 Python Methods

You can also use any of the methods defined on a variable's type. The value returned from the method invocation is used as the value of the expression. Here is an example that uses methods defined on strings (where `page.title` is a string):

```
{{ page.title.capitalize() }}
```

This works for methods on user-defined types. For example, if variable `f` of type `Foo` has a method `bar` defined on it, you can do the following:

```
{{ f.bar(value) }}
```

Operator methods also work as expected. For example, `%` implements printf-style for strings:

```
{{ "Hello, %s!" % name }}
```

Although you should prefer the `.format` method for that case (which is a bit contrived in the context of rendering a template):

```
{{ "Hello, {}!".format(name) }}
```

## 5.14 List of Builtin Filters

| | | | | |
|---|---|---|---|---|
| abs() | forceescape() | map() | select() | unique() |
| attr() | format() | max() | selectattr() | upper() |
| batch() | groupby() | min() | slice() | urlencode() |
| capitalize() | indent() | pprint() | sort() | urlize() |
| center() | int() | random() | string() | wordcount() |
| default() | items() | reject() | striptags() | wordwrap() |
| dictsort() | join() | rejectattr() | sum() | xmlattr() |
| escape() | last() | replace() | title() | |
| filesizeformat() | length() | reverse() | tojson() | |
| first() | list() | round() | trim() | |
| float() | lower() | safe() | truncate() | |

jinja-filters.**abs**(*x*, */*)

    Return the absolute value of the argument.

jinja-filters.**attr**(*obj: Any*, *name:* *str*) → Union[*jinja2.runtime.Undefined*, Any]

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

    See *Notes on subscriptions* for more details.

jinja-filters.**batch**(*value: 't.Iterable[V]'*, *linecount:* *int*, *fill_with: 't.Optional[V]' = None*) → 't.Iterator[t.List[V]]'

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

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

jinja-filters.**capitalize**(*s:* *str*) → str

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

jinja-filters.**center**(*value:* *str*, *width:* *int = 80*) → str

    Centers the value in a field of a given width.

jinja-filters.**default**(*value: V*, *default_value: V = ''*, *boolean:* *bool = False*) → V

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

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

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

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

    Changed in version 2.11: It's now possible to configure the `Environment` with `ChainableUndefined` to make the *default* filter work on nested elements and attributes that may contain undefined values in the chain without getting an `UndefinedError`.

        **Aliases**
          d

jinja-filters.**dictsort**(*value: Mapping[K, V]*, *case_sensitive:* *bool = False*, *by: 'te.Literal["key", "value"]' = 'key'*, *reverse:* *bool = False*) → *List*[*Tuple*[K, V]]

    Sort a dict and yield (key, value) pairs. Python dicts may not be in the order you want to display them in, so sort them first.

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

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

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

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

jinja-filters.**escape**(*value*)

Replace the characters &, <, >, ', and " in the string with HTML-safe sequences. Use this if you need to display text that might contain such characters in HTML.

If the object has an __html__ method, it is called and the return value is assumed to already be safe for HTML.

> **Parameters**
> > **s** – An object to be converted to a string and escaped.
>
> **Returns**
> > A Markup string with the escaped text.
>
> **Aliases**
> > e

jinja-filters.**filesizeformat**(*value: Union[str, float, int]*, *binary: bool = False*) → str

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

jinja-filters.**first**(*seq: 't.Iterable[V]'*) → 't.Union[V, Undefined]'

Return the first item of a sequence.

jinja-filters.**float**(*value: Any*, *default: float = 0.0*) → float

Convert the value into a floating point number. If the conversion doesn't work it will return 0.0. You can override this default using the first parameter.

jinja-filters.**forceescape**(*value: 't.Union[str, HasHTML]'*) → markupsafe.Markup

Enforce HTML escaping. This will probably double escape variables.

jinja-filters.**format**(*value: str*, *\*args: Any*, *\*\*kwargs: Any*) → str

Apply the given values to a printf-style format string, like string % values.

```
{{ "%s, %s!"|format(greeting, name) }}
Hello, World!
```

In most cases it should be more convenient and efficient to use the % operator or str.format().

```
{{ "%s, %s!" % (greeting, name) }}
{{ "{}, {}!".format(greeting, name) }}
```

jinja-filters.**groupby**(*value: 't.Iterable[V]'*, *attribute: Union[str, int]*, *default: Optional[Any] = None*, *case_sensitive: bool = False*) → 't.List[_GroupTuple]'

Group a sequence of objects by an attribute using Python's itertools.groupby(). The attribute can use dot

notation for nested access, like `"address.city"`. Unlike Python's `groupby`, the values are sorted first so only one group is returned for each unique value.

For example, a list of `User` objects with a `city` attribute can be rendered in groups. In this example, `grouper` refers to the `city` value of the group.

```
<ul>{% for city, items in users|groupby("city") %}
  <li>{{ city }}
    <ul>{% for user in items %}
      <li>{{ user.name }}
    {% endfor %}</ul>
  </li>
{% endfor %}</ul>
```

`groupby` yields namedtuples of `(grouper, list)`, which can be used instead of the tuple unpacking above. `grouper` is the value of the attribute, and `list` is the items with that value.

```
<ul>{% for group in users|groupby("city") %}
  <li>{{ group.grouper }}: {{ group.list|join(", ") }}
{% endfor %}</ul>
```

You can specify a `default` value to use if an object in the list does not have the given attribute.

```
<ul>{% for city, items in users|groupby("city", default="NY") %}
  <li>{{ city }}: {{ items|map(attribute="name")|join(", ") }}</li>
{% endfor %}</ul>
```

Like the *sort()* filter, sorting and grouping is case-insensitive by default. The `key` for each group will have the case of the first item in that group of values. For example, if a list of users has cities `["CA", "NY", "ca"]`, the "CA" group will have two values. This can be disabled by passing `case_sensitive=True`.

Changed in version 3.1: Added the `case_sensitive` parameter. Sorting and grouping is case-insensitive by default, matching other filters that do comparisons.

Changed in version 3.0: Added the `default` parameter.

Changed in version 2.6: The attribute supports dot notation for nested access.

`jinja-filters.`**`indent`**(*s: str*, *width: Union[int, str] = 4*, *first: bool = False*, *blank: bool = False*) → str

Return a copy of the string with each line indented by 4 spaces. The first line and blank lines are not indented by default.

> **Parameters**
>
> - **width** – Number of spaces, or a string, to indent by.
> - **first** – Don't skip indenting the first line.
> - **blank** – Don't skip indenting empty lines.

Changed in version 3.0: `width` can be a string.

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

Rename the `indentfirst` argument to `first`.

`jinja-filters.`**`int`**(*value: Any*, *default: int = 0*, *base: int = 10*) → int

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

jinja-filters.**items**(*value: Union[Mapping[K, V],* jinja2.runtime.Undefined*]*) → Iterator[*Tuple*[K, V]]

> Return an iterator over the (`key, value`) items of a mapping.
>
> `x|items` is the same as `x.items()`, except if `x` is undefined an empty iterator is returned.
>
> This filter is useful if you expect the template to be rendered with an implementation of Jinja in another programming language that does not have a `.items()` method on its mapping type.
>
> ```
> <dl>
> {% for key, value in my_dict|items %}
>     <dt>{{ key }}
>     <dd>{{ value }}
> {% endfor %}
> </dl>
> ```
>
> New in version 3.1.

jinja-filters.**join**(*value: Iterable*, *d:* str *= ''*, *attribute: Union[*str*,* int*, NoneType] = None*) → str

> Return a string which is the concatenation of the strings in the sequence. The separator between elements is an empty string per default, you can define it with the optional parameter:
>
> ```
> {{ [1, 2, 3]|join('|') }}
>     -> 1|2|3
>
> {{ [1, 2, 3]|join }}
>     -> 123
> ```
>
> It is also possible to join certain attributes of an object:
>
> ```
> {{ users|join(', ', attribute='username') }}
> ```
>
> New in version 2.6: The *attribute* parameter was added.

jinja-filters.**last**(*seq: 't.Reversible[V]'*) → 't.Union[V, Undefined]'

> Return the last item of a sequence.
>
> Note: Does not work with generators. You may want to explicitly convert it to a list:
>
> ```
> {{ data | selectattr('name', '==', 'Jinja') | list | last }}
> ```

jinja-filters.**length**(*obj*, */*)

> Return the number of items in a container.
>
> > **Aliases**
> > count

jinja-filters.**list**(*value: 't.Iterable[V]'*) → 't.List[V]'

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

jinja-filters.**lower**(*s:* str) → str

> Convert a value to lowercase.

jinja-filters.**map**(*value: Iterable*, *\*args: Any*, *\*\*kwargs: Any*) → Iterable

> Applies a filter on a sequence of objects or looks up an attribute. This is useful when dealing with lists of objects but you are really only interested in a certain value of it.
>
> The basic usage is mapping on an attribute. Imagine you have a list of users but you are only interested in a list of usernames:

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

You can specify a `default` value to use if an object in the list does not have the given attribute.

```
{{ users|map(attribute="username", default="Anonymous")|join(", ") }}
```

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

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

Similar to a generator comprehension such as:

```
(u.username for u in users)
(getattr(u, "username", "Anonymous") for u in users)
(do_lower(x) for x in titles)
```

Changed in version 2.11.0: Added the `default` parameter.

New in version 2.7.

`jinja-filters.max`(*value: 't.Iterable[V]'*, *case_sensitive: bool = False*, *attribute: Union[str, int, NoneType] = None*) → 't.Union[V, Undefined]'

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.

`jinja-filters.min`(*value: 't.Iterable[V]'*, *case_sensitive: bool = False*, *attribute: Union[str, int, NoneType] = None*) → 't.Union[V, Undefined]'

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 min value of this attribute.

`jinja-filters.pprint`(*value: Any*) → str

Pretty print a variable. Useful for debugging.

`jinja-filters.random`(*seq: 't.Sequence[V]'*) → 't.Union[V, Undefined]'

Return a random item from the sequence.

jinja-filters.**reject**(*value: 't.Iterable[V]', *args: Any, **kwargs: Any*) → 't.Iterator[V]'

>   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") }}
```

>   Similar to a generator comprehension such as:

```
(n for n in numbers if not test_odd(n))
```

>   New in version 2.7.

jinja-filters.**rejectattr**(*value: 't.Iterable[V]', *args: Any, **kwargs: Any*) → 't.Iterator[V]'

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

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

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

>   Similar to a generator comprehension such as:

```
(u for user in users if not user.is_active)
(u for user in users if not test_none(user.email))
```

>   New in version 2.7.

jinja-filters.**replace**(*s: str, old: str, new: str, count: Optional[int] = None*) → str

>   Return a copy of the value with all occurrences of a substring replaced with a new one. The first argument is the substring that should be replaced, the second is the replacement string. If the optional third argument `count` is given, only the first `count` occurrences are replaced:

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

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

jinja-filters.**reverse**(*value: Union[str, Iterable[V]]*) → Union[str, Iterable[V]]

>   Reverse the object or return an iterator that iterates over it the other way round.

jinja-filters.**round**(*value: float, precision: int = 0, method: 'te.Literal["common", "ceil", "floor"]' = 'common'*) → float

>   Round the number to a given precision. The first parameter specifies the precision (default is `0`), the second the rounding method:

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

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

---

**5.14. List of Builtin Filters**                                                                    **79**

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

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

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

jinja-filters.**safe**(*value: str*) → markupsafe.Markup

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

jinja-filters.**select**(*value: 't.Iterable[V]', *args: Any, **kwargs: Any*) → 't.Iterator[V]'

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") }}
```

Similar to a generator comprehension such as:

```
(n for n in numbers if test_odd(n))
(n for n in numbers if test_divisibleby(n, 3))
```

New in version 2.7.

jinja-filters.**selectattr**(*value: 't.Iterable[V]', *args: Any, **kwargs: Any*) → 't.Iterator[V]'

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

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

Example usage:

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

Similar to a generator comprehension such as:

```
(u for user in users if user.is_active)
(u for user in users if test_none(user.email))
```

New in version 2.7.

jinja-filters.**slice**(*value: 't.Collection[V]', slices: int, fill_with: 't.Optional[V]' = None*) → 't.Iterator[t.List[V]]'

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

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

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

jinja-filters.**sort**(*value: 't.Iterable[V]'*, *reverse: bool = False*, *case_sensitive: bool = False*, *attribute: Union[str, int, NoneType] = None*) → *'t.List[V]'*

Sort an iterable using Python's sorted().

```
{% for city in cities|sort %}
    ...
{% endfor %}
```

> **Parameters**
>
> - **reverse** – Sort descending instead of ascending.
>
> - **case_sensitive** – When sorting strings, sort upper and lower case separately.
>
> - **attribute** – When sorting objects or dicts, an attribute or key to sort by. Can use dot notation like `"address.city"`. Can be a list of attributes like `"age,name"`.

