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.

Contents:

- Version 2.1.1
- Version 2.1
- Version 2.0
- Version 2.0rc1

Index

_

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X

Python Module Index

j

	j	
−	**jinja2**	*public Jinja API*
	jinja2.ext	
	jinja2.nativetypes	
	jinja2.nodes	
	jinja2.sandbox	

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.

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
```

Dependencies¶

These will be installed automatically when installing Jinja.

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

Optional Dependencies¶

These distributions will not be installed automatically.

- Babel provides translation support in templates.

Contents

Navigation

- Overview
  - Previous: Jinja
  - Next: API

Quick search

Go

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.

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`.

```
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`.

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

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

```
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.

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.

▶ Changelog

`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.

▶ Changelog

`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.

▶ Changelog

`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.

▶ Changelog

`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** (*str* | *None*)
- **line_comment_prefix** (*str* | *None*)
- **trim_blocks** (*bool*)
- **lstrip_blocks** (*bool*)
- **newline_sequence** (*te.Literal['\n', '\r\n', '\r']*)
- **keep_trailing_newline** (*bool*)
- **extensions** (*Sequence[str | Type[Extension]]*)
- **optimized** (*bool*)
- **undefined** (*Type[Undefined]*)
- **finalize** (*Callable[[...], Any] | None*)
- **autoescape** (*bool* | *Callable[[str | None], bool]*)
- **loader** (*BaseLoader* | *None*)
- **cache_size** (*int*)
- **auto_reload** (*bool*)
- **bytecode_cache** (*BytecodeCache* | *None*)
- **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** (*str* | *None*)
- **line_comment_prefix** (*str* | *None*)
- **trim_blocks** (*bool*)
- **lstrip_blocks** (*bool*)
- **newline_sequence** (*te.Literal['\n', '\r\n', '\r']*)
- **keep_trailing_newline** (*bool*)
- **extensions** (*Sequence[str | Type[Extension]]*)
- **optimized** (*bool*)
- **undefined** (*Type[Undefined]*)
- **finalize** (*Callable[[...], Any]* | *None*)
- **autoescape** (*bool* | *Callable[[str | None], bool]*)

- **loader** (*BaseLoader* | *None*)
- **cache_size** (*int*)
- **auto_reload** (*bool*)
- **bytecode_cache** (*BytecodeCache* | *None*)
- **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.

▶ Changelog

Parameters:

extension (*str* | *Type*[*Extension*])

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

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
```

▶ Changelog

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.

▶ Changelog

Parameters:

- **target** (*str* | *PathLike*[*str*])
- **extensions** (*Collection*[*str*] | *None*)
- **filter_func** (*Callable*[[*str*], *bool*] | *None*)
- **zip** (*str* | *None*)
- **log_function** (*Callable*[[*str*], *None*] | *None*)

- **ignore_errors** (*bool*)

Return type:

None

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.

▶ Changelog

Parameters:

- **extensions** (*Collection[str] | None*)
- **filter_func** (*Callable[[str], bool] | None*)

Return type:

List[str]

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

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** (*str* | *Template*) – Name of the template to load. When loading templates from the filesystem, "/" is used as the path separator, even on Windows.
- **parent** (*str* | *None*) – The name of the parent template importing this template. `join_path()` can be used to implement name transformations with this.
- **globals** (*MutableMapping[str, Any] | None*) – 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

▶ Changelog

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*[*str* | *Template*]) – List of template names to try loading in order.
- **parent** (*str* | *None*) – The name of the parent template importing this template. `join_path()` can be used to implement name transformations with this.
- **globals** (*MutableMapping*[*str*, *Any*] | *None*) – 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

▶ Changelog

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.

