Jinja2 Documentation

Release 2.0

Armin Ronacher

October 17, 2010

CONTENTS

CHAPTER

ONE

INTRODUCTION

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

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

1.1 Prerequisites

Jinja2 needs at least **Python 2.4** to run. Additionally a working C-compiler that can create python extensions should be installed for the debugger if you are using Python 2.4.

If you don't have a working C-compiler and you are trying to install the source release with the debug-support you will get a compiler error.

1.2 Installation

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

1.2.1 As a Python egg (via easy_install)

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

```
easy_install Jinja2
pip install Jinja2
```

This will install a Jinja2 egg in your Python installation's site-packages directory.

(If you are installing from the windows command line omit the *sudo* and make sure to run the command as user with administrator rights)

1.2.2 From the tarball release

1. Download the most recent tarball from the download page
2. Unpack the tarball
3. `sudo python setup.py install`

Note that you either have to have setuptools or distribute installed, the latter is preferred.

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

1.2.3 Installing the development version

1. Install git
2. `git clone git://github.com/mitsuhiko/jinja2.git`
3. `cd jinja2`
4. `ln -s jinja2 /usr/lib/python2.X/site-packages`

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

1.2.4 More Speed with MarkupSafe

As of version 2.5.1 Jinja2 will check for an installed MarkupSafe module. If it can find it, it will use the Markup class of that module instead of the one that comes with Jinja2. *MarkupSafe* replaces the older speedups module that came with Jinja2 and has the advantage that is has a better setup script and will automatically attempt to install the C version and nicely fall back to a pure Python implementation if that is not possible.

The C implementation of MarkupSafe is much faster and recommended when using Jinja2 with autoescaping.

1.2.5 Enable the debug support Module

By default Jinja2 will not compile the debug support module. Enabling this will fail if you don't have the Python headers or a working compiler. This is often the case if you are installing Jinja2 from a windows machine.

Because the debug support is only necessary for Python 2.4 you will not have to do this unless you run 2.4:

```
sudo python setup.py --with-debugsupport install
```

1.3 Basic API Usage

This section gives you a brief introduction to the Python API for Jinja2 templates.

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

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

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

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

1.4 Experimental Python 3 Support

Jinja 2.3 brings experimental support for Python 3. It means that all unittests pass on the new version, but there might still be small bugs in there and behavior might be inconsistent. If you notice any bugs, please provide feedback in the Jinja bug tracker.

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

CHAPTER

TWO

API

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

2.1 Basics

Jinja2 uses a central object called the template `Environment`. Instances of this class are used to store the configuration, global objects and are used to load templates from the file system or other locations. Even if you are creating templates from 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 it's however useful to have multiple environments side by side, if different configurations are in use.

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

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

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

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

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

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

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

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

2.2 Unicode

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

Python 2.x supports two ways of representing string objects. One is the *str* type and the other is the *unicode* type, both of which extend a type called *basestring*. Unfortunately the default is *str* which should not be used to store text based information unless only ASCII characters are used. With Python 2.6 it is possible to make *unicode* the default on a per module level and with Python 3 it will be the default.

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

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

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

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

It is not possible to use Jinja2 to process non-Unicode data. The reason for this is that Jinja2 uses Unicode already on the language level. For example Jinja2 treats the non-breaking space as valid whitespace inside expressions which requires knowledge of the encoding or operating on an Unicode string.

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

Another important thing is how Jinja2 is handling string literals in templates. A naive implementation would be using Unicode strings for all string literals but it turned out in the past that this is problematic as some libraries are typechecking against *str* explicitly. For example *datetime.strftime* does not accept Unicode arguments. To not break it completely Jinja2 is returning *str* for strings that fit into ASCII and for everything else *unicode*:

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

2.3 High Level API

The high-level API is the API you will use in the application to load and render Jinja2 templates. The *Low Level API* on the other side is only useful if you want to dig deeper into Jinja2 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 the possible initialization parameters:

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

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

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

variable_end_string The string marking the end of a print statement. Defaults to `'}}'`.

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

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

line_statement_prefix If given and a string, this will be used as prefix for line based statements. See also *Line Statements*.

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

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.

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

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

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.

`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 global variables. These variables are always available in a template. As long as no

template was loaded it's safe to modify this dict. For more details see *The Global Namespace*. For valid object names have a look at *Notes on Identifiers*.

overlay([*options*])
Create a new overlay environment that shares all the data with the current environment except of 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.

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 ommited.

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 shold be passed to the undefined object, even if a custom *hint* is provided. This gives undefined objects the possibility to enhance the error message.

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

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

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

If the template does not exist a `TemplateNotFound` exception is raised. Changed in version 2.4: If *name* is a `Template` object it is returned from the function unchanged.

select_template(*names*, *parent=None*, *globals=None*)
Works like `get_template()` but tries a number of templates before it fails. If it cannot find any of the templates, it will raise a `TemplatesNotFound` exception. New in version 2.3.Changed in version 2.4: If *names* contains a `Template` object it is returned from the function unchanged.

get_or_select_template(*template_name_or_list*, *parent=None*, *globals=None*)
Does a typecheck and dispatches to `select_template()` if an iterable of template names is given, otherwise to `get_template()`. New in version 2.3.

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.

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.

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.

class jinja2.**Template**
The central template object. This class represents a compiled template and is used to evaluate it.

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

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

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

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

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

globals
The dict with the globals of that template. It's unsafe to modify this dict as it may be shared

with other templates or the environment that loaded the template.

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 unicode string.

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 unicode strings.

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

stream([*context*])
Works exactly like `generate()` but returns a `TemplateStream`.

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

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

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.

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 unicode string is yielded.

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

disable_buffering()
Disable the output buffering.

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

dump(*fp*, *encoding=None*, *errors='strict'*)
Dump the complete stream into a file or file-like object. Per default unicode strings are written, if you want to encode before writing specifiy an *encoding*.

Example usage:

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

2.4 Autoescaping

New in version 2.4. As of Jinja 2.4 the preferred way to do autoescaping is to enable the *Autoescape Extension* and 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).

Here a recommended setup that enables autoescaping for templates ending in `'.html'`, `'.htm'` and `'.xml'` and disabling it by default for all other extensions:

```
def guess_autoescape(template_name):
    if template_name is None or '.' not in template_name:
        return False
    ext = template_name.rsplit('.', 1)[1]
    return ext in ('html', 'htm', 'xml')

env = Environment(autoescape=guess_autoescape,
                  loader=PackageLoader('mypackage'),
                  extensions=['jinja2.ext.autoescape'])
```

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.

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

2.5 Notes on Identifiers

Jinja2 uses the regular Python 2.x naming rules. Valid identifiers have to match `[a-zA-Z_][a-zA-Z0-9_]*`. As a matter of fact non ASCII characters are currently not allowed. This limitation will probably go away as soon as unicode identifiers are fully specified for Python 3.

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.unicode*. The regular expression for filter and test identifiers is `[a-zA-Z_][a-zA-Z0-9_]*(\.[a-zA-Z_][a-zA-Z0-9_]*)*`.

2.6 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):
  ...
UndefinedError: 'foo' is undefined
```

_undefined_hint
Either *None* or an unicode 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.`**`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):
  ...
UndefinedError: 'foo' is undefined
```

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):
  ...
UndefinedError: 'foo' is undefined
>>> not foo
Traceback (most recent call last):
  ...
UndefinedError: 'foo' is undefined
>>> foo + 42
Traceback (most recent call last):
  ...
UndefinedError: 'foo' is undefined
```

Undefined objects are created by calling `undefined`.

Implementation

`Undefined` objects are implemented by overriding the special *__underscore__* methods. For example the default `Undefined` class implements *__unicode__* in a way that it returns an empty string, however *__int__* and others still fail with an exception. To allow conversion to int by returning `0` you can implement your own:

```
class NullUndefined(Undefined):
    def __int__(self):
        return 0
    def __float__(self):
        return 0.0
```

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

```
class NonIterableUndefined(Undefined):
    __iter__ = Undefined._fail_with_undefined_error
```

2.7 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 `contextfunction()`s 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.

`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 is a `contextfunction()` or `environmentfunction()`.

resolve(*key*)
Looks up a variable like *__getitem__* or *get* but returns an `Undefined` object with the name of the name looked up.

get_exported()
Get a new dict with the exported variables.

get_all()
Return a copy of the complete context as dict including the exported variables.

Implementation

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

When a template accesses a variable the template does not define, Jinja2 looks up the variable in the context, after that the variable is treated as if it was defined in the template.

2.8 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 file(path) as f:
            source = f.read().decode('utf-8')
        return source, path, lambda: mtime == getmtime(path)
```

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

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

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

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

Here a list of the builtin loaders Jinja2 provides:

class `jinja2.`**FileSystemLoader** (*searchpath*, *encoding='utf-8'*)
Loads templates from the file system. This loader can find templates in folders on the file system and is the preferred way to load them.