The sort is stable, it does not change the relative order of elements that compare equal. This makes it is possible to chain sorts on different attributes and ordering.

```
{% for user in users|sort(attribute="name")
    |sort(reverse=true, attribute="age") %}
    ...
{% endfor %}
```

As a shortcut to chaining when the direction is the same for all attributes, pass a comma separate list of attributes.

```
{% for user in users|sort(attribute="age,name") %}
    ...
{% endfor %}
```

Changed in version 2.11.0: The `attribute` parameter can be a comma separated list of attributes, e.g. `"age, name"`.

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

jinja-filters.**string**(*value*)

Convert an object to a string if it isn't already. This preserves a `Markup` string rather than converting it back to a basic string, so it will still be marked as safe and won't be escaped again.

```
>>> value = escape("<User 1>")
>>> value
Markup('&lt;User 1&gt;')
```

```
>>> escape(str(value))
Markup('&amp;lt;User 1&amp;gt;')
>>> escape(soft_str(value))
Markup('&lt;User 1&gt;')
```

jinja-filters.**striptags**(*value: 't.Union[str, HasHTML]'*) → str

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

jinja-filters.**sum**(*iterable: 't.Iterable[V]'*, *attribute: Union[str, int, NoneType] = None*, *start: V = 0*) → V

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

  It is also possible to sum up only certain attributes:

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

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

jinja-filters.**title**(*s: str*) → str

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

jinja-filters.**tojson**(*value: Any*, *indent: Optional[int] = None*) → markupsafe.Markup

  Serialize an object to a string of JSON, and mark it safe to render in HTML. This filter is only for use in HTML documents.

  The returned string is safe to render in HTML documents and `<script>` tags. The exception is in HTML attributes that are double quoted; either use single quotes or the `|forceescape` filter.

  > **Parameters**
  >
  >   • **value** – The object to serialize to JSON.
  >
  >   • **indent** – The `indent` parameter passed to `dumps`, for pretty-printing the value.

  New in version 2.9.

jinja-filters.**trim**(*value: str*, *chars: Optional[str] = None*) → str

  Strip leading and trailing characters, by default whitespace.

jinja-filters.**truncate**(*s: str*, *length: int = 255*, *killwords: bool = False*, *end: str = '...'*, *leeway: Optional[int] = None*) → str

  Return a truncated copy of the string. The length is specified with the first parameter which defaults to `255`. If the second parameter is `true` the filter will cut the text at length. Otherwise it will discard the last word. If the text was in fact truncated it will append an ellipsis sign (`"..."`). If you want a different ellipsis sign than `"..."` you can specify it using the third parameter. Strings that only exceed the length by the tolerance margin given in the fourth parameter will not be truncated.

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

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

jinja-filters.**unique**(*value: 't.Iterable[V]'*, *case_sensitive: [bool](#) = False*, *attribute: Union[[str](#), [int](#), NoneType]* *= None*) → 't.Iterator[V]'

Returns a list of unique items from the given iterable.

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

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

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

jinja-filters.**upper**(*s: [str](#)*) → [str](#)

Convert a value to uppercase.

jinja-filters.**urlencode**(*value: Union[[str](#), Mapping[[str](#), Any], Iterable[Tuple[[str](#), Any]]]*) → [str](#)

Quote data for use in a URL path or query using UTF-8.

Basic wrapper around [urllib.parse.quote()](#) when given a string, or [urllib.parse.urlencode()](#) for a dict or iterable.

> **Parameters**
> **value** – Data to quote. A string will be quoted directly. A dict or iterable of (key, value) pairs will be joined as a query string.

When given a string, "/" is not quoted. HTTP servers treat "/" and "%2F" equivalently in paths. If you need quoted slashes, use the |replace("/", "%2F") filter.

New in version 2.7.

jinja-filters.**urlize**(*value: [str](#)*, *trim_url_limit: Optional[[int](#)] = None*, *nofollow: [bool](#) = False*, *target:* *Optional[[str](#)] = None*, *rel: Optional[[str](#)] = None*, *extra_schemes: Optional[Iterable[[str](#)]]* *= None*) → [str](#)

Convert URLs in text into clickable links.

This may not recognize links in some situations. Usually, a more comprehensive formatter, such as a Markdown library, is a better choice.

Works on http://, https://, www., mailto:, and email addresses. Links with trailing punctuation (periods, commas, closing parentheses) and leading punctuation (opening parentheses) are recognized excluding the punctuation. Email addresses that include header fields are not recognized (for example, mailto:address@example.com?cc=copy@example.com).

> **Parameters**
>
> - **value** – Original text containing URLs to link.
>
> - **trim_url_limit** – Shorten displayed URL values to this length.
>
> - **nofollow** – Add the rel=nofollow attribute to links.
>
> - **target** – Add the target attribute to links.
>
> - **rel** – Add the rel attribute to links.
>
> - **extra_schemes** – Recognize URLs that start with these schemes in addition to the default behavior. Defaults to env.policies["urlize.extra_schemes"], which defaults to no extra schemes.

Changed in version 3.0: The `extra_schemes` parameter was added.

Changed in version 3.0: Generate `https://` links for URLs without a scheme.

Changed in version 3.0: The parsing rules were updated. Recognize email addresses with or without the `mailto:` scheme. Validate IP addresses. Ignore parentheses and brackets in more cases.

Changed in version 2.8: The `target` parameter was added.

jinja-filters.**wordcount**(*s: str*) → int

> Count the words in that string.

jinja-filters.**wordwrap**(*s: str*, *width: int = 79*, *break_long_words: bool = True*, *wrapstring: Optional[str] = None*, *break_on_hyphens: bool = True*) → str

> Wrap a string to the given width. Existing newlines are treated as paragraphs to be wrapped separately.
>
> > **Parameters**
> >
> > - **s** – Original text to wrap.
> >
> > - **width** – Maximum length of wrapped lines.
> >
> > - **break_long_words** – If a word is longer than `width`, break it across lines.
> >
> > - **break_on_hyphens** – If a word contains hyphens, it may be split across lines.
> >
> > - **wrapstring** – String to join each wrapped line. Defaults to `Environment.newline_sequence`.
>
> Changed in version 2.11: Existing newlines are treated as paragraphs wrapped separately.
>
> Changed in version 2.11: Added the `break_on_hyphens` parameter.
>
> Changed in version 2.7: Added the `wrapstring` parameter.

jinja-filters.**xmlattr**(*d: Mapping[str, Any]*, *autospace: bool = True*) → str

> Create an SGML/XML attribute string based on the items in a dict. All values that are neither *none* nor *undefined* are automatically escaped:

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

> Results in something like this:

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

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

## 5.15 List of Builtin Tests

| | | | | |
|---|---|---|---|---|
| *boolean()* | *even()* | *in()* | *mapping()* | *sequence()* |
| *callable()* | *false()* | *integer()* | *ne()* | *string()* |
| *defined()* | *filter()* | *iterable()* | *none()* | *test()* |
| *divisibleby()* | *float()* | *le()* | *number()* | *true()* |
| *eq()* | *ge()* | *lower()* | *odd()* | *undefined()* |
| *escaped()* | *gt()* | *lt()* | *sameas()* | *upper()* |

jinja-tests.**boolean**(*value: Any*) → bool

> Return true if the object is a boolean value.
>
> New in version 2.11.

jinja-tests.**callable**(*obj, /*)

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

jinja-tests.**defined**(*value: Any*) → bool

> Return true if the variable is defined:

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

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

jinja-tests.**divisibleby**(*value: int, num: int*) → bool

> Check if a variable is divisible by a number.

jinja-tests.**eq**(*a, b, /*)

> Same as a == b.
>
> > **Aliases**
> > ==, equalto

jinja-tests.**escaped**(*value: Any*) → bool

> Check if the value is escaped.

jinja-tests.**even**(*value: int*) → bool

> Return true if the variable is even.

jinja-tests.**false**(*value: Any*) → bool

> Return true if the object is False.
>
> New in version 2.11.

jinja-tests.**filter**(*value: str*) → bool

> Check if a filter exists by name. Useful if a filter may be optionally available.

```
{% if 'markdown' is filter %}
    {{ value | markdown }}
{% else %}
```

```
    {{ value }}
{% endif %}
```

New in version 3.0.

jinja-tests.**float**(*value: Any*) → bool

Return true if the object is a float.

New in version 2.11.

jinja-tests.**ge**(*a*, *b*, */*)

Same as a >= b.

> **Aliases**
>> >=

jinja-tests.**gt**(*a*, *b*, */*)

Same as a > b.

> **Aliases**
>> >, greaterthan

jinja-tests.**in**(*value: Any*, *seq: Container*) → bool

Check if value is in seq.

New in version 2.10.

jinja-tests.**integer**(*value: Any*) → bool

Return true if the object is an integer.

New in version 2.11.

jinja-tests.**iterable**(*value: Any*) → bool

Check if it's possible to iterate over an object.

jinja-tests.**le**(*a*, *b*, */*)

Same as a <= b.

> **Aliases**
>> <=

jinja-tests.**lower**(*value: str*) → bool

Return true if the variable is lowercased.

jinja-tests.**lt**(*a*, *b*, */*)

Same as a < b.

> **Aliases**
>> <, lessthan

jinja-tests.**mapping**(*value: Any*) → bool

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

New in version 2.6.

jinja-tests.**ne**(*a*, *b*, */*)

Same as a != b.

> **Aliases**
>> !=

---

jinja-tests.**none**(*value: Any*) → bool

> Return true if the variable is none.

jinja-tests.**number**(*value: Any*) → bool

> Return true if the variable is a number.

jinja-tests.**odd**(*value: int*) → bool

> Return true if the variable is odd.

jinja-tests.**sameas**(*value: Any, other: Any*) → bool

> Check if an object points to the same memory address than another object:

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

jinja-tests.**sequence**(*value: Any*) → bool

> Return true if the variable is a sequence. Sequences are variables that are iterable.

jinja-tests.**string**(*value: Any*) → bool

> Return true if the object is a string.

jinja-tests.**test**(*value: str*) → bool

> Check if a test exists by name. Useful if a test may be optionally available.

```
{% if 'loud' is test %}
    {% if value is loud %}
        {{ value|upper }}
    {% else %}
        {{ value|lower }}
    {% endif %}
{% else %}
    {{ value }}
{% endif %}
```

> New in version 3.0.

jinja-tests.**true**(*value: Any*) → bool

> Return true if the object is True.

> New in version 2.11.

jinja-tests.**undefined**(*value: Any*) → bool

> Like *defined()* but the other way round.

jinja-tests.**upper**(*value: str*) → bool

> Return true if the variable is uppercased.

---

**5.15. List of Builtin Tests**

## 5.16 List of Global Functions

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

jinja-globals.**range**([*start*], *stop*[, *step*])

> Return a list containing an arithmetic progression of integers. range(i, j) returns [i, i+1, i+2, ...,
> j-1]; start (!) defaults to 0. When step is given, it specifies the increment (or decrement). For example,
> range(4) and range(0, 4, 1) return [0, 1, 2, 3]. The end point is omitted! These are exactly the valid
> indices for a list of 4 elements.
>
> This is useful to repeat a template block multiple times, e.g. to fill a list. Imagine you have 7 users in the list but
> you want to render three empty items to enforce a height with CSS:

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

jinja-globals.**lipsum**(*n=5*, *html=True*, *min=20*, *max=100*)

> Generates some lorem ipsum for the template. By default, five paragraphs of HTML are generated with each
> paragraph between 20 and 100 words. If html is False, regular text is returned. This is useful to generate simple
> contents for layout testing.

jinja-globals.**dict**(\\\*\**items*)

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

**class** jinja-globals.**cycler**(\\\**items*)

> Cycle through values by yielding them one at a time, then restarting once the end is reached.
>
> Similar to loop.cycle, but can be used outside loops or across multiple loops. For example, render a list of
> folders and files in a list, alternating giving them "odd" and "even" classes.

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

> > **Parameters**
> >
> > > **items** – Each positional argument will be yielded in the order given for each cycle.
>
> New in version 2.1.
>
> **property current**
>
> > Return the current item. Equivalent to the item that will be returned next time *next()* is called.

**next**()

> Return the current item, then advance *current* to the next item.

**reset**()

> Resets the current item to the first item.

**class** jinja-globals.**joiner**(*sep=', '*)

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

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

New in version 2.1.

**class** jinja-globals.**namespace**(...)

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

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

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

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

New in version 2.10.

## 5.17 Extensions

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

### 5.17.1 i18n

If the *i18n Extension* is enabled, it's possible to mark text in the template as translatable. To mark a section as translatable, use a `trans` block:

```
{% trans %}Hello, {{ user }}!{% endtrans %}
```

Inside the block, no statements are allowed, only text and simple variable tags.

Variable tags can only be a name, not attribute access, filters, or other expressions. To use an expression, bind it to a name in the `trans` tag for use in the block.

```
{% trans user=user.username %}Hello, {{ user }}!{% endtrans %}
```

To bind more than one expression, separate each with a comma (`,`).

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

To pluralize, specify both the singular and plural forms separated by the `pluralize` tag.

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

By default, the first variable in a block is used to determine whether to use singular or plural form. If that isn't correct, specify the variable used for pluralizing as a parameter to `pluralize`.

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

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

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

This results in `This is %(book_title)s. You should read it!` in the translation file.

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

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

If the translation depends on the context that the message appears in, the `pgettext` and `npgettext` functions take a `context` string as the first argument, which is used to select the appropriate translation. To specify a context with the `{% trans %}` tag, provide a string as the first token after `trans`.

```
{% trans "fruit" %}apple{% endtrans %}
{% trans "fruit" trimmed count -%}
    1 apple
```

```
{%- pluralize -%}
    {{ count }} apples
{%- endtrans %}
```

New in version 3.1: A context can be passed to the `trans` tag to use `pgettext` and `npgettext`.

It's possible to translate strings in expressions with these functions:

- `_(message)`: Alias for `gettext`.

- `gettext(message)`: Translate a message.

- `ngettext(singluar, plural, n)`: Translate a singular or plural message based on a count variable.

- `pgettext(context, message)`: Like `gettext()`, but picks the translation based on the context string.

- `npgettext(context, singular, plural, n)`: Like `npgettext()`, but picks the translation based on the context string.

You can print a translated string like this:

```
{{ _("Hello, World!") }}
```

To use placeholders, use the `format` filter.

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

Always use keyword arguments to `format`, as other languages may not use the words in the same order.

If *New Style Gettext* calls are activated, using placeholders is easier. Formatting is part of the `gettext` call instead of using the `format` filter.

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

The `ngettext` function's format string automatically receives the count as a `num` parameter in addition to the given parameters.

## 5.17.2 Expression Statement

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

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

### 5.17.3 Loop Controls

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

Here's a loop that skips every second item:

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

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

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

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

### 5.17.4 Debug Statement

If the *Debug Extension* is enabled, a `{% debug %}` tag will be available to dump the current context as well as the available filters and tests. This is useful to see what's available to use in the template without setting up a debugger.

```
<pre>{% debug %}</pre>
```

```
{'context': {'cycler': <class 'jinja2.utils.Cycler'>,
             ...,
             'namespace': <class 'jinja2.utils.Namespace'>},
 'filters': ['abs', 'attr', 'batch', 'capitalize', 'center', 'count', 'd',
             ..., 'urlencode', 'urlize', 'wordcount', 'wordwrap', 'xmlattr'],
 'tests': ['!=', '<', '<=', '==', '>', '>=', 'callable', 'defined',
           ..., 'odd', 'sameas', 'sequence', 'string', 'undefined', 'upper']}
```

### 5.17.5 With Statement

New in version 2.3.

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

With in a nutshell:

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

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

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

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

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

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

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

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

**Extension**

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

## 5.18 Autoescape Overrides

New in version 2.4.

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

Example:

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

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

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

**Extension**

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

# EXTENSIONS

Jinja supports extensions that can add extra filters, tests, globals or even extend the parser. The main motivation of extensions is to move often used code into a reusable class like adding support for internationalization.

## 6.1 Adding Extensions

Extensions are added to the Jinja environment at creation time. To add an extension pass a list of extension classes or import paths to the `extensions` parameter of the *Environment* constructor. The following example creates a Jinja environment with the i18n extension loaded:

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

To add extensions after creation time, use the *add_extension()* method:

```
jinja_env.add_extension('jinja2.ext.debug')
```

## 6.2 i18n Extension

**Import name:** `jinja2.ext.i18n`

The i18n extension can be used in combination with gettext or Babel. When it's enabled, Jinja provides a `trans` statement that marks a block as translatable and calls gettext.

After enabling, an application has to provide functions for `gettext`, `ngettext`, and optionally `pgettext` and `npgettext`, either globally or when rendering. A `_()` function is added as an alias to the `gettext` function.

### 6.2.1 Environment Methods

After enabling the extension, the environment provides the following additional methods:

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

> Installs a translation globally for the environment. The `translations` object must implement `gettext`, `ngettext`, and optionally `pgettext` and `npgettext`. `gettext.NullTranslations`, `gettext.GNUTranslations`, and Babels `Translations` are supported.
>
> Changed in version 3.0: Added `pgettext` and `npgettext`.
>
> Changed in version 2.5: Added new-style gettext support.

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

> Install no-op gettext functions. This is useful if you want to prepare the application for internationalization but don't want to implement the full system yet.
>
> Changed in version 2.5: Added new-style gettext support.

jinja2.Environment.**install_gettext_callables**(*gettext*, *ngettext*, *newstyle=False*, *pgettext=None*, *npgettext=None*)

> Install the given `gettext`, `ngettext`, `pgettext`, and `npgettext` callables into the environment. They should behave exactly like `gettext.gettext()`, `gettext.ngettext()`, `gettext.pgettext()` and `gettext.npgettext()`.
>
> If `newstyle` is activated, the callables are wrapped to work like newstyle callables. See *New Style Gettext* for more information.
>
> Changed in version 3.0: Added `pgettext` and `npgettext`.
>
> New in version 2.5: Added new-style gettext support.

jinja2.Environment.**uninstall_gettext_translations**()

> Uninstall the environment's globally installed translation.

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 the string was extracted from embedded Python code).
>
> - `message` is the string itself, or a tuple of strings for functions with multiple arguments.
>
> If Babel is installed, see *Babel* to extract the strings.

For a web application that is available in multiple languages but gives all the users the same language (for example, multilingual forum software installed for a French community), the translation may be installed when the environment is created.

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

The `get_gettext_translations` function would return the translator for the current configuration, for example by using `gettext.find`.

The usage of the `i18n` extension for template designers is covered in *the template documentation*.

## 6.2.2 Whitespace Trimming

New in version 2.10.

Within `{% trans %}` blocks, it can be useful to trim line breaks and whitespace so that the block of text looks like a simple string with single spaces in the translation file.

Linebreaks and surrounding whitespace can be automatically trimmed by enabling the `ext.i18n.trimmed` *policy*.

### 6.2.3 New Style Gettext

New in version 2.5.

New style gettext calls are less to type, less error prone, and support autoescaping better.

You can use "new style" gettext calls by setting `env.newstyle_gettext = True` or passing `newstyle=True` to `env.install_translations`. They are fully supported by the Babel extraction tool, but might not work as expected with other extraction tools.

With standard `gettext` calls, string formatting is a separate step done with the `|format` filter. This requires duplicating work for `ngettext` calls.

```
{{ gettext("Hello, World!") }}
{{ gettext("Hello, %(name)s!")|format(name=name) }}
{{ ngettext(
    "%(num)d apple", "%(num)d apples", apples|count
  )|format(num=apples|count) }}
{{ pgettext("greeting", "Hello, World!") }}
{{ npgettext(
    "fruit", "%(num)d apple", "%(num)d apples", apples|count
  )|format(num=apples|count) }}
```

New style `gettext` make formatting part of the call, and behind the scenes enforce more consistency.

```
{{ gettext("Hello, World!") }}
{{ gettext("Hello, %(name)s!", name=name) }}
{{ ngettext("%(num)d apple", "%(num)d apples", apples|count) }}
{{ pgettext("greeting", "Hello, World!") }}
{{ npgettext("fruit", "%(num)d apple", "%(num)d apples", apples|count) }}
```

The advantages of newstyle gettext are:

- There's no separate formatting step, you don't have to remember to use the `|format` filter.

- Only named placeholders are allowed. This solves a common problem translators face because positional placeholders can't switch positions meaningfully. Named placeholders always carry semantic information about what value goes where.

- String formatting is used even if no placeholders are used, which makes all strings use a consistent format. Remember to escape any raw percent signs as `%%`, such as `100%%`.

- The translated string is marked safe, formatting performs escaping as needed. Mark a parameter as `|safe` if it has already been escaped.

## 6.3 Expression Statement

**Import name:** `jinja2.ext.do`

The "do" aka expression-statement extension adds a simple `do` tag to the template engine that works like a variable expression but ignores the return value.

## 6.4 Loop Controls

**Import name:** `jinja2.ext.loopcontrols`

This extension adds support for `break` and `continue` in loops. After enabling, Jinja provides those two keywords which work exactly like in Python.

## 6.5 With Statement

**Import name:** `jinja2.ext.with_`

Changed in version 2.9: This extension is now built-in and no longer does anything.

## 6.6 Autoescape Extension

**Import name:** `jinja2.ext.autoescape`

Changed in version 2.9: This extension was removed and is now built-in. Enabling the extension no longer does anything.

## 6.7 Debug Extension

**Import name:** `jinja2.ext.debug`

Adds a `{% debug %}` tag to dump the current context as well as the available filters and tests. This is useful to see what's available to use in the template without setting up a debugger.

## 6.8 Writing Extensions

By writing extensions you can add custom tags to Jinja. This is a non-trivial task and usually not needed as the default tags and expressions cover all common use cases. The i18n extension is a good example of why extensions are useful. Another one would be fragment caching.

When writing extensions you have to keep in mind that you are working with the Jinja template compiler which does not validate the node tree you are passing to it. If the AST is malformed you will get all kinds of compiler or runtime errors that are horrible to debug. Always make sure you are using the nodes you create correctly. The API documentation below shows which nodes exist and how to use them.

## 6.9 Example Extensions

### 6.9.1 Cache

The following example implements a `cache` tag for Jinja by using the cachelib library:

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


class FragmentCacheExtension(Extension):
    # a set of names that trigger the extension.
    tags = {"cache"}

    def __init__(self, environment):
        super().__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 = 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)
```

(continues on next page)

```
        return rv
```

And here is how you use it in an environment:

```python
from jinja2 import Environment
from cachelib import SimpleCache

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

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

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

## 6.9.2 Inline `gettext`

The following example demonstrates using `Extension.filter_stream()` to parse calls to the `_()` gettext function inline with static data without needing Jinja blocks.

```
<h1>_(Welcome)</h1>
<p>_(This is a paragraph)</p>
```

It requires the i18n extension to be loaded and configured.

```python
import re

from jinja2.exceptions import TemplateSyntaxError
from jinja2.ext import Extension
from jinja2.lexer import count_newlines
from jinja2.lexer import Token


_outside_re = re.compile(r"\\?(gettext|_)\(")
_inside_re = re.compile(r"\\?[()]")


class InlineGettext(Extension):
    """This extension implements support for inline gettext blocks::

        <h1>_(Welcome)</h1>
        <p>_(This is a paragraph)</p>

    Requires the i18n extension to be loaded and configured.
    """

    def filter_stream(self, stream):
```

---

```python
        paren_stack = 0

    for token in stream:
        if token.type != "data":
            yield token
            continue

        pos = 0
        lineno = token.lineno

        while True:
            if not paren_stack:
                match = _outside_re.search(token.value, pos)
            else:
                match = _inside_re.search(token.value, pos)
            if match is None:
                break
            new_pos = match.start()
            if new_pos > pos:
                preval = token.value[pos:new_pos]
                yield Token(lineno, "data", preval)
                lineno += count_newlines(preval)
            gtok = match.group()
            if gtok[0] == "\\":
                yield Token(lineno, "data", gtok[1:])
            elif not paren_stack:
                yield Token(lineno, "block_begin", None)
                yield Token(lineno, "name", "trans")
                yield Token(lineno, "block_end", None)
                paren_stack = 1
            else:
                if gtok == "(" or paren_stack > 1:
                    yield Token(lineno, "data", gtok)
                paren_stack += -1 if gtok == ")" else 1
                if not paren_stack:
                    yield Token(lineno, "block_begin", None)
                    yield Token(lineno, "name", "endtrans")
                    yield Token(lineno, "block_end", None)
            pos = match.end()

        if pos < len(token.value):
            yield Token(lineno, "data", token.value[pos:])

    if paren_stack:
        raise TemplateSyntaxError(
            "unclosed gettext expression",
            token.lineno,
            stream.name,
            stream.filename,
        )
```

# 6.10 Extension API

## 6.10.1 Extension

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 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).

> **Parameters**
> **environment** (*Environment*) –

**identifier**

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

**tags**

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

**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)
```

> **Parameters**
>
> - **name** (*str*) –
>
> - **lineno** (*Optional[int]*) –
>
> **Return type**
> ExtensionAttribute

**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*.

> **Parameters**
>
> - **name** (*str*) –
>
> - **args** (*Optional[List[Expr]]*) –
>
> - **kwargs** (*Optional[List[Keyword]]*) –
>
> - **dyn_args** (*Optional[Expr]*) –
>
> - **dyn_kwargs** (*Optional[Expr]*) –

- **lineno** (*Optional[int]*) –

> **Return type**
>> 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 *Token*s, but it doesn't have to return a *TokenStream*.

> **Parameters**
>> **stream** (TokenStream) –

> **Return type**
>> *Union*[*TokenStream*, *Iterable*[*Token*]]

**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.

> **Parameters**
>> **parser** (Parser) –

> **Return type**
>> *Union*[Node, *List*[Node]]

**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.

> **Parameters**
>> - **source** (*str*) –
>> - **name** (*Optional[str]*) –
>> - **filename** (*Optional[str]*) –

> **Return type**
>> str

## 6.10.2 Parser

The parser passed to *Extension.parse()* provides ways to parse expressions of different types. The following methods may be used by extensions:

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

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

> **Parameters**
>> - **environment** (Environment) –
>> - **source** (*str*) –
>> - **name** (*Optional[str]*) –
>> - **filename** (*Optional[str]*) –
>> - **state** (*Optional[str]*) –

**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, passed line number or last line number as well as the current name and filename.

> **Parameters**
>
> - **msg** (*str*) –
> - **lineno** (*Optional[int]*) –
> - **exc** (*Type[TemplateSyntaxError]*) –
>
> **Return type**
> te.NoReturn

**free_identifier**(*lineno=None*)

Return a new free identifier as *InternalName*.

> **Parameters**
> **lineno** (*Optional[int]*) –
>
> **Return type**
> InternalName

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

Parse an assignment target. As Jinja 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. If *with_namespace* is enabled, a namespace assignment may be parsed.

> **Parameters**
>
> - **with_tuple** (*bool*) –
> - **name_only** (*bool*) –
> - **extra_end_rules** (*Optional[Tuple[str, ...]]*) –
> - **with_namespace** (*bool*) –
>
> **Return type**
> *Union*[NSRef, Name, Tuple]

**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.

> **Parameters**
> **with_condexpr** (*bool*) –

> **Return type**
> Expr

**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.
>
> > **Parameters**
> > - **end_tokens** (*Tuple[str, ...]*) –
> > - **drop_needle** (*bool*) –
> >
> > **Return type**
> > *List*[Node]

**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.
>
> > **Parameters**
> > - **simplified** (*bool*) –
> > - **with_condexpr** (*bool*) –
> > - **extra_end_rules** (*Optional[Tuple[str, ...]]*) –
> > - **explicit_parentheses** (*bool*) –
> >
> > **Return type**
> > *Union*[Tuple, Expr]

**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*.
>
> > **Parameters**
> > - **generator** (*Iterable[Token]*) –
> > - **name** (*Optional[str]*) –
> > - **filename** (*Optional[str]*) –
>
> **current**
> > The current *Token*.

**__next__**()

> Go one token ahead and return the old one.
>
> Use the built-in `next()` instead of calling this directly.
>
> > **Return type**
> > > Token

**property eos:** `bool`

> 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()*.
>
> > **Parameters**
> > > **expr** (`str`) –
> >
> > **Return type**
> > > Token

**look**()

> Look at the next token.
>
> > **Return type**
> > > Token

**next_if**(*expr*)

> Perform the token test and return the token if it matched. Otherwise the return value is *None*.
>
> > **Parameters**
> > > **expr** (`str`) –
> >
> > **Return type**
> > > *Optional*[Token]

**push**(*token*)

> Push a token back to the stream.
>
> > **Parameters**
> > > **token** (Token) –
> >
> > **Return type**
> > > None

**skip**(*n=1*)

> Got n tokens ahead.
>
> > **Parameters**
> > > **n** (`int`) –
> >
> > **Return type**
> > > None

**skip_if**(*expr*)

> Like *next_if()* but only returns *True* or *False*.
>
> > **Parameters**
> > > **expr** (`str`) –
> >
> > **Return type**
> > > `bool`

---

**class** `jinja2.lexer.Token`(*lineno*, *type*, *value*)

> **Parameters**
>
> > - **lineno** (*int*) –
> > - **type** (*str*) –
> > - **value** (*str*) –
>
> **lineno**
>
> > The line number of the token
>
> **type**
>
> > The type of the token. This string is interned so you may compare it 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 a token type or `'token_type:token_value'`. This can only test against string values and types.
> >
> > > **Parameters**
> > > > **expr** (*str*) –
> > >
> > > **Return type**
> > > > bool
>
> **test_any**(*\*iterable*)
>
> > Test against multiple token expressions.
> >
> > > **Parameters**
> > > > **iterable** (*str*) –
> > >
> > > **Return type**
> > > > bool

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

`jinja2.lexer.`**count_newlines**(*value*)

> Count the number of newline characters in the string. This is useful for extensions that filter a stream.
>
> > **Parameters**
> > > **value** (*str*) –
> >
> > **Return type**
> > > int

## 6.10.3 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 Python code objects. Extensions that provide custom statements can return nodes to execute custom Python code.

The list below describes all nodes that are currently available. The AST may change between Jinja 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 Jinja 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.
>
> > **Parameters**
> >
> > - **fields** (*Any*) –
> > - **attributes** (*Any*) –
>
> **find**(*node_type*)
>
> > Find the first node of a given type. If no such node exists the return value is *None*.
> >
> > > **Parameters**
> > > **node_type** (*Type[_NodeBound]*) –
> > >
> > > **Return type**
> > > *Optional*[*_NodeBound*]
>
> **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.
> >
> > > **Parameters**
> > > **node_type** (*Union[Type[_NodeBound], Tuple[Type[_NodeBound], ...]]*) –
> > >
> > > **Return type**
> > > *Iterator*[*_NodeBound*]
>
> **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.
> >
> > > **Parameters**
> > >
> > > - **exclude** (*Optional[Container[str]]*) –
> > > - **only** (*Optional[Container[str]]*) –
> > >
> > > **Return type**
> > > *Iterator*[*Node*]
>
> **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.
> >
> > > **Parameters**
> > >
> > > - **exclude** (*Optional[Container[str]]*) –
> > > - **only** (*Optional[Container[str]]*) –

> > **Return type**
> > *Iterator*[*Tuple*[str, *Any*]]

> **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.
> >
> > > **Parameters**
> > > **ctx** (*str*) –
> > >
> > > **Return type**
> > > Node

> **set_environment**(*environment*)
>
> > Set the environment for all nodes.
> >
> > > **Parameters**
> > > **environment** (*Environment*) –
> > >
> > > **Return type**
> > > *Node*

> **set_lineno**(*lineno*, *override=False*)
>
> > Set the line numbers of the node and children.
> >
> > > **Parameters**
> > >
> > > - **lineno** (*int*) –
> > > - **override** (*bool*) –
> > >
> > > **Return type**
> > > Node

**class** jinja2.nodes.**Expr**

> Baseclass for all expressions.
>
> > **Node type**
> > *Node*
>
> > **Parameters**
> >
> > - **fields** (*Any*) –
> > - **attributes** (*Any*) –

> **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.
> >
> > > **Parameters**
> > > **eval_ctx** (*Optional*[*EvalContext*]) –
> > >
> > > **Return type**
> > > *Any*

**can_assign**()
>    Check if it's possible to assign something to this node.

>    > **Return type**
>    >    [bool](...)

**class** jinja2.nodes.**_FilterTestCommon**(*node*, *name*, *args*, *kwargs*, *dyn_args*, *dyn_kwargs*)

>    **Node type**
>    >    [*Expr*](...)

>    **Parameters**
>    >    - **fields** (*Any*) –
>    >    - **attributes** (*Any*) –

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

>    Apply a filter to an expression. `name` is the name of the filter, the other fields are the same as [`Call`](...).

>    If `node` is `None`, the filter is being used in a filter block and is applied to the content of the block.

>    **Node type**
>    >    [*_FilterTestCommon*](...)

>    **Parameters**
>    >    - **fields** (*Any*) –
>    >    - **attributes** (*Any*) –

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

>    Apply a test to an expression. `name` is the name of the test, the other field are the same as [`Call`](...).

>    Changed in version 3.0: `as_const` shares the same logic for filters and tests. Tests check for volatile, async, and `@pass_context` etc. decorators.

>    **Node type**
>    >    [*_FilterTestCommon*](...)

>    **Parameters**
>    >    - **fields** (*Any*) –
>    >    - **attributes** (*Any*) –

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

>    Baseclass for all binary expressions.

>    **Node type**
>    >    [*Expr*](...)

>    **Parameters**
>    >    - **fields** (*Any*) –
>    >    - **attributes** (*Any*) –

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

>    Add the left to the right node.

>    **Node type**
>    >    [*BinExpr*](...)

>    **Parameters**

- **fields** (*Any*) –

- **attributes** (*Any*) –

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

Short circuited AND.

> **Node type**
> [*BinExpr*](#)

> **Parameters**

> > - **fields** (*Any*) –
> >
> > - **attributes** (*Any*) –

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

Divides the left by the right node.

> **Node type**
> [*BinExpr*](#)

> **Parameters**

> > - **fields** (*Any*) –
> >
> > - **attributes** (*Any*) –

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

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

> **Node type**
> [*BinExpr*](#)

> **Parameters**

> > - **fields** (*Any*) –
> >
> > - **attributes** (*Any*) –

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

Left modulo right.

> **Node type**
> [*BinExpr*](#)

> **Parameters**

> > - **fields** (*Any*) –
> >
> > - **attributes** (*Any*) –

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

Multiplies the left with the right node.

> **Node type**
> [*BinExpr*](#)

> **Parameters**

> > - **fields** (*Any*) –
> >
> > - **attributes** (*Any*) –

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

Short circuited OR.

> **Node type**
> [*BinExpr*](#)
>
> **Parameters**
> - **fields** (*Any*) –
> - **attributes** (*Any*) –

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

Left to the power of right.

> **Node type**
> [*BinExpr*](#)
>
> **Parameters**
> - **fields** (*Any*) –
> - **attributes** (*Any*) –

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

Subtract the right from the left node.

> **Node type**
> [*BinExpr*](#)
>
> **Parameters**
> - **fields** (*Any*) –
> - **attributes** (*Any*) –

**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.

> **Node type**
> [*Expr*](#)
>
> **Parameters**
> - **fields** (*Any*) –
> - **attributes** (*Any*) –

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

Compares an expression with some other expressions. *ops* must be a list of [*Operand*](#)s.

> **Node type**
> [*Expr*](#)
>
> **Parameters**
> - **fields** (*Any*) –
> - **attributes** (*Any*) –

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

> Concatenates the list of expressions provided after converting them to strings.
>
> > **Node type**
> >> [*Expr*](#)
> >
> > **Parameters**
> >> • **fields** (*[Any](#)*) –
> >>
> >> • **attributes** (*[Any](#)*) –

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

> A conditional expression (inline if expression). ({{ foo if bar else baz }})
>
> > **Node type**
> >> [*Expr*](#)
> >
> > **Parameters**
> >> • **fields** (*[Any](#)*) –
> >>
> >> • **attributes** (*[Any](#)*) –

**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'))
> ```
>
> This is basically equivalent to using the [*pass_context()*](#) decorator when using the high-level API, which causes a reference to the context to be passed as the first argument to a function.
>
> > **Node type**
> >> [*Expr*](#)
> >
> > **Parameters**
> >> • **fields** (*[Any](#)*) –
> >>
> >> • **attributes** (*[Any](#)*) –

**class** jinja2.nodes.**DerivedContextReference**

> Return the current template context including locals. Behaves exactly like [*ContextReference*](#), but includes local variables, such as from a `for` loop.
>
> New in version 2.11.
>
> > **Node type**
> >> [*Expr*](#)
> >
> > **Parameters**
> >> • **fields** (*[Any](#)*) –
> >>
> >> • **attributes** (*[Any](#)*) –

**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.

> **Node type**
>> [*Expr*](#)
>
> **Parameters**
>> • **fields** (*Any*) –
>>
>> • **attributes** (*Any*) –

**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*](#)
>
> **Parameters**
>> • **fields** (*Any*) –
>>
>> • **attributes** (*Any*) –

**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.
>
> **Node type**
>> [*Expr*](#)
>
> **Parameters**
>> • **fields** (*Any*) –
>>
>> • **attributes** (*Any*) –

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

> Get an attribute or item from an expression and prefer the item.
>
> **Node type**
>> [*Expr*](#)
>
> **Parameters**
>> • **fields** (*Any*) –
>>
>> • **attributes** (*Any*) –

**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.
>
> **Node type**
>> [*Expr*](#)
>
> **Parameters**
>> • **fields** (*Any*) –
>>
>> • **attributes** (*Any*) –

**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 treated specially by the compiler.

> > **Node type**
> > *Expr*

**class** jinja2.nodes.**Literal**

> Baseclass for literals.

> > **Node type**
> > *Expr*

> > **Parameters**

> > > • **fields** (*Any*) –

> > > • **attributes** (*Any*) –

**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).

> > **Node type**
> > *Literal*

> > **Parameters**

> > > • **fields** (*Any*) –

> > > • **attributes** (*Any*) –

**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*

> > **Parameters**

> > > • **fields** (*Any*) –

> > > • **attributes** (*Any*) –

**class** jinja2.nodes.**List**(*items*)

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

> > **Node type**
> > *Literal*

> > **Parameters**

> > > • **fields** (*Any*) –

> > > • **attributes** (*Any*) –

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

> A constant template string.

> > **Node type**
> > *Literal*

> > **Parameters**

> > > • **fields** (*Any*) –

> > > • **attributes** (*Any*) –

**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.

> **Node type**
> > *Literal*
>
> **Parameters**
> > - **fields** (*Any*) –
> > - **attributes** (*Any*) –

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

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

> **Node type**
> > *Expr*
>
> **Parameters**
> > - **fields** (*Any*) –
> > - **attributes** (*Any*) –

**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.

> **Node type**
> > *Expr*
>
> **Parameters**
> > - **fields** (*Any*) –
> > - **attributes** (*Any*) –

**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*
>
> **Parameters**
> > - **fields** (*Any*) –
> > - **attributes** (*Any*) –

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

Reference to a namespace value assignment

> **Node type**
> > *Expr*
>
> **Parameters**

- **fields** (*Any*) –

- **attributes** (*Any*) –

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

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

> **Node type**
> > [*Expr*](#)
>
> **Parameters**
> > - **fields** (*Any*) –
> >
> > - **attributes** (*Any*) –

**class** jinja2.nodes.**UnaryExpr**(*node*)

Baseclass for all unary expressions.

> **Node type**
> > [*Expr*](#)
>
> **Parameters**
> > - **fields** (*Any*) –
> >
> > - **attributes** (*Any*) –

**class** jinja2.nodes.**Neg**(*node*)

Make the expression negative.

> **Node type**
> > [*UnaryExpr*](#)
>
> **Parameters**
> > - **fields** (*Any*) –
> >
> > - **attributes** (*Any*) –

**class** jinja2.nodes.**Not**(*node*)

Negate the expression.

> **Node type**
> > [*UnaryExpr*](#)
>
> **Parameters**
> > - **fields** (*Any*) –
> >
> > - **attributes** (*Any*) –

**class** jinja2.nodes.**Pos**(*node*)

Make the expression positive (noop for most expressions)

> **Node type**
> > [*UnaryExpr*](#)
>
> **Parameters**
> > - **fields** (*Any*) –
> >
> > - **attributes** (*Any*) –

**class** jinja2.nodes.**Helper**

> Nodes that exist in a specific context only.
>
> > **Node type**
> > > [*Node*](#)
> >
> > **Parameters**
> > > - **fields** (*Any*) –
> > > - **attributes** (*Any*) –

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

> A key, value pair for keyword arguments where key is a string.
>
> > **Node type**
> > > [*Helper*](#)
> >
> > **Parameters**
> > > - **fields** (*Any*) –
> > > - **attributes** (*Any*) –

**class** jinja2.nodes.**Operand**(*op*, *expr*)

> Holds an operator and an expression.
>
> > **Node type**
> > > [*Helper*](#)
> >
> > **Parameters**
> > > - **fields** (*Any*) –
> > > - **attributes** (*Any*) –

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

> A key, value pair for dicts.
>
> > **Node type**
> > > [*Helper*](#)
> >
> > **Parameters**
> > > - **fields** (*Any*) –
> > > - **attributes** (*Any*) –

**class** jinja2.nodes.**Stmt**

> Base node for all statements.
>
> > **Node type**
> > > [*Node*](#)
> >
> > **Parameters**
> > > - **fields** (*Any*) –
> > > - **attributes** (*Any*) –

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

> Assigns an expression to a target.
>
> > **Node type**
> > > [*Stmt*](#)

> Parameters
>
> > - **fields** (*Any*) –
> >
> > - **attributes** (*Any*) –

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

> Assigns a block to a target.
>
> > **Node type**
> > > [*Stmt*](#)
> >
> > **Parameters**
> >
> > > - **fields** (*Any*) –
> > >
> > > - **attributes** (*Any*) –

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

> A node that represents a block.
>
> Changed in version 3.0.0: the *required* field was added.
>
> > **Node type**
> > > [*Stmt*](#)
> >
> > **Parameters**
> >
> > > - **fields** (*Any*) –
> > >
> > > - **attributes** (*Any*) –

class jinja2.nodes.**Break**

> Break a loop.
>
> > **Node type**
> > > [*Stmt*](#)
> >
> > **Parameters**
> >
> > > - **fields** (*Any*) –
> > >
> > > - **attributes** (*Any*) –

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.
>
> > **Node type**
> > > [*Stmt*](#)
> >
> > **Parameters**
> >
> > > - **fields** (*Any*) –
> > >
> > > - **attributes** (*Any*) –

class jinja2.nodes.**Continue**

> Continue a loop.
>
> > **Node type**
> > > [*Stmt*](#)
> >
> > **Parameters**
> >
> > > - **fields** (*Any*) –

- **attributes** (*Any*) –

**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))])
```

> **Node type**
> > *Stmt*
>
> **Parameters**
>
> > - **fields** (*Any*) –
> >
> > - **attributes** (*Any*) –

**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*
>
> **Parameters**
>
> > - **fields** (*Any*) –
> >
> > - **attributes** (*Any*) –

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

A statement that evaluates an expression and discards the result.

> **Node type**
> > *Stmt*
>
> **Parameters**
>
> > - **fields** (*Any*) –
> >
> > - **attributes** (*Any*) –

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

Represents an extends statement.

> **Node type**
> > *Stmt*
>
> **Parameters**
>
> > - **fields** (*Any*) –
> >
> > - **attributes** (*Any*) –

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

Node for filter sections.

> **Node type**
> > *Stmt*
>
> **Parameters**

- **fields** (*Any*) –

- **attributes** (*Any*) –

**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*.

> **Node type**
> *Stmt*

> **Parameters**
> - **fields** (*Any*) –
>
> - **attributes** (*Any*) –

**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.

> **Node type**
> *Stmt*

> **Parameters**
> - **fields** (*Any*) –
>
> - **attributes** (*Any*) –

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

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

> **Node type**
> *Stmt*

> **Parameters**
> - **fields** (*Any*) –
>
> - **attributes** (*Any*) –

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

A node that represents the import tag.

> **Node type**
> *Stmt*

> **Parameters**
> - **fields** (*Any*) –
>
> - **attributes** (*Any*) –

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

A node that represents the include tag.

> **Node type**
>> *Stmt*
>
> **Parameters**
>
> > - **fields** (*Any*) –
> >
> > - **attributes** (*Any*) –

**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.
>
> > **Node type**
> >> *Stmt*
> >
> > **Parameters**
> >
> > > - **fields** (*Any*) –
> > >
> > > - **attributes** (*Any*) –

**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*
> >
> > **Parameters**
> >
> > > - **fields** (*Any*) –
> > >
> > > - **attributes** (*Any*) –

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

> An overlay scope for extensions. This is a largely unoptimized scope that however can be used to introduce completely arbitrary variables into a sub scope from a dictionary or dictionary like object. The *context* field has to evaluate to a dictionary object.
>
> Example usage:

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

> New in version 2.10.
>
> > **Node type**
> >> *Stmt*
> >
> > **Parameters**
> >
> > > - **fields** (*Any*) –
> > >
> > > - **attributes** (*Any*) –

**class** jinja2.nodes.**Scope**(*body*)

> An artificial scope.
>
> > **Node type**
> >> *Stmt*
> >
> > **Parameters**
> >
> > > - **fields** (*Any*) –

     • **attributes** (*Any*) –

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

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

    New in version 2.9.3.

        **Node type**
            *Stmt*

        **Parameters**

            • **fields** (*Any*) –

            • **attributes** (*Any*) –

**class** jinja2.nodes.**Template**(*body*)

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

        **Node type**
            *Node*

        **Parameters**

            • **fields** (*Any*) –

            • **attributes** (*Any*) –

**exception** jinja2.nodes.**Impossible**

    Raised if the node could not perform a requested action.

# INTEGRATION

## 7.1 Flask

The Flask web application framework, also maintained by Pallets, uses Jinja templates by default. Flask sets up a Jinja environment and template loader for you, and provides functions to easily render templates from view functions.

## 7.2 Django

Django supports using Jinja as its template engine, see https://docs.djangoproject.com/en/stable/topics/templates/ #support-for-template-engines.

## 7.3 Babel

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

Gettext messages are extracted from both `trans` tags and code expressions.

To extract gettext messages from templates, the project needs a Jinja section in its Babel extraction method mapping file:

```
[jinja2: **/templates/**.html]
encoding = utf-8
```

The syntax related options of the `Environment` are also available as configuration values in the mapping file. For example, to tell the extractor that templates use `%` as `line_statement_prefix` you can use this code:

```
[jinja2: **/templates/**.html]
encoding = utf-8
line_statement_prefix = %
```

*Extensions* may also be defined by passing a comma separated list of import paths as the `extensions` value. The i18n extension is added automatically.

Template syntax errors are ignored by default. The assumption is that tests will catch syntax errors in templates. If you don't want to ignore errors, add `silent = false` to the settings.

## 7.4 Pylons

It's easy to integrate Jinja into a Pylons application.

The template engine is configured in `config/environment.py`. The configuration for Jinja looks something like this:

```python
from jinja2 import Environment, PackageLoader
config['pylons.app_globals'].jinja_env = Environment(
    loader=PackageLoader('yourapplication', 'templates')
)
```

After that you can render Jinja templates by using the `render_jinja` function from the `pylons.templating` module.

Additionally it's a good idea to set the Pylons `c` object to strict mode. By default attribute access on missing attributes on the `c` object returns an empty string and not an undefined object. To change this add this to `config/environment.py`:

```python
config['pylons.strict_c'] = True
```

# SWITCHING FROM OTHER TEMPLATE ENGINES

This is a brief guide on some of the differences between Jinja syntax and other template languages. See *Template Designer Documentation* for a comprehensive guide to Jinja syntax and features.

## 8.1 Django

If you have previously worked with Django templates, you should find Jinja very familiar. Many of the syntax elements look and work the same. However, Jinja provides some more syntax elements, and some work a bit differently.

This section covers the template changes. The API, including extension support, is fundamentally different so it won't be covered here.

Django supports using Jinja as its template engine, see https://docs.djangoproject.com/en/stable/topics/templates/ #support-for-template-engines.

### 8.1.1 Method Calls

In Django, methods are called implicitly, without parentheses.

```
{% for page in user.get_created_pages %}
    ...
{% endfor %}
```

In Jinja, using parentheses is required for calls, like in Python. This allows you to pass variables to the method, which is not possible in Django. This syntax is also used for calling macros.

```
{% for page in user.get_created_pages() %}
    ...
{% endfor %}
```

### 8.1.2 Filter Arguments

In Django, one literal value can be passed to a filter after a colon.

```
{{ items|join:", " }}
```

In Jinja, filters can take any number of positional and keyword arguments in parentheses, like function calls. Arguments can also be variables instead of literal values.

```
{{ items|join(", ") }}
```

### 8.1.3 Tests

In addition to filters, Jinja also has "tests" used with the `is` operator. This operator is not the same as the Python operator.

```
{% if user.user_id is odd %}
    {{ user.username|e }} is odd
{% else %}
    hmm. {{ user.username|e }} looks pretty normal
{% endif %}
```

### 8.1.4 Loops

In Django, the special variable for the loop context is called `forloop`, and the `empty` is used for no loop items.

```
{% for item in items %}
    {{ item }}
{% empty %}
    No items!
{% endfor %}
```

In Jinja, the special variable for the loop context is called `loop`, and the `else` block is used for no loop items.

```
{% for item in items %}
    {{ loop.index}}. {{ item }}
{% else %}
    No items!
{% endfor %}
```

### 8.1.5 Cycle

In Django, the `{% cycle %}` can be used in a for loop to alternate between values per loop.

```
{% for user in users %}
    <li class="{% cycle 'odd' 'even' %}">{{ user }}</li>
{% endfor %}
```

In Jinja, the `loop` context has a `cycle` method.

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

A cycler can also be assigned to a variable and used outside or across loops with the `cycle()` global function.

## 8.2 Mako

You can configure Jinja to look more like Mako:

```
env = Environment(
    block_start_string="<%",
    block_end_string="%>",
    variable_start_string="${",
    variable_end_string="}",
    comment_start_string="<%doc>",
    commend_end_string="</%doc>",
    line_statement_prefix="%",
    line_comment_prefix="##",
)
```

With an environment configured like that, Jinja should be able to interpret a small subset of Mako templates without any changes.

Jinja does not support embedded Python code, so you would have to move that out of the template. You could either process the data with the same code before rendering, or add a global function or filter to the Jinja environment.

The syntax for defs (which are called macros in Jinja) 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 Jinja 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 %>
```

# TIPS AND TRICKS

This part of the documentation shows some tips and tricks for Jinja templates.

## 9.1 Null-Default Fallback

Jinja supports dynamic inheritance and does not distinguish between parent and child template as long as no *extends* tag is visited. While this leads to the surprising behavior that everything before the first *extends* tag including whitespace is printed out instead of being ignored, it can be used for a neat trick.

Usually child templates extend from one template that adds a basic HTML skeleton. However it's possible to put the *extends* tag into an *if* tag to only extend from the layout template if the *standalone* variable evaluates to false which it does per default if it's not defined. Additionally a very basic 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 'default.html' %}{% endif -%}
<!DOCTYPE html>
<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 %}
```

## 9.2 Alternating Rows

If you want to have different styles for each row of a table or list 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 number of strings. Each time this tag is encountered the next item from the list is rendered.

## 9.3 Highlighting Active Menu Items

Often you want to have a navigation bar with an active navigation item. This is really simple to achieve. Because assignments outside of *block*s in child templates are global and executed before the layout template is evaluated it's possible to define the active menu item in the child template:

```
{% extends "layout.html" %}
{% set active_page = "index" %}
```

The layout template can then access *active_page*. Additionally it makes sense 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>
...
```

## 9.4 Accessing the parent Loop

The special *loop* variable always points to the innermost loop. If it's desired 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>
```

# FREQUENTLY ASKED QUESTIONS

## 10.1 Why is it called Jinja?

"Jinja" is a Japanese Shinto shrine, or temple, and temple and template share a similar English pronunciation. It is not named after the city in Uganda.

## 10.2 How fast is Jinja?

Jinja is relatively fast among template engines because it compiles and caches template code to Python code, so that the template does not need to be parsed and interpreted each time. Rendering a template becomes as close to executing a Python function as possible.

Jinja also makes extensive use of caching. Templates are cached by name after loading, so future uses of the template avoid loading. The template loading itself uses a bytecode cache to avoid repeated compiling. The caches can be external to persist across restarts. Templates can also be precompiled and loaded as fast Python imports.

We dislike benchmarks because they don't reflect real use. Performance depends on many factors. Different engines have different default configurations and tradeoffs that make it unclear how to set up a useful comparison. Often, database access, API calls, and data processing have a much larger effect on performance than the template engine.

## 10.3 Isn't it a bad idea to put logic in templates?

Without a doubt you should try to remove as much logic from templates as possible. With less logic, the template is easier to understand, has fewer potential side effects, and is faster to compile and render. But a template without any logic means processing must be done in code before rendering. A template engine that does that is shipped with Python, called `string.Template`, and while it's definitely fast it's not convenient.

Jinja's features such as blocks, statements, filters, and function calls make it much easier to write expressive templates, with very few restrictions. Jinja doesn't allow arbitrary Python code in templates, or every feature available in the Python language. This keeps the engine easier to maintain, and keeps templates more readable.

Some amount of logic is required in templates to keep everyone happy. Too much logic in the template can make it complex to reason about and maintain. It's up to you to decide how your application will work and balance how much logic you want to put in the template.

## 10.4 Why is HTML escaping not the default?

Jinja provides a feature that can be enabled to escape HTML syntax in rendered templates. However, it is disabled by default.

Jinja is a general purpose template engine, it is not only used for HTML documents. You can generate plain text, LaTeX, emails, CSS, JavaScript, configuration files, etc. HTML escaping wouldn't make sense for any of these document types.

While automatic escaping means that you are less likely have an XSS problem, it also requires significant extra processing during compiling and rendering, which can reduce performance. Jinja uses MarkupSafe for escaping, which provides optimized C code for speed, but it still introduces overhead to track escaping across methods and formatting.

# BSD-3-CLAUSE LICENSE

Copyright 2007 Pallets

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, IN-CIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSI-NESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CON-TRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

# CHANGES

## 12.1 Version 3.2.0

Unreleased

## 12.2 Version 3.1.2

Released 2022-04-28

- Add parameters to `Environment.overlay` to match `__init__`. #1645
- Handle race condition in `FileSystemBytecodeCache`. #1654

## 12.3 Version 3.1.1

Released 2022-03-25

- The template filename on Windows uses the primary path separator. #1637

## 12.4 Version 3.1.0

Released 2022-03-24

- Drop support for Python 3.6. #1534
- Remove previously deprecated code. #1544
    - `WithExtension` and `AutoEscapeExtension` are built-in now.
    - `contextfilter` and `contextfunction` are replaced by `pass_context`. `evalcontextfilter` and `evalcontextfunction` are replaced by `pass_eval_context`. `environmentfilter` and `environmentfunction` are replaced by `pass_environment`.
    - `Markup` and `escape` should be imported from MarkupSafe.
    - Compiled templates from very old Jinja versions may need to be recompiled.
    - Legacy resolve mode for `Context` subclasses is no longer supported. Override `resolve_or_missing` instead of `resolve`.
    - `unicode_urlencode` is renamed to `url_quote`.

- Add support for native types in macros. #1510

- The {% trans %} tag can use pgettext and npgettext by passing a context string as the first token in the tag, like {% trans "title" %}. #1430

- Update valid identifier characters from Python 3.6 to 3.7. #1571

- Filters and tests decorated with @async_variant are pickleable. #1612

- Add items filter. #1561

- Subscriptions ([0], etc.) can be used after filters, tests, and calls when the environment is in async mode. #1573

- The groupby filter is case-insensitive by default, matching other comparison filters. Added the case_sensitive parameter to control this. #1463

- Windows drive-relative path segments in template names will not result in FileSystemLoader and PackageLoader loading from drive-relative paths. #1621

## 12.5 Version 3.0.3

Released 2021-11-09

- Fix traceback rewriting internals for Python 3.10 and 3.11. #1535

- Fix how the native environment treats leading and trailing spaces when parsing values on Python 3.10. #1537

- Improve async performance by avoiding checks for common types. #1514

- Revert change to hash(Node) behavior. Nodes are hashed by id again #1521

- PackageLoader works when the package is a single module file. #1512

## 12.6 Version 3.0.2

Released 2021-10-04

- Fix a loop scoping bug that caused assignments in nested loops to still be referenced outside of it. #1427

- Make compile_templates deterministic for filter and import names. #1452#1453

- Revert an unintended change that caused Undefined to act like StrictUndefined for the in operator. #1448

- Imported macros have access to the current template globals in async environments. #1494

- PackageLoader will not include a current directory (.) path segment. This allows loading templates from the root of a zip import. #1467

## 12.7 Version 3.0.1

Released 2021-05-18

- Update MarkupSafe dependency to >= 2.0. #1418

- Mark top-level names as exported so type checking understands imports in user projects. #1426

- Fix some types that weren't available in Python 3.6.0. #1433

- The deprecation warning for unneeded autoescape and with_ extensions shows more relevant context. #1429

- Fixed calling deprecated `jinja2.Markup` without an argument. Use `markupsafe.Markup` instead. #1438

- Calling sync `render` for an async template uses `asyncio.run` on Python >= 3.7. This fixes a deprecation that Python 3.10 introduces. #1443

## 12.8 Version 3.0.0

Released 2021-05-11

- Drop support for Python 2.7 and 3.5.

- Bump MarkupSafe dependency to >=1.1.

- Bump Babel optional dependency to >=2.1.

- Remove code that was marked deprecated.

- Add type hinting. #1412

- Use **PEP 451** API to load templates with *PackageLoader*. #1168

- Fix a bug that caused imported macros to not have access to the current template's globals. #688

- Add ability to ignore `trim_blocks` using +%}. #1036

- Fix a bug that caused custom async-only filters to fail with constant input. #1279

- Fix UndefinedError incorrectly being thrown on an undefined variable instead of `Undefined` being returned on `NativeEnvironment` on Python 3.10. #1335

- Blocks can be marked as `required`. They must be overridden at some point, but not necessarily by the direct child. #1147

- Deprecate the `autoescape` and `with` extensions, they are built-in to the compiler. #1203

- The `urlize` filter recognizes `mailto:` links and takes `extra_schemes` (or `env.policies["urlize. extra_schemes"]`) to recognize other schemes. It tries to balance parentheses within a URL instead of ignoring trailing characters. The parsing in general has been updated to be more efficient and match more cases. URLs without a scheme are linked as `https://` instead of `http://`. #522 #827 #1172, #1195

- Filters that get attributes, such as `map` and `groupby`, can use a false or empty value as a default. #1331

- Fix a bug that prevented variables set in blocks or loops from being accessed in custom context functions. #768

- Fix a bug that caused scoped blocks from accessing special loop variables. #1088

- Update the template globals when calling `Environment.get_template(globals=...)` even if the template was already loaded. #295

- Do not raise an error for undefined filters in unexecuted if-statements and conditional expressions. #842

- Add `is filter` and `is test` tests to test if a name is a registered filter or test. This allows checking if a filter is available in a template before using it. Test functions can be decorated with `@pass_environment`, `@pass_eval_context`, or `@pass_context`. #842, #1248

- Support `pgettext` and `npgettext` (message contexts) in i18n extension. #441

- The `|indent` filter's `width` argument can be a string to indent by. #1167

- The parser understands hex, octal, and binary integer literals. #1170

- `Undefined.__contains__` (`in`) raises an `UndefinedError` instead of a `TypeError`. #1198

- `Undefined` is iterable in an async environment. #1294

- `NativeEnvironment` supports async mode. #1362

- Template rendering only treats \n, \r\n and \r as line breaks. Other characters are left unchanged. #769 #952 #1313

- |groupby filter takes an optional `default` argument. #1359

- The function and filter decorators have been renamed and unified. The old names are deprecated. #1381

  - `pass_context` replaces `contextfunction` and `contextfilter`.

  - `pass_eval_context` replaces `evalcontextfunction` and `evalcontextfilter`.

  - `pass_environment` replaces `environmentfunction` and `environmentfilter`.

- Async support no longer requires Jinja to patch itself. It must still be enabled with `Environment(enable_async=True)`. #1390

- Overriding `Context.resolve` is deprecated, override `resolve_or_missing` instead. #1380

## 12.9 Version 2.11.3

Released 2021-01-31

- Improve the speed of the `urlize` filter by reducing regex backtracking. Email matching requires a word character at the start of the domain part, and only word characters in the TLD. #1343

## 12.10 Version 2.11.2

Released 2020-04-13

- Fix a bug that caused callable objects with `__getattr__`, like `Mock` to be treated as a `contextfunction()`. #1145

- Update `wordcount` filter to trigger *Undefined* methods by wrapping the input in `soft_str()`. #1160

- Fix a hang when displaying tracebacks on Python 32-bit. #1162

- Showing an undefined error for an object that raises `AttributeError` on access doesn't cause a recursion error. #1177

- Revert changes to *PackageLoader* from 2.10 which removed the dependency on setuptools and pkg_resources, and added limited support for namespace packages. The changes caused issues when using Pytest. Due to the difficulty in supporting Python 2 and **PEP 451** simultaneously, the changes are reverted until 3.0. #1182

- Fix line numbers in error messages when newlines are stripped. #1178

- The special `namespace()` assignment object in templates works in async environments. #1180

- Fix whitespace being removed before tags in the middle of lines when `lstrip_blocks` is enabled. #1138

- *NativeEnvironment* doesn't evaluate intermediate strings during rendering. This prevents early evaluation which could change the value of an expression. #1186

## 12.11 Version 2.11.1

Released 2020-01-30

- Fix a bug that prevented looking up a key after an attribute (`{{ data.items[1:] }}`) in an async template. #1141

## 12.12 Version 2.11.0

Released 2020-01-27

- Drop support for Python 2.6, 3.3, and 3.4. This will be the last version to support Python 2.7 and 3.5.
- Added a new `ChainableUndefined` class to support getitem and getattr on an undefined object. #977
- Allow `{%+` syntax (with NOP behavior) when `lstrip_blocks` is disabled. #748
- Added a `default` parameter for the `map` filter. #557
- Exclude environment globals from *meta.find_undeclared_variables()*. #931
- Float literals can be written with scientific notation, like 2.56e-3. #912, #922
- Int and float literals can be written with the '_' separator for legibility, like 12_345. #923
- Fix a bug causing deadlocks in `LRUCache.setdefault`. #1000
- The `trim` filter takes an optional string of characters to trim. #828
- A new `jinja2.ext.debug` extension adds a `{% debug %}` tag to quickly dump the current context and available filters and tests. #174, #798#983
- Lexing templates with large amounts of whitespace is much faster. #857, #858
- Parentheses around comparisons are preserved, so `{{ 2 * (3 < 5) }}` outputs "2" instead of "False". #755, #938
- Add new `boolean`, `false`, `true`, `integer` and `float` tests. #824
- The environment's `finalize` function is only applied to the output of expressions (constant or not), not static template data. #63
- When providing multiple paths to `FileSystemLoader`, a template can have the same name as a directory. #821
- Always return *Undefined* when omitting the `else` clause in a `{{ 'foo' if bar }}` expression, regardless of the environment's `undefined` class. Omitting the `else` clause is a valid shortcut and should not raise an error when using *StrictUndefined*. #710, #1079
- Fix behavior of `loop` control variables such as `length` and `revindex0` when looping over a generator. #459#751#794, #993
- Async support is only loaded the first time an environment enables it, in order to avoid a slow initial import. #765
- In async environments, the `|map` filter will await the filter call if needed. #913
- In for loops that access `loop` attributes, the iterator is not advanced ahead of the current iteration unless `length`, `revindex`, `nextitem`, or `last` are accessed. This makes it less likely to break `groupby` results. #555, #1101
- In async environments, the `loop` attributes `length` and `revindex` work for async iterators. #1101
- In async environments, values from attribute/property access will be awaited if needed. #1101
- `PackageLoader` doesn't depend on setuptools or pkg_resources. #970

- `PackageLoader` has limited support for **PEP 420** namespace packages. #1097

- Support `os.PathLike` objects in `FileSystemLoader` and `ModuleLoader`. #870

- *NativeTemplate* correctly handles quotes between expressions. `"'{{ a }}', '{{ b }}'"` renders as the tuple `('1', '2')` rather than the string `'1, 2'`. #1020

- Creating a *NativeTemplate* directly creates a *NativeEnvironment* instead of a default *Environment*. #1091

- After calling `LRUCache.copy()`, the copy's queue methods point to the correct queue. #843

- Compiling templates always writes UTF-8 instead of defaulting to the system encoding. #889

- `|wordwrap` filter treats existing newlines as separate paragraphs to be wrapped individually, rather than creating short intermediate lines. #175

- Add `break_on_hyphens` parameter to `|wordwrap` filter. #550

- Cython compiled functions decorated as context functions will be passed the context. #1108

- When chained comparisons of constants are evaluated at compile time, the result follows Python's behavior of returning `False` if any comparison returns `False`, rather than only the last one. #1102

- Tracebacks for exceptions in templates show the correct line numbers and source for Python >= 3.7. #1104

- Tracebacks for template syntax errors in Python 3 no longer show internal compiler frames. #763

- Add a `DerivedContextReference` node that can be used by extensions to get the current context and local variables such as `loop`. #860

- Constant folding during compilation is applied to some node types that were previously overlooked. #733

- `TemplateSyntaxError.source` is not empty when raised from an included template. #457

- Passing an `Undefined` value to `get_template` (such as through `extends`, `import`, or `include`), raises an `UndefinedError` consistently. `select_template` will show the undefined message in the list of attempts rather than the empty string. #1037

- `TemplateSyntaxError` can be pickled. #1117

## 12.13 Version 2.10.3

Released 2019-10-04

- Fix a typo in Babel entry point in `setup.py` that was preventing installation.

## 12.14 Version 2.10.2

Released 2019-10-04

- Fix Python 3.7 deprecation warnings.

- Using `range` in the sandboxed environment uses `xrange` on Python 2 to avoid memory use. #933

- Use Python 3.7's better traceback support to avoid a core dump when using debug builds of Python 3.7. #1050

# 12.15 Version 2.10.1

Released 2019-04-06

- SandboxedEnvironment securely handles `str.format_map` in order to prevent code execution through untrusted format strings. The sandbox already handled `str.format`.

# 12.16 Version 2.10

Released 2017-11-08

- Added a new extension node called `OverlayScope` which can be used to create an unoptimized scope that will look up all variables from a derived context.

- Added an `in` test that works like the in operator. This can be used in combination with `reject` and `select`.

- Added `previtem` and `nextitem` to loop contexts, providing access to the previous/next item in the loop. If such an item does not exist, the value is undefined.

- Added `changed(*values)` to loop contexts, providing an easy way of checking whether a value has changed since the last iteration (or rather since the last call of the method)

- Added a `namespace` function that creates a special object which allows attribute assignment using the `set` tag. This can be used to carry data across scopes, e.g. from a loop body to code that comes after the loop.

- Added a `trimmed` modifier to `{% trans %}` to strip linebreaks and surrounding whitespace. Also added a new policy to enable this for all `trans` blocks.

- The `random` filter is no longer incorrectly constant folded and will produce 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 types instead 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 could modify template context.

- Fixed a bug where having many `{% elif ... %}` blocks resulted in a "too many levels of indentation" error. These blocks now compile to native `elif ..:` instead of `else:  if ..:` #759

## 12.17 Version 2.9.6

Released 2017-04-03

- Fixed custom context behavior in fast resolve mode #675

## 12.18 Version 2.9.5

Released 2017-01-28

- Restored the original repr of the internal `_GroupTuple` because this caused issues with ansible and it was an unintended change. #654
- Added back support for custom contexts that override the old `resolve` method since it was hard for people to spot that this could cause a regression.
- Correctly use the buffer for the else block of for loops. This caused invalid syntax errors to be caused on 2.x and completely wrong behavior on Python 3 #669
- Resolve an issue where the `{% extends %}` tag could not be used with async environments. #668
- Reduce memory footprint slightly by reducing our unicode database dump we use for identifier matching on Python 3 #666
- Fixed autoescaping not working for macros in async compilation mode. #671

## 12.19 Version 2.9.4

Released 2017-01-10

- Solved some warnings for string literals. #646
- Increment the bytecode cache version which was not done due to an oversight before.
- Corrected bad code generation and scoping for filtered loops. #649
- Resolved an issue where top-level output silencing after known extend blocks could generate invalid code when blocks where contained in if statements. #651
- Made the `truncate.leeway` default configurable to improve compatibility with older templates.

## 12.20 Version 2.9.3

Released 2017-01-08

- Restored the use of blocks in macros to the extend that was possible before. On Python 3 it would render a generator repr instead of the block contents. #645
- Set a consistent behavior for assigning of variables in inner scopes when the variable is also read from an outer scope. This now sets the intended behavior in all situations however it does not restore the old 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 the new scoping rules. In some more exotic cases a variable overridden in a local scope would not make it into a block.

- Change the code generation of the `with` statement to be in line with the new scoping rules. This resolves some unlikely bugs in edge cases. This also introduces a new internal `With` node that can be used by extensions.

## 12.21 Version 2.9.2

Released 2017-01-08

- Fixed a regression that caused for loops to not be able to use the same variable for the target as well as source iterator. #640

- Add support for a previously unknown behavior of macros. It used to be possible in some circumstances to explicitly provide a caller argument to macros. While badly buggy and unintended it turns out that this is a common case that gets copy pasted around. To not completely break backwards compatibility with the most common cases it's now possible to provide an explicit keyword argument for caller if it's given an explicit default. #642

## 12.22 Version 2.9.1

Released 2017-01-07

- Resolved a regression with call block scoping for macros. Nested caller blocks that used the same identifiers as outer macros could refer to the wrong variable incorrectly.

## 12.23 Version 2.9

Released 2017-01-07, codename Derivation

- Change cache key definition in environment. This fixes a performance regression 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 operations not 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 undefined values.
- `map` and friends will now give better error messages if you forgot to quote the parameter.
- Depend on MarkupSafe 0.23 or higher.
- Improved the `truncate` filter to support better truncation in case the string is barely truncated at all.
- Change the logic for macro autoescaping to be based on the runtime autoescaping information at call time instead of macro define time.
- Ported a modified version of the `tojson` filter from Flask to Jinja and 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 with the `compiler.ascii_str` policy.

- Tests now no longer accept an arbitrary expression as first argument but a restricted one. This means that you can now properly use multiple tests in one expression without extra parentheses. In particular you can now write `foo is divisibleby 2 or foo is divisibleby 3` as you would expect.

- Greatly changed the scoping system to be more consistent with what template designers and developers expect. There is now no more magic difference between the different include and import constructs. Context is now always propagated the same way. The only remaining differences is the defaults for `with context` and `without context`.

- The `with` and `autoescape` tags are now built-in.

- Added the new `select_autoescape` function which helps configuring better autoescaping easier.

- Fixed a runtime error in the sandbox when attributes of async generators were accessed.

## 12.24 Version 2.8.1

Released 2016-12-29

- 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 sandboxed with the same rules as in Jinja. This solves various information leakage problems that can occur with format strings.

## 12.25 Version 2.8

Released 2015-07-26, codename Replacement

- 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 available instead of load names.

- Fixed loop length calculation for some iterators.

- Changed how Jinja enforces strings to be native strings in Python 2 to work when people break their default encoding.

- Added `make_logging_undefined` which returns an undefined object that logs failures into a logger.

- If unmarshalling of cached data fails the template will be reloaded 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. This change makes {% macro m(x, y=1, z) %} a syntax error. The previous behavior for this code was broken anyway (resulting in the default value being applied to y).

- Add ability to use custom subclasses of `jinja2.compiler.CodeGenerator` and `jinja2.runtime.Context` by adding two new attributes to the environment (`code_generator_class` and `context_class`). #404

- Added support for context/environment/evalctx decorator functions on the finalize callback of the environment.

- Escape query strings for urlencode properly. Previously slashes were not escaped in that place.

- Add 'base' parameter to 'int' filter.

## 12.26 Version 2.7.3

Released 2014-06-06

- Security issue: Corrected the security fix for the cache folder. This fix was provided by RedHat.

## 12.27 Version 2.7.2

Released 2014-01-10

- Prefix loader was not forwarding the locals properly to inner loaders. This is now fixed.
- Security issue: Changed the default folder for the filesystem cache to be user specific and read and write protected on UNIX systems. See Debian bug 734747 for more information.

## 12.28 Version 2.7.1

Released 2013-08-07

- Fixed a bug with `call_filter` not working properly on environment and context filters.
- Fixed lack of Python 3 support for bytecode caches.
- Reverted support for defining blocks in included templates as this broke existing templates for users.
- Fixed some warnings with hashing of undefineds and nodes if Python is run with warnings for Python 3.
- Added support for properly hashing undefined objects.
- Fixed a bug with the title filter not working on already uppercase strings.

## 12.29 Version 2.7

Released 2013-05-20, codename Translation

- Choice and prefix loaders now dispatch source and template lookup separately in order to work in combination with module loaders as advertised.
- Fixed filesizeformat.
- Added a non-silent option for babel extraction.

- Added `urlencode` filter that automatically quotes values for URL safe usage with utf-8 as only supported encoding. If applications want to change this encoding they can override the filter.

- Added `keep-trailing-newline` configuration to environments and templates to optionally preserve the final trailing newline.

- Accessing `last` on the loop context no longer causes the iterator to 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 `wordwrap` filter.

- 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 side of blocks.

- Added `map`, `select`, `reject`, `selectattr` and `rejectattr` filters.

- Added support for `loop.depth` to figure out how deep inside a recursive loop the code is.

- Disabled py_compile for pypy and python 3.

## 12.30 Version 2.6

Released 2011-07-24, codename Convolution

- Internal attributes now raise an internal attribute error now instead of returning an undefined. This fixes problems when passing undefined objects to Python semantics expecting APIs.

- Traceback support now works properly for PyPy. (Tested with 1.4)

- Implemented operator intercepting for sandboxed environments. This allows application developers to disable builtin operators for better security. (For instance limit the mathematical operators to actual integers instead of longs)

- Groupby filter now supports dotted notation for grouping by attributes of 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 code from bytecode caches to prevent segfaults on invalid opcodes. The segfault in earlier Jinja versions here was not a Jinja bug but a limitation in the underlying Python interpreter. If you notice Jinja segfaulting in earlier versions after an upgrade of the Python interpreter you don't have to upgrade, it's enough to flush the bytecode cache. This just no longer makes this necessary, Jinja will automatically detect these cases now.

- The sum filter can now sum up values by attribute. This is a backwards incompatible change. The argument to the filter previously was the optional starting index which defaults to zero. This now became the second 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 objects as 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 with a similar interface.

## 12.31 Version 2.5.5

Released 2010-10-18

- Built documentation is no longer part of release.

## 12.32 Version 2.5.4

Released 2010-10-17

- Fixed extensions not loading properly with overlays.
- Work around a bug in cpython for the debugger that causes segfaults on 64bit big-endian architectures.

## 12.33 Version 2.5.3

Released 2010-10-17

- Fixed an operator precedence error introduced in 2.5.2. Statements like "-foo.bar" had their implicit parentheses applied around the first part of the expression ("(-foo).bar") instead of the more correct "-(foo.bar)".

## 12.34 Version 2.5.2

Released 2010-08-18

- Improved setup.py script to better work with assumptions people might still have from it (`--with-speedups`).
- Fixed a packaging error that excluded the new debug support.

## 12.35 Version 2.5.1

Released 2010-08-17

- StopIteration exceptions raised by functions called from templates are now intercepted and converted to undefineds. This solves a lot of debugging grief. (StopIteration is used internally to abort 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 else than the pluralize count will no longer raise a `KeyError`.
- Removed builtin markup class and switched to markupsafe. For backwards compatibility the pure Python implementation still exists but is pulled from markupsafe by the Jinja developers. The debug support went into a separate feature called "debugsupport" and is disabled by default because it is only relevant for Python 2.4
- Fixed an issue with unary operators having the wrong precedence.

## 12.36  Version 2.5

Released 2010-05-29, codename Incoherence

- Improved the sort filter (should have worked like this for a long time) by adding support for case insensitive searches.

- Fixed a bug for getattribute constant folding.

- Support for newstyle gettext translations which result in a nicer in-template user interface and more consistent catalogs.

- It's now possible to register extensions after an environment was created.

## 12.37  Version 2.4.1

Released 2010-04-20

- Fixed an error reporting bug for undefined.

## 12.38  Version 2.4

Released 2010-04-13, codename Correlation

- The environment template loading functions now transparently pass through a template object if it was passed to it. This makes it possible to import or extend from a template object that was passed to the template.

- Added a `ModuleLoader` that can load templates from precompiled sources. The environment now features a method to compile the templates from a configured loader into a zip file or folder.

- The _speedups C extension now supports Python 3.

- Added support for autoescaping toggling sections and support for evaluation contexts.

- Extensions have a priority now.

## 12.39  Version 2.3.1

Released 2010-02-19

- Fixed an error reporting bug on all python versions

- Fixed an error reporting bug on Python 2.4

## 12.40 Version 2.3

Released 2010-02-10, codename 3000 Pythons

- Fixes issue with code generator that causes unbound variables to be generated if set was used in if-blocks and other small identifier problems.

- Include tags are now able to select between multiple templates and take the first that exists, if a list of templates is given.

- Fixed a problem with having call blocks in outer scopes that have an argument that is also used as local variable in an inner frame #360.

- Greatly improved error message reporting #339

- Implicit tuple expressions can no longer be totally empty. This change makes {% if %} a syntax error now. #364

- Added support for translator comments if extracted via babel.

- Added with-statement extension.

- Experimental Python 3 support.

## 12.41 Version 2.2.1

Released 2009-09-14

- Fixes some smaller problems for Jinja on Jython.

## 12.42 Version 2.2

Released 2009-09-13, codename Kong

- Include statements can now be marked with `ignore missing` to skip non 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 grammar required 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 templates that were unused in the callers frame #327

- Fixed a bug that caused internal errors if names where used as iteration variable 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 the environment because it was clashing with a method of the same name.

- Speedup extension is now disabled by default.

## 12.43 Version 2.1.1

Released 2008-12-25

- Fixed a translation error caused by looping over empty recursive loops.

## 12.44 Version 2.1

Released 2008-11-23, codename Yasuzō

- Fixed a bug with nested loops and the special loop variable. Before the change an inner loop overwrote the loop variable from the outer one after iteration.

- Fixed a bug with the i18n extension that caused the explicit pluralization block 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 an undefined object now instead of raising an index error. This was a bug caused by eager optimizing.

- The i18n extension looks up `foo.ugettext` now followed by `foo.gettext` if an translations object is installed. This makes dealing with custom translations classes easier.

- Fixed a confusing behavior with conditional extending. loops were partially executed under some conditions even though they were not part of a visible area.

- Added `sort` filter that works like `dictsort` but for arbitrary sequences.

- Fixed a bug with empty statements in macros.

- Implemented a bytecode cache system.

- The template context is now weakref-able

- Inclusions and imports "with context" forward all variables now, not only the initial context.

- Added a cycle helper called `cycler`.

- Added a joining helper called `joiner`.

- Added a `compile_expression` method to the environment that allows compiling of Jinja expressions into callable Python objects.

- Fixed an escaping bug in urlize

## 12.45 Version 2.0

Released 2008-07-17, codename Jinjavitus

- The subscribing of objects (looking up attributes and items) changed from slightly. It's now possible to give attributes or items a higher priority by either using dot-notation lookup or the bracket syntax. This also changed the AST slightly. `Subscript` is gone and was replaced with `Getitem` and `Getattr`.

- Added support for preprocessing and token stream filtering for extensions. This would allow extensions to allow simplified gettext calls in template data and something similar.

- Added `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 evaluates to `false`.

- Improved error reporting for undefined values by providing a position.

- `filesizeformat` filter uses decimal prefixes now per default and can be set to binary mode with the second parameter.

- Fixed bug in finalizer

## 12.46 Version 2.0rc1

Released 2008-06-09

- First release of Jinja 2.

# PYTHON MODULE INDEX

## j

# Symbols

## U

## V

## W

## X