▶ Changelog

Parameters:

- **template_name_or_list** (*str* | *Template* | *List*[*str* | *Template*])
- **parent** (*str* | *None*)
- **globals** (*MutableMapping*[*str*, *Any*] | *None*)

Return type:

Template

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

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

Parameters:

- **source** (*str* | *Template*) – Jinja source to compile into a template.
- **globals** (*MutableMapping*[*str*, *Any*] | *None*) – 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** (*Type*[*Template*] | *None*) – Return an instance of this `Template` class.

Return type:

Template

class jinja2.Template(*source*, *block_start_string=BLOCK_START_STRING*, *block_end_string=BLOCK_END_STRING*, *variable_start_string=VARIABLE_START_STRING*, *variable_end_string=VARIABLE_END_STRING*, *comment_start_string=COMMENT_START_STRING*, *comment_end_string=COMMENT_END_STRING*,

line_statement_prefix=LINE_STATEMENT_PREFIX, line_comment_prefix=LINE_COMMENT_PREFIX, trim_blocks=TRIM_BLOCKS, lstrip_blocks=LSTRIP_BLOCKS, newline_sequence=NEWLINE_SEQUENCE, keep_trailing_newline=KEEP_TRAILING_NEWLINE, extensions=(), optimized=True, undefined=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** (*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** (*str* | *None*)
- **line_comment_prefix** (*str* | *None*)
- **trim_blocks** (*bool*)
- **lstrip_blocks** (*bool*)
- **newline_sequence** (*te.Literal['\n', '\r\n', '\r']*)
- **keep_trailing_newline** (*bool*)
- **extensions** (*Sequence[str | Type[Extension]]*)
- **optimized** (*bool*)
- **undefined** (*Type[Undefined]*)
- **finalize** (*Callable[[...], Any]* | *None*)
- **autoescape** (*bool* | *Callable[[str | None], 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:

AsyncGenerator[str, object]

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** (*Dict*[*str*, *Any*] | *None*)
- **shared** (*bool*)
- **locals** (*Mapping*[*str*, *Any*] | *None*)

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]*)

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** (*str* | *IO[bytes]*)
- **encoding** (*str* | *None*)
- **errors** (*str* | *None*)

Return type:

None

disable_buffering()¶

Disable the output buffering.

Return type:

None

enable_buffering(*size=5*)¶

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

Parameters:

size (*int*)

Return type:

None

Autoescaping¶

▶ Changelog

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.

▶ Changelog

Parameters:

- **enabled_extensions** (*Collection*[*str*])
- **disabled_extensions** (*Collection*[*str*])
- **default_for_string** (*bool*)
- **default** (*bool*)

Return type:

Callable[[str | None], 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:

```
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:

```
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).

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_]*)*`.

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** (*str* | *None*)
- **obj** (*Any*)
- **name** (*str* | *None*)
- **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
```

▶ Changelog

Parameters:

- **hint** (*str* | *None*)
- **obj** (*Any*)
- **name** (*str* | *None*)
- **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** (*str* | *None*)
- **obj** (*Any*)
- **name** (*str* | *None*)
- **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** (*str* | *None*)
- **obj** (*Any*)
- **name** (*str* | *None*)
- **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=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
)
```

▶ Changelog

Parameters:

- **logger** (*logging.Logger* | *None*) – 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.

```
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:

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

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** (*str* | *None*)
- **blocks** (*Dict[str, Callable[[Context], Iterator[str]]]*)
- **globals** (*MutableMapping[str, Any]* | *None*)

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[[...], Any]*)
- **args** (*Any*)
- **kwargs** (*Any*)

Return type:

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** (*Any*) – The value to return if the key is not found.

Return type:

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:

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

get_exported()¶

Get a new dict with the exported variables.

Return type:

Dict[str, *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*]

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() %}
```

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:

```
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 `TemplateNotFound` 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, str | None, *Callable*[[], bool] | None]

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** (*MutableMapping*[*str*, *Any*] | *None*)

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** (*str* | *os.PathLike*[*str*] | *Sequence*[*str* | *os.PathLike*[*str*]]) – 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.

▶ Changelog

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.

▶ Changelog

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], str | Tuple[str, str | None, Callable[[], bool] | None] | None]*)

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 (*str* | *os.PathLike[str]* | *Sequence[str | os.PathLike[str]]*)

Bytecode Cache¶

Jinja 2.1 and higher support external bytecode caching. Bytecode caches make it possible to store the generated bytecode 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.

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

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

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

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`.

reset()¶

Resets the bucket (unloads the bytecode).

Return type:

None

load_bytecode(*f*)¶

Loads bytecode from a file or file like object.

Parameters:

f (*BinaryIO*)

Return type:

None

write_bytecode(*f*)¶

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

Parameters:

f (*IO[bytes]*)

Return type:

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

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** (*str* | *None*)
- **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** (*int* | *None*)
- **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.

▶ Changelog

Async Support¶

▶ Changelog

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:

- 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.

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.

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()`.

▶ Changelog

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()`.

▶ Changelog

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.

▶ Changelog

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:

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

Parameters:

obj (*Any*)

Return type:

bool

Exceptions¶

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

Baseclass for all template errors.

Parameters:

message (*str* | *None*)

Return type:

None

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

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

Parameters:

message (*str* | *None*)

Return type:

None

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

Raised if a template does not exist.

▶ Changelog

Parameters:

- **name** (*str* | *Undefined* | *None*)
- **message** (*str* | *None*)

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.

▶ Changelog

Parameters:

- **names** (*Sequence[str | Undefined]*)
- **message** (*str | None*)

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** (*str | None*)
- **filename** (*str | None*)

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 (*str | None*)

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** (*str* | *None*)
- **filename** (*str* | *None*)

Return type:

None

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:

```
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.

```
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
```

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:

```
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 test. The object is passed as the first argument, making the value being tested the second argument.

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

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** (*str* | *None*)

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`.

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.

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** (*str* | *None*)
- **filename** (*str* | *None*)

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** (*str* | *None*)
- **filename** (*str* | *None*)

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** (*str* | *None*)
- **filename** (*str* | *None*)

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** (*Dict[str, Any] | None*)
- **shared** (*bool*)
- **locals** (*Mapping[str, Any] | None*)

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()`.

The Meta API¶

▶ Changelog

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[str | None]

Contents

Navigation

- Overview
    - Previous: Introduction
    - Next: Sandbox

Quick search

jinja.palletsprojects.com

Verifying you are human. This may take a few seconds.

jinja.palletsprojects.com needs to review the security of your connection before proceeding.

jinja.palletsprojects.com

Verifying you are human. This may take a few seconds.

jinja.palletsprojects.com needs to review the security of your connection before proceeding.

jinja.palletsprojects.com

Verifying you are human. This may take a few seconds.

jinja.palletsprojects.com needs to review the security of your connection before proceeding.

jinja.palletsprojects.com

Verifying you are human. This may take a few seconds.

jinja.palletsprojects.com needs to review the security of your connection before proceeding.

jinja.palletsprojects.com

Verifying you are human. This may take a few seconds.

jinja.palletsprojects.com needs to review the security of your connection before proceeding.

jinja.palletsprojects.com

Verifying you are human. This may take a few seconds.

jinja.palletsprojects.com needs to review the security of your connection before proceeding.

jinja.palletsprojects.com

Verifying you are human. This may take a few seconds.

jinja.palletsprojects.com needs to review the security of your connection before proceeding.

jinja.palletsprojects.com

Verifying you are human. This may take a few seconds.

jinja.palletsprojects.com needs to review the security of your connection before proceeding.

jinja.palletsprojects.com

Verifying you are human. This may take a few seconds.

jinja.palletsprojects.com needs to review the security of your connection before proceeding.

jinja.palletsprojects.com

Verifying you are human. This may take a few seconds.

jinja.palletsprojects.com needs to review the security of your connection before proceeding.