The Twig Book
This work is licensed under the new BSD license.
# Contents at a Glance

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>6</td>
</tr>
<tr>
<td>Installation</td>
<td>8</td>
</tr>
<tr>
<td>Twig for Template Designers</td>
<td>10</td>
</tr>
<tr>
<td>Twig for Developers</td>
<td>24</td>
</tr>
<tr>
<td>Extending Twig</td>
<td>34</td>
</tr>
<tr>
<td>Twig Internals</td>
<td>49</td>
</tr>
<tr>
<td>Deprecated Features</td>
<td>52</td>
</tr>
<tr>
<td>Recipes</td>
<td>56</td>
</tr>
<tr>
<td>Coding Standards</td>
<td>66</td>
</tr>
<tr>
<td>apply</td>
<td>68</td>
</tr>
<tr>
<td>autoescape</td>
<td>69</td>
</tr>
<tr>
<td>block</td>
<td>71</td>
</tr>
<tr>
<td>deprecated</td>
<td>72</td>
</tr>
<tr>
<td>do</td>
<td>73</td>
</tr>
<tr>
<td>embed</td>
<td>74</td>
</tr>
<tr>
<td>extends</td>
<td>77</td>
</tr>
<tr>
<td>filter</td>
<td>82</td>
</tr>
<tr>
<td>flush</td>
<td>83</td>
</tr>
<tr>
<td>for</td>
<td>84</td>
</tr>
<tr>
<td>from</td>
<td>88</td>
</tr>
<tr>
<td>if</td>
<td>89</td>
</tr>
<tr>
<td>import</td>
<td>91</td>
</tr>
<tr>
<td>include</td>
<td>92</td>
</tr>
<tr>
<td>macro</td>
<td>94</td>
</tr>
<tr>
<td>sandbox</td>
<td>97</td>
</tr>
<tr>
<td>set</td>
<td>98</td>
</tr>
<tr>
<td>spaceless</td>
<td>100</td>
</tr>
<tr>
<td>use</td>
<td>102</td>
</tr>
<tr>
<td>verbatim</td>
<td>105</td>
</tr>
<tr>
<td>with</td>
<td>106</td>
</tr>
<tr>
<td>abs</td>
<td>107</td>
</tr>
<tr>
<td>batch</td>
<td>108</td>
</tr>
<tr>
<td>capitalize</td>
<td>110</td>
</tr>
<tr>
<td>convert_encoding</td>
<td>111</td>
</tr>
<tr>
<td>date</td>
<td>112</td>
</tr>
<tr>
<td>date_modify</td>
<td>114</td>
</tr>
</tbody>
</table>
Contents at a Glance

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>115</td>
</tr>
<tr>
<td>escape</td>
<td>116</td>
</tr>
<tr>
<td>filter</td>
<td>119</td>
</tr>
<tr>
<td>first</td>
<td>121</td>
</tr>
<tr>
<td>format</td>
<td>122</td>
</tr>
<tr>
<td>join</td>
<td>123</td>
</tr>
<tr>
<td>json_encode</td>
<td>124</td>
</tr>
<tr>
<td>keys</td>
<td>125</td>
</tr>
<tr>
<td>last</td>
<td>126</td>
</tr>
<tr>
<td>length</td>
<td>127</td>
</tr>
<tr>
<td>lower</td>
<td>128</td>
</tr>
<tr>
<td>map</td>
<td>129</td>
</tr>
<tr>
<td>merge</td>
<td>130</td>
</tr>
<tr>
<td>nl2br</td>
<td>132</td>
</tr>
<tr>
<td>number_format</td>
<td>133</td>
</tr>
<tr>
<td>raw</td>
<td>135</td>
</tr>
<tr>
<td>reduce</td>
<td>136</td>
</tr>
<tr>
<td>replace</td>
<td>137</td>
</tr>
<tr>
<td>reverse</td>
<td>138</td>
</tr>
<tr>
<td>round</td>
<td>140</td>
</tr>
<tr>
<td>slice</td>
<td>141</td>
</tr>
<tr>
<td>sort</td>
<td>143</td>
</tr>
<tr>
<td>spaceless</td>
<td>144</td>
</tr>
<tr>
<td>split</td>
<td>146</td>
</tr>
<tr>
<td>striptags</td>
<td>148</td>
</tr>
<tr>
<td>title</td>
<td>149</td>
</tr>
<tr>
<td>trim</td>
<td>150</td>
</tr>
<tr>
<td>upper</td>
<td>152</td>
</tr>
<tr>
<td>url_encode</td>
<td>153</td>
</tr>
<tr>
<td>attribute</td>
<td>154</td>
</tr>
<tr>
<td>block</td>
<td>155</td>
</tr>
<tr>
<td>constant</td>
<td>156</td>
</tr>
<tr>
<td>cycle</td>
<td>157</td>
</tr>
<tr>
<td>date</td>
<td>158</td>
</tr>
<tr>
<td>dump</td>
<td>160</td>
</tr>
<tr>
<td>include</td>
<td>162</td>
</tr>
<tr>
<td>max</td>
<td>164</td>
</tr>
<tr>
<td>min</td>
<td>165</td>
</tr>
<tr>
<td>parent</td>
<td>166</td>
</tr>
<tr>
<td>random</td>
<td>167</td>
</tr>
<tr>
<td>range</td>
<td>168</td>
</tr>
<tr>
<td>source</td>
<td>170</td>
</tr>
<tr>
<td>template_from_string</td>
<td>171</td>
</tr>
<tr>
<td>constant</td>
<td>173</td>
</tr>
<tr>
<td>defined</td>
<td>174</td>
</tr>
<tr>
<td>divisible by</td>
<td>175</td>
</tr>
<tr>
<td>empty</td>
<td>176</td>
</tr>
</tbody>
</table>
Chapter 1
Introduction

Welcome to the documentation for Twig, the flexible, fast, and secure template engine for PHP.
Twig is both designer and developer friendly by sticking to PHP’s principles and adding functionality useful for templating environments.

The key-features are...

- **Fast**: Twig compiles templates down to plain optimized PHP code. The overhead compared to regular PHP code was reduced to the very minimum.
- **Secure**: Twig has a sandbox mode to evaluate untrusted template code. This allows Twig to be used as a template language for applications where users may modify the template design.
- **Flexible**: Twig is powered by a flexible lexer and parser. This allows the developer to define their own custom tags and filters, and to create their own DSL.

Twig is used by many Open-Source projects like Symfony, Drupal8, eZPublish, phpBB, Matomo, OroCRM; and many frameworks have support for it as well like Slim, Yii, Laravel, and Codeigniter — just to name a few.

Prerequisites

Twig needs at least **PHP 5.5.0** to run.

Installation

The recommended way to install Twig is via Composer:

```
composer require "twig/twig:^1.0"
```
To learn more about the other installation methods, read the installation chapter; it also explains how to install the Twig C extension.

## Basic API Usage

This section gives you a brief introduction to the PHP API for Twig.

```php
Listing 1-2
require_once '/path/to/vendor/autoload.php';
$loader = new Twig\Loader\ArrayLoader(['index' => 'Hello {{ name }}!', ]);  
$twig = new Twig\Environment($loader);
echo $twig->render('index', ['name' => 'Fabien']);
```

Twig uses a loader (`Twig\Loader\ArrayLoader`) to locate templates, and an environment (`Twig\Environment`) to store its configuration.

The `render()` method loads the template passed as a first argument and renders it with the variables passed as a second argument.

As templates are generally stored on the filesystem, Twig also comes with a filesystem loader:

```php
Listing 1-3
$loader = new Twig\Loader\FilesystemLoader('/path/to/templates');
$twig = new Twig\Environment($loader, [
  'cache' => '/path/to/compilation_cache',
]);

echo $twig->render('index.html', ['name' => 'Fabien']);
```
You have multiple ways to install Twig.

### Installing the Twig PHP package

Install Composer\(^1\) and run the following command:

```
composer require "twig/twig:^1.0"
```

### Installing the C extension

*New in version 1.4:* The C extension was added in Twig 1.4.

Twig comes with an **optional** C extension that improves the performance of the Twig runtime engine. Note that this extension does not replace the PHP code but only provides an optimized version of the `\Twig\Template::getAttribute()` method; you must still install the regular PHP code. The C extension is only compatible and useful for **PHP5**.

Install it like any other PHP extensions:

```
cd ext/twig
phpize
./configure
make
make install
```

For Windows:

1. Setup the build environment following the [PHP documentation]\(^2\)
2. Put Twig's C extension source code into `C:\php-sdk\phpdev\vcXXX\x86\php-source-directory\ext\twig`
3. Use the `configure --disable-all --enable-cli --enable-twig=shared` command instead of step 14

---

1. https://getcomposer.org/download/
4. `nmake`

5. Copy the `C:\php-sdk\phpdev\vcXX\x86\php-source-directory\Release\TS\php_twig.dll` file to your PHP setup.

For Windows ZendServer, ZTS is not enabled as mentioned in Zend Server FAQ. You have to use `configure --disable-all --disable-zts --enable-cli --enable-twig=shared` to be able to build the twig C extension for ZendServer.

The built DLL will be available in `C:\php-sdk\phpdev\vcXX\x86\php-source-directory\Release`

Finally, enable the extension in your `php.ini` configuration file:

```ini
extension=twig.so # For Unix systems
extension=php_twig.dll # For Windows systems
```

And from now on, Twig will automatically compile your templates to take advantage of the C extension.
Chapter 3
Twig for Template Designers

This document describes the syntax and semantics of the template engine and will be most useful as reference to those creating Twig templates.

Synopsis

A template is a regular 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 template’s logic.

Below is a minimal template that illustrates a few basics. We will cover further details later on:

```
<!DOCTYPE html>
<html>
<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>
```

There are two kinds of delimiters: `{% ... %}` and `{{ ... }}`. The first one is used to execute statements such as for-loops, the latter outputs the result of an expression.

IDEs Integration

Many IDEs support syntax highlighting and auto-completion for Twig:
• Textmate via the Twig bundle
• Vim via the Jinja syntax plugin or the vim-twig plugin
• Netbeans via the Twig syntax plugin (until 7.1, native as of 7.2)
• PhpStorm (native as of 2.1)
• Eclipse via the Twig plugin
• Sublime Text via the Twig bundle
• GtkSourceView via the Twig language definition (used by gedit and other projects)
• Coda and SubEthaEdit via the Twig syntax mode
• Coda 2 via the other Twig syntax mode
• Komodo and Komodo Edit via the Twig highlight/syntax check mode
• Notepad++ via the Notepad++ Twig Highlighter
• Emacs via web-mode.el
• Atom via the PHP-twig for atom
• Visual Studio Code via the Twig pack

Also, TwigFiddle is an online service that allows you to execute Twig templates from a browser; it supports all versions of Twig.

Variables

The application passes variables to the templates for manipulation in the template. Variables may have attributes or elements you can access, too. The visual representation of a variable depends heavily on the application providing it.

Use a dot (.) to access attributes of a variable (methods or properties of a PHP object, or items of a PHP array):

Listing 3-2

It's important to know that the curly braces are not part of the variable but the print statement. When accessing variables inside tags, don't put the braces around them.

1. https://github.com/Anomareh/PHP-Twig.tmbundle
5. https://github.com/pulse00/Twig-Eclipse-Plugin
6. https://github.com/Anomareh/PHP-Twig.tmbundle
8. https://github.com/bobthecow/Twig-HTML.mode
11. http://web-mode.org/
Implementation

For convenience’s sake \texttt{foo.bar} does the following things on the PHP layer:

- check if \texttt{foo} is an array and \texttt{bar} a valid element;
- if not, and if \texttt{foo} is an object, check that \texttt{bar} is a valid property;
- if not, and if \texttt{foo} is an object, check that \texttt{bar} is a valid method (even if \texttt{bar} is the constructor - use \_\_construct\_\_() instead);
- if not, and if \texttt{foo} is an object, check that \texttt{getBar} is a valid method;
- if not, and if \texttt{foo} is an object, check that \texttt{isBar} is a valid method;
- if not, return a \texttt{null} value.

Twig also supports a specific syntax for accessing items on PHP arrays, \texttt{foo['bar']}:  

- check if \texttt{foo} is an array and \texttt{bar} a valid element;
- if not, return a \texttt{null} value.

If a variable or attribute does not exist, you will receive a \texttt{null} value when the \texttt{strict_variables} option is set to \texttt{false}; alternatively, if \texttt{strict_variables} is set, Twig will throw an error (see environment options).

If you want to access a dynamic attribute of a variable, use the attribute function instead. The \texttt{attribute} function is also useful when the attribute contains special characters (like \texttt{-} that would be interpreted as the minus operator):

```{1}
{# equivalent to the non-working foo.data-foo #}
{{attribute(foo, 'data-foo')}}
```

Global Variables

The following variables are always available in templates:

- \_\_self\_: references the current template;
- \_context\_: references the current context;
- \_charset\_: references the current charset.

Setting Variables

You can assign values to variables inside code blocks. Assignments use the set tag:

```{1}
{% set foo = 'foo' %}
{% set foo = [1, 2] %}
{% set foo = {'foo': 'bar'} %}
```

Filters

Variables can be modified by filters. Filters are separated from the variable by a pipe symbol (|). Multiple
filters can be chained. The output of one filter is applied to the next.

The following example removes all HTML tags from the name and title-cases it:

```twig
Listing 3-5
1  {{ name|striptags|title }}
```

Filters that accept arguments have parentheses around the arguments. This example joins the elements of a list by commas:

```twig
Listing 3-6
1  {{ list|join(’, ’) }}
```

To apply a filter on a section of code, wrap it with the apply tag:

```twig
Listing 3-7
1 (% apply upper %)
2  This text becomes uppercase
3 (% endapply %)
```

Go to the filters page to learn more about built-in filters.

The apply tag was introduced in Twig 1.40; use the filter tag with previous versions.

Functions

Functions can be called to generate content. Functions are called by their name followed by parentheses (()) and may have arguments.

For instance, the range function returns a list containing an arithmetic progression of integers:

```twig
Listing 3-8
1 (% for i in range(0, 3) %)
2  {{ i }},
3 (% endfor %)
```

Go to the functions page to learn more about the built-in functions.

Named Arguments

New in version 1.12: Support for named arguments was added in Twig 1.12.

```twig
Listing 3-9
1 (% for i in range(low=1, high=10, step=2) %)
2  {{ i }},
3 (% endfor %)
```

Using named arguments makes your templates more explicit about the meaning of the values you pass as arguments:

```twig
Listing 3-10
1  {{ data|convert_encoding(’UTF-8’, ’iso-2022-jp’) }}
2
3 (# versus #)
4```
Named arguments also allow you to skip some arguments for which you don’t want to change the default value:

```twig
Listing 3-11
1 (# the first argument is the date format, which defaults to the global date format if null is passed #)
2 {{ "now"|date(null, "Europe/Paris") }}
3
4 (# or skip the format value by using a named argument for the time zone #)
5 {{ "now"|date(timezone="Europe/Paris") }}
```

You can also use both positional and named arguments in one call, in which case positional arguments must always come before named arguments:

```twig
Listing 3-12
1 {{ "now"|date('d/m/Y H:i', timezone="Europe/Paris") }}
```

Each function and filter documentation page has a section where the names of all arguments are listed when supported.

## Control Structure

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

For example, to display a list of users provided in a variable called **users**, use the `for` tag:

```twig
Listing 3-13
1 <h1>Members</h1>
2 <ul>
3  {% for user in users %}
4  <li>{{ user.username|e }}</li>
5  {% endfor %}
6 </ul>
```

The `if` tag can be used to test an expression:

```twig
Listing 3-14
1 (% if users|length > 0 %)
2 <ul>
3  {% for user in users %}
4  <li>{{ user.username|e }}</li>
5  {% endfor %}
6 </ul>
7 (% endif %)
```

Go to the tags page to learn more about the built-in tags.

## Comments

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

```yaml
Listing 3-15
---
# note: disabled template because we no longer use this
{% for user in users %}
   ...
{% endfor %}
#}
```

## Including other Templates

The include function is useful to include a template and return the rendered content of that template into the current one:

```yaml
Listing 3-16
---
{{ include('sidebar.html') }}
```

By default, included templates have access to the same context as the template which includes them. This means that any variable defined in the main template will be available in the included template too:

```yaml
Listing 3-17
---
{% for box in boxes %}
   {{ include('render_box.html') }}
{% endfor %}
```

The included template `render_box.html` is able to access the `box` variable.

The name of the template depends on the template loader. For instance, the `Twig\Loader\FilesystemLoader` allows you to access other templates by giving the filename. You can access templates in subdirectories with a slash:

```yaml
Listing 3-18
---
{{ include('sections/articles/sidebar.html') }}
```

This behavior depends on the application embedding Twig.

## Template Inheritance

The most powerful part of Twig 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.

It's easier to understand the concept by starting with an example.

Let's define a base template, `base.html`, which defines an HTML skeleton document that might be used for a two-column page:

```yaml
Listing 3-19
---
<!DOCTYPE html>
<html>
<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 2011 by <a href="http://domain.invalid/">you</a>.
   {% endblock %}</div>
</body>
```

The `base.html` template is the base template that contains the common elements of the site. It defines blocks for the `head`, `content`, and `footer` sections.

Child templates can override these blocks to add their specific content. For example, a child template `article.html` might look like this:

```yaml
Listing 3-20
---
{% extends "base.html" %}
{% block content %}
   <h1>{{ article.title }}</h1>
   <p>{{ article.body }}</p>
{% endblock %}
```

This child template inherits the `base.html` template and overrides the `content` block to add the title and body of the article.

Template inheritance is a powerful feature of Twig that allows you to create reusable and maintainable templates. It's an essential tool for any template designer.
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.

A child template might look like this:

```html
{% extends "base.html" %}

{% block title %}Index{% endblock %}

{% block head %}
    {{ parent() }}
    <style type="text/css">
        .important { color: #336699; }
    </style>
{% endblock %}

{% block content %}
    <h1>Index</h1>
    <p class="important">
        Welcome to my awesome homepage.
    </p>
{% endblock %}

{% block sidebar %}
    <h3>Table Of Contents</h3>
    ... 
    {{ parent() }}
{% 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.

Note that since the child template doesn't define the `footer` block, the value from the parent template is used instead.

It's possible to render the contents of the parent block by using the `parent` function. This gives back the results of the parent block:

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

The documentation page for the `extends` tag describes more advanced features like block nesting, scope, dynamic inheritance, and conditional inheritance.

Twig also supports multiple inheritance via "horizontal reuse" with the help of the `use` tag.
**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.

Twig supports both, automatic escaping is enabled by default.

The automatic escaping strategy can be configured via the autoescape option and defaults to `html`.