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

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

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

class `jinja2.`**PackageLoader** (*package_name*, *package_path='templates'*, *encoding='utf-8'*)
Load templates from python eggs or packages. It is constructed with the name of the python package and the path to the templates in that package:

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

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

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

class `jinja2.`**DictLoader** (*mapping*)
Loads a template from a python dict. It's passed a dict of unicode strings bound to template names. This loader is useful for unittesting:

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

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

class `jinja2.`**FunctionLoader** (*load_func*)
A loader that is passed a function which does the loading. The function becomes the name of the template passed and has to return either an unicode string with the template source, a tuple in the form `(source, filename, uptodatefunc)` or *None* if the template does not exist.

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

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

class `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.

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.

2.9 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, instanciate 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 Jinja2.

`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.

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.

clear ()
: Clears the cache. This method is not used by Jinja2 but should be implemented to allow applications to clear the bytecode cache used by a particular environment.

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.

environment
: The `Environment` that created the bucket.

key
: The unique cache key for this bucket

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

write_bytecode (*f*)
: Dump the bytecode into the file or file like object passed.

load_bytecode (*f*)
: Loads bytecode from a file or file like object.

bytecode_from_string (*string*)
: Load bytecode from a string.

bytecode_to_string ()
: Return the bytecode as string.

reset ()
: Resets the bucket (unloads the bytecode).

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 the system temporary items folder is used.

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.

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

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

(Unfortunately the django cache interface is not compatible because it does not support storing binary data, only unicode. 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:

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.

2.10 Utilities

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

`jinja2.`**environmentfilter** (*f*)
Decorator for marking evironment dependent filters. The current `Environment` is passed to the filter as first argument.

`jinja2.`**contextfilter** (*f*)
Decorator for marking context dependent filters. The current `Context` will be passed as first argument.

`jinja2.`**evalcontextfilter** (*f*)
Decorator for marking eval-context dependent filters. An eval context object is passed as first argument. For more information about the eval context, see *Evaluation Context*. New in version 2.4.

`jinja2.`**environmentfunction** (*f*)
This decorator can be used to mark a function or method as environment callable. This decorator works exactly like the `contextfunction()` decorator just that the first argument is the active `Environment` and not context.

`jinja2.`**contextfunction** (*f*)
This decorator can be used to mark a function or method context callable. A context callable is passed the active `Context` as first argument when called from the template. This is useful if a function wants to get access to the context or functions provided on the context object. For example a function that returns a sorted list of template variables the current template exports could look like this:

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

`jinja2.`**evalcontextfunction** (*f*)
This decoraotr can be used to mark a function or method as an eval context callable. This is similar to the `contextfunction()` but instead of passing the context, an evaluation context object is passed. For more information about the eval context, see *Evaluation Context*. New in version 2.4.

`jinja2.`**`escape`**`(`*s*`)`

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

The return value is a `Markup` string.

`jinja2.`**`clear_caches`**`()`

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

`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
```

class `jinja2.`**`Markup`**`(`[*string*]`)`

Marks a string as being safe for inclusion in HTML/XML output without needing to be escaped. This implements the *__html__* interface a couple of frameworks and web applications use. `Markup` is a direct subclass of *unicode* and provides all the methods of *unicode* just that it escapes arguments passed and always returns *Markup*.

The *escape* function returns markup objects so that double escaping can't happen.

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

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

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

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

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

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

classmethod **`escape`**`(`*s*`)`

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

unescape()

Unescape markup again into an unicode string. This also resolves known HTML4 and XHTML entities:

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

striptags()

Unescape markup into an unicode string and strip all tags. This also resolves known HTML4 and XHTML entities. Whitespace is normalized to one:

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

Note

The Jinja2 `Markup` class is compatible with at least Pylons and Genshi. It's expected that more template engines and framework will pick up the *__html__* concept soon.

2.11 Exceptions

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

Baseclass for all template errors.

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

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

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

Raised if a template does not exist.

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

Like `TemplateNotFound` but raised if multiple templates are selected. This is a subclass of `TemplateNotFound` exception, so just catching the base exception will catch both. New in version 2.2.

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

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

message

The error message as utf-8 bytestring.

lineno

The line number where the error occurred

name

The load name for the template as unicode string.

filename

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

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

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.

2.12 Custom Filters

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

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

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

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

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

Inside the template it can then be used as follows:

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

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

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

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

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

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

Context filters work the same just that the first argument is the current active `Context` rather then the environment.

2.13 Evaluation Context

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

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

In previous Jinja versions filters and functions were marked as environment callables in order to check for the autoescape status from the environment. In new versions it's encouraged to check the setting from the evaluation context instead.

Previous versions:

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

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

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

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

The evaluation context must not be modified at runtime. 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.

`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*.

2.14 Custom Tests

Tests work like filters just that there is no way for a test to get access to the environment or context and that they can't be chained. The return value of a test should be *True* or *False*. The purpose of a test is to give the template designers the possibility to perform type and conformability checks.

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

```
import math

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

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

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

A template designer can then use the test like this:

```
{% if 42 is prime %}
    42 is a prime number
{% else %}
```

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

2.15 The Global Namespace

Variables stored in the `Environment.globals` dict are special as they are available for imported templates too, even if they are imported without context. This is the place where you can put variables and functions that should be available all the time. Additionally `Template.globals` exist that are variables available to a specific template that are available to all `render()` calls.

2.16 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.

`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 Jinja2 extensions* this gives you a good overview of the node tree generated.

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

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

`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 it to the context without adding the globals.

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

`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 unicode 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 Jinja2 versions will try not to change it in a backwards incompatible way but modifications in the Jinja2 core may shine through. For example if Jinja2 introduces a new AST node in later versions that may be returned by `parse()`.

2.17 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)
set(['bar'])
```

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.

`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.

CHAPTER

THREE

SANDBOX

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

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

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

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 occour during the rendering so the caller has to ensure that all exceptions are catched.

`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.

`is_safe_callable``(`*obj*`)`

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

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

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

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

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

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

Mark a function or method as unsafe:

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

`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 overriden.

```
>>> from jinja2.sandbox import is_internal_attribute
>>> is_internal_attribute(lambda: None, "func_code")
True
>>> is_internal_attribute((lambda x:x).func_code, 'co_code')
True
>>> is_internal_attribute(str, "upper")
False
```

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

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

```
>>> 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 (such as unicode) *False* is returned.

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

Note

The Jinja2 sandbox alone is no solution for perfect security. Especially for web applications you have to keep in mind that users may create templates with arbitrary HTML in so it's crucial to ensure that (if you are running multiple users on the same server) they can't harm each other via JavaScript insertions and much more.

Also the sandbox is only as good as the configuration. We stronly recommend only passing non-shared resources to the template and use some sort of whitelisting for attributes.

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

CHAPTER

FOUR

TEMPLATE DESIGNER DOCUMENTATION

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

4.1 Synopsis

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

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

Below is a minimal template that illustrates a few basics. We will cover the details later in that document:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html lang="en">
<head>
    <title>My Webpage</title>
</head>
<body>
    <ul id="navigation">
    {% for item in navigation %}
        <li><a href="{{ item.href }}">{{ item.caption }}</a></li>
    {% endfor %}
    </ul>

    <h1>My Webpage</h1>
    {{ a_variable }}
</body>
</html>
```

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

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

4.2 Variables

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

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

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

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

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

Implementation

For convenience sake `foo.bar` in Jinja2 does the following things on the Python layer:

- check if there is an attribute called *bar* on *foo*.
- if there is not, check if there is an item `'bar'` in *foo*.
- if there is not, return an undefined object.

`foo['bar']` on the other hand works mostly the same with the a small difference in the order:

- check if there is an item `'bar'` in *foo*.
- if there is not, check if there is an attribute called *bar* on *foo*.
- if there is not, return an undefined object.

This is important if an object has an item or attribute with the same name. Additionally there is the `attr()` filter that just looks up attributes.

4.3 Filters

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

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

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

4.4 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 if *name* is defined.

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

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

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

4.5 Comments

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

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

4.6 Whitespace Control

In the default configuration whitespace is not further modified by the template engine, so each whitespace (spaces, tabs, newlines etc.) is returned unchanged. If the application configures Jinja to *trim_blocks* the first newline after a a template tag is removed automatically (like in PHP).

But you can also strip whitespace in templates by hand. If you put an minus sign (-) to the start or end of an block (for example a for tag), a comment or variable expression you can remove the whitespaces after or before that block:

```
{% 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.

Note

You must not use a whitespace between the tag and the minus sign.

valid:

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

invalid:

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

4.7 Escaping

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

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

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

For bigger sections it makes sense to mark a block *raw*. For example to put Jinja syntax as example into a template you can use this snippet:

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

4.8 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
```

4.9 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.

4.9.1 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 PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html lang="en">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    {% block head %}
    <link rel="stylesheet" href="style.css" />
    <title>{% block title %}{% endblock %} - My Webpage</title>
    {% endblock %}
</head>
<body>
    <div id="content">{% block content %}{% endblock %}</div>
    <div id="footer">
        {% block footer %}
        &copy; Copyright 2008 by <a href="http://domain.invalid/">you</a>.
        {% endblock %}
    </div>
</body>
```

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

4.9.2 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 on 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, first it locates the parent. The extends tag should be the first tag in the template. Everything before it is printed out normally and may cause confusion. For details about this behavior and how to take advantage of it, see *Null-Master Fallback*.

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

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

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

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

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

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

4.9.3 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 %}
```

4.9.4 Named Block End-Tags

Jinja2 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.

4.9.5 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.

4.9.6 Template Objects

Changed in version 2.4. If a template object was passed to the template context you can extend from that object as well. Assuming the calling code passes a layout template as *layout_template* to the environment, this code works:

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

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

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

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

- escaping everything except of safe values will also mean that Jinja is escaping variables known to not include HTML such as numbers which is a huge performance hit.
- The information about the safety of a variable is very fragile. It could happen that by coercing safe and unsafe values the return value is double escaped HTML.

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

4.10.2 Working with Automatic Escaping

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

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

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

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

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

4.11.1 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>
```

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

Variable	Description
loop.index	The current iteration of the loop. (1 indexed)
loop.index0	The current iteration of the loop. (0 indexed)
loop.revindex	The number of iterations from the end of the loop (1 indexed)
loop.revindex0	The number of iterations from the end of the loop (0 indexed)
loop.first	True if first iteration.
loop.last	True if last iteration.
loop.length	The number of items in the sequence.
loop.cycle	A helper function to cycle between a list of sequences. See the explanation below.

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

With 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 replacement block by using *else*:

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

It is also possible to use loops recursively. This is useful if you are dealing with recursive data such as sitemaps. 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>
```

4.11.2 If

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

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

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

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

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

4.11.3 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.

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

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

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

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

If the macro was 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 list of values.

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

caller If the macro was called from a *call* tag the caller is stored in this variable as macro which can be called.

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

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

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

defaults A tuple of default values.

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

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

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

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

4.11.4 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 replacement for loops. Generally speaking a call block works exactly like an macro, just that it doesn't have a name.

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

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

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

4.11.5 Filters

Filter sections allow you to apply regular Jinja2 filters on a block of template data. Just wrap the code in the special *filter* section:

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

4.11.6 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() %}
```

4.11.7 Extends

The *extends* tag can be used to extend a template from another one. You can have multiple of them in a file but only one of them may be executed at the time. See the section about *Template Inheritance* above.

4.11.8 Block

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

4.11.9 Include

The *include* statement is useful to include a template and return the rendered contents of that file into the current namespace:

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

Included templates have access to the variables of the active context by default. For more details about context behavior of imports and includes see *Import Context Behavior*.

From Jinja 2.2 onwards you can mark an include with `ignore missing` in which case Jinja will ignore the statement if the template to be ignored does not exist. When combined with `with` or `without context` it has to be placed *before* the context visibility statement. Here some valid examples:

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

New in version 2.2. You can also provide a list of templates that are checked for existence before inclusion. The first template that exists will be included. If *ignore missing* is given, it will fall back to rendering nothing if none of the templates exist, otherwise it will raise an exception.

Example:

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

Changed in version 2.4: If a template object was passed to the template context you can include that object using *include*.

4.11.10 Import

Jinja2 supports putting often used code into macros. These macros can go into different templates and get imported from there. This works similar to the import statements in Python. It's important to know that imports are cached and imported templates don't have access to the current template variables, just the globals by defualt. For more details about context behavior of imports and includes see *Import Context Behavior*.

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

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

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

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

The easiest and most flexible is importing the whole module into a variable. That way you can access the attributes:

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

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

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

Macros and variables starting with one ore 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.

4.12 Import Context Behavior

Per default included templates are passed the current context and imported templates 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 however can be changed of course 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 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, but in Jinja 2.1.

4.13 Expressions

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

4.13.1 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 (for example as arguments to function calls, filters or just to extend or include a template).

42 / 42.23: Integers and floating point numbers are created by just writing the number down. If a dot is present the number is a float, otherwise an integer. Keep in mind that for Python `42` and `42.0` is something different.

['list', 'of', 'objects']: Everything between two brackets is a list. Lists are useful to store sequential data in or to iterate over them. For example you can easily create a list of links using lists and tuples with a for loop:

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

('tuple', 'of', 'values'): Tuples are like lists, just that you can't modify them. If the tuple only has one item you have to end it with a comma. Tuples are usually used to represent items of two or more elements. See the example above for more details.

{'dict': 'of', '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, when writing *True* expands to an undefined variable that is considered false, all three of them can be written in title case too (*True*, *False*, and *None*). However for consistency (all Jinja identifiers are lowercase) you should use the lowercase versions.

4.13.2 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 at the ~ operator. `{{ 1 + 1 }}` is `2`.

- Substract 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`.

4.13.3 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.

4.13.4 Logic

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

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

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

not negate a statement (see below).

(expr) group an expression.

Note

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)`.

4.13.5 Other Operators

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

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

is Performs a *test*.

| Applies a *filter*.

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

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

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

4.13.6 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 'master.html' %}
```

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

The *else* part is optional. If not provided the else block implicitly evaluates into an undefined object:

```
{{ '[%s]' % page.title if page.title }}
```

4.14 List of Builtin Filters

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

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

See *Notes on subscriptions* for more details.

batch (*value*, *linecount*, *fill_with=None*)
A filter that batches items. It works pretty much like *slice* just the other way round. It returns a list of lists with the given number of items. If you provide a second parameter this is used to fill missing items. See this example:

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

capitalize (*s*)
Capitalize a value. The first character will be uppercase, all others lowercase.

center (*value*, *width=80*)
Centers the value in a field of a given width.

default (*value*, *default_value=u''*, *boolean=False*)
If the value is undefined it will return the passed default value, otherwise the value of the variable:

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

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

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

Aliases d

dictsort (*value, case_sensitive=False, by='key'*)
Sort a dict and yield (key, value) pairs. Because python dicts are unsorted you may want to use this function to order them by either key or value:

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

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

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

escape (*s*)
Convert the characters &, <, >, ' and " in string s to HTML-safe sequences. Use this if you need to display text that might contain such characters in HTML. Marks return value as markup string.

Aliases e

filesizeformat (*value, binary=False*)
Format the value like a 'human-readable' file size (i.e. 13 KB, 4.1 MB, 102 bytes, etc). Per default decimal prefixes are used (mega, giga, etc.), if the second parameter is set to *True* the binary prefixes are used (mebi, gibi).

first (*seq*)
Return the first item of a sequence.

float (*value, default=0.0*)
Convert the value into a floating point number. If the conversion doesn't work it will return `0.0`. You can override this default using the first parameter.

forceescape (*value*)
Enforce HTML escaping. This will probably double escape variables.

format (*value, *args, **kwargs*)
Apply python string formatting on an object:

```
{{ "%s - %s"|format("Hello?", "Foo!") }}
    -> Hello? - Foo!
```

groupby (*value, attribute*)
Group a sequence of objects by a common attribute.

If you for example have a list of dicts or objects that represent persons with *gender*, *first_name* and *last_name* attributes and you want to group all users by genders you can do something like the following snippet:

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

Additionally it's possible to use tuple unpacking for the grouper and list:

```
<ul>
{% for grouper, list in persons|groupby('gender') %}
    ...
{% endfor %}
</ul>
```

As you can see the item we're grouping by is stored in the *grouper* attribute and the *list* contains all the objects that have this grouper in common.

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

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

int (*value, default=0*)
Convert the value into an integer. If the conversion doesn't work it will return `0`. You can override this default using the first parameter.

join (*eval_ctx, value, d=u''*)
Return a string which is the concatenation of the strings in the sequence. The separator between elements is an empty string per default, you can define it with the optional parameter:

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

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

last (*seq*)
Return the last item of a sequence.

length (*object*)
Return the number of items of a sequence or mapping.

Aliases `count`

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

lower (*s*)
Convert a value to lowercase.

pprint (*value, verbose=False*)
Pretty print a variable. Useful for debugging.

With Jinja 1.2 onwards you can pass it a parameter. If this parameter is truthy the output will be more verbose (this requires *pretty*)

random (*seq*)
Return a random item from the sequence.

replace (*eval_ctx, s, old, new, count=None*)
Return a copy of the value with all occurrences of a substring replaced with a new one. The first argument is the substring that should be replaced, the second is the replacement string. If the optional third argument `count` is given, only the first `count` occurrences are replaced:

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

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

reverse (*value*)
Reverse the object or return an iterator the iterates over it the other way round.

round (*value, precision=0, method='common'*)
Round the number to a given precision. The first parameter specifies the precision (default is `0`), the second the rounding method:

- `'common'` rounds either up or down
- `'ceil'` always rounds up
- `'floor'` always rounds down

If you don't specify a method `'common'` is used.

```
{{ 42.55|round }}
    -> 43.0
{{ 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
```

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

slice (*value, slices, fill_with=None*)
Slice an iterator and return a list of lists containing those items. Useful if you want to create a div containing three ul tags that represent columns:

```
<div class="columwrapper">
  {%- for column in items|slice(3) %}
    <ul class="column-{{ loop.index }}">
    {%- for item in column %}
      <li>{{ item }}</li>
    {%- endfor %}
    </ul>
  {%- endfor %}
</div>
```

If you pass it a second argument it's used to fill missing values on the last iteration.

sort (*value, reverse=False, case_sensitive=False*)
Sort an iterable. Per default it sorts ascending, if you pass it true as first argument it will reverse the sorting.

If the iterable is made of strings the third parameter can be used to control the case sensitiveness of the comparison which is disabled by default.

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

string (*s*)
Make a string unicode if it isn't already. That way a markup string is not converted back to unicode.

striptags (*value*)
Strip SGML/XML tags and replace adjacent whitespace by one space.

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

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

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

truncate (*s, length=255, killwords=False, end='...'*)
Return a truncated copy of the string. The length is specified with the first parameter which defaults to `255`. If the second parameter is `true` the filter will cut the text at length. Otherwise it will try to save the last word. If the text was in fact truncated it will append an ellipsis sign (`"..."`). If you want a different ellipsis sign than `"..."` you can specify it using the third parameter.

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

urlize (*eval_ctx, value, trim_url_limit=None, nofollow=False*)
Converts URLs in plain text into clickable links.

If you pass the filter an additional integer it will shorten the urls to that number. Also a third argument exists that makes the urls "nofollow":

```
{{ mytext|urlize(40, true) }}
    links are shortened to 40 chars and defined with rel="nofollow"
```

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

wordwrap (*s, width=79, break_long_words=True*)
Return a copy of the string passed to the filter wrapped after `79` characters. You can override this default using the first parameter. If you set the second parameter to *false* Jinja will not split words apart if they are longer than *width*.

xmlattr (*_eval_ctx, d, autospace=True*)
Create an SGML/XML attribute string based on the items in a dict. All values that are neither *none* nor *undefined* are automatically escaped:

```
<ul{{ {'class': 'my_list', 'missing': none,
        'id': 'list-%d'|format(variable)}|xmlattr }}>
...
</ul>
```

Results in something like this:

```
<ul class="my_list" id="list-42">
...
</ul>
```

As you can see it automatically prepends a space in front of the item if the filter returned something unless the second parameter is false.

4.15 List of Builtin Tests

callable (*object*)
Return whether the object is callable (i.e., some kind of function). Note that classes are callable, as are instances with a __call__() method.

defined (*value*)
Return true if the variable is defined:

```
{% if variable is defined %}
    value of variable: {{ variable }}
{% else %}
```

```
    variable is not defined
{% endif %}
```

See the `default()` filter for a simple way to set undefined variables.

`divisibleby`(*value*, *num*)
Check if a variable is divisible by a number.

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

`even`(*value*)
Return true if the variable is even.

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

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

`none`(*value*)
Return true if the variable is none.

`number`(*value*)
Return true if the variable is a number.

`odd`(*value*)
Return true if the variable is odd.

`sameas`(*value*, *other*)
Check if an object points to the same memory address than another object:

```
{% if foo.attribute is sameas false %}
    the foo attribute really is the `False` singleton
{% endif %}
```

`sequence`(*value*)
Return true if the variable is a sequence. Sequences are variables that are iterable.

`string`(*value*)
Return true if the object is a string.

`undefined`(*value*)
Like `defined()` but the other way round.

`upper`(*value*)
Return true if the variable is uppercased.

4.16 List of Global Functions

The following functions are available in the global scope by default:

`range`([*start*], *stop*[, *step*])
Return a list containing an arithmetic progression of integers. range(i, j) returns [i, i+1, i+2, ..., j-1]; start (!) defaults to 0. When step is given, it specifies the increment (or decrement). For example, range(4) returns [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 for example 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>
```

lipsum(*n=5, html=True, min=20, max=100*)
Generates some lorem ipsum for the template. Per default five paragraphs with HTML are generated each paragraph between 20 and 100 words. If html is disabled regular text is returned. This is useful to generate simple contents for layout testing.

dict(***items*)
A convenient alternative to dict literals. `{'foo': 'bar'}` is the same as `dict(foo='bar')`.

class **cycler**(**items*)
The cycler allows you to cycle among values similar to how *loop.cycle* works. Unlike *loop.cycle* however you can use this cycler outside of loops or over multiple loops.

This is for example very useful if you want to show a list of folders and files, with the folders on top, but both in the same list with alternating row colors.

The following example shows how *cycler* can be used:

```
{% set row_class = cycler('odd', 'even') %}
<ul class="browser">
{% for folder in folders %}
  <li class="folder {{ row_class.next() }}">{{ folder|e }}</li>
{% endfor %}
{% for filename in files %}
  <li class="file {{ row_class.next() }}">{{ filename|e }}</li>
{% endfor %}
</ul>
```

A cycler has the following attributes and methods:

reset()
Resets the cycle to the first item.

next()
Goes one item a head and returns the then current item.

current
Returns the current item.

new in Jinja 2.1

class **joiner**(*sep=', '*)
A tiny helper that can be use to "join" multiple sections. A joiner is passed a string and will return that string every time it's calld, except the first time in which situation 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 Jinja 2.1

4.17 Extensions

The following sections cover the built-in Jinja2 extensions that may be enabled by the application. The application could also provide further extensions not covered by this documentation. In that case there should be a separate document explaining the extensions.

4.17.1 i18n

If the i18n extension is enabled it's possible to mark parts in the template as translatable. To mark a section as translatable you can use *trans*:

```
<p>{% trans %}Hello {{ user }}!{% endtrans %}</p>
```

To translate a template expression — say, using template filters or just accessing an attribute of an object — you need to bind the expression to a name for use within the translation block:

```
<p>{% trans user=user.username %}Hello {{ user }}!{% endtrans %}</p>
```

If you need to bind more than one expression inside a *trans* tag, separate the pieces with a comma (`,`):

```
{% trans book_title=book.title, author=author.name %}
This is {{ book_title }} by {{ author }}
{% endtrans %}
```

Inside trans tags no statements are allowed, only variable tags are.

To pluralize, specify both the singular and plural forms with the *pluralize* tag, which appears between *trans* and *endtrans*:

```
{% trans count=list|length %}
There is {{ count }} {{ name }} object.
{% pluralize %}
There are {{ count }} {{ name }} objects.
{% endtrans %}
```

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

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

It's also possible to translate strings in expressions. For that purpose three functions exist:

_ *gettext*: translate a single string - *ngettext*: translate a pluralizable string - _: alias for *gettext*

For example you can print a translated string easily this way:

```
{{ _('Hello World!') }}
```

To use placeholders you can use the *format* filter:

```
{{ _('Hello %(user)s!')|format(user=user.username) }}
```

For multiple placeholders always use keyword arguments to *format* as other languages may not use the words in the same order. Changed in version 2.5. If newstyle gettext calls are activated (*Newstyle Gettext*), using placeholders is a lot easier:

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

Note that the *ngettext* function's format string automatically recieves the count as *num* parameter additionally to the regular parameters.

4.17.2 Expression Statement

If the expression-statement extension is loaded a tag called *do* is available that works exactly like the regular variable expression (`{{ ... }}`) just that it doesn't print anything. This can be used to modify lists:

```
{% do navigation.append('a string') %}
```

4.17.3 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 eached the processing is stopped and continues with the next iteration.

Here a loop that skips every second item:

```
{% for user in users %}
    {%- if loop.index is even %}{% continue %}{% endif %}
    ...
{% endfor %}
```

Likewise a look that stops processing after the 10th iteration:

```
{% for user in users %}
    {%- if loop.index >= 10 %}{% break %}{% endif %}
{%- endfor %}
```

4.17.4 With Statement

New in version 2.3. If the application enables the *With Statement* it is possible to use the *with* keyword in templates. This 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 %}
```

4.18 Autoescape Extension

New in version 2.4. If the application enables the *Autoescape Extension* one 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 the *endautoescape* the behavior is reverted to what it was before.

CHAPTER

FIVE

EXTENSIONS

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

5.1 Adding Extensions

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

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

5.2 i18n Extension

Import name: *jinja2.ext.i18n*

Jinja2 currently comes with one extension, the i18n extension. It can be used in combination with gettext or babel. If the i18n extension is enabled Jinja2 provides a *trans* statement that marks the wrapped string as translatable and calls *gettext*.

After enabling dummy _ function that forwards calls to *gettext* is added to the environment globals. An internationalized application then has to provide at least an *gettext* and optoinally a *ngettext* function into the namespace. Either globally or for each rendering.

5.2.1 Environment Methods

After enabling of the extension the environment provides the following additional methods:

`jinja2.Environment.`**`install_gettext_translations`**(*translations*, *newstyle=False*)
Installs a translation globally for that environment. The tranlations object provided must implement at least *ugettext* and *ungettext*. The *gettext.NullTranslations* and *gettext.GNUTranslations* classes as well as Babels *Translations* class are supported. Changed in version 2.5: newstyle gettext added

`jinja2.Environment.`**`install_null_translations`**(*newstyle=False*)
Install dummy gettext functions. This is useful if you want to prepare the application for internationalization but don't want to implement the full internationalization system yet. Changed in version 2.5: newstyle gettext added

`jinja2.Environment.`**`install_gettext_callables`**(*gettext*, *ngettext*, *newstyle=False*)
Installs the given *gettext* and *ngettext* callables into the environment as globals. They

are supposed to behave exactly like the standard library's `gettext.ugettext()` and `gettext.ungettext()` functions.

If *newstyle* is activated, the callables are wrapped to work like newstyle callables. See *Newstyle Gettext* for more information. New in version 2.5.

`jinja2.Environment.`**`uninstall_gettext_translations`**`()`
Uninstall the translations again.

`jinja2.Environment.`**`extract_translations`**`(`*source*`)`
Extract localizable strings from the given template node or source.

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

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

If Babel is installed *the babel integration* can be used to extract strings for babel.

For a web application that is available in multiple languages but gives all the users the same language (for example a multilingual forum software installed for a French community) may load the translations once and add the translation methods to the environment at environment generation time:

```
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 as part *of the template documentation*.

5.2.2 Newstyle Gettext

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

What's the big difference between standard and newstyle gettext calls? In general they are less to type and less error prone. Also if they are used in an autoescaping environment they better support automatic escaping. Here some common differences between old and new calls:

standard gettext:

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

newstyle gettext looks like this instead:

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

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

Furthermore with newstyle gettext, string formatting is also used if no placeholders are used which makes all strings behave exactly the same. Last but not least are newstyle gettext calls able to properly mark strings for autoescaping which solves lots of escaping related issues many templates are experiencing over time when using autoescaping.

5.3 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.

5.4 Loop Controls

Import name: *jinja2.ext.loopcontrols*

This extension adds support for *break* and *continue* in loops. After enabling Jinja2 provides those two keywords which work exactly like in Python.

5.5 With Statement

Import name: *jinja2.ext.with_* New in version 2.3. This extension adds support for the with keyword. Using this keyword it is possible to enforce a nested scope in a template. Variables can be declared directly in the opening block of the with statement or using a standard *set* statement directly within.

5.6 Autoescape Extension

Import name: *jinja2.ext.autoescape* New in version 2.4. The autoescape extension allows you to toggle the autoescape feature from within the template. If the environment's `autoescape` setting is set to *False* it can be activated, if it's *True* it can be deactivated. The setting overriding is scoped.

5.7 Writing Extensions

By writing extensions you can add custom tags to Jinja2. This is a non trival 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 Jinja2 template compiler which does not validate the node tree you are possing 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.

5.7.1 Example Extension

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

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


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

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

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

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

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

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

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

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

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

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

And here is how you use it in an environment:

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

env = Environment(extensions=[FragmentCacheExtension])
```

```
env.fragment_cache = SimpleCache()
```

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

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

5.7.2 Extension API

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

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

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

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

`identifier`

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.

`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.

`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`.

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

`parse`(*parser*)

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

`attr`(*name, lineno=None*)

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

```
self.attr('_my_attribute', lineno=lineno)
```

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

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

5.7.3 Parser API

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 Jinja2 uses. It's passed to extensions and can be used to parse expressions or statements.

`filename`

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

`name`

The load name of the template.

`stream`

The current `TokenStream`

`parse_expression`(*with_condexpr=True*)

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

`parse_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.

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

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

`parse_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.

`free_identifier`(*lineno=None*)

Return a new free identifier as `InternalName`.

`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.

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

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

current
The current `Token`.

push (*token*)
Push a token back to the stream.

look ()
Look at the next token.

eos
Are we at the end of the stream?

skip (*n=1*)
Got n tokens ahead.

next ()
Go one token ahead and return the old one

next_if (*expr*)
Perform the token test and return the token if it matched. Otherwise the return value is *None*.

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

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

class `jinja2.lexer.`**Token**
Token class.

lineno
The line number of the token

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

test_any (**iterable*)
Test against multiple token expressions.

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.

5.7.4 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 Jinja2 versions but will stay backwards compatible.

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

class `jinja2.nodes.`**`Node`**

Baseclass for all Jinja2 nodes. There are a number of nodes available of different types. There are three major types:

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

All nodes have fields and attributes. Fields may be other nodes, lists, or arbitrary values. Fields are passed to the constructor as regular positional arguments, attributes as keyword arguments. Each node has two attributes: *lineno* (the line number of the node) and *environment*. The *environment* attribute is set at the end of the parsing process for all nodes automatically.

`find`(*node_type*)
Find the first node of a given type. If no such node exists the return value is *None*.

`find_all`(*node_type*)
Find all the nodes of a given type. If the type is a tuple, the check is performed for any of the tuple items.

`iter_child_nodes`(*exclude=None*, *only=None*)
Iterates over all direct child nodes of the node. This iterates over all fields and yields the values of they are nodes. If the value of a field is a list all the nodes in that list are returned.

`iter_fields`(*exclude=None*, *only=None*)
This method iterates over all fields that are defined and yields `(key, value)` tuples. Per default all fields are returned, but it's possible to limit that to some fields by providing the *only* parameter or to exclude some using the *exclude* parameter. Both should be sets or tuples of field names.

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

`set_environment`(*environment*)
Set the environment for all nodes.

`set_lineno`(*lineno*, *override=False*)
Set the line numbers of the node and children.

class `jinja2.nodes.`**`Expr`**

Baseclass for all expressions.

Node type `Node`

`as_const`(*eval_ctx=None*)
Return the value of the expression as constant or raise `Impossible` if this was not possible.

An `EvalContext` can be provided, if none is given a default context is created which requires the nodes to have an attached environment. Changed in version 2.4: the *eval_ctx* parameter was added.

`can_assign`()
Check if it's possible to assign something to this node.

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

Baseclass for all binary expressions.

Node type `Expr`

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

Add the left to the right node.

Node type `BinExpr`

class `jinja2.nodes.`**`And`**(*left*, *right*)
Short circuited AND.

Node type `BinExpr`

class `jinja2.nodes.`**`Div`**(*left*, *right*)
Divides the left by the right node.

Node type `BinExpr`

class `jinja2.nodes.`**`FloorDiv`**(*left*, *right*)
Divides the left by the right node and truncates conver the result into an integer by truncating.

Node type `BinExpr`

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

Node type `BinExpr`

class `jinja2.nodes.`**`Mul`**(*left*, *right*)
Multiplies the left with the right node.

Node type `BinExpr`

class `jinja2.nodes.`**`Or`**(*left*, *right*)
Short circuited OR.

Node type `BinExpr`

class `jinja2.nodes.`**`Pow`**(*left*, *right*)
Left to the power of right.

Node type `BinExpr`

class `jinja2.nodes.`**`Sub`**(*left*, *right*)
Substract the right from the left node.

Node type `BinExpr`

class `jinja2.nodes.`**`Call`**(*node*, *args*, *kwargs*, *dyn_args*, *dyn_kwargs*)
Calls an expression. *args* is a list of arguments, *kwargs* a 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`

class `jinja2.nodes.`**`Compare`**(*expr*, *ops*)
Compares an expression with some other expressions. *ops* must be a list of `Operand`s.

Node type `Expr`

class `jinja2.nodes.`**`Concat`**(*nodes*)
Concatenates the list of expressions provided after converting them to unicode.

Node type `Expr`

class `jinja2.nodes.`**`CondExpr`**(*test*, *expr1*, *expr2*)
A conditional expression (inline if expression). (`{{ foo if bar else baz }}`)

Node type `Expr`

class `jinja2.nodes.`**`ContextReference`**
Returns the current template context. It can be used like a `Name` node, with a `'load'` ctx and will return 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'))
```

Node type `Expr`

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`

class `jinja2.nodes.`**`ExtensionAttribute`**`(`*identifier*, *name*`)`
Returns the attribute of an extension bound to the environment. The identifier is the identifier of the `Extension`.

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

Node type `Expr`

class `jinja2.nodes.`**`Filter`**`(`*node*, *name*, *args*, *kwargs*, *dyn_args*, *dyn_kwargs*`)`
This node applies a filter on an expression. *name* is the name of the filter, the rest of the fields are the same as for `Call`.

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

Node type `Expr`

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`

class `jinja2.nodes.`**`Getitem`**`(`*node*, *arg*, *ctx*`)`
Get an attribute or item from an expression and prefer the item.

Node type `Expr`

class `jinja2.nodes.`**`ImportedName`**`(`*importname*`)`
If created with an import name the import name is returned on 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`

class `jinja2.nodes.`**`InternalName`**`(`*name*`)`
An internal name in the compiler. You cannot create these nodes yourself but the parser provides a `free_identifier()` method that creates a new identifier for you. This identifier is not available from the template and is not threated specially by the compiler.

Node type `Expr`

class `jinja2.nodes.`**`Literal`**
Baseclass for literals.

Node type `Expr`

class `jinja2.nodes.`**`Const`**`(`*value*`)`
All constant values. The parser will return this node for 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`

class `jinja2.nodes.`**`Dict`**`(`*items*`)`
Any dict literal such as `{1: 2, 3: 4}`. The items must be a list of `Pair` nodes.

Node type `Literal`

class `jinja2.nodes.`**`List`**`(`*items*`)`
Any list literal such as `[1, 2, 3]`

Node type `Literal`

class `jinja2.nodes.`**`TemplateData`**(*data*)
A constant template string.

Node type `Literal`

class `jinja2.nodes.`**`Tuple`**(*items*, *ctx*)
For loop unpacking and some other things like multiple arguments for subscripts. Like for `Name` *ctx* specifies if the tuple is used for loading the names or storing.

Node type `Literal`

class `jinja2.nodes.`**`MarkSafe`**(*expr*)
Mark the wrapped expression as safe (wrap it as *Markup*).

Node type `Expr`

class `jinja2.nodes.`**`MarkSafeIfAutoescape`**(*expr*)
Mark the wrapped expression as safe (wrap it as *Markup*) but only if autoescaping is active. New in version 2.5.

Node type `Expr`

class `jinja2.nodes.`**`Name`**(*name*, *ctx*)
Looks up a name or stores a value in a name. The *ctx* of the node can be one of the following values:

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

Node type `Expr`

class `jinja2.nodes.`**`Slice`**(*start*, *stop*, *step*)
Represents a slice object. This must only be used as argument for `Subscript`.

Node type `Expr`

class `jinja2.nodes.`**`Test`**(*node*, *name*, *args*, *kwargs*, *dyn_args*, *dyn_kwargs*)
Applies a test on an expression. *name* is the name of the test, the rest of the fields are the same as for `Call`.

Node type `Expr`

class `jinja2.nodes.`**`UnaryExpr`**(*node*)
Baseclass for all unary expressions.

Node type `Expr`

class `jinja2.nodes.`**`Neg`**(*node*)
Make the expression negative.

Node type `UnaryExpr`

class `jinja2.nodes.`**`Not`**(*node*)
Negate the expression.

Node type `UnaryExpr`

class `jinja2.nodes.`**`Pos`**(*node*)
Make the expression positive (noop for most expressions)

Node type `UnaryExpr`

class `jinja2.nodes.`**`Helper`**
Nodes that exist in a specific context only.

Node type `Node`

class `jinja2.nodes.`**`Keyword`**(*key, value*)
A key, value pair for keyword arguments where key is a string.

Node type `Helper`

class `jinja2.nodes.`**`Operand`**(*op, expr*)
Holds an operator and an expression. The following operators are available: `%`, `**`, `*`, `+`, `-`, `//`, `/`, `eq`, `gt`, `gteq`, `in`, `lt`, `lteq`, `ne`, `not`, `notin`

Node type `Helper`

class `jinja2.nodes.`**`Pair`**(*key, value*)
A key, value pair for dicts.

Node type `Helper`

class `jinja2.nodes.`**`Stmt`**
Base node for all statements.

Node type `Node`

class `jinja2.nodes.`**`Assign`**(*target, node*)
Assigns an expression to a target.

Node type `Stmt`

class `jinja2.nodes.`**`Block`**(*name, body, scoped*)
A node that represents a block.

Node type `Stmt`

class `jinja2.nodes.`**`Break`**
Break a loop.

Node type `Stmt`

class `jinja2.nodes.`**`CallBlock`**(*call, args, defaults, body*)
Like a macro without a name but a call instead. *call* is called with the unnamed macro as *caller* argument this node holds.

Node type `Stmt`

class `jinja2.nodes.`**`Continue`**
Continue a loop.

Node type `Stmt`

class `jinja2.nodes.`**`EvalContextModifier`**(*options*)
Modifies the eval context. For each option that should be modified, a `Keyword` has to be added to the `options` list.

Example to change the *autoescape* setting:

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

Node type `Stmt`

class `jinja2.nodes.`**`ScopedEvalContextModifier`**(*options, body*)
Modifies the eval context and reverts it later. Works exactly like `EvalContextModifier` but will only modify the `EvalContext` for nodes in the `body`.

Node type `EvalContextModifier`

class `jinja2.nodes.`**`ExprStmt`**(*node*)
A statement that evaluates an expression and discards the result.

Node type `Stmt`

class `jinja2.nodes.`**`Extends`**(*template*)
Represents an extends statement.

Node type `Stmt`

class `jinja2.nodes.`**`FilterBlock`**(*body*, *filter*)
Node for filter sections.

Node type `Stmt`

class `jinja2.nodes.`**`For`**(*target*, *iter*, *body*, *else_*, *test*, *recursive*)
The for loop. *target* is the target for the iteration (usually a `Name` or `Tuple`), *iter* the iterable. *body* is a 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`

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`

class `jinja2.nodes.`**`If`**(*test*, *body*, *else_*)
If *test* is true, *body* is rendered, else *else_*.

Node type `Stmt`

class `jinja2.nodes.`**`Import`**(*template*, *target*, *with_context*)
A node that represents the import tag.

Node type `Stmt`

class `jinja2.nodes.`**`Include`**(*template*, *with_context*, *ignore_missing*)
A node that represents the include tag.

Node type `Stmt`

class `jinja2.nodes.`**`Macro`**(*name*, *args*, *defaults*, *body*)
A macro definition. *name* is the name of the macro, *args* a list 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`

class `jinja2.nodes.`**`Output`**(*nodes*)
A node that holds multiple expressions which are then printed out. This is used both for the *print* statement and the regular template data.

Node type `Stmt`

class `jinja2.nodes.`**`Scope`**(*body*)
An artificial scope.

Node type `Stmt`

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`

exception `jinja2.nodes.`**`Impossible`**
Raised if the node could not perform a requested action.

CHAPTER

SIX

INTEGRATION

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

6.1 Babel Integration

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

Gettext messages extracted from both *trans* tags and code expressions.

To extract gettext messages from templates, the project needs a Jinja2 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 extraction 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 *extensions* value. The i18n extension is added automatically.

6.2 Pylons

With Pylons 0.9.7 onwards it's incredible easy to integrate Jinja into a Pylons powered application.

The template engine is configured in *config/environment.py*. The configuration for Jinja2 looks something like that:

```
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 into strict mode. Per default any attribute to not existing attributes on the *c* object return an empty string and not an undefined object. To change this just use this snippet and add it into your *config/environment.py*:

```
config['pylons.strict_c'] = True
```

6.3 TextMate

Inside the *ext* folder of Jinja2 there is a bundle for TextMate that supports syntax highlighting for Jinja1 and Jinja2 for text based templates as well as HTML. It also contains a few often used snippets.

6.4 Vim

A syntax plugin for Vim exists in the Vim-scripts directory as well as the ext folder of Jinja2. The script supports Jinja1 and Jinja2. Once installed two file types are available *jinja* and *htmljinja*. The first one for text based templates, the latter for HTML templates.

Copy the files into your *syntax* folder.

CHAPTER

SEVEN

SWITCHING FROM OTHER TEMPLATE ENGINES

If you have used a different template engine in the past and want to swtich to Jinja2 here is a small guide that shows the basic syntatic and semantic changes between some common, similar text template engines for Python.

7.1 Jinja1

Jinja2 is mostly compatible with Jinja1 in terms of API usage and template syntax. The differences between Jinja1 and 2 are explained in the following list.

7.1.1 API

Loaders Jinja2 uses a different loader API. Because the internal representation of templates changed there is no longer support for external caching systems such as memcached. The memory consumed by templates is comparable with regular Python modules now and external caching doesn't give any advantage. If you have used a custom loader in the past have a look at the new *loader API*.

Loading templates from strings In the past it was possible to generate templates from a string with the default environment configuration by using *jinja.from_string*. Jinja2 provides a `Template` class that can be used to do the same, but with optional additional configuration.

Automatic unicode conversion Jinja1 performed automatic conversion of bytestrings in a given encoding into unicode objects. This conversion is no longer implemented as it was inconsistent as most libraries are using the regular Python ASCII bytestring to Unicode conversion. An application powered by Jinja2 *has to* use unicode internally everywhere or make sure that Jinja2 only gets unicode strings passed.

i18n Jinja1 used custom translators for internationalization. i18n is now available as Jinja2 extension and uses a simpler, more gettext friendly interface and has support for babel. For more details see *i18n Extension*.

Internal methods Jinja1 exposed a few internal methods on the environment object such as *call_function*, *get_attribute* and others. While they were marked as being an internal method it was possible to override them. Jinja2 doesn't have equivalent methods.

Sandbox Jinja1 was running sandbox mode by default. Few applications actually used that feature so it became optional in Jinja2. For more details about the sandboxed execution see `SandboxedEnvironment`.

Context Jinja1 had a stacked context as storage for variables passed to the environment. In Jinja2 a similar object exists but it doesn't allow modifications nor is it a singleton. As inheritance is dynamic now multiple context objects may exist during template evaluation.

Filters and Tests Filters and tests are regular functions now. It's no longer necessary and allowed to use factory functions.

7.1.2 Templates

Jinja2 has mostly the same syntax as Jinja1. What's different is that macros require parentheses around the argument list now.

Additionally Jinja2 allows dynamic inheritance now and dynamic includes. The old helper function *rendertemplate* is gone now, *include* can be used instead. Includes no longer import macros and variable assignments, for that the new *import* tag is used. This concept is explained in the *Import* documentation.

Another small change happened in the *for*-tag. The special loop variable doesn't have a *parent* attribute, instead you have to alias the loop yourself. See *Accessing the parent Loop* for more details.

7.2 Django

If you have previously worked with Django templates, you should find Jinja2 very familiar. In fact, most of the syntax elements look and work the same.

However, Jinja2 provides some more syntax elements covered in the documentation and some work a bit different.

This section covers the template changes. As the API is fundamentally different we won't cover it here.

7.2.1 Method Calls

In Django method calls work implicitly. With Jinja2 you have to specify that you want to call an object. Thus this Django code:

```
{% for page in user.get_created_pages %}
    ...
{% endfor %}
```

will look like this in Jinja:

```
{% for page in user.get_created_pages() %}
    ...
{% endfor %}
```

This allows you to pass variables to the function which is also used for macros which is not possible in Django.

7.2.2 Conditions

In Django you can use the following constructs to check for equality:

```
{% ifequal foo "bar" %}
    ...
{% else %}
    ...
{% endifequal %}
```

In Jinja2 you can use the normal if statement in combination with operators:

```
{% if foo == 'bar' %}
    ...
{% else %}
```

```
    ...
{% endif %}
```

You can also have multiple elif branches in your template:

```
{% if something %}
    ...
{% elif otherthing %}
    ...
{% elif foothing %}
    ...
{% else %}
    ...
{% endif %}
```

7.2.3 Filter Arguments

Jinja2 provides more than one argument for filters. Also the syntax for argument passing is different. A template that looks like this in Django:

```
{{ items|join:", " }}
```

looks like this in Jinja2:

```
{{ items|join(', ') }}
```

In fact it's a bit more verbose but it allows different types of arguments - including variables - and more than one of them.

7.2.4 Tests

In addition to filters there also are tests you can perform using the is operator. Here are some examples:

```
{% if user.user_id is odd %}
    {{ user.username|e }} is odd
{% else %}
    hmm. {{ user.username|e }} looks pretty normal
{% endif %}
```

7.2.5 Loops

For loops work very similar to Django, the only incompatibility is that in Jinja2 the special variable for the loop context is called *loop* and not *forloop* like in Django.

7.2.6 Cycle

The `{% cycle %}` tag does not exist in Jinja because of it's implicit nature. However you can achieve mostly the same by using the *cycle* method on a loop object.

The following Django template:

```
{% for user in users %}
    <li class="{% cycle 'odd' 'even' %}">{{ user }}</li>
{% endfor %}
```

Would look like this in Jinja:

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

There is no equivalent of `{% cycle ... as variable %}`.

7.3 Mako

If you have used Mako so far and want to switch to Jinja2 you can configure Jinja2 to look more like Mako:

```
env = Environment('<%', '%>', '${', '}', '%')
```

Once the environment is configure like that Jinja2 should be able to interpret a small subset of Mako templates. Jinja2 does not support embedded Python code so you would have to move that out of the template. The syntax for defs (in Jinja2 defs are called macros) and template inheritance is different too. The following Mako template:

```
<%inherit file="layout.html" />
<%def name="title()">Page Title</%def>
<ul>
% for item in list:
    <li>${item}</li>
% endfor
</ul>
```

Looks like this in Jinja2 with the above configuration:

```
<% extends "layout.html" %>
<% block title %>Page Title<% endblock %>
<% block body %>
<ul>
% for item in list:
    <li>${item}</li>
% endfor
</ul>
<% endblock %>
```

CHAPTER

EIGHT

TIPS AND TRICKS

This part of the documentation shows some tips and tricks for Jinja2 templates.

8.1 Null-Master Fallback

Jinja2 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 igored, 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 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 'master.html' %}{% endif -%}
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<title>{% block title %}The Page Title{% endblock %}</title>
<link rel="stylesheet" href="style.css" type="text/css">
{% block body %}
  <p>This is the page body.</p>
{% endblock %}
```

8.2 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 amount of strings. Each time this tag is encountered the next item from the list is rendered.

8.3 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 defined 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>
...
```

8.4 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>
```

CHAPTER

NINE

FREQUENTLY ASKED QUESTIONS

This page answers some of the often asked questions about Jinja.

9.1 Why is it called Jinja?

The name Jinja was chosen because it's the name of a Japanese temple and temple and template share a similar pronunciation. It is not named after the capital city of Uganda.

9.2 How fast is it?

We really hate benchmarks especially since they don't reflect much. The performance of a template depends on many factors and you would have to benchmark different engines in different situations. The benchmarks from the testsuite show that Jinja2 has a similar performance to Mako and is between 10 and 20 times faster than Django's template engine or Genshi. These numbers should be taken with tons of salt as the benchmarks that took these numbers only test a few performance related situations such as looping. Generally speaking the performance of a template engine doesn't matter much as the usual bottleneck in a web application is either the database or the application code.

9.3 How Compatible is Jinja2 with Django?

The default syntax of Jinja2 matches Django syntax in many ways. However this similarity doesn't mean that you can use a Django template unmodified in Jinja2. For example filter arguments use a function call syntax rather than a colon to separate filter name and arguments. Additionally the extension interface in Jinja is fundamentally different from the Django one which means that your custom tags won't work any longer.

Generally speaking you will use much less custom extensions as the Jinja template system allows you to use a certain subset of Python expressions which can replace most Django extensions. For example instead of using something like this:

```
{% load comments %}
{% get_latest_comments 10 as latest_comments %}
{% for comment in latest_comments %}
    ...
{% endfor %}
```

You will most likely provide an object with attributes to retrieve comments from the database:

```
{% for comment in models.comments.latest(10) %}
    ...
{% endfor %}
```

Or directly provide the model for quick testing:

```
{% for comment in Comment.objects.order_by('-pub_date')[:10] %}
    ...
{% endfor %}
```

Please keep in mind that even though you may put such things into templates it still isn't a good idea. Queries should go into the view code and not the template!

9.4 Isn't it a terrible idea to put Logic into Templates?

Without a doubt you should try to remove as much logic from templates as possible. But templates without any logic mean that you have to do all the processing in the code which is boring and stupid. A template engine that does that is shipped with Python and called *string.Template*. Comes without loops and if conditions and is by far the fastest template engine you can get for Python.

So some amount of logic is required in templates to keep everyone happy. And Jinja leaves it pretty much to you how much logic you want to put into templates. There are some restrictions in what you can do and what not.

Jinja2 neither allows you to put arbitrary Python code into templates nor does it allow all Python expressions. The operators are limited to the most common ones and more advanced expressions such as list comprehensions and generator expressions are not supported. This keeps the template engine easier to maintain and templates more readable.

9.5 Why is Autoescaping not the Default?

There are multiple reasons why automatic escaping is not the default mode and also not the recommended one. While automatic escaping of variables means that you will less likely have an XSS problem it also causes a huge amount of extra processing in the template engine which can cause serious performance problems. As Python doesn't provide a way to mark strings as unsafe Jinja has to hack around that limitation by providing a custom string class (the `Markup` string) that safely interacts with safe and unsafe strings.

With explicit escaping however the template engine doesn't have to perform any safety checks on variables. Also a human knows not to escape integers or strings that may never contain characters one has to escape or already HTML markup. For example when iterating over a list over a table of integers and floats for a table of statistics the template designer can omit the escaping because he knows that integers or floats don't contain any unsafe parameters.

Additionally Jinja2 is a general purpose template engine and not only used for HTML/XML generation. For example you may generate LaTeX, emails, CSS, JavaScript, or configuration files.

9.6 Why is the Context immutable?

When writing a `contextfunction()` or something similar you may have noticed that the context tries to stop you from modifying it. If you have managed to modify the context by using an internal context API you may have noticed that changes in the context don't seem to be visible in the template. The reason for this is that Jinja uses the context only as primary data source for template variables for performance reasons.

If you want to modify the context write a function that returns a variable instead that one can assign to a variable by using set:

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

9.7 What is the speedups module and why is it missing?

To achieve a good performance with automatic escaping enabled, the escaping function was also implemented in pure C in older Jinja2 releases and used if Jinja2 was installed with the speedups module.

Because this feature itself is very useful for non-template engines as well it was moved into a separate project on PyPI called MarkupSafe.

Jinja2 no longer ships with a C implementation of it but only the pure Python implementation. It will however check if MarkupSafe is available and installed, and if it is, use the Markup class from MarkupSafe.

So if you want the speedups, just import MarkupSafe.

9.8 My tracebacks look weird. What's happening?

If the debugsupport module is not compiled and you are using a Python installation without ctypes (Python 2.4 without ctypes, Jython or Google's AppEngine) Jinja2 is unable to provide correct debugging information and the traceback may be incomplete. There is currently no good workaround for Jython or the AppEngine as ctypes is unavailable there and it's not possible to use the debugsupport extension.

9.9 Why is there no Python 2.3 support?

Python 2.3 is missing a lot of features that are used heavily in Jinja2. This decision was made as with the upcoming Python 2.6 and 3.0 versions it becomes harder to maintain the code for older Python versions. If you really need Python 2.3 support you either have to use Jinja 1 or other templating engines that still support 2.3.

9.10 My Macros are overriden by something

In some situations the Jinja scoping appears arbitrary:

layout.tmpl:

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

child.tmpl:

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

This will print `LAYOUT` in Jinja2. This is a side effect of having the parent template evaluated after the child one. This allows child templates passing information to the parent template. To avoid this issue rename the macro or variable in the parent template to have an uncommon prefix.

CHAPTER

TEN

JINJA2 CHANGELOG

10.1 Version 2.5.3

(bugfix release, released on October 17th 2010)

- 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)".

10.2 Version 2.5.2

(bugfix release, released on August 18th 2010)

- 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.

10.3 Version 2.5.1

(bugfix release, released on August 17th 2010)

- 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 Jinja2 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 precendence.

10.4 Version 2.5

(codename Incoherence, relased on May 29th 2010)

- 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. (*Newstyle Gettext*)
- it's now possible to register extensions after an environment was created.

10.5 Version 2.4.1

(bugfix release, released on April 20th 2010)

- fixed an error reporting bug for undefineds.

10.6 Version 2.4

(codename Correlation, released on April 13th 2010)

- 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 (*Evaluation Context*).
- extensions have a priority now.

10.7 Version 2.3.1

(bugfix release, released on February 19th 2010)

- fixed an error reporting bug on all python versions
- fixed an error reporting bug on Python 2.4

10.8 Version 2.3

(3000 Pythons, released on February 10th 2010)

- 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 %}...{% endif %}` a syntax error now. (#364)

- added support for translator comments if extracted via babel.
- added with-statement extension.
- experimental Python 3 support.

10.9 Version 2.2.1

(bugfix release, released on September 14th 2009)

- fixes some smaller problems for Jinja2 on Jython.

10.10 Version 2.2

(codename Kong, released on September 13th 2009)

- 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.

10.11 Version 2.1.1

(Bugfix release)

- Fixed a translation error caused by looping over empty recursive loops.

10.12 Version 2.1

(codename Yasuz, released on November 23rd 2008)

- 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. (*Bytecode Cache*)
- 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

10.13 Version 2.0

(codename jinjavitus, released on July 17th 2008)

- 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`.

  For more information see *the implementation details*.
- added support for preprocessing and token stream filtering for extensions. This would allow extensions to allow simplified gettext calls in template data and something similar.
- added `jinja2.environment.TemplateStream.dump()`.
- added missing support for implicit string literal concatenation. `{{ "foo" "bar" }}` is equivalent to `{{ "foobar" }}`
- *else* is optional for conditional expressions. If not given it 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

10.14 Version 2.0rc1

(no codename, released on June 9th 2008)

- first release of Jinja2