**Working with Manual Escaping**

If manual escaping is enabled, it is **your** responsibility to escape variables if needed. What to escape? Any variable that comes from an untrusted source.

Escaping works by using the `escape` or `e` filter:

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

By default, the `escape` filter uses the `html` strategy, but depending on the escaping context, you might want to explicitly use an other strategy:

```twig
{{ user.username|e('js') }}
{{ user.username|e('css') }}
{{ user.username|e('url') }}
{{ user.username|e('html_attr') }}
```

**Working with Automatic Escaping**

Whether automatic escaping is enabled or not, you can mark a section of a template to be escaped or not by using the `autoescape` tag:

```twig
{% autoescape %}

Everything will be automatically escaped in this block (using the HTML strategy)

{% endautoescape %}
```

By default, auto-escaping uses the `html` escaping strategy. If you output variables in other contexts, you need to explicitly escape them with the appropriate escaping strategy:

```twig
{% autoescape 'js' %}

Everything will be automatically escaped in this block (using the JS strategy)

{% endautoescape %}
```

**Escaping**

It is sometimes desirable or even necessary to have Twig 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:

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

For bigger sections it makes sense to mark a block verbatim.
Macros

New in version 1.12: Support for default argument values was added in Twig 1.12.

Macros are comparable with functions in regular programming languages. They are useful to reuse HTML fragments to not repeat yourself. They are described in the macro tag documentation.

Expressions

Twig allows expressions everywhere.

The operator precedence is as follows, with the lowest-precedence operators listed first: ?: (ternary operator), b-and, b-xor, b-or, or, and, ==, !, <, >, <=, <, in, matches, starts with, ends with, ., +, -, ~, *, /, //, %, is (tests), **, ??, | (filters), [], and ..

Literals

New in version 1.5: Support for hash keys as names and expressions was added in Twig 1.5.

The simplest form of expressions are literals. Literals are representations for PHP types such as strings, numbers, and arrays. 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). A string can contain a delimiter if it is preceded by a backslash (\) -- like in 'It\'s good'. If the string contains a backslash (e.g. 'c:\Program Files') escape it by doubling it (e.g. 'c:\\Program Files').
- 42 / 42.23: Integers and floating point numbers are created by writing the number down. If a dot is present the number is a float, otherwise an integer.
- ["foo", "bar"]: Arrays are defined by a sequence of expressions separated by a comma (,) and wrapped with squared brackets ([]).
- {"foo": "bar"): Hashes are defined by a list of keys and values separated by a comma (,) and wrapped with curly braces ({}):
{ 2: 'foo', 4: 'bar' }

{% set foo = 'foo' %}

{ (foo): 'foo', (1 + 1): 'bar', (foo ~ 'b'): 'baz' }

• **true** / **false**: **true** represents the true value, **false** represents the false value.

• **null**: **null** represents no specific value. This is the value returned when a variable does not exist. **none** is an alias for **null**.

Arrays and hashes can be nested:

Using double-quoted or single-quoted strings has no impact on performance but string interpolation is only supported in double-quoted strings.

**Math**

Twig allows you to do math in templates; the following operators are supported:

• +: Adds two numbers together (the operands are casted to numbers). {{ 1 + 1 }} is 2.

• -: Subtracts the second number from the first one. {{ 3 - 2 }} is 1.

• /: Divides two numbers. The returned value will be a floating point number. {{ 1 / 2 }} is {{ 0.5 }}.

• %: Calculates the remainder of an integer division. {{ 11 % 7 }} is 4.

• //: Divides two numbers and returns the floored integer result. {{ 20 // 7 }} is 2, {{ -20 // 7 }} is -3 (this is just syntactic sugar for the round filter).

• *: Multiplies the left operand with the right one. {{ 2 * 2 }} would return 4.

• **: Raises the left operand to the power of the right operand. {{ 2 ** 3 }} would return 8.

**Logic**

You can combine multiple expressions with the following operators:

• and: Returns true if the left and the right operands are both true.

• or: Returns true if the left or the right operand is true.

• not: Negates a statement.

• (expr): Groups an expression.

Twig also supports bitwise operators (**b-and**, **b-xor**, and **b-or**).
Operators are case sensitive.

Comparisons

The following comparison operators are supported in any expression: ==, !=, <, >, >=, and <=.

You can also check if a string starts with or ends with another string:

```twig
{% if 'Fabien' starts with 'F' %}
{% endif %}
{% if 'Fabien' ends with 'n' %}
{% endif %}
```

For complex string comparisons, the matches operator allows you to use regular expressions:

```twig
{% if phone matches '/^[\d\d]*$/' %}
{% endif %}
```

Containment Operator

The in operator performs containment test. It returns true if the left operand is contained in the right:

```twig
{{ 1 in [1, 2, 3] }}
{{ 'cd' in 'abcde' }}
```

You can use this filter to perform a containment test on strings, arrays, or objects implementing the Traversable interface.

To perform a negative test, use the not in operator:

```twig
{% if 1 not in [1, 2, 3] %}
{% endif %}
```
### Test Operator

The `is` operator performs tests. Tests can be used to test a variable against a common expression. The right operand is name of the test:

```twig
{% if not (1 in [1, 2, 3]) %}
```

Tests can accept arguments too:

```twig
{% if post.status is constant('Post::PUBLISHED') %}
```

Tests can be negated by using the `is not` operator:

```twig
{% if post.status is not constant('Post::PUBLISHED') %}
```

Go to the tests page to learn more about the built-in tests.

### Other Operators

**New in version 1.12.0:** Support for the extended ternary operator was added in Twig 1.12.0.

The following operators don't fit into any of the other categories:

- `|`: Applies a filter.
- `..`: Creates a sequence based on the operand before and after the operator (this is syntactic sugar for the range function):

```twig
{{ 1..5 }}
```

Note that you must use parentheses when combining it with the filter operator due to the operator precedence rules:

```twig
(1..5)|join(', ')
```

- `~`: Converts all operands into strings and concatenates them. `{{ "Hello " ~ name ~ "!" }}` would return (assuming `name` is `'John`) `Hello John!`.
- `.`: Gets an attribute of a variable.
- `?:`: The ternary operator:

```twig
{{ foo ? 'yes' : 'no' }}
```
• ??: The null-coalescing operator:

```twig
Listing 3-40
1 (# returns the value of foo if it is defined and not null, 'no' otherwise #)
2 {{ foo ?? 'no' }}
```

**String Interpolation**

*New in version 1.5:* String interpolation was added in Twig 1.5.

String interpolation (`#{expression}`) allows any valid expression to appear within a *double-quoted string*. The result of evaluating that expression is inserted into the string:

```twig
Listing 3-41
1 {{ "foo #{bar} baz" }}
2 {{ "foo #{1 + 2} baz" }}
```

**Whitespace Control**

*New in version 1.1:* Tag level whitespace control was added in Twig 1.1.

*New in version 1.139:* Tag level Line whitespace control was added in Twig 1.39.

The first newline after a template tag is removed automatically (like in PHP). Whitespace is not further modified by the template engine, so each whitespace (spaces, tabs, newlines etc.) is returned unchanged.

You can also control whitespace on a per tag level. By using the whitespace control modifiers on your tags, you can trim leading and or trailing whitespace.

Twig supports two modifiers:

- **Whitespace trimming** via the `-` modifier: Removes all whitespace (including newlines);
- **Line whitespace trimming** via the `~` modifier: Removes all whitespace (excluding newlines). Using this modifier on the right disables the default removal of the first newline inherited from PHP.

The modifiers can be used on either side of the tags like in `{{- or -%}}` and they consume all whitespace for that side of the tag. It is possible to use the modifiers on one side of a tag or on both sides:

```twig
Listing 3-42
1 {% set value = 'no spaces' %}
2 {%- No leading/trailing whitespace -%}
3 {%- if true -%}
4 {% (~ value -%)}
5 {%- endif -%}
6 {# output 'no spaces' #}
7
8 <li>{{ value }}</li>
9 {# outputs '<li>no spaces</li>' #}
10
11 <li>{{ value }}</li>
12 {# outputs '<li>no spaces</li>' #}
13
14 <li>{{ value }}</li>
15 {# outputs '<li>no spaces</li>' #}
16
17 <li>{{ value }}</li>
18 {# outputs '<li>no spaces</li>' #}
```
In addition to the whitespace modifiers, Twig also has a `spaceless` filter that removes whitespace between HTML tags:

```twig
{% apply spaceless %}
<div>
  <strong>foo bar</strong>
</div>
{% endapply %}

{% # output will be <div><strong>foo bar</strong></div> #}
```

Note that the `apply` tag was introduced in Twig 1.40; use the `filter` tag with previous versions.

**Extensions**

Twig can be extended. If you want to create your own extensions, read the Creating an Extension chapter.
Chapter 4
Twig for Developers

This chapter describes the API to Twig 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 Twig templates.

Basics

Twig uses a central object called the environment (of class \Twig\Environment). Instances of this class are used to store the configuration and extensions, and are used to load templates.

Most applications create one \Twig\Environment object on application initialization and use that to load templates. In some cases, it might be useful to have multiple environments side by side, with different configurations.

The typical way to configure Twig to load templates for an application looks roughly like this:

```php
require_once '/path/to/vendor/autoload.php';
$loader = new \Twig\Loader\FilesystemLoader('/path/to/templates');
$twig = new \Twig\Environment($loader, [
    'cache' => '/path/to/compilation_cache',
]);
```

This creates a template environment with a default configuration and a loader that looks up templates in the /path/to/templates/ directory. Different loaders are available and you can also write your own if you want to load templates from a database or other resources.
Notice that the second argument of the environment is an array of options. The `cache` option is a compilation cache directory, where Twig caches the compiled templates to avoid the parsing phase for subsequent requests. It is very different from the cache you might want to add for the evaluated templates. For such a need, you can use any available PHP cache library.

## Rendering Templates

To load a template from a Twig environment, call the `load()` method which returns a `Twig\TemplateWrapper` instance:

```php
Listing 4.2
$template = $twig->load('index.html');
```

Before Twig 1.28, use `loadTemplate()` instead which returns a `Twig\Template` instance.

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

```php
Listing 4.3
echo $template->render(['the' => 'variables', 'go' => 'here']);
```

The `display()` method is a shortcut to output the rendered template.

You can also load and render the template in one fell swoop:

```php
Listing 4.4
echo $twig->render('index.html', ['the' => 'variables', 'go' => 'here']);
```

**New in version 1.28:** The possibility to render blocks from the API was added in Twig 1.28. If a template defines blocks, they can be rendered individually via the `renderBlock()` call:

```php
Listing 4.5
echo $template->renderBlock('block_name', ['the' => 'variables', 'go' => 'here']);
```

## Environment Options

When creating a new `Twig\Environment` instance, you can pass an array of options as the constructor second argument:

```php
Listing 4.6
$twig = new Twig\Environment($loader, ['debug' => true]);
```

The following options are available:

- `debug` *boolean*
When set to `true`, the generated templates have a `__toString()` method that you can use to display the generated nodes (default to `false`).

- `charset string` (defaults to `utf-8`)
  The charset used by the templates.

- `base_template_class string` (defaults to `\Twig\Template`)
  The base template class to use for generated templates.

- `cache string or false`
  An absolute path where to store the compiled templates, or `false` to disable caching (which is the default).

- `auto_reload boolean`
  When developing with Twig, it's useful to recompile the template whenever the source code changes. If you don't provide a value for the `auto_reload` option, it will be determined automatically based on the `debug` value.

- `strict_variables boolean`
  If set to `false`, Twig will silently ignore invalid variables (variables and or attributes/methods that do not exist) and replace them with a `null` value. When set to `true`, Twig throws an exception instead (default to `false`).

- `autoescape string or boolean`
  If set to `true`, HTML auto-escaping will be enabled by default for all templates (default to `true`).
  As of Twig 1.8, you can set the escaping strategy to use (`html`, `js`, `false` to disable).
  As of Twig 1.9, you can set the escaping strategy to use (`css`, `url`, `html_attr`, or a PHP callback that takes the template name and must return the escaping strategy to use -- the callback cannot be a function name to avoid collision with built-in escaping strategies).
  As of Twig 1.17, the `filename` escaping strategy (renamed to `name` as of Twig 1.27) determines the escaping strategy to use for a template based on the template filename extension (this strategy does not incur any overhead at runtime as auto-escaping is done at compilation time.)

- `optimizations integer`
  A flag that indicates which optimizations to apply (default to `-1` -- all optimizations are enabled; set it to `0` to disable).

### Loaders

Loaders are responsible for loading templates from a resource such as the file system.

### Compilation Cache

All template loaders can cache the compiled templates on the filesystem for future reuse. It speeds up Twig a lot as templates are only compiled once; and the performance boost is even larger if you use a PHP accelerator such as OPCache. See the `cache` and `auto_reload` options of `\Twig\Environment` above for more information.

### Built-in Loaders

Here is a list of the built-in loaders:
\Twig\Loader\FilesystemLoader

New in version 1.10: The prependPath() and support for namespaces were added in Twig 1.10.
New in version 1.27: Relative paths support was added in Twig 1.27.

\Twig\Loader\FilesystemLoader 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:

Listing 4-7
```
$loader = new \Twig\Loader\FilesystemLoader($templateDir);
```

It can also look for templates in an array of directories:

Listing 4-8
```
$loader = new \Twig\Loader\FilesystemLoader([$templateDir1, $templateDir2]);
```

With such a configuration, Twig will first look for templates in $templateDir1 and if they do not exist, it will fallback to look for them in the $templateDir2.

You can add or prepend paths via the addPath() and prependPath() methods:

Listing 4-9
```
$loader->addPath($templateDir3);
$loader->prependPath($templateDir4);
```

The filesystem loader also supports namespaced templates. This allows to group your templates under different namespaces which have their own template paths.

When using the setPaths(), addPath(), and prependPath() methods, specify the namespace as the second argument (when not specified, these methods act on the "main" namespace):

Listing 4-10
```
$loader->addPath($templateDir, 'admin');
```

Namespaced templates can be accessed via the special @namespace_name/template_path notation:

Listing 4-11
```
$twig->render('@admin/index.html', []);
```

\Twig\Loader\FilesystemLoader support absolute and relative paths. Using relative paths is preferred as it makes the cache keys independent of the project root directory (for instance, it allows warming the cache from a build server where the directory might be different from the one used on production servers):

Listing 4-12
```
$loader = new \Twig\Loader\FilesystemLoader('templates', getcwd().'/..');
```

When not passing the root path as a second argument, Twig uses getcwd() for relative paths.

\Twig\Loader\ArrayLoader

\Twig\Loader\ArrayLoader loads a template from a PHP array. It is passed an array of strings bound to template names:

Listing 4-13
```
$loader = new \Twig\Loader\ArrayLoader([1
  'index.html' => 'Hello {{ name }}!',2
  ]);3
$twig = new \Twig\Environment($loader);4
echo $twig->render('index.html', ['name' => 'Fabien']);
```
This loader is very useful for unit testing. It can also be used for small projects where storing all templates in a single PHP file might make sense.

When using the Array loaders with a cache mechanism, you should know that a new cache key is generated each time a template content "changes" (the cache key being the source code of the template). If you don't want to see your cache grows out of control, you need to take care of clearing the old cache file by yourself.

When looking for a template, Twig tries each loader in turn and returns as soon as the template is found. When rendering the index.html template from the above example, Twig will load it with $loader2 but the base.html template will be loaded from $loader1.

You can also add loaders via the addLoader() method.

Create your own Loader

All loaders implement the Twig\Loader\LoaderInterface:

```
interface Twig_LoaderInterface
{
    /**
     * Gets the source code of a template, given its name.
     * @param  string $name string The name of the template to load
     * @return string The template source code
     */
    function getSource($name);
}
```
* Gets the cache key to use for the cache for a given template name.
* @param string $name string The name of the template to load
* @return string The cache key
*/
function getCacheKey($name);

/**
 * Returns true if the template is still fresh.
 *
 * @param string    $name The template name
 * @param timestamp $time The last modification time of the cached template
 */
function isFresh($name, $time);
}

The `isFresh()` method must return `true` if the current cached template is still fresh, given the last modification time, or `false` otherwise.

As of Twig 1.27, you should also implement \Twig\Loader\SourceContextLoaderInterface to avoid deprecation notices.

As of Twig 1.11.0, you can also implement \Twig\Loader\ExistsLoaderInterface to make your loader faster when used with the chain loader.

## Using Extensions

Twig extensions are packages that add new features to Twig. Register an extension via the `addExtension()` method:

```php
$twig->addExtension(new \Twig\Extension\SandboxExtension());
```

Twig comes bundled with the following extensions:

- `TwigExtensionCoreExtension`: Defines all the core features of Twig.
- `TwigExtensionDebugExtension`: Defines the `dump` function to help debug template variables.
- `TwigExtensionEscaperExtension`: Adds automatic output-escaping and the possibility to escape/unescape blocks of code.
- `TwigExtensionSandboxExtension`: Adds a sandbox mode to the default Twig environment, making it safe to evaluate untrusted code.
- `TwigExtensionProfilerExtension`: Enabled the built-in Twig profiler (as of Twig 1.18).
- `TwigExtensionOptimizerExtension`: Optimizes the node tree before compilation.
- `TwigExtensionStringLoaderExtension`: Defined the `template_from_string` function to allow loading templates from string in a template.
The Core, Escaper, and Optimizer extensions are registered by default.

**Built-in Extensions**

This section describes the features added by the built-in extensions.

Read the chapter about extending Twig to learn how to create your own extensions.

**Core Extension**

The core extension defines all the core features of Twig:

- Tags;
- Filters;
- Functions;
- Tests.

**Escaper Extension**

The escaper extension adds automatic output escaping to Twig. It defines a tag, autoescape, and a filter, raw.

When creating the escaper extension, you can switch on or off the global output escaping strategy:

```php
$escaper = new \Twig\Extension\EscaperExtension('html');
$twig->addExtension($escaper);
```

If set to html, all variables in templates are escaped (using the html escaping strategy), except those using the raw filter:

```twig
{{ article.to_html|raw }}
```

You can also change the escaping mode locally by using the autoescape tag (see the autoescape doc for the syntax used before Twig 1.8):

```twig
{% autoescape 'html' %}
  {{ var }}
  {{ var.raw }}  # var won't be escaped
  {{ var.escape }}  # var won't be double-escaped
{% endautoescape %}
```
The autoescape tag has no effect on included files.

The escaping rules are implemented as follows:

- Literals (integers, booleans, arrays, ...) used in the template directly as variables or filter arguments are never automatically escaped:

```
{{ "Twig<br />" }} {# won't be escaped #}
{{ set text = "Twig<br />" }} {# will be escaped #}
```

- Expressions which the result is a literal or a variable marked safe are never automatically escaped:

```
{{ foo ? "Twig<br />" : "<br />Twig" }} {# won't be escaped #}
{{ true ? text : "<br />Twig" }} {# will be escaped #}
{{ false ? text : "<br />Twig" }} {# won't be escaped #}
{{ foo ? text | raw : "<br />Twig" }} {# won't be escaped #}
```

- Escaping is applied before printing, after any other filter is applied:

```
{% autoescape 'js' %}
\%
{{ var | raw }} {# will be escaped #}
{{ var | escape('html') }} {# will be escaped for HTML and JavaScript #}
{{ var | escape('js') }} {# won't be double-escaped #}
\%
```

- The raw filter should only be used at the end of the filter chain:

```
{{ var | raw | upper }} {# is equivalent to {{ var|upper|escape }} #}
```

- Automatic escaping is not applied if the last filter in the chain is marked safe for the current context (e.g. html or js). escape and escape('html') are marked safe for HTML, escape('js') is marked safe for JavaScript, raw is marked safe for everything.

```
{% autoescape 'js' %}
\%
{{ var | escape('html') }} {# will be escaped for HTML and JavaScript #}
{{ var | escape('js') }} {# won't be double-escaped #}
\%
```

Note that autoescaping has some limitations as escaping is applied on expressions after evaluation. For instance, when working with concatenation, `{{ foo|raw ~ bar }}` won't give the expected result as escaping is applied on the result of the concatenation, not on the individual variables (so, the raw filter won't have any effect here).

### Sandbox Extension

The sandbox extension can be used to evaluate untrusted code. Access to unsafe attributes and methods
is prohibited. The sandbox security is managed by a policy instance. By default, Twig comes with one policy class: \Twig\Sandbox\SecurityPolicy. This class allows you to white-list some tags, filters, properties, and methods:

```php
Listing 4-25
$tags = ['if'];
$filters = ['upper'];
$methods = ['Article' => ['getTitle', 'getBody'], ];
$properties = ['Article' => ['title', 'body'], ];
$functions = ['range'];
$policy = new \Twig\Sandbox\SecurityPolicy($tags, $filters, $methods, $properties, $functions);
```

With the previous configuration, the security policy will only allow usage of the if tag, and the upper filter. Moreover, the templates will only be able to call the getTitle() and getBody() methods on Article objects, and the title and body public properties. Everything else won't be allowed and will generate a \Twig\Sandbox\SecurityError exception.

The policy object is the first argument of the sandbox constructor:

```php
Listing 4-26
$sandbox = new \Twig\Extension\SandboxExtension($policy);
$twig->addExtension($sandbox);
```

By default, the sandbox mode is disabled and should be enabled when including untrusted template code by using the sandbox tag:

```twig
Listing 4-27
{% sandbox %}
{% include 'user.html' %}
{% endsandbox %}
```

You can sandbox all templates by passing true as the second argument of the extension constructor:

```php
Listing 4-28
$sandbox = new \Twig\Extension\SandboxExtension($policy, true);
```

Profiler Extension

*New in version 1.18:* The Profile extension was added in Twig 1.18.

The profiler extension enables a profiler for Twig templates; it should only be used on your development machines as it adds some overhead:

```php
Listing 4-29
$profile = new \Twig\Profiler\Profile();
$twig->addExtension(new \Twig\Extension\ProfilerExtension($profile));
$dumper = new \Twig\Profiler\Dumper\TextDumper();
echo $dumper->dump($profile);
```

A profile contains information about time and memory consumption for template, block, and macro executions.

You can also dump the data in a Blackfire.io¹ compatible format:

```php
Listing 4-30
$dumper = new \Twig\Profiler\Dumper\BlackfireDumper();
file_put_contents('/path/to/profile.prof', $dumper->dump($profile));
```

Upload the profile to visualize it (create a free account² first):

---

1. https://blackfire.io/

---

Chapter 4: Twig for Developers | 32
### Optimizer Extension

The `optimizer` extension optimizes the node tree before compilation:

```php
Listing 4-32
$twig->addExtension(new \Twig\Extension\OptimizerExtension());
```

By default, all optimizations are turned on. You can select the ones you want to enable by passing them to the constructor:

```php
Listing 4-33
$optimizer = new \Twig\Extension\OptimizerExtension(\Twig\NodeVisitor\OptimizerNodeVisitor::OPTIMIZE_FOR);
$twig->addExtension($optimizer);
```

Twig supports the following optimizations:

- `\Twig\NodeVisitor\OptimizerNodeVisitor::OPTIMIZE_ALL`, enables all optimizations (this is the default value).
- `\Twig\NodeVisitor\OptimizerNodeVisitor::OPTIMIZE_NONE`, disables all optimizations. This reduces the compilation time, but it can increase the execution time and the consumed memory.
- `\Twig\NodeVisitor\OptimizerNodeVisitor::OPTIMIZE_FOR`, optimizes the `for` tag by removing the `loop` variable creation whenever possible.
- `\Twig\NodeVisitor\OptimizerNodeVisitor::OPTIMIZE_RAW_FILTER`, removes the `raw` filter whenever possible.
- `\Twig\NodeVisitor\OptimizerNodeVisitor::OPTIMIZE_VAR_ACCESS`, simplifies the creation and access of variables in the compiled templates whenever possible.

### Exceptions

Twig can throw exceptions:

- `\Twig\Error\Error`: The base exception for all errors.
- `\Twig\Error\SyntaxError`: Thrown to tell the user that there is a problem with the template syntax.
- `\Twig\Error\RuntimeError`: Thrown when an error occurs at runtime (when a filter does not exist for instance).
- `\Twig\Error\LoaderError`: Thrown when an error occurs during template loading.
- `\Twig\Sandbox\SecurityError`: Thrown when an unallowed tag, filter, or method is called in a sandboxed template.
This section describes how to extend Twig as of **Twig 1.12**. If you are using an older version, read the legacy chapter instead.

Twig can be extended in many ways; you can add extra tags, filters, tests, operators, global variables, and functions. You can even extend the parser itself with node visitors.

The first section of this chapter describes how to extend Twig. If you want to reuse your changes in different projects or if you want to share them with others, you should then create an extension as described in the following section.

When extending Twig without creating an extension, Twig won’t be able to recompile your templates when the PHP code is updated. To see your changes in real-time, either disable template caching or package your code into an extension (see the next section of this chapter).

Before extending Twig, you must understand the differences between all the different possible extension points and when to use them.

First, remember that Twig has two main language constructs:
• {{ }}: used to print the result of an expression evaluation;
• {% %}: used to execute statements.

To understand why Twig exposes so many extension points, let’s see how to implement a *Lorem ipsum* generator (it needs to know the number of words to generate).

You can use a *lipsum* tag:

```
{%- %lipsum 40%-%}
```

That works, but using a tag for *lipsum* is not a good idea for at least three main reasons:

- *lipsum* is not a language construct;
- The tag outputs something;
- The tag is not flexible as you cannot use it in an expression:

```
{{ 'some text' ~ %lipsum 40% ~ 'some more text'}}
```

In fact, you rarely need to create tags; and that’s good news because tags are the most complex extension point.

Now, let’s use a *lipsum* filter:

```
{{ 40|lipsum }}
```

Again, it works. But a filter should transform the passed value to something else. Here, we use the value to indicate the number of words to generate (so, 40 is an argument of the filter, not the value we want to transform).

Next, let’s use a *lipsum* function:

```
{{ lipsum(40) }}
```

Here we go. For this specific example, the creation of a function is the extension point to use. And you can use it anywhere an expression is accepted:

```
{{ 'some text' ~ lipsum(40) ~ 'some more text'}}
{% set lipsum = lipsum(40) %}
```

Lastly, you can also use a *global* object with a method able to generate lorem ipsum text:

```
{{ text.lipsum(40) }}
```

As a rule of thumb, use functions for frequently used features and global objects for everything else.

Keep in mind the following when you want to extend Twig:

<table>
<thead>
<tr>
<th>What?</th>
<th>Implementation difficulty?</th>
<th>How often?</th>
<th>When?</th>
</tr>
</thead>
<tbody>
<tr>
<td>macro</td>
<td>simple</td>
<td>frequent</td>
<td>Content generation</td>
</tr>
<tr>
<td>global</td>
<td>simple</td>
<td>frequent</td>
<td>Helper object</td>
</tr>
<tr>
<td>function</td>
<td>simple</td>
<td>frequent</td>
<td>Content generation</td>
</tr>
<tr>
<td>filter</td>
<td>simple</td>
<td>frequent</td>
<td>Value transformation</td>
</tr>
<tr>
<td>tag</td>
<td>complex</td>
<td>rare</td>
<td>DSL language construct</td>
</tr>
<tr>
<td>What?</td>
<td>Implementation difficulty?</td>
<td>How often?</td>
<td>When?</td>
</tr>
<tr>
<td>----------</td>
<td>----------------------------</td>
<td>------------</td>
<td>------------</td>
</tr>
<tr>
<td>test</td>
<td>simple</td>
<td>rare</td>
<td>Boolean decision</td>
</tr>
<tr>
<td>operator</td>
<td>simple</td>
<td>rare</td>
<td>Values transformation</td>
</tr>
</tbody>
</table>

**Globals**

A global variable is like any other template variable, except that it’s available in all templates and macros:

```php
$twig = new \Twig\Environment($loader);
$twig->addGlobal('text', new Text());
```

You can then use the `text` variable anywhere in a template:

```twig
{{ text.lipsum(40) }}
```

**Filters**

Creating a filter consists of associating a name with a PHP callable:

```php
// an anonymous function
$filter = new \Twig\TwigFilter('rot13', function ($string) {
    return str_rot13($string);
});

// or a simple PHP function
$filter = new \Twig\TwigFilter('rot13', 'str_rot13');

// or a class static method
$filter = new \Twig\TwigFilter('rot13', ['SomeClass', 'rot13Filter']);

// or a class method
$filter = new \Twig\TwigFilter('rot13', [$this, 'rot13Filter']);
```

The first argument passed to the `\Twig\TwigFilter` constructor is the name of the filter you will use in templates and the second one is the PHP callable to associate with it.

Then, add the filter to the Twig environment:

```php
$twig = new \Twig\Environment($loader);
$twig->addFilter($filter);
```

And here is how to use it in a template:

```twig
{{ 'Twig'|rot13 }}
```

When called by Twig, the PHP callable receives the left side of the filter (before the pipe |) as the first argument and the extra arguments passed to the filter (within parentheses ( )) as extra arguments.

For instance, the following code:

```twig
{{ 'TWIG'|lower }}
```
is compiled to something like the following:

```php
<?php
echo strtolower('TWIG') ?>

<?php
echo twig_date_format_filter($now, 'd/m/Y') ?>
```

The `Twig\TwigFilter` class takes an array of options as its last argument:

```php
$filter = new \Twig\TwigFilter('rot13', 'str_rot13', $options);
```

Environment-aware Filters

If you want to access the current environment instance in your filter, set the `needs_environment` option to `true`; Twig will pass the current environment as the first argument to the filter call:

```php
$filter = new \Twig\TwigFilter('rot13', function ($env, $string) {
    // get the current charset for instance
    $charset = $env->getCharset();
    return str_rot13($string);
}, ['needs_environment' => true]);
```

Context-aware Filters

If you want to access the current context in your filter, set the `needs_context` option to `true`; Twig will pass the current context as the first argument to the filter call (or the second one if `needs_environment` is also set to `true`):

```php
$filter = new \Twig\TwigFilter('rot13', function ($context, $string) {
    // ...
}, ['needs_context' => true]);
```

Automatic Escaping

If automatic escaping is enabled, the output of the filter may be escaped before printing. If your filter acts as an escaper (or explicitly outputs HTML or JavaScript code), you will want the raw output to be printed. In such a case, set the `is_safe` option:

```php
$filter = new \Twig\TwigFilter('nl2br', 'nl2br', ['is_safe' => ['html']]);
```

Some filters may need to work on input that is already escaped or safe, for example when adding (safe) HTML tags to originally unsafe output. In such a case, set the `pre_escape` option to escape the input data before it is run through your filter:

```php
$filter = new \Twig\TwigFilter('somefilter', 'somefilter', ['pre_escape' => 'html', 'is_safe' => ['html']]);
```

Variadic Filters

*New in version 1.19:* Support for variadic filters was added in Twig 1.19.

When a filter should accept an arbitrary number of arguments, set the `is_variadic` option to `true`; Twig will pass the extra arguments as the last argument to the filter call as an array:
Dynamic Filters

A filter name containing the special * character is a dynamic filter and the * part will match any string:

```php
$filter = new \Twig\TwigFilter('*_path', function ($name, $arguments) {
// ...
});
```

The following filters are matched by the above defined dynamic filter:

- product_path
- category_path

A dynamic filter can define more than one dynamic parts:

```php
$filter = new \Twig\TwigFilter('*_path_*', function ($name, $suffix, $arguments) {
// ...
});
```

The filter receives all dynamic part values before the normal filter arguments, but after the environment and the context. For instance, a call to `'*foo'|a_path_b()` will result in the following arguments to be passed to the filter: (`'a'`, `'b'`, `'foo'`).

Deprecated Filters

*New in version 1.21:* Support for deprecated filters was added in Twig 1.21.

You can mark a filter as being deprecated by setting the `deprecated` option to `true`. You can also give an alternative filter that replaces the deprecated one when that makes sense:

```php
$filter = new \Twig\TwigFilter('obsolete', function () {
// ...
}, ['deprecated' => true, 'alternative' => 'new_one']);
```

When a filter is deprecated, Twig emits a deprecation notice when compiling a template using it. See Displaying Deprecation Notices for more information.

Functions

Functions are defined in the exact same way as filters, but you need to create an instance of `\Twig\TwigFunction`:

```php
$twig = new \Twig\Environment($loader);
$function = new \Twig\TwigFunction('function_name', function () {
// ...
});
$twig->addFunction($function);
```

Functions support the same features as filters, except for the `pre_escape` and `preserves_safety` options.
Tests

Tests are defined in the exact same way as filters and functions, but you need to create an instance of \Twig\TwigTest:

```php
$twig = new \Twig\Environment($loader);
$test = new \Twig\TwigTest('test_name', function () {
    // ...
});
$twig->addTest($test);
```

Tests allow you to create custom application specific logic for evaluating boolean conditions. As a simple example, let's create a Twig test that checks if objects are 'red':

```php
$twig = new \Twig\Environment($loader);
$test = new \Twig\TwigTest('red', function ($value) {
    if (isset($value->color) && $value->color == 'red') {
        return true;
    } else if (isset($value->paint) && $value->paint == 'red') {
        return true;
    }
    return false;
});
$twig->addTest($test);
```

Test functions must always return `true/false`.

When creating tests you can use the `node_class` option to provide custom test compilation. This is useful if your test can be compiled into PHP primitives. This is used by many of the tests built into Twig:

```php
$twig = new \Twig\Environment($loader);
$test = new \Twig\TwigTest('odd', null, [ 'node_class' => '\Twig\Node\Expression\Test\OddTest' ]);;
$twig->addTest($test);
```

```php
class Twig_Node_Expression_Test_Odd extends \Twig\Node\Expression\TestExpression
{
    public function compile(\Twig\Compiler $compiler)
    {
        $compiler->raw('(');
        $compiler->subcompile($this->getNode('node'))
            ->raw(' % 2 == 1')
            ->raw(')');
    }
}
```

The above example shows how you can create tests that use a node class. The node class has access to one sub-node called `node`. This sub-node contains the value that is being tested. When the `odd` filter is used in code such as:

```twig
{% if my_value is odd %}
```

The `node` sub-node will contain an expression of `my_value`. Node-based tests also have access to the `arguments` node. This node will contain the various other arguments that have been provided to your test.

**New in version 1.36:** Dynamic tests support was added in Twig 1.36.

If you want to pass a variable number of positional or named arguments to the test, set the
**is_variadic** option to **true**. Tests support dynamic names (see dynamic filters for the syntax).

## Tags

One of the most exciting features of a template engine like Twig is the possibility to define new **language constructs**. This is also the most complex feature as you need to understand how Twig's internals work.

Most of the time though, a tag is not needed:

- If your tag generates some output, use a **function** instead.
- If your tag modifies some content and returns it, use a **filter** instead.

For instance, if you want to create a tag that converts a Markdown formatted text to HTML, create a **markdown** filter instead:

```twig
{{ '**markdown** text'|markdown }}
```

If you want use this filter on large amounts of text, wrap it with the apply tag:

```twig
{% apply markdown %}
Title
=====
Much better than creating a tag as you can **compose** filters.
{% endapply %}
```

The **apply** tag was introduced in Twig 1.40; use the **filter** tag with previous versions.

- If your tag does not output anything, but only exists because of a side effect, create a **function** that returns nothing and call it via the **filter** tag.

For instance, if you want to create a tag that logs text, create a **log** function instead and call it via the do tag:

```twig
{% do log('Log some things') %}
```

If you still want to create a tag for a new language construct, great!

Let’s create a **set** tag that allows the definition of simple variables from within a template. The tag can be used like follows:

```twig
{% set name = "value" %}
{{ name }}
```

# should output value #
The `set` tag is part of the Core extension and as such is always available. The built-in version is slightly more powerful and supports multiple assignments by default.

Three steps are needed to define a new tag:

- Defining a Token Parser class (responsible for parsing the template code);
- Defining a Node class (responsible for converting the parsed code to PHP);
- Registering the tag.

### Registering a new tag

Add a tag by calling the `addTokenParser` method on the `Twig\Environment` instance:

```php
$twig = new Twig\Environment($loader);
$twig->addTokenParser(new Project_Set_TokenParser());
```

### Defining a Token Parser

Now, let's see the actual code of this class:

```php
class Project_Set_TokenParser extends Twig\TokenParser\AbstractTokenParser {
    public function parse(Twig\Token $token) {
        $parser = $this->parser;
        $stream = $parser->getStream();
        $name = $stream->expect(Twig\Token::NAME_TYPE)->getValue();
        $stream->expect(Twig\Token::OPERATOR_TYPE, '=');
        $value = $parser->getExpressionParser()->parseExpression();
        $stream->expect(Twig\Token::BLOCK_END_TYPE);
        return new Project_Set_Node($name, $value, $token->getLine(), $this->getTag());
    }

    public function getTag() {
        return 'set';
    }
}
```

The `getTag()` method must return the tag we want to parse, here `set`.

The `parse()` method is invoked whenever the parser encounters a `set` tag. It should return a `Twig\Node\Node` instance that represents the node (the `Project_Set_Node` calls creating is explained in the next section).

The parsing process is simplified thanks to a bunch of methods you can call from the token stream ($this->parser->getStream()):

- `getCurrent()`: Gets the current token in the stream.
- `next()`: Moves to the next token in the stream, *but returns the old one*.
- `test($type), test($value) or test($type, $value)`: Determines whether the current token is of a particular type or value (or both). The value may be an array of several possible values.
- `expect($type[, $value[, $message]])`: If the current token isn’t of the given type/value a syntax error is thrown. Otherwise, if the type and value are correct, the token is returned and the stream moves to
the next token.

- `look()`: Looks at the next token without consuming it.

Parsing expressions is done by calling the `parseExpression()` like we did for the `set` tag.

Reading the existing `TokenParser` classes is the best way to learn all the nitty-gritty details of the parsing process.

### Defining a Node

The `Project_Set_Node` class itself is quite short:

```php
Listing 5-34
class Project_Set_Node extends \Twig\Node\Node {
    public function __construct($name, \Twig\Node\Expression\AbstractExpression $value, $line, $tag = null) {
        parent::__construct(['value' => $value], ['name' => $name], $line, $tag);
    }

    public function compile(\Twig\Compiler $compiler) {
        $compiler->addDebugInfo($this)->write('$context[' . $this->getAttribute('name') . ']' . ' = ')
            ->subcompile($this->getNode('value'))->raw(';\n')
            ;
    }
}
```

The compiler implements a fluid interface and provides methods that helps the developer generate beautiful and readable PHP code:

- `subcompile()`: Compiles a node.
- `raw()`: Writes the given string as is.
- `write()`: Writes the given string by adding indentation at the beginning of each line.
- `string()`: Writes a quoted string.
- `repr()`: Writes a PHP representation of a given value (see `\Twig\Node\ForNode` for a usage example).
- `addDebugInfo()`: Adds the line of the original template file related to the current node as a comment.
- `indent()`: Indents the generated code (see `\Twig\Node\BlockNode` for a usage example).
- `outdent()`: Outdents the generated code (see `\Twig\Node\BlockNode` for a usage example).

### Creating an Extension

The main motivation for writing an extension is to move often used code into a reusable class like adding support for internationalization. An extension can define tags, filters, tests, operators, functions, and node visitors.

Most of the time, it is useful to create a single extension for your project, to host all the specific tags and filters you want to add to Twig.
When packaging your code into an extension, Twig is smart enough to recompile your templates whenever you make a change to it (when `auto_reload` is enabled).

An extension is a class that implements the following interface:

```php
interface Twig_ExtensionInterface
{
    /**
     * Initializes the runtime environment.
     * This is where you can load some file that contains filter functions for instance.
     * @deprecated since 1.23 (to be removed in 2.0), implement \Twig\Extension\InitRuntimeInterface instead
     */
    function initRuntime(Twig\Environment $environment);

    /**
     * Returns the token parser instances to add to the existing list.
     * @return (Twig_TokenParserInterface|Twig_TokenParserBrokerInterface)[]
     */
    function getTokenParsers();

    /**
     * Returns the node visitor instances to add to the existing list.
     * @return \Twig\NodeVisitor\NodeVisitorInterface[]
     */
    function getNodeVisitors();

    /**
     * Returns a list of filters to add to the existing list.
     * @return \Twig\TwigFilter[]
     */
    function getFilters();

    /**
     * Returns a list of tests to add to the existing list.
     * @return \Twig\TwigTest[]
     */
    function getTests();

    /**
     * Returns a list of functions to add to the existing list.
     * @return \Twig\TwigFunction[]
     */
    function getFunctions();

    /**
     * Returns a list of operators to add to the existing list.
     * @return array<array> First array of unary operators, second array of binary operators
     */
    function getOperators();

    /**
     * Returns a list of global variables to add to the existing list.
     * @return array An array of global variables
     */
}
```
To keep your extension class clean and lean, inherit from the built-in \Twig\Extension\AbstractExtension class instead of implementing the interface as it provides empty implementations for all methods:

```php
Listing 5.36
class ProjectTwig_Extension extends \Twig\Extension\AbstractExtension
{
}
```

This extension does nothing for now. We will customize it in the next sections.

Prior to Twig 1.26, you must implement the `getName()` method which must return a unique identifier for the extension.

You can save your extension anywhere on the filesystem, as all extensions must be registered explicitly to be available in your templates.

You can register an extension by using the `addExtension()` method on your main `Environment` object:

```php
Listing 5.37
$twig = new \Twig\Environment;$loader);
$twig->addExtension(now ProjectTwig_Extension());
```

The Twig core extensions are great examples of how extensions work.

### Globals

Global variables can be registered in an extension via the `getGlobals()` method:

```php
Listing 5.38
1 class ProjectTwig_Extension extends \Twig\Extension\AbstractExtension implements \Twig\Extension\GlobalsInterface
2 {
3     public function getGlobals()
4     {
```
return [
    'text' => new Text(),
];

// ...

Functions

Functions can be registered in an extension via the `getFunctions()` method:

```php
Listing 5.39
class Project_Twig_Extension extends \Twig\Extension\AbstractExtension
{
    public function getFunctions()
    {
        return [
            new \Twig\TwigFunction('lipsum', 'generate_lipsum'),
        ];
    }
    // ...
}
```

Filters

To add a filter to an extension, you need to override the `getFilters()` method. This method must return an array of filters to add to the Twig environment:

```php
Listing 5.40
class Project_Twig_Extension extends \Twig\Extension\AbstractExtension
{
    public function getFilters()
    {
        return [
            new \Twig\TwigFilter('rot13', 'str_rot13'),
        ];
    }
    // ...
}
```

Tags

Adding a tag in an extension can be done by overriding the `getTokenParsers()` method. This method must return an array of tags to add to the Twig environment:

```php
Listing 5.41
class Project_Twig_Extension extends \Twig\Extension\AbstractExtension
{
    public function getTokenParsers()
    {
        return [new Project_Set_TokenParser()];
    }
    // ...
}
```

In the above code, we have added a single new tag, defined by the `Project_Set_TokenParser` class. The `Project_Set_TokenParser` class is responsible for parsing the tag and compiling it to PHP.
Operators

The `getOperators()` method lets you add new operators. Here is how to add the `!`, `||`, and `&&` operators:

```php
Listing 5-42
class Project_Twig_Extension extends \Twig\Extension\AbstractExtension
{
    public function getOperators()
    {
        return [
            '!' => ['precedence' => 50, 'class' => '\Twig\Node\Expression\Unary\NotUnary'],
            '||' => ['precedence' => 10, 'class' => '\Twig\Node\Expression\Binary\OrBinary',
                      'associativity' => \Twig\ExpressionParser::OPERATOR_LEFT],
            '&&' => ['precedence' => 15, 'class' => '\Twig\Node\Expression\Binary\AndBinary',
                      'associativity' => \Twig\ExpressionParser::OPERATOR_LEFT],
        ];
    }
}
```

Tests

The `getTests()` method lets you add new test functions:

```php
Listing 5-43
class Project_Twig_Extension extends \Twig\Extension\AbstractExtension
{
    public function getTests()
    {
        return [
            new \Twig\TwigTest('even', 'twig_test_even'),
        ];
    }
}
```

Definition vs Runtime

Twig filters, functions, and tests runtime implementations can be defined as any valid PHP callable:

- **functions/static methods**: Simple to implement and fast (used by all Twig core extensions); but it is hard for the runtime to depend on external objects;
- **closures**: Simple to implement;
- **object methods**: More flexible and required if your runtime code depends on external objects.

The simplest way to use methods is to define them on the extension itself:

```php
Listing 5-44
class Project_Twig_Extension extends \Twig\Extension\AbstractExtension
{
    private $rot13Provider;
    public function __construct($rot13Provider)
    {
        $this->rot13Provider = $rot13Provider;
    }
    public function getFunctions()
    {
        return [
        ];
    }
}
```
This is very convenient but not recommended as it makes template compilation depend on runtime dependencies even if they are not needed (think for instance as a dependency that connects to a database engine).

As of Twig 1.26, you can decouple the extension definitions from their runtime implementations by registering a `\Twig\RuntimeLoader\RuntimeLoaderInterface` instance on the environment that knows how to instantiate such runtime classes (runtime classes must be autoload-able):

```php
Listing 5.45
class RuntimeLoader implements \Twig\RuntimeLoader\RuntimeLoaderInterface
{
    public function load($class)
    {
        // implement the logic to create an instance of $class
        // and inject its dependencies
        // most of the time, it means using your dependency injection container
        if ('Project_Twig_RuntimeExtension' === $class) {
            return new $class(new Rot13Provider());
        } else {
            // ...
        }
    }
}
	$twig->addRuntimeLoader(new RuntimeLoader());
```

As of Twig 1.32, Twig comes with a PSR-11 compatible runtime loader (`\Twig\RuntimeLoader\ContainerRuntimeLoader`).

It is now possible to move the runtime logic to a new `Project_Twig_RuntimeExtension` class and use it directly in the extension:

```php
Listing 5.46
class Project_Twig_RuntimeExtension
{
    private $rot13Provider;
    public function __construct($rot13Provider)
    {
        $this->rot13Provider = $rot13Provider;
    }
    public function rot13($value)
    {
        return $this->rot13Provider->rot13($value);
    }
}
class Project_Twig_Extension extends \Twig\Extension\AbstractExtension
{
```
Testing an Extension

Functional Tests

You can create functional tests for extensions by creating the following file structure in your test directory:

```
Fixtures/
  filters/
    foo.test
    bar.test
  functions/
    foo.test
    bar.test
  tags/
    foo.test
    bar.test
IntegrationTest.php
```

The `IntegrationTest.php` file should look like this:

```
class Project_Tests_IntegrationTest extends \Twig\Test\IntegrationTestCase
{
  public function getExtensions()
  {
    return [
      new \Twig\TwigFunction('rot13', ['Project_Twig_RuntimeExtension', 'rot13']),
      new \Twig\TwigFunction('rot13', 'Project_Twig_RuntimeExtension::rot13'),
    ];
  }

  public function getFixturesDir()
  {
    return __DIR__.'/Fixtures/';
  }
}
```

Fixtures examples can be found within the Twig repository `tests/Twig/Fixtures` directory.

Node Tests

Testing the node visitors can be complex, so extend your test cases from `\Twig\Test\NodeTestCare`. Examples can be found in the Twig repository `tests/Twig/Node` directory.

---

1. https://github.com/twigphp/Twig/tree/1.x/tests/Fixtures
2. https://github.com/twigphp/Twig/tree/1.x/tests/Node
Twig Internals

Chapter 6

Twig Internals

Twig is very extensible and you can hack it. Keep in mind that you should probably try to create an extension before hacking the core, as most features and enhancements can be handled with extensions. This chapter is also useful for people who want to understand how Twig works under the hood.

How does Twig work?

The rendering of a Twig template can be summarized into four key steps:

- **Load** the template: If the template is already compiled, load it and go to the *evaluation* step, otherwise:
  - First, the **lexer** tokenizes the template source code into small pieces for easier processing;
  - Then, the **parser** converts the token stream into a meaningful tree of nodes (the Abstract Syntax Tree);
  - Finally, the **compiler** transforms the AST into PHP code.
- **Evaluate** the template: It means calling the `display()` method of the compiled template and passing it the context.

The Lexer

The lexer tokenizes a template source code into a token stream (each token is an instance of `Twig\Token`, and the stream is an instance of `Twig\TokenStream`). The default lexer recognizes 13 different token types:

- `Twig\Token::BLOCK_START_TYPE`, `Twig\Token::BLOCK_END_TYPE`: Delimiters for blocks `(% %)`
- `Twig\Token::VAR_START_TYPE`, `Twig\Token::VAR_END_TYPE`: Delimiters for variables `{{ }}`
- `Twig\Token::TEXT_TYPE`: A text outside an expression;
- `Twig\Token::NAME_TYPE`: A name in an expression;
- `Twig\Token::NUMBER_TYPE`: A number in an expression;
- `Twig\Token::STRING_TYPE`: A string in an expression;
- `Twig\Token::OPERATOR_TYPE`: An operator;
• \Twig\Token::PUNCTUATION_TYPE: A punctuation sign;
• \Twig\Token::INTERPOLATION_START_TYPE, \Twig\Token::INTERPOLATION_END_TYPE (as of Twig 1.5): Delimiters for string interpolation;
• \Twig\Token::EOF_TYPE: Ends of template.

You can manually convert a source code into a token stream by calling the `tokenize()` method of an environment:

```php
Listing 6-1

$stream = $twig->tokenize(new \Twig\Source($source, $identifier));
```

New in version 1.27: `\Twig\Source` was introduced in version 1.27, pass the source and the identifier directly on previous versions.

As the stream has a `__toString()` method, you can have a textual representation of it by echoing the object:

```php
Listing 6-2
echo $stream."
```

Here is the output for the `Hello {{ name }}` template:

```twig
Listing 6-3
  TEXT_TYPE(Hello )
  VAR_START_TYPE()
  NAME_TYPE(name)
  VAR_END_TYPE()
  EOF_TYPE()
```

The default lexer (`\Twig\Lexer`) can be changed by calling the `setLexer()` method:

```php
Listing 6-4

$twig->setLexer($lexer);
```

The Parser

The parser converts the token stream into an AST (Abstract Syntax Tree), or a node tree (an instance of `\Twig\Node\ModuleNode`). The core extension defines the basic nodes like: `for`, `if`, ... and the expression nodes.

You can manually convert a token stream into a node tree by calling the `parse()` method of an environment:

```php
Listing 6-5

$nodes = $twig->parse($stream);
```

Echoing the node object gives you a nice representation of the tree:

```php
Listing 6-6
echo $nodes."
```

Here is the output for the `Hello {{ name }}` template:

```twig
Listing 6-7
  \Twig\Node\ModuleNode(
  \Twig\Node\TextNode(Hello )
  \Twig\Node\PrintNode(
    \Twig\Node\Expression\NameExpression(name)
  )
)
```
The default parser (`\Twig\TokenParser\AbstractTokenParser`) can be changed by calling the `setParser()` method:

```
$twig->setParser($parser);
```

---

The Compiler

The last step is done by the compiler. It takes a node tree as an input and generates PHP code usable for runtime execution of the template.

You can manually compile a node tree to PHP code with the `compile()` method of an environment:

```
$php = $twig->compile($nodes);
```

The generated template for a `Hello {{ name }}` template reads as follows (the actual output can differ depending on the version of Twig you are using):

```
/* Hello {{ name }} */

class __TwigTemplate_1121b6f109fe93ebe8c6e22e3712bceb extends \Twig\Template
{
protected function doDisplay(array $context, array $blocks = [])
{
// line 1
    echo "Hello ";
    echo twig_escape_filter($this->env, (isset($context["name"])) ? $context["name"] : null), "html",
null, true);
}

// some more code
```

The default compiler (`\Twig\Compiler`) can be changed by calling the `setCompiler()` method:

```
$twig->setCompiler($compiler);
```

---
Chapter 7

Deprecated Features

This document lists all deprecated features in Twig. Deprecated features are kept for backward compatibility and removed in the next major release (a feature that was deprecated in Twig 1.x is removed in Twig 2.0).

Deprecation Notices

As of Twig 1.21, Twig generates deprecation notices when a template uses deprecated features. See Displaying Deprecation Notices for more information.

Macros

As of Twig 2.0, macros imported in a file are not available in child templates anymore (via an include call for instance). You need to import macros explicitly in each file where you are using them.

Token Parsers

- As of Twig 1.x, the token parser broker sub-system is deprecated. The following class and interface will be removed in 2.0:
  - Twig_TokenParserBrokerInterface
  - Twig_TokenParserBroker
- As of Twig 1.27, \Twig\Parser::getFilename() is deprecated. From a token parser, use $this->parser->getStream()->getSourceContext()->getPath() instead.
- As of Twig 1.27, \Twig\Parser::getEnvironment() is deprecated.

Extensions

- As of Twig 1.x, the ability to remove an extension is deprecated and the \Twig\
Environment::removeExtension() method will be removed in 2.0.

- As of Twig 1.23, the \Twig\Extension\ExtensionInterface::initRuntime() method is deprecated. You have two options to avoid the deprecation notice: if you implement this method to store the environment for your custom filters, functions, or tests, use the needs_environment option instead; if you have more complex needs, explicitly implement \Twig\Extension\InitRuntimeInterface (not recommended).
- As of Twig 1.23, the \Twig\Extension\ExtensionInterface::getGlobals() method is deprecated. Implement \Twig\Extension\GlobalsInterface to avoid deprecation notices.
- As of Twig 1.26, the \Twig\Extension\ExtensionInterface::getName() method is deprecated and it is not used internally anymore.

**PEAR**

PEAR support has been discontinued in Twig 1.15.1, and no PEAR packages are provided anymore. Use Composer instead.

**Filters**

- As of Twig 1.x, use \Twig\TwigFilter to add a filter. The following classes and interfaces will be removed in 2.0:
  - Twig_FilterInterface
  - Twig_FilterCallableInterface
  - Twig_Filter
  - Twig_Filter_Function
  - Twig_Filter_Method
  - Twig_Filter_Node
- As of Twig 2.x, the Twig_SimpleFilter class is deprecated and will be removed in Twig 3.x (use \Twig\TwigFilter instead). In Twig 2.x, Twig_SimpleFilter is just an alias for \Twig\TwigFilter.

**Functions**

- As of Twig 1.x, use \Twig\TwigFunction to add a function. The following classes and interfaces will be removed in 2.0:
  - Twig_FunctionInterface
  - Twig_FunctionCallableInterface
  - Twig_Function
  - Twig_Function_Function
  - Twig_Function_Method
  - Twig_Function_Node
- As of Twig 2.x, the Twig_SimpleFunction class is deprecated and will be removed in Twig 3.x (use \Twig\TwigFunction instead). In Twig 2.x, Twig_SimpleFunction is just an alias for \Twig\TwigFunction.

**Tests**

- As of Twig 1.x, use \Twig\TwigTest to add a test. The following classes and interfaces will be removed in 2.0:
  - Twig_TestInterface
As of Twig 2.x, the `Twig_SimpleTest` class is deprecated and will be removed in Twig 3.x (use `Twig\TwigTest` instead). In Twig 2.x, `Twig_SimpleTest` is just an alias for `Twig\TwigTest`.

The `sameas` and `divisibleby` tests are deprecated in favor of `same as` and `divisible by` respectively.

**Tags**

- As of Twig 1.x, the `raw` tag is deprecated. You should use `verbatim` instead.

**Nodes**

- As of Twig 1.x, `Node::toXml()` is deprecated and will be removed in Twig 2.0.
- As of Twig 1.26, `Node::$nodes` should only contain `Twig\Node\Node` instances, storing a null value is deprecated and won’t be possible in Twig 2.x.
- As of Twig 1.27, the `filename` attribute on `Twig\Node\ModuleNode` is deprecated. Use `getName()` instead.
- As of Twig 1.27, the `Twig\Node\Node::getFilename()`/`Twig\Node\Node::getLine()` methods are deprecated, use `Twig\Node\Node::getTemplateName()`/`Twig\Node\Node::getTemplateLine()` instead.

**Interfaces**

- As of Twig 2.x, the following interfaces are deprecated and empty (they will be removed in Twig 3.0):
  - `Twig_CompilerInterface` (use `Twig\Compiler` instead)
  - `Twig_LexerInterface` (use `Twig\Lexer` instead)
  - `Twig_NodeInterface` (use `Twig\Node\Node` instead)
  - `Twig_ParserInterface` (use `Twig\Parser` instead)
  - `Twig\Loader\ExistsLoaderInterface` (merged with `Twig\Loader\LoaderInterface`)
  - `Twig\Loader\SourceContextLoaderInterface` (merged with `Twig\Loader\LoaderInterface`)
  - `Twig_TemplateInterface` (use `Twig\Template` instead, and use those constants `TwigTemplate::ANY_CALL`, `TwigTemplate::ARRAY_CALL`, `TwigTemplate::METHOD_CALL`)

**Compiler**

- As of Twig 1.26, the `Twig\Compiler::getFilename()` has been deprecated. You should not use it anyway as its values is not reliable.
- As of Twig 1.27, the `Twig\Compiler::addIndentation()` has been deprecated. Use `Twig\Compiler::write('')` instead.

**Loaders**

- As of Twig 1.x, `Twig\Loader\String` is deprecated and will be removed in 2.0. You can render a string via `Twig\Environment::createTemplate()`.
- As of Twig 1.27, `Twig\Loader\LoaderInterface::getSource()` is deprecated. Implement `Twig\Loader\SourceContextLoaderInterface` instead and use `getSourceContext()`.
Node Visitors

- Because of the removal of \Twig\NodeInterface in 2.0, you need to extend \Twig\NodeVisitor\AbstractNodeVisitor instead of implementing \Twig\NodeVisitor\NodeVisitorInterface directly to make your node visitors compatible with both Twig 1.x and 2.x.

Globals

- As of Twig 2.x, the ability to register a global variable after the runtime or the extensions have been initialized is not possible anymore (but changing the value of an already registered global is possible).
- As of Twig 1.x, using the \_self global variable to get access to the current \Twig\Template instance is deprecated; most usages only need the current template name, which will continue to work in Twig 2.0. In Twig 2.0, \_self returns the current template name instead of the current \Twig\Template instance. If you are using {{ \_self.templateName }}, just replace it with {{ \_self }}.

Miscellaneous

- As of Twig 1.x, \Twig\Environment::clearTemplateCache(), \Twig\Environment::writeCacheFile(), \Twig\Environment::clearCacheFiles(), \Twig\Environment::getCacheFilename(), \Twig\Environment::getTemplateClassPrefix(), \Twig\Environment::getLexer(), \Twig\Environment::getParser(), and \Twig\Environment::getCompiler() are deprecated and will be removed in 2.0.
- As of Twig 1.x, \Twig\Template::getEnvironment() and \Twig_TemplateInterface::getEnvironment() are deprecated and will be removed in 2.0.
- As of Twig 1.21, setting the environment option autoescape to true is deprecated and will be removed in 2.0. Use "html" instead.
- As of Twig 1.27, \Twig\Error\Error::getTemplateFile() and \Twig\Error\Error::setTemplateFile() are deprecated. Use \Twig\Error\Error::getTemplateName() and \Twig\Error\Error::setTemplateName() instead.
- As of Twig 1.27, \Twig\Template::getSource() is deprecated. Use \Twig\Template::getSourceContext() instead.
- As of Twig 1.27, \Twig\Parser::addHandler() and \Twig\Parser::addNodeVisitor() are deprecated and will be removed in 2.0.
- As of Twig 1.29, some classes are marked as being final via the @final annotation. Those classes will be marked as final in 2.0.
Displaying Deprecation Notices

*New in version 1.21:* This works as of Twig 1.21.

Deprecated features generate deprecation notices (via a call to the `trigger_error()` PHP function). By default, they are silenced and never displayed nor logged.

To remove all deprecated feature usages from your templates, write and run a script along the lines of the following:

```php
require_once __DIR__ . '/vendor/autoload.php';
$twig = create_your_twig_env();
$deprecations = new \Twig\Util\DeprecationCollector($twig);
print_r($deprecations->collectDir(__DIR__ . '/templates'));
```

The `collectDir()` method compiles all templates found in a directory, catches deprecation notices, and return them.

If your templates are not stored on the filesystem, use the `collect()` method instead. `collect()` takes a `Traversable` which must return template names as keys and template contents as values (as done by `\Twig\Util\TemplateDirIterator`).

However, this code won't find all deprecations (like using deprecated some Twig classes). To catch all notices, register a custom error handler like the one below:

```php
$deprecations = [];
```
set_error_handler(function ($type, $msg) use ($deprecations) {
    if (E_USER_DEPRECATED === $type) {
        $deprecations[] = $msg;
    }
});

// run your application
print_r($deprecations);

Note that most deprecation notices are triggered during compilation, so they won’t be generated when templates are already cached.

If you want to manage the deprecation notices from your PHPUnit tests, have a look at the symfony/phpunit-bridge package, which eases the process.

Making a Layout conditional

Working with Ajax means that the same content is sometimes displayed as is, and sometimes decorated with a layout. As Twig layout template names can be any valid expression, you can pass a variable that evaluates to true when the request is made via Ajax and choose the layout accordingly:

```twig
{% extends request.ajax ? "base_ajax.html" : "base.html" %}
{% block content %}
    This is the content to be displayed.
{% endblock %}
```

Making an Include dynamic

When including a template, its name does not need to be a string. For instance, the name can depend on the value of a variable:

```twig
{% include var ~ '_foo.html' %}
```

If `var` evaluates to `index`, the `index_foo.html` template will be rendered.

As a matter of fact, the template name can be any valid expression, such as the following:

```twig
{% include var|default('index') ~ '_foo.html' %}
```

Overriding a Template that also extends itself

A template can be customized in two different ways:

1. https://github.com/symfony/phpunit-bridge
• **Inheritance**: A template *extends* a parent template and overrides some blocks;
• **Replacement**: If you use the filesystem loader, Twig loads the first template it finds in a list of configured directories; a template found in a directory replaces another one from a directory further in the list.

But how do you combine both: replace a template that also extends itself (aka a template in a directory further in the list)?

Let's say that your templates are loaded from both `.../templates/mysite` and `.../templates/default` in this order. The `page.twig` template, stored in `.../templates/default` reads as follows:

```
# page.twig#
{% extends "layout.twig" %}
{% block content %}
{% endblock %}
```

You can replace this template by putting a file with the same name in `.../templates/mysite`. And if you want to extend the original template, you might be tempted to write the following:

```
#{ page.twig in .../templates/mysite #}
{% extends "page.twig" %} #{ from .../templates/default #}
```

However, this will not work as Twig will always load the template from `.../templates/mysite`. It turns out it is possible to get this to work, by adding a directory right at the end of your template directories, which is the parent of all of the other directories: `.../templates` in our case. This has the effect of making every template file within our system uniquely addressable. Most of the time you will use the "normal" paths, but in the special case of wanting to extend a template with an overriding version of itself we can reference its parent's full, unambiguous template path in the extends tag:

```
#{ page.twig in .../templates/mysite #}
{% extends "default/page.twig" %} #{ from .../templates #}
```

This recipe was inspired by the following Django wiki page: [https://code.djangoproject.com/wiki/ExtendingTemplates](https://code.djangoproject.com/wiki/ExtendingTemplates)

### Customizing the Syntax

Twig allows some syntax customization for the block delimiters. It's not recommended to use this feature as templates will be tied with your custom syntax. But for specific projects, it can make sense to change the defaults.

To change the block delimiters, you need to create your own lexer object:

```
$twig = new Twig\Environment();
$lexer = new Twig\Lexer($twig, [
    'tag_comment' => ['#', '#'],
    'tag_block'   => ['%', '%'],
]);
```
Here are some configuration example that simulates some other template engines syntax:

Listing 8-10

```php
// Ruby erb syntax
$lexer = new Twig\Lexer($twig, [
    'tag_comment' => ['<%#', '%>'],
    'tag_block' => ['<%', '%>'],
    'tag_variable' => ['<%=', '%>'],
]);

// SGML Comment Syntax
$lexer = new Twig\Lexer($twig, [
    'tag_comment' => ['<!--#', '-->'],
    'tag_block' => ['<!--', '-->'],
    'tag_variable' => ['{$', '}'],
]);

// Smarty like
$lexer = new Twig\Lexer($twig, [
    'tag_comment' => '{*', '*}'],
    'tag_block' => '{', '}'],
    'tag_variable' => '{$', '}'],
]);
```

Using dynamic Object Properties

When Twig encounters a variable like `article.title`, it tries to find a `title` public property in the `article` object.

It also works if the property does not exist but is rather defined dynamically thanks to the magic `__get()` method; you need to also implement the `__isset()` magic method like shown in the following snippet of code:

Listing 8-11

```php
class Article {
    public function __get($name) {
        if ('title' == $name) {
            return 'The title';
        }
    }
    // throw some kind of error

    public function __isset($name) {
        if ('title' == $name) {
            return true;
        }
    }
    return false;
}
```

Accessing the parent Context in Nested Loops

Sometimes, when using nested loops, you need to access the parent context. The parent context is always
accessible via the `loop.parent` variable. For instance, if you have the following template data:

```php
Listing 8-12
$data = [
    'topics' => [
        'topic1' => ['Message 1 of topic 1', 'Message 2 of topic 1'],
        'topic2' => ['Message 1 of topic 2', 'Message 2 of topic 2'],
    ],
];
```

And the following template to display all messages in all topics:

```twig
Listing 8-13
{% for topic, messages in topics %}
  * {{ loop.index }}: {{ topic }}
  {% for message in messages %}
    - {{ loop.parent.loop.index }}.{{ loop.index }}: {{ message }}
  {% endfor %}
{% endfor %}
```

The output will be similar to:

```
* 1: topic1
  - 1.1: The message 1 of topic 1
  - 1.2: The message 2 of topic 1
* 2: topic2
  - 2.1: The message 1 of topic 2
  - 2.2: The message 2 of topic 2
```

In the inner loop, the `loop.parent` variable is used to access the outer context. So, the index of the current `topic` defined in the outer for loop is accessible via the `loop.parent.loop.index` variable.

### Defining undefined Functions and Filters on the Fly

When a function (or a filter) is not defined, Twig defaults to throw a `\Twig\Error\SyntaxError` exception. However, it can also call a **callback** (any valid PHP callable) which should return a function (or a filter).

For filters, register callbacks with `registerUndefinedFilterCallback()`. For functions, use `registerUndefinedFunctionCallback()`:

```php
Listing 8-15
// auto-register all native PHP functions as Twig functions
// don't try this at home as it's not secure at all!
$twig->registerUndefinedFunctionCallback(function ($name) {
    if (function_exists($name)) {
        return new \Twig\TwigFunction($name, $name);
    }

    // if the callable is not able to return a valid function (or filter), it must return false.
    return false;
});
```

If the callable is not able to return a valid function (or filter), it must return `false`.

If you register more than one callback, Twig will call them in turn until one does not return `false`.

---

As the resolution of functions and filters is done during compilation, there is no overhead when registering these callbacks.

Validating the Template Syntax

When template code is provided by a third-party (through a web interface for instance), it might be interesting to validate the template syntax before saving it. If the template code is stored in a `$template` variable, here is how you can do it:

```php
try {
    $twig->parse($twig->tokenize(new \Twig\Source($template)));
    // the $template is valid
} catch (\Twig\Error\SyntaxError $e) {
    // $template contains one or more syntax errors
}
```

If you iterate over a set of files, you can pass the filename to the `tokenize()` method to get the filename in the exception message:

```php
foreach ($files as $file) {
    try {
        $twig->parse($twig->tokenize(new \Twig\Source($template, $file->getFilename(), $file)));
        // the $template is valid
    } catch (\Twig\Error\SyntaxError $e) {
        // $template contains one or more syntax errors
    }
}
```

*New in version 1.27:* `\Twig\Source` was introduced in version 1.27, pass the source and the identifier directly on previous versions.

This method won't catch any sandbox policy violations because the policy is enforced during template rendering (as Twig needs the context for some checks like allowed methods on objects).

Refreshing modified Templates when OPcache or APC is enabled

When using OPcache with `opcache.validate_timestamps` set to 0 or APC with `apc.stat` set to 0 and Twig cache enabled, clearing the template cache won't update the cache.

To get around this, force Twig to invalidate the bytecode cache:

```php
$twig = new \Twig\Environment($loader, [
    'cache' => new \Twig\Cache\FilesystemCache('/some/cache/path', \Twig\Cache\FilesystemCache::FORCE_BYTECODE_INVALIDATION),
    // ...
]);
```
Before Twig 1.22, you should extend `\Twig\Environment` instead:

```php
class OpCacheAwareTwigEnvironment extends \Twig\Environment
{
    protected function writeCacheFile($file, $content)
    {
        parent::writeCacheFile($file, $content);

        // Compile cached file into bytecode cache
        if (function_exists('opcache_invalidate') && filter_var(ini_get('opcache.enable'), FILTER_VALIDATE_BOOLEAN)) {
            opcache_invalidate($file, true);
        } elseif (function_exists('apc_compile_file')) {
            apc_compile_file($file);
        }
    }
}
```

Reusing a stateful Node Visitor

When attaching a visitor to a `\Twig\Environment` instance, Twig uses it to visit all templates it compiles. If you need to keep some state information around, you probably want to reset it when visiting a new template.

This can be achieved with the following code:

```php
protected $someTemplateState = [];

public function enterNode(Twig_NodeInterface $node, \Twig\Environment $env)
{
    if ($node instanceof Twig_Node_ModuleNode) {
        // reset the state as we are entering a new template
        $this->someTemplateState = [];
    }
    // ...
    return $node;
}
```

Using a Database to store Templates

If you are developing a CMS, templates are usually stored in a database. This recipe gives you a simple PDO template loader you can use as a starting point for your own.

First, let’s create a temporary in-memory SQLite3 database to work with:

```php
$dbh = new PDO('sqlite::memory:');
$dbh->exec('CREATE TABLE templates (name STRING, source STRING, last_modified INTEGER)');
$base = '{% block content %}{% endblock %}'
$index = '{% extends "base.twig" %}
{% block content %}Hello {{ name }}{% endblock %}
';
$now = time();
```
We have created a simple `templates` table that hosts two templates: `base.twig` and `index.twig`. Now, let's define a loader able to use this database:

```php
class DatabaseTwigLoader implements \Twig\Loader\LoaderInterface, \Twig\Loader\ExistsLoaderInterface, \Twig\Loader\SourceContextLoaderInterface {
    protected $dbh;

    public function __construct(PDO $dbh) {
        $this->dbh = $dbh;
    }

    public function getSource($name) {
        if (false === $source = $this->getValue('source', $name)) {
            throw new \Twig\Error\LoaderError(sprintf('Template "%s" does not exist.', $name));
        }

        return $source;
    }

    public function getSourceContext($name) {
        if (false === $source = $this->getValue('source', $name)) {
            throw new \Twig\Error\LoaderError(sprintf('Template "%s" does not exist.', $name));
        }

        return new \Twig\Source($source, $name);
    }

    public function exists($name) {
        return $name === $this->getValue('name', $name);
    }

    public function getCacheKey($name) {
        return $name;
    }

    public function isFresh($name, $time) {
        if (false === $lastModified = $this->getValue('last_modified', $name)) {
            return false;
        }

        return $lastModified <= $time;
    }

    protected function getValue($column, $name) {
        $sth = $this->dbh->prepare('SELECT '.$column.' FROM templates WHERE name = :name');
        $sth->execute([':name' => (string) $name]);
        return $sth->fetchColumn();
    }
}
```

Finally, here is an example on how you can use it:

```php
$loader = new DatabaseTwigLoader($dbh);```
Using different Template Sources

This recipe is the continuation of the previous one. Even if you store the contributed templates in a database, you might want to keep the original/base templates on the filesystem. When templates can be loaded from different sources, you need to use the \Twig\Loader\ChainLoader loader.

As you can see in the previous recipe, we reference the template in the exact same way as we would have done it with a regular filesystem loader. This is the key to be able to mix and match templates coming from the database, the filesystem, or any other loader for that matter: the template name should be a logical name, and not the path from the filesystem:

```php
$loader1 = new DatabaseTwigLoader($dbh);
$loader2 = new Twig\Loader\ArrayLoader(['base.twig' => '{% block content %}{% endblock %}', ]);
$loader = new Twig\Loader\ChainLoader([$loader1, $loader2]);
$twig = new Twig\Environment($loader);

echo $twig->render('index.twig', ['name' => 'Fabien']);
```

Now that the **base.twig** templates is defined in an array loader, you can remove it from the database, and everything else will still work as before.

Loading a Template from a String

From a template, you can load a template stored in a string via the `template_from_string` function (available as of Twig 1.11 via the \Twig\Extension\StringLoaderExtension extension):

```php
{{ include(template_from_string("Hello {{ name }}")) }}
```

From PHP, it's also possible to load a template stored in a string via `Twig\Environment::createTemplate()` (available as of Twig 1.18):

```php
$template = $twig->createTemplate('hello {{ name }}');
echo $template->render(['name' => 'Fabien']);
```

Never use the Twig_Loader_String loader, which has severe limitations.

Using Twig and AngularJS in the same Templates

Mixing different template syntaxes in the same file is not a recommended practice as both AngularJS and Twig use the same delimiters in their syntax: {{ and }}.
Still, if you want to use AngularJS and Twig in the same template, there are two ways to make it work depending on the amount of AngularJS you need to include in your templates:

- Escaping the AngularJS delimiters by wrapping AngularJS sections with the `{% verbatim %}` tag or by escaping each delimiter via `{{ '{' }}` and `{{ '}' }}`;
- Changing the delimiters of one of the template engines (depending on which engine you introduced last):
  - For AngularJS, change the interpolation tags using the `interpolateProvider` service, for instance at the module initialization time:

```javascript
angular.module('myApp', []).config(function($interpolateProvider) {
  $interpolateProvider.startSymbol('{{').endSymbol('}}');
});
```
  
  - For Twig, change the delimiters via the `tag_variable` Lexer option:

```javascript
$env->setLexer(new Twig Lexer($env, [
  'tag_variable' => ['{{', '}}'],
]));
```
Chapter 9

Coding Standards

When writing Twig templates, we recommend you to follow these official coding standards:

- Put one (and only one) space after the start of a delimiter ({{, {%}, and {{#}}) and before the end of a delimiter (}}), %}, and {{#}}):

Listing 9-1

```
1 {{ foo }}
2 {% comment %}
3 {% if foo %}[% endif %]
```

When using the whitespace control character, do not put any spaces between it and the delimiter:

Listing 9-2

```
1 {{- foo -}}
2 {#- comment -#}
3 {%- if foo %}{%- endif -%}
```

- Put one (and only one) space before and after the following operators: comparison operators (==, !=, <, >, >=, <=), math operators (+, -, /, *, %, //, **), logic operators (not, and, or, ~, is, in, and the ternary operator (?:)):

Listing 9-3

```
1 {{ 1 + 2 }}
2 {{ foo * bar }}
3 {{ true ? true : false }}
```

- Put one (and only one) space after the : sign in hashes and , in arrays and hashes:

Listing 9-4

```
1 {{ [1, 2, 3] }}
2 {{ ['foo': 'bar'] }}
```

- Do not put any spaces after an opening parenthesis and before a closing parenthesis in expressions:

Listing 9-5

```
1 {{ 1 + (2 * 3) }}
```

- Do not put any spaces before and after string delimiters:

Listing 9-6

```
1 {{ 'foo' }}
2 {{ "foo" }}
```
• Do not put any spaces before and after the following operators: |, . . . , []:

Listing 9-7
1 {{ foo|upper|lower }}
2 {{ user.name }}
3 {{ user.name }}
4 {% for i in 1..12 %}{% endfor %}

• Do not put any spaces before and after the parenthesis used for filter and function calls:

Listing 9-8
1 {{ foo|default('foo') }}
2 {{ range(1..10) }}

• Do not put any spaces before and after the opening and the closing of arrays and hashes:

Listing 9-9
1 {{ [1, 2, 3] }}
2 {{ {'foo': 'bar'} }}

• Use lower cased and underscored variable names:

Listing 9-10
1 {% set foo = 'foo' %}
2 {% set foo_bar = 'foo' %}

• Indent your code inside tags (use the same indentation as the one used for the target language of the rendered template):

Listing 9-11
1 {% block foo %}
2  {% if true %}
3    true
4  {% endif %}
5  {% endblock %}
New in version 1.40: The `apply` tag was added in Twig 1.40.

The `apply` tag allows you to apply Twig filters on a block of template data:

```
{% apply upper %}
This text becomes uppercase
{% endapply %}
```

You can also chain filters and pass arguments to them:

```
{% apply lower|escape('html') %}
<strong>SOME TEXT</strong>
{% endapply %}
```

`# outputs "&lt;strong&gt;some text&lt;/strong;&quot;`
Whether automatic escaping is enabled or not, you can mark a section of a template to be escaped or not by using the `autoescape` tag:

```
{% autoescape %}
Everything will be automatically escaped in this block
using the HTML strategy
{% endautoescape %}

{% autoescape 'html' %}
Everything will be automatically escaped in this block
using the HTML strategy
{% endautoescape %}

{% autoescape 'js' %}
Everything will be automatically escaped in this block
using the js escaping strategy
{% endautoescape %}

{% autoescape false %}
Everything will be outputted as is in this block
{% endautoescape %}
```
Before Twig 1.8, the syntax was different:

Listing 11-2

```twig
{% autoescape true %}
  Everything will be automatically escaped in this block
  using the HTML strategy
{% endautoescape %}

{% autoescape false %}
  Everything will be outputted as is in this block
{% endautoescape %}

{% autoescape true js %}
  Everything will be automatically escaped in this block
  using the js escaping strategy
{% endautoescape %}
```

When automatic escaping is enabled everything is escaped by default except for values explicitly marked as safe. Those can be marked in the template by using the `raw` filter:

Listing 11-3

```twig
{% autoescape %}
  {{ safe_value|raw }}
{% endautoescape %}
```

Functions returning template data (like macros and parent) always return safe markup.

Twig is smart enough to not escape an already escaped value by the `escape` filter.

Twig does not escape static expressions:

Listing 11-4

```twig
{% set hello = '<strong>Hello</strong>' %}

{{ hello }}

{{ '<strong>world</strong>' }}
```

Will be rendered "<strong>Hello</strong> world".

The chapter Twig for Developers gives more information about when and how automatic escaping is applied.
Blocks are used for inheritance and act as placeholders and replacements at the same time. They are documented in detail in the documentation for the extends tag.

Block names should consist of alphanumeric characters, and underscores. Dashes are not permitted.

*block, parent, use, extends*
Chapter 13

deprecated

New in version 1.36: and 2.6 The **deprecated** tag was added in Twig 1.36 and 2.6.

Twig generates a deprecation notice (via a call to the `trigger_error()` PHP function) where the **deprecated** tag is used in a template:

```twig
{% deprecated 'The "base.twig" template is deprecated, use "layout.twig" instead.' %}
{% extends 'layout.twig' %}
```

Also you can deprecate a block in the following way:

```twig
{% block hey %}
  {% deprecated 'The "hey" block is deprecated, use "greet" instead.' %}
  {{ block('greet') }}
{% endblock %}

{% block greet %}
  Hey you!
{% endblock %}
```

Note that by default, the deprecation notices are silenced and never displayed nor logged. See Displaying Deprecation Notices to learn how to handle them.
New in version 1.5: The do tag was added in Twig 1.5.

The do tag works exactly like the regular variable expression ({{ ... }}) just that it doesn't print anything:

```plaintext
{% do 1 + 2 %}
```
New in version 1.8: The embed tag was added in Twig 1.8.

The embed tag combines the behaviour of include and extends. It allows you to include another template's contents, just like include does. But it also allows you to override any block defined inside the included template, like when extending a template.

Think of an embedded template as a "micro layout skeleton".

```twig
{% embed "teasers_skeleton.twig" %}
  # These blocks are defined in "teasers_skeleton.twig" #
  # and we override them right here:  
  #
  {% block left_teaser %}
    Some content for the left teaser box
  {% endblock %}
  #
  {% block right_teaser %}
    Some content for the right teaser box
  {% endblock %}
{% endembed %}
```

The embed tag takes the idea of template inheritance to the level of content fragments. While template inheritance allows for "document skeletons", which are filled with life by child templates, the embed tag allows you to create "skeletons" for smaller units of content and re-use and fill them anywhere you like.

Since the use case may not be obvious, let's look at a simplified example. Imagine a base template shared by multiple HTML pages, defining a single block named "content":

```text
page layout

block "content"

(child template to put content here)
```
Some pages ("foo" and "bar") share the same content structure - two vertically stacked boxes:

```
1 2 3 4 5 6 7 8 9 10 11
Page layout
 ┌─ block "content" ┐
 │               │
 │     ┌─ block "top" ┐  │
 │     │               │  │
 │     │               │  └─ block "bottom"
 │               │
 │               │
 └──────────────┘
```

While other pages ("boom" and "baz") share a different content structure - two boxes side by side:

```
1 2 3 4 5 6 7 8 9 10 11
Page layout
 ┌─ block "content" ┐
 │               │
 │     ┌─ block "left" ┐  │
 │     │               │  │
 │     │               │  └─ block "right"
 │               │
 │               │
 └──────────────┘
```

Without the `embed` tag, you have two ways to design your templates:

- Create two "intermediate" base templates that extend the master layout template: one with vertically stacked boxes to be used by the "foo" and "bar" pages and another one with side-by-side boxes for the "boom" and "baz" pages.
- Embed the markup for the top/bottom and left/right boxes into each page template directly.

These two solutions do not scale well because they each have a major drawback:

- The first solution may indeed work for this simplified example. But imagine we add a sidebar, which may again contain different, recurring structures of content. Now we would need to create intermediate base templates for all occurring combinations of content structure and sidebar structure... and so on.
- The second solution involves duplication of common code with all its negative consequences: any change involves finding and editing all affected copies of the structure, correctness has to be verified for each copy, copies may go out of sync by careless modifications etc.

In such a situation, the `embed` tag comes in handy. The common layout code can live in a single base template, and the two different content structures, let's call them "micro layouts" go into separate templates which are embedded as necessary:

Page template `foo.twig`:

```
1 (% extends "layout_skeleton.twig" %)
2 (% block content %)
3 (% embed "vertical_boxes_skeleton.twig" %)
4 (% block top %)
5 Some content for the top box
6 (% endblock %)
7 (% block bottom %)
8 Some content for the bottom box
9 (% endblock %)
10
11
12
```

---

*PDF brought to you by [Sensio](http://sensio.com) generated on February 19, 2020*
And here is the code for `vertical_boxes_skeleton.twig`:

```twig
<% embed "base" with {'foo': 'bar'} %>...
<% endembed %>

Listing 15.6
1 <div class="top_box">
2 (% block top %)
3 Top box default content
4 (% endblock %)
5 </div>
6 7 <div class="bottom_box">
8 (% block bottom %)
9 Bottom box default content
10 (% endblock %)
11 </div>
```

The goal of the `vertical_boxes_skeleton.twig` template is to factor out the HTML markup for the boxes.

The `embed` tag takes the exact same arguments as the `include` tag:

```twig
<% embed "base" with {'foo': 'bar'} %>...
<% endembed %>
<% embed "base" with {'foo': 'bar'} only %>...
<% endembed %>
<% embed "base" ignore missing %>...
<% endembed %>
```

As embedded templates do not have "names", auto-escaping strategies based on the template name won't work as expected if you change the context (for instance, if you embed a CSS/JavaScript template into an HTML one). In that case, explicitly set the default auto-escaping strategy with the `autoescape` tag.

`include`
The `extends` tag can be used to extend a template from another one.

Like PHP, Twig does not support multiple inheritance. So you can only have one `extends` tag called per rendering. However, Twig supports horizontal reuse.

Let's define a base template, `base.html`, which defines a simple HTML skeleton document:

```html
<!DOCTYPE html>
<html>
<head>{% block head %}
    <link rel="stylesheet" href="style.css" />
    <title>{% block title %}My Webpage{% endblock %}</title>
{% endblock %}
</head>
<body>
    <div id="content">{% block content %}{% endblock %}</div>
    <div id="footer"><% block footer %>
    </% endblock %>
</div>
</body>
</html>
```

In this example, the block tags define four blocks that child templates can fill in.

All the `block` tag does is to tell the template engine that a child template may override those portions of the template.
Child Template

A child template might look like this:

```plaintext
Listing 16-2
{% extends "base.html" %}
{% block title %}Index{% endblock %}
{% block head %}
{{ parent() }}
{% endblock %}
<style type="text/css">
.important { color: #336699; }
</style>
{% 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.

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 `block` function:

```plaintext
Listing 16-3
<title>{% block title %}{% endblock %}</title>
<h1>{{ block('title') }}</h1>
{% block body %}{% endblock %}
```

Parent Blocks

It's possible to render the contents of the parent block by using the `parent` function. This gives back the results of the parent block:

```plaintext
Listing 16-4
{% block sidebar %}
<h3>Table Of Contents</h3>
...
{{ parent() }}
{% endblock %}
```

Named Block End-Tags

Twig allows you to put the name of the block after the end tag for better readability (the name after the `endblock` word must match the block name):

```plaintext
Listing 16-5
{% block sidebar %}
{% block inner_sidebar %}
...
```
Blocks can be nested for more complex layouts. Per default, blocks have access to variables from outer scopes:

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

## Block Shortcuts

For blocks with little content, it's possible to use a shortcut syntax. The following constructs do the same thing:

```twig
{% block title %}
{{ page_title|title }}
{% endblock %}
```

```twig```
{% block title page_title|title %}
```twig```

## Dynamic Inheritance

Twig supports dynamic inheritance by using a variable as the base template:

```twig
{% extends some_var %}
```

If the variable evaluates to a `\Twig\Template` or a `\Twig\TemplateWrapper` instance, Twig will use it as the parent template:

```twig```
// {% extends layout %}
// deprecated as of Twig 1.28
$layout = $twig->loadTemplate('some_layout_template.twig');
// as of Twig 1.28
$layout = $twig->load('some_layout_template.twig');
$twig->display('template.twig', ['layout' => $layout]);
```twig```

**New in version 1.2:** The possibility to pass an array of templates has been added in Twig 1.2. You can also provide a list of templates that are checked for existence. The first template that exists will be used as a parent:

```twig```
{% extends ['layout.html', 'base_layout.html'] %}
```twig```
Conditional Inheritance

As the template name for the parent can be any valid Twig expression, it's possible to make the inheritance mechanism conditional:

```twig
{% extends standalone ? "minimum.html" : "base.html" %}
```

In this example, the template will extend the "minimum.html" layout template if the `standalone` variable evaluates to `true`, and "base.html" otherwise.

How do blocks work?

A block provides a way to change how a certain part of a template is rendered but it does not interfere in any way with the logic around it.

Let's take the following example to illustrate how a block works and more importantly, how it does not work:

```twig
{% for post in posts %}
  {% block post %}
    <h1>{{ post.title }}</h1>
    <p>{{ post.body }}</p>
  {% endblock %}
{% endfor %}
```

If you render this template, the result would be exactly the same with or without the `block` tag. The block inside the `for` loop is just a way to make it overridable by a child template:

```twig
{% extends "base.twig" %}
{% block post %}
  <article>
    <header>{{ post.title }}</header>
    <section>{{ post.text }}</section>
  </article>
{% endblock %}
```

Now, when rendering the child template, the loop is going to use the block defined in the child template instead of the one defined in the base one; the executed template is then equivalent to the following one:

```twig
{% for post in posts %}
  <article>
    <header>{{ post.title }}</header>
    <section>{{ post.text }}</section>
  </article>
{% endfor %}
```

Let's take another example: a block included within an `if` statement:

```twig
{% if posts is empty %}
  {% block head %}
    <meta name="robots" content="noindex, follow">
  {% endblock head %}
{% endif %}
```
Contrary to what you might think, this template does not define a block conditionally; it just makes overridable by a child template the output of what will be rendered when the condition is true.

If you want the output to be displayed conditionally, use the following instead:

```latex
\begin{verbatim}
{\% block head \%
\{\parent[\}

{% if posts is empty %}
\meta name="robots" content="noindex, follow">
{% endif %}

{% endblock head \%
\end{verbatim}
```

*block, block, parent, use*
As of Twig 1.40, you should use the `apply` tag instead which does the same thing except that the wrapped template data is not scoped.

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

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

You can also chain filters and pass arguments to them:

```twig
{% filter lower%escape('html')%}
<br>Some Text</strong>
{% endfilter %}
```

Listing 17-2

```
5 (# outputs "&lt;strong&gt;some text&lt;/strong&gt;"
```
New in version 1.5: The flush tag was added in Twig 1.5. The `flush` tag tells Twig to flush the output buffer:

Listing 18-1  

```
{% flush %}
```

Internally, Twig uses the PHP `flush` function.

1. https://secure.php.net/flush
Chapter 19

for

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

```plaintext
Listing 19-1
1 <h1>Members</h1>
2 <ul>
3   (% for user in users %)
4     <li>{{ user.username }}</li>
5   (% endfor %)
6 </ul>
```

A sequence can be either an array or an object implementing the Traversable interface.

If you do need to iterate over a sequence of numbers, you can use the .. operator:

```plaintext
Listing 19-2
1 (% for i in 0..10 %)
2   * {{ i }}
3 (% endfor %)
```

The above snippet of code would print all numbers from 0 to 10.

It can be also useful with letters:

```plaintext
Listing 19-3
1 (% for letter in 'a'..'z' %)
2   * {{ letter }}
3 (% endfor %)
```

The .. operator can take any expression at both sides:
The loop variable

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

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop.index</td>
<td>The current iteration of the loop. (1 indexed)</td>
</tr>
<tr>
<td>loop.index0</td>
<td>The current iteration of the loop. (0 indexed)</td>
</tr>
<tr>
<td>loop.revindex</td>
<td>The number of iterations from the end of the loop (1 indexed)</td>
</tr>
<tr>
<td>loop.revindex0</td>
<td>The number of iterations from the end of the loop (0 indexed)</td>
</tr>
<tr>
<td>loop.first</td>
<td>True if first iteration</td>
</tr>
<tr>
<td>loop.last</td>
<td>True if last iteration</td>
</tr>
<tr>
<td>loop.length</td>
<td>The number of items in the sequence</td>
</tr>
<tr>
<td>loop.parent</td>
<td>The parent context</td>
</tr>
</tbody>
</table>

The loop.length, loop.revindex, loop.revindex0, and loop.last variables are only available for PHP arrays, or objects that implement the Countable interface. They are also not available when looping with a condition.

New in version 1.2: The if modifier support has been added in Twig 1.2.

Adding a condition

As of Twig 1.41, use the filter filter instead, or an if condition inside the for body (if your condition depends on a variable updated inside the loop and you are not using the loop variable).

Unlike in PHP, 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 not
active:

```html
Listing 19-6
1  <ul>
2   (% for user in users if user.active %)
3   <li>{{ user.username | e }}</li>
4   (% endfor %)
5  </ul>
```

The advantage is that the special loop variable will count correctly thus not counting the users not iterated over. Keep in mind that properties like `loop.last` will not be defined when using loop conditions.

Using the `loop` variable within the condition is not recommended as it will probably not be doing what you expect it to. For instance, adding a condition like `loop.index > 4` won't work as the index is only incremented when the condition is true (so the condition will never match).

The `else` Clause

If no iteration took place because the sequence was empty, you can render a replacement block by using `else`:

```html
Listing 19-7
1  <ul>
2   (% for user in users %)
3   <li>{{ user.username | e }}</li>
4   (% else %)
5   <li><em>no user found</em></li>
6   (% endif %)
7  </ul>
```

Iterating over Keys

By default, a loop iterates over the values of the sequence. You can iterate on keys by using the `keys` filter:

```html
Listing 19-8
1  <h1>Members</h1>
2  <ul>
3   (% for key in users.keys %)
4   <li>{{ key }}</li>
5   (% endfor %)
6  </ul>
```

Iterating over Keys and Values

You can also access both keys and values:

```html
Listing 19-9
1  <h1>Members</h1>
2  <ul>
3   (% for key, user in users %)
```
Iterating over a Subset

You might want to iterate over a subset of values. This can be achieved using the slice filter:

```html
<h1>Top Ten Members</h1>
<ul>
  {% for user in users|slice(0, 10) %}
    <li>{{ user.username|e }}</li>
  {% endfor %}
</ul>
```
The `from` tag imports macro names into the current namespace. The tag is documented in detail in the documentation for the macro tag.
The `if` statement in Twig is comparable with the if statements of PHP.

In the simplest form you can use it to test if an expression evaluates to `true`:

Listing 21-1

```twig
{% if online == false %}
<p>Our website is in maintenance mode. Please, come back later.</p>
{% endif %}
```

You can also test if an array is not empty:

Listing 21-2

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

If you want to test if the variable is defined, use `if users is defined` instead.

You can also use `not` to check for values that evaluate to `false`:

Listing 21-3

```twig
{% if not user.subscribed %}
<p>You are not subscribed to our mailing list.</p>
{% endif %}
```

For multiple conditions, `and` and `or` can be used:

Listing 21-4

```twig
{% if temperature > 18 and temperature < 27 %}
```
It’s a nice day for a walk in the park.

For multiple branches `elseif` and `else` can be used like in PHP. You can use more complex expressions there too:

```plaintext
{% if product.stock > 10 %}
Available
{% elseif product.stock > 0 %}
Only {{ product.stock }} left!
{% else %}
Sold-out!
{% endif %}
```

The rules to determine if an expression is **true** or **false** are the same as in PHP; here are the edge cases rules:

<table>
<thead>
<tr>
<th>Value</th>
<th>Boolean evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>empty string</td>
<td>false</td>
</tr>
<tr>
<td>numeric zero</td>
<td>false</td>
</tr>
<tr>
<td>NAN (Not A Number)</td>
<td>true</td>
</tr>
<tr>
<td>INF (Infinity)</td>
<td>true</td>
</tr>
<tr>
<td>whitespace-only string</td>
<td>true</td>
</tr>
<tr>
<td>string &quot;0&quot; or '0'</td>
<td>false</td>
</tr>
<tr>
<td>empty array</td>
<td>false</td>
</tr>
<tr>
<td>null</td>
<td>false</td>
</tr>
<tr>
<td>non-empty array</td>
<td>true</td>
</tr>
<tr>
<td>object</td>
<td>true</td>
</tr>
</tbody>
</table>
Chapter 22

import

The **import** tag imports macro names in a local variable. The tag is documented in detail in the documentation for the macro tag.
The `include` statement includes a template and returns the rendered content of that file:

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

As of Twig 1.12, it is recommended to use the `include` function instead as it provides the same features with a bit more flexibility:

- The `include` function is semantically more "correct" (including a template outputs its rendered contents in the current scope; a tag should not display anything);
- It's easier to store the rendered template in a variable when using the `include` function:

```
{% set content = include('template.html') %}
```

- The `include` function does not impose any specific order for arguments thanks to named arguments.

Included templates have access to the variables of the active context.

If you are using the filesystem loader, the templates are looked for in the paths defined by it.

You can add additional variables by passing them after the `with` keyword:

```
{% include 'template.html' with {'foo': 'bar'} %}
```
You can disable access to the context by appending the `only` keyword:

```twig
{% set vars = {'foo': 'bar'}.%}
{% include 'template.html' with vars only%}
```

When including a template created by an end user, you should consider sandboxing it. More information in the Twig for Developers chapter and in the sandbox tag documentation.

The template name can be any valid Twig expression:

```twig
{% include some_var %}
{% include ajax ? 'ajax.html' : 'not_ajax.html'%}
```

And if the expression evaluates to a `Twig\Template` or a `Twig\TemplateWrapper` instance, Twig will use it directly:

```twig
// {% include template %}
// deprecated as of Twig 1.28
$template = $twig->loadTemplate('some_template.twig');
// as of Twig 1.28
$template = $twig->load('some_template.twig');
$twig->display('template.twig', ['template' => $template]);
```

**New in version 1.2:** The `ignore missing` feature has been added in Twig 1.2.

You can mark an include with `ignore missing` in which case Twig will ignore the statement if the template to be included does not exist. It has to be placed just after the template name. Here some valid examples:

```twig
{% include 'sidebar.html' ignore missing %}
{% include 'sidebar.html' ignore missing with {'foo': 'bar'}.%}
{% include 'sidebar.html' ignore missing only %}
```

**New in version 1.2:** The possibility to pass an array of templates has been added in Twig 1.2.

You can also provide a list of templates that are checked for existence before inclusion. The first template that exists will be included:

```twig
{% include ['page_detailed.html', 'page.html']%}
```

If `ignore missing` is given, it will fall back to rendering nothing if none of the templates exist, otherwise it will throw an exception.
New in version 1.12: The possibility to define default values for arguments in the macro signature was added in Twig 1.12.

Macros are comparable with functions in regular programming languages. They are useful to reuse template fragments to not repeat yourself.

Macros are defined in regular templates.

Imagine having a generic helper template that define how to render HTML forms via macros (called forms.html):

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

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

Each macro argument can have a default value (here `text` is the default value for `type` if not provided in the call).

Before Twig 1.12, defining default argument values was done via the `default` filter in the macro body:

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

Macros differ from native PHP functions in a few ways:
• Arguments of a macro are always optional.
• If extra positional arguments are passed to a macro, they end up in the special `varargs` variable as a list of values.

But as with PHP functions, macros don't have access to the current template variables.

💡 You can pass the whole context as an argument by using the special `_context` variable.

## Import

There are two ways to import macros. You can import the complete template containing the macros into a local variable (via the `import` tag) or only import specific macros from the template (via the `from` tag).

To import all macros from a template into a local variable, use the `import` tag:

```
{% import "forms.html" as forms %}
```

The above `import` call imports the `forms.html` file (which can contain only macros, or a template and some macros), and import the macros as items of the `forms` local variable.

The macros can then be called at will in the current template:

```
<p>{{ forms.input('username') }}</p>
<p>{{ forms.input('password', null, 'password') }}</p>
```

When you want to use a macro in another macro from the same file, you need to import it locally:

```
{% macro input(name, value, type, size) %}
  <input type="{{ type | default('text') }}" name="{{ name }}" value="{{ value|e }}" size="{{ size | default(20) }}" />
{% endmacro %}
{% macro wrapped_input(name, value, type, size) %}
  {% import _self as forms %}
  <div class="field">
    {{ forms.input(name, value, type, size) }}
  </div>
{% endmacro %}
```

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

```
{% from 'forms.html' import input as input_field, textarea %
<p>{{ input_field('password', '', 'password') }}</p>
<p>{{ textarea('comment') }}</p>
```
Importing macros using `import` or `from` is local to the current file. The imported macros are not available in included templates or child templates; you need to explicitly re-import macros in each file.

To import macros from the current file, use the special `_self` variable:

```
{%
import _self as forms %}
<p>{{ forms.input('username') }}</p>
```

When you define a macro in the template where you are going to use it, you might be tempted to call the macro directly via `_self.input()` instead of importing it; even if it seems to work, this is just a side-effect of the current implementation and it won't work anymore in Twig 2.x.

#### Named Macro End-Tags

Twig allows you to put the name of the macro after the end tag for better readability (the name after the `endmacro` word must match the macro name):

```
{% macro input() %}
...
{% endmacro input %}
```
The **sandbox** tag can be used to enable the sandboxing mode for an included template, when sandboxing is not enabled globally for the Twig environment:

```twig
{% sandbox %}
{% include 'user.html' %}
{% endsandbox %}
```

The **sandbox** tag is only available when the sandbox extension is enabled (see the Twig for Developers chapter).

The **sandbox** tag can only be used to sandbox an include tag and it cannot be used to sandbox a section of a template. The following example won't work:

```twig
{% sandbox %}
{% for i in 1..2 %}
{{ i }}
{% endfor %}
{% endsandbox %}
```
Inside code blocks you can also assign values to variables. Assignments use the `set` tag and can have multiple targets.

Here is how you can assign the `bar` value to the `foo` variable:

```twig
{# displays bar #}
{# displays bar #}
```

The assigned value can be any valid Twig expression:

```twig
{# displays [1, 2] #}
{# displays {'foo': 'bar'} #}
{# displays 'foo' ~ 'bar' #}
```

Several variables can be assigned in one block:

```twig
{# is equivalent to #}
{# is equivalent to #}
```

The `set` tag can also be used to 'capture' chunks of text:

```twig
{# endset #}
```

```
```
If you enable automatic output escaping, Twig will only consider the content to be safe when capturing chunks of text.

Note that loops are scoped in Twig; therefore a variable declared inside a \texttt{for} loop is not accessible outside the loop itself:

\begin{verbatim}
{% for item in list %}
{% set foo = item %}
{% endfor %}
#{ foo is NOT available #}
\end{verbatim}

If you want to access the variable, just declare it before the loop:

\begin{verbatim}
{% set foo = "" %}
{% for item in list %}
{% set foo = item %}
{% endfor %}
#{ foo is available #}
\end{verbatim}
As of Twig 1.38, use the spaceless filter instead.

Use the `spaceless` tag to remove whitespace between HTML tags, not whitespace within HTML tags or whitespace in plain text:

```twig
{% spaceless %}
<div>
  <strong>foo</strong>
</div>
{% endspaceless %}

#{ output will be <div><strong>foo</strong></div> #}
```

This tag is not meant to "optimize" the size of the generated HTML content but merely to avoid extra whitespace between HTML tags to avoid browser rendering quirks under some circumstances.

If you want to optimize the size of the generated HTML content, gzip compress the output instead.
If you want to create a tag that actually removes all extra whitespace in an HTML string, be warned that this is not as easy as it seems to be (think of `textarea` or `pre` tags for instance). Using a third-party library like Tidy is probably a better idea.

For more information on whitespace control, read the dedicated section of the documentation and learn how you can also use the whitespace control modifier on your tags.
Chapter 28

use

New in version 1.1: Horizontal reuse was added in Twig 1.1.

Horizontal reuse is an advanced Twig feature that is hardly ever needed in regular templates. It is mainly used by projects that need to make template blocks reusable without using inheritance.

Template inheritance is one of the most powerful features of Twig but it is limited to single inheritance; a template can only extend one other template. This limitation makes template inheritance simple to understand and easy to debug:

```
{% extends "base.html" %}
{% block title %}{% endblock %}
{% block content %}{% endblock %}
```

Horizontal reuse is a way to achieve the same goal as multiple inheritance, but without the associated complexity:

```
{% extends "base.html" %}
{% use "blocks.html" %}
{% block title %}{% endblock %}
{% block content %}{% endblock %}
```

The `use` statement tells Twig to import the blocks defined in `blocks.html` into the current template (it's like macros, but for blocks):

```
{% use "blocks.html" %}
{% block sidebar %}{% endblock %}
```
In this example, the **use** statement imports the **sidebar** block into the main template. The code is mostly equivalent to the following one (the imported blocks are not outputted automatically):

```twig
{% extends "base.html" %}
{% block sidebar %}{% endblock %}
{% block title %}{% endblock %}
{% block content %}{% endblock %}
```

The **use** tag only imports a template if it does not extend another template, if it does not define macros, and if the body is empty. But it can use other templates.

Because **use** statements are resolved independently of the context passed to the template, the template reference cannot be an expression.

The main template can also override any imported block. If the template already defines the **sidebar** block, then the one defined in **blocks.html** is ignored. To avoid name conflicts, you can rename imported blocks:

```twig
{% extends "base.html" %}
{% use "blocks.html" with sidebar as base_sidebar, title as base_title %}
{% block sidebar %}{% parent() %}{% endblock %}
{% block title %}{% endblock %}
{% block content %}{% endblock %}
```

New in version 1.3: The **parent()** support was added in Twig 1.3. The **parent()** function automatically determines the correct inheritance tree, so it can be used when overriding a block defined in an imported template:

```twig
{% extends "base.html" %}
{% use "blocks.html" %}
{% block sidebar %}{{ parent() }}{% endblock %}
{% block title %}{% endblock %}
{% block content %}{% endblock %}
```

In this example, **parent()** will correctly call the **sidebar** block from the **blocks.html** template.
In Twig 1.2, renaming allows you to simulate inheritance by calling the "parent" block:

```twig
{% extends "base.html" %}
{% use "blocks.html" with sidebar as parent_sidebar %}
{% block sidebar %}
  {{ block('parent_sidebar') }}
{% endblock %}
```

You can use as many `use` statements as you want in any given template. If two imported templates define the same block, the latest one wins.
New in version 1.12: The `verbatim` tag was added in Twig 1.12 (it was named `raw` before).

The `verbatim` tag marks sections as being raw text that should not be parsed. For example to put Twig syntax as example into a template you can use this snippet:

```twig
{% verbatim %}
<ul>
{% for item in seq %}
<li>{{ item }}</li>
{% endfor %}
</ul>
{% endverbatim %}
```

The `verbatim` tag works in the exact same way as the old `raw` tag, but was renamed to avoid confusion with the `raw` filter.
Chapter 30

with

New in version 1.28: The `with` tag was added in Twig 1.28.

Use the `with` tag to create a new inner scope. Variables set within this scope are not visible outside of the scope:

```
{% with %}
  {% set foo = 42 %}
  {{ foo }}
{% endwith %}
foo is 42 here
foo is not visible here any longer
```

Instead of defining variables at the beginning of the scope, you can pass a hash of variables you want to define in the `with` tag; the previous example is equivalent to the following one:

```
{% with { foo: 42 } %}
  {{ foo }}
{% endwith %}
# only foo is defined#
# bar is not defined#
```

By default, the inner scope has access to the outer scope context; you can disable this behavior by appending the `only` keyword:

```
{% set bar = 'bar' %}
{% with { foo: 42 } only %}
  {{ foo }}
{% endwith %}
```

# only foo is defined#
# bar is not defined#
The `abs` filter returns the absolute value.

```
Listing 31-1
1  {# number = -5 #}
2  {{ number|abs }}
3  {# outputs 5 #}
```

Internally, Twig uses the PHP `abs`\(^1\) function.

---

1. https://secure.php.net/abs
Chapter 32

batch

New in version 1.12.3: The **batch** filter was added in Twig 1.12.3.

The **batch** filter "batches" items by returning a list of lists with the given number of items. A second parameter can be provided and used to fill in missing items:

```twig
{% set items = ['a', 'b', 'c', 'd', 'e', 'f', 'g'] %}
<table>
{% for row in items|batch(3, 'No item') %}
  <tr>
    {% for column in row %}
    <td>{{ column }}</td>
    {% endfor %}
  </tr>
{% endfor %}
</table>
```

The above example will be rendered as:

```
<table>
<tr><td>a</td><td>b</td><td>c</td></tr>
<tr><td>d</td><td>e</td><td>f</td></tr>
<tr><td>g</td><td>No item</td><td>No item</td></tr>
</table>
```
Arguments

- **size**: The size of the batch; fractional numbers will be rounded up
- **fill**: Used to fill in missing items
- **preserve_keys**: Whether to preserve keys or not
The `capitalize` filter capitalizes a value. The first character will be uppercase, all others lowercase:

```
1  {{ 'my first car'|capitalize }}
2  # outputs 'My first car'
```
Chapter 34

convert_encoding

New in version 1.4: The `convert_encoding` filter was added in Twig 1.4.

The `convert_encoding` filter converts a string from one encoding to another. The first argument is the expected output charset and the second one is the input charset:

```twig
{{ data|convert_encoding('UTF-8', 'iso-2022-jp') }}
```

This filter relies on the `iconv` or `mbstring` extension, so one of them must be installed. In case both are installed, `mbstring` is used by default (Twig before 1.8.1 uses `iconv` by default).

Arguments

- **to**: The output charset
- **from**: The input charset

---

1. https://secure.php.net/iconv
2. https://secure.php.net/mbstring
3. https://secure.php.net/mbstring
4. https://secure.php.net/iconv
New in version 1.1: The timezone support has been added in Twig 1.1.
New in version 1.5: The default date format support has been added in Twig 1.5.
New in version 1.6.1: The default timezone support has been added in Twig 1.6.1.
New in version 1.11.0: The introduction of the false value for the timezone was introduced in Twig 1.11.0

The `date` filter formats a date to a given format:

```
{{ post.published_at | date("m/d/Y") }}
```

The format specifier is the same as supported by `date`\(^1\), except when the filtered data is of type `DateInterval`\(^2\), when the format must conform to `DateInterval::format`\(^3\) instead.

The `date` filter accepts strings (it must be in a format supported by the `strtotime`\(^4\) function), `DateTime`\(^5\) instances, or `DateInterval`\(^6\) instances. For instance, to display the current date, filter the word "now":

```
{{ "now" | date("m/d/Y") }}
```

To escape words and characters in the date format use `\ \` in front of each character:

```
{{ post.published_at | date("F jS \a\t g:ia") }}
```

If the value passed to the `date` filter is `null`, it will return the current date by default. If an empty string is desired instead of the current date, use a ternary operator:

```
{{ post.published_at is empty ? "" : post.published_at | date("m/d/Y") }}
```

If no format is provided, Twig will use the default one: `F j Y H:i`. This default can be changed

---

1. https://secure.php.net/date
2. https://secure.php.net/DateInterval
3. https://secure.php.net/DateInterval.format
4. https://secure.php.net/strtotime
5. https://secure.php.net/DateTime
6. https://secure.php.net/DateInterval
by calling the `setDateFormat()` method on the `core` extension instance. The first argument is the default format for dates and the second one is the default format for date intervals:

```php
Listing 35-5
1 $twig = new \Twig\Environment($loader);
2 $twig->getExtension('\Twig\Extension\CoreExtension')->setDateFormat('d/m/Y', '%d days');
3 // before Twig 1.26
4 $twig->getExtension('core')->setDateFormat('d/m/Y', '%d days');
```

Timezone

By default, the date is displayed by applying the default timezone (the one specified in php.ini or declared in Twig -- see below), but you can override it by explicitly specifying a timezone:

```twig
Listing 35-6
1 {{ post.published_at|date("m/d/Y", "Europe/Paris") }}
```

If the date is already a DateTime object, and if you want to keep its current timezone, pass `false` as the timezone value:

```twig
Listing 35-7
1 {{ post.published_at|date("m/d/Y", false) }}
```

The default timezone can also be set globally by calling `setTimezone()`:

```php
Listing 35-8
1 $twig = new \Twig\Environment($loader);
2 $twig->getExtension('\Twig\Extension\CoreExtension')->setTimezone('Europe/Paris');
3 // before Twig 1.26
4 $twig->getExtension('core')->setTimezone('Europe/Paris');
```

Arguments

- `format`: The date format
- `timezone`: The date timezone
New in version 1.9.0: The date_modify filter has been added in Twig 1.9.0. The date_modify filter modifies a date with a given modifier string:

```twig
{{ post.published_at|date_modify("+1 day")|date("m/d/Y") }}
```

The date_modify filter accepts strings (it must be in a format supported by the strtotime\(^1\) function) or DateTime\(^2\) instances. You can combine it with the date filter for formatting.

### Arguments

- modifier: The modifier

---

1. https://secure.php.net/strtotime
2. https://secure.php.net/DateTime
Chapter 37

default

The default filter returns the passed default value if the value is undefined or empty, otherwise the value of the variable:

```javascript
{{ var|default('var is not defined') }}
{{ var.foo|default('foo item on var is not defined') }}
{{ var['foo']|default('foo item on var is not defined') }}
{{ ''|default('passed var is empty') }}
```

When using the default filter on an expression that uses variables in some method calls, be sure to use the default filter whenever a variable can be undefined:

```javascript
{{ var.method(foo|default('foo'))|default('foo') }}
```

Read the documentation for the defined and empty tests to learn more about their semantics.

Arguments

- default: The default value
Chapter 38

escape

New in version 1.9.0: The css, url, and html_attr strategies were added in Twig 1.9.0.
New in version 1.14.0: The ability to define custom escapers was added in Twig 1.14.0.

The escape filter escapes a string using strategies that depend on the context.

By default, it uses the HTML escaping strategy:

```
Listing 38-1
1 <p>
2 {{ user.username|escape }}</p>
```

For convenience, the e filter is defined as an alias:

```
Listing 38-2
1 <p>
2 {{ user.username|e }}</p>
```

The escape filter can also be used in other contexts than HTML thanks to an optional argument which defines the escaping strategy to use:

```
Listing 38-3
1 {{ user.username|e }}
2 (# is equivalent to #)
3 {{ user.username|e('html') }}
```

And here is how to escape variables included in JavaScript code:

```
Listing 38-4
1 {{ user.username|escape('js') }}
2 {{ user.username|e('js') }}
```

The escape filter supports the following escaping strategies for HTML documents:

- html: escapes a string for the HTML body context.
- js: escapes a string for the JavaScript context.
- css: escapes a string for the CSS context. CSS escaping can be applied to any string being inserted into CSS and escapes everything except alphanumeric.
- url: escapes a string for the URI or parameter contexts. This should not be used to escape an entire
URI; only a subcomponent being inserted.
• html_attr: escapes a string for the HTML attribute context.

Note that doing contextual escaping in HTML documents is hard and choosing the right escaping strategy depends on a lot of factors. Please, read related documentation like the OWASP prevention cheat sheet to learn more about this topic.

Internally, escape uses the PHP native htmlspecialchars function for the HTML escaping strategy.

When using automatic escaping, Twig tries to not double-escape a variable when the automatic escaping strategy is the same as the one applied by the escape filter; but that does not work when using a variable as the escaping strategy:

```
Listing 38-5
1 { % set strategy = 'html' % }
2 3 { % autoescape 'html' % }
4 {{ var|escape('html') }} {# won't be double-escaped #}
5 {{ var|escape(strategy) }} {# will be double-escaped #}
6 { % endautoescape %}
```

When using a variable as the escaping strategy, you should disable automatic escaping:

```
Listing 38-6
1 { % set strategy = 'html' % }
2 3 { % autoescape 'html' % }
4 {{ var|escape(strategy)|raw }} {# won't be double-escaped #}
5 { % endautoescape %}
```

Custom Escapers

You can define custom escapers by calling the setEscaper() method on the core extension instance. The first argument is the escaper name (to be used in the escape call) and the second one must be a valid PHP callable:

```
Listing 38-7
1 $twig = new \Twig\Environment($loader);
2 $twig->getExtension('\Twig\Extension\CoreExtension')->setEscaper('csv', 'csv_escaper');
3 // before Twig 1.26
4 $twig->getExtension('core')->setEscaper('csv', 'csv_escaper');
```

When called by Twig, the callable receives the Twig environment instance, the string to escape, and the charset.

---
2. https://secure.php.net/htmlspecialchars
Built-in escapers cannot be overridden mainly because they should be considered as the final implementation and also for better performance.

**Arguments**

- `strategy`: The escaping strategy
- `charset`: The string charset
New in version 1.41: The `filter` filter was added in Twig 1.41 and 2.10.

The `filter` filter filters elements of a sequence or a mapping using an arrow function. The arrow function receives the value of the sequence or mapping:

```twig
Listing 39-1
\[
\begin{align*}
\text{\%} & \text{ set sizes} = [34, 36, 38, 40, 42] \% \\
\text{% for } v \text{ in sizes} & | \text{filter}(v \Rightarrow v > 38) | \text{join(', ')} \\
\text{% output 40, 42 }%
\end{align*}
\]

Combined with the `for` tag, it allows to filter the items to iterate over:

```twig
Listing 39-2
\[
\begin{align*}
\text{% for } v \text{ in sizes} & | \text{filter}(v \Rightarrow v > 38) -\% \\
\text{% output 40, 42 }%
\end{align*}
\]

It also works with mappings:

```twig
Listing 39-3
\[
\begin{align*}
\text{% set sizes} & = \{ \\
\text{xs: } 34, \\
\text{s: } 36, \\
\text{m: } 38, \\
\text{l: } 40, \\
\text{xl: } 42, \\
\text{} \}
\end{align*}
\]

```twig
Listing 39-4
\[
\begin{align*}
\text{% for k, v in sizes} & | \text{filter}((v \Rightarrow v > 38) | \text{and} k \neq \text{"xl"}) -\% \\
\text{% output 40 l = 40 }%
\end{align*}
\]
Note that the arrow function has access to the current context.

**Arguments**

- `array`: The sequence or mapping
- `arrow`: The arrow function
Chapter 40

first

New in version 1.12.2: The `first` filter was added in Twig 1.12.2.

The `first` filter returns the first "element" of a sequence, a mapping, or a string:

```
{{ [1, 2, 3, 4]|first }}
# outputs 1#
```

```
{{ { a: 1, b: 2, c: 3, d: 4}|first }}
# outputs 1#
```

```
{{ '1234'|first }}
# outputs 1#
```

It also works with objects implementing the `Traversable`\(^1\) interface.

---

The `format` filter formats a given string by replacing the placeholders (placeholders follows the `sprintf` notation):

```php
{{ "I like %s and %s." | format(foo, "bar") }}
```

Listing 41.1

1. Outputs I like foo and bar
2. if the foo parameter equals to the foo string. #}

replace

1. https://secure.php.net/sprintf
Chapter 42

join

New in version 1.37: and 2.6.1 The **and** argument was added in Twig 1.37 and 2.6.1.

The **join** filter returns a string which is the concatenation of the items of a sequence:

```
Listing 42-1
1  {{ [1, 2, 3]|join }}
2  (# returns 123 #)
```

The separator between elements is an empty string per default, but you can define it with the optional first parameter:

```
Listing 42-2
1  {{ [1, 2, 3]|join('|') }}
2  (# outputs 1|2|3 #)
```

A second parameter can also be provided that will be the separator used between the last two items of the sequence:

```
Listing 42-3
1  {{ [1, 2, 3]|join(', ', 'and ') }}
2  (# outputs 1, 2 and 3 #)
```

**Arguments**

- glue: The separator
- and: The separator for the last pair of input items
Chapter 43

json_encode

The `json_encode` filter returns the JSON representation of a value:

Listing 43-1

```
  {{ data|json_encode() }}
```

Internally, Twig uses the PHP `json_encode`\(^1\) function.

Arguments

- **options**: A bitmask of `json_encode` options:\(^2\):
  ```
  {{ data|json_encode(constant('JSON_PRETTY_PRINT')) }}
  ```
  ```
  data|json_encode(constant('JSON_PRETTY_PRINT') b-or
  constant('JSON_HEX_QUOT'))}}
  ```

---

1. [https://secure.php.net/json_encode](https://secure.php.net/json_encode)
Chapter 44

keys

The **keys** filter returns the keys of an array. It is useful when you want to iterate over the keys of an array:

```
Listing 44.1
1  (% for key in array|keys %)
2  ...
3  (% endfor %)
```
New in version 1.12.2: The \texttt{last} filter was added in Twig 1.12.2.

The \texttt{last} filter returns the last "element" of a sequence, a mapping, or a string:

\begin{verbatim}
Listing 45-1
1 {{ [1, 2, 3, 4]|last }}
2 (# outputs 4 #)
3
4 {{ { a: 1, b: 2, c: 3, d: 4 }|last }}
5 (# outputs 4 #)
6
7 {{ '1234'|last }}
8 (# outputs 4 #)
\end{verbatim}

It also works with objects implementing the \texttt{Traversable}\textsuperscript{1} interface.

New in version 1.33: Support for the `__toString()` magic method has been added in Twig 1.33.

The `length` filter returns the number of items of a sequence or mapping, or the length of a string.

For objects that implement the `Countable` interface, `length` will use the return value of the `count()` method.

For objects that implement the `__toString()` magic method (and not `Countable`), it will return the length of the string provided by that method.

For objects that implement the `IteratorAggregate` interface, `length` will use the return value of the `iterator_count()` method.

```twig
{% if users|length > 10 %}
    ...
{% endif %}
```
Chapter 47

lower

The **lower** filter converts a value to lowercase:

```
{{ 'WELCOME' | lower }}
```

# outputs 'welcome' #
Chapter 48

map

New in version 1.41: The map filter was added in Twig 1.41 and 2.10.

The map filter applies an arrow function to the elements of a sequence or a mapping. The arrow function receives the value of the sequence or mapping:

Listing 48-1
5
6 {{ people|map(p => "#{p.first} #{p.last}"|join(' ', )) }}
7 (# outputs Bob Smith, Alice Dupond #)

The arrow function also receives the key as a second argument:

Listing 48-2
1 (% set people = [ 2  "Bob": "Smith", 3  "Alice": "Dupond", 4 ] %)
5
6 {{ people|map((last, first) => "#{first} #{last}"|join(' ', )) }}
7 (# outputs Bob Smith, Alice Dupond #)

Note that the arrow function has access to the current context.

Arguments

- arrow: The arrow function
Chapter 49

merge

The **merge** filter merges an array with another array:

```noaatex
\begin{verbatim}
{% set values = [1, 2] %}
{% set values = values | merge(["apple", "orange"]) %}
{# values now contains [1, 2, "apple", "orange"] #}
\end{verbatim}
```

New values are added at the end of the existing ones.

The **merge** filter also works on hashes:

```noaatex
\begin{verbatim}
{% set items = {"apple": "fruit", "orange": "fruit", "peugeot": "unknown"} %}
{% set items = items | merge({"peugeot": "car", "renault": "car"}) %}
{# items now contains { 'apple': 'fruit', 'orange': 'fruit', 'peugeot': 'car', 'renault': 'car' } #}
\end{verbatim}
```

For hashes, the merging process occurs on the keys: if the key does not already exist, it is added but if the key already exists, its value is overridden.

If you want to ensure that some values are defined in an array (by given default values), reverse the two elements in the call:

```noaatex
\begin{verbatim}
{% set items = {"apple": "fruit", "orange": "fruit"} %}
{% set items = {"apple": "unknown"} | merge(items) %}
{# items now contains { 'apple': 'fruit', 'orange': 'fruit' } #}
\end{verbatim}
```
Internally, Twig uses the PHP `array_merge` function. It supports Traversable objects by transforming those to arrays.

1. https://secure.php.net/array_merge
New in version 1.5: The \texttt{nl2br} filter was added in Twig 1.5.

The \texttt{nl2br} filter inserts HTML line breaks before all newlines in a string:

```twig
{{ "I like Twig.
You will like it too."|nl2br }}
# outputs
I like Twig.<br />
You will like it too.
#}
```

The \texttt{nl2br} filter pre-escapes the input before applying the transformation.
Chapter 51

**number_format**

*New in version 1.5:* The `number_format` filter was added in Twig 1.5

The `number_format` filter formats numbers. It is a wrapper around PHP's `number_format`\(^1\) function:

```
{{ 200.35 | number_format }}
```

You can control the number of decimal places, decimal point, and thousands separator using the additional arguments:

```
{{ 9800.333 | number_format(2, '.', ',') }}
```

To format negative numbers, wrap the number with parentheses (needed because of Twig's precedence of operators:

```
{{ -(9800.333) | number_format(2, '.', ',') }}
```

If no formatting options are provided then Twig will use the default formatting options of:

- 0 decimal places.
- . as the decimal point.
- , as the thousands separator.

These defaults can be changed through the core extension:

```
$twig = new \Twig\Environment($loader);
$twig->getExtension('\Twig\Extension\CoreExtension')->setNumberFormat(3, '.', ',');
```

```
// before Twig 1.26
$twig->getExtension('core')->setNumberFormat(3, '.', ',');
```

The defaults set for `number_format` can be over-ridden upon each call using the additional parameters.

---

\(^1\) [https://secure.php.net/number_format](https://secure.php.net/number_format)
Arguments

- `decimal`: The number of decimal points to display
- `decimal_point`: The character(s) to use for the decimal point
- `thousand_sep`: The character(s) to use for the thousands separator
The **raw** filter marks the value as being "safe", which means that in an environment with automatic escaping enabled this variable will not be escaped if **raw** is the last filter applied to it:

```twig
{% autoescape %}
{{ var | raw }} # var won't be escaped #
{% endautoescape %}
```

This note only applies to Twig before versions 1.39 and 2.8.

Be careful when using the **raw** filter inside expressions:

```twig
{% autoescape %
{% set hello = '<strong>Hello</strong>' %
{% set hola = '<strong>Hola</strong>' %

{{ false ? '<strong>Hola</strong>' : hello | raw }}

does not render the same as

{{ false ? hola : hello | raw }}

but renders the same as

{{ (false ? hola : hello) | raw }}
{% endautoescape %}
```

The first ternary statement is not escaped: **hello** is marked as being safe and Twig does not escape static values (see escape). In the second ternary statement, even if **hello** is marked as safe, **hola** remains unsafe and so is the whole expression. The third ternary statement is marked as safe and the result is not escaped.
New in version 1.41: The reduce filter was added in Twig 1.41 and 2.10.

The reduce filter iteratively reduces a sequence or a mapping to a single value using an arrow function, so as to reduce it to a single value. The arrow function receives the return value of the previous iteration and the current value of the sequence or mapping:

```twig
{%
  set numbers = [1, 2, 3]
%
%
{%
  numbers|reduce((carry, v) => carry + v)}
%
#{ output 6 #}
```

The reduce filter takes an initial value as a second argument:

```twig
{%
  numbers|reduce((carry, v) => carry + v, 10)}
%
#{ output 16 #}
```

Note that the arrow function has access to the current context.

**Arguments**

- **arrow**: The arrow function
- **initial**: The initial value
Chapter 54

replace

The **replace** filter formats a given string by replacing the placeholders (placeholders are free-form):

```plaintext
{{ "I like %this% and %that%." | replace({"%this%": foo, "%that%": "bar"}) }}
```

- **outputs**: `I like foo and bar`

```
{{ "I like this and --that--." | replace({'this': foo, '--that--': "bar"}) }}
```

- **outputs**: `I like foo and bar`

### Arguments

- **from**: The placeholder values

  ```plaintext
  format
  ```
Chapter 55

reverse

New in version 1.6: Support for strings has been added in Twig 1.6.
The `reverse` filter reverses a sequence, a mapping, or a string:

```
{% for user in users|reverse %}
...
{% endfor %}
```

```
{{ '1234'|reverse }}
```

# outputs 4321

For sequences and mappings, numeric keys are not preserved. To reverse them as well, pass `true` as an argument to the `reverse` filter:

```
{% for key, value in {1: "a", 2: "b", 3: "c"}|reverse %}
  {{ key }}:
  {{ value }}
{% endfor %}
```

# output: 3: c 2: b 1: a

```
{% for key, value in {1: "a", 2: "b", 3: "c"}|reverse(true) %}
  {{ key }}:
  {{ value }}
{% endfor %}
```

# output: 3: c 2: b 1: a
It also works with objects implementing the `Traversable`\(^1\) interface.

**Arguments**

- `preserve_keys`: Preserve keys when reversing a mapping or a sequence.

---

1. https://secure.php.net/Traversable
New in version 1.15.0: The `round` filter was added in Twig 1.15.0.

The `round` filter rounds a number to a given precision:

```twig
{{ 42.55 | round }}
# outputs 43#

{{ 42.55 | round(1, 'floor') }}
# outputs 42.5#
```

The `round` filter takes two optional arguments; the first one specifies the precision (default is 0) and the second the rounding method (default is `common`):

- `common` rounds either up or down (rounds the value up to precision decimal places away from zero, when it is half way there -- making 1.5 into 2 and -1.5 into -2);
- `ceil` always rounds up;
- `floor` always rounds down.

The `//` operator is equivalent to `|round(0, 'floor')`.

**Arguments**

- **precision**: The rounding precision
- **method**: The rounding method
Chapter 57

slice

New in version 1.6: The slice filter was added in Twig 1.6.

The slice filter extracts a slice of a sequence, a mapping, or a string:

```twig
{% for i in [1, 2, 3, 4, 5]|slice(1, 2) %}
  (% will iterate over 2 and 3 %)
{% endfor %}

{{ '12345'|slice(1, 2) }}

(# outputs 23 #)

You can use any valid expression for both the start and the length:

```twig
{% for i in [1, 2, 3, 4, 5]|slice(start, length) %}
  (% # ... #)
{% endfor %}

# ... #

{{ '12345'|slice(1:2) }}

(# will display "23" #)

# you can omit the first argument -- which is the same as 0 #

{{ '12345'|slice(2:) }}

(# will display "345" #)

As syntactic sugar, you can also use the [] notation:

```twig
{% for i in [1, 2, 3, 4, 5]|array_slice %}
  (% # ... #)
{% endfor %}

{{ '12345'|array_slice(1:2) }}

(# will display "23" #)

# you can omit the first argument -- which is the same as 0 #

{{ '12345'|array_slice(2:) }}

(# will display "345" #)

# you can omit the last argument -- which will select everything till the end #

{{ '12345'|array_slice[:] }}

(# will display "12345" #)

The slice filter works as the array_slice\(^1\) PHP function for arrays and mb_substr\(^2\) for strings with a fallback to substr\(^3\).

1. https://secure.php.net/array_slice
2. https://secure.php.net/mb-substr
3. https://secure.php.net/substr
If the start is non-negative, the sequence will start at that start in the variable. If start is negative, the sequence will start that far from the end of the variable.

If length is given and is positive, then the sequence will have up to that many elements in it. If the variable is shorter than the length, then only the available variable elements will be present. If length is given and is negative then the sequence will stop that many elements from the end of the variable. If it is omitted, then the sequence will have everything from offset up until the end of the variable.

It also works with objects implementing the `Traversable` interface.

**Arguments**

- `start`: The start of the slice
- `length`: The size of the slice
- `preserve_keys`: Whether to preserve key or not (when the input is an array)

Chapter 58

sort

The `sort` filter sorts an array:

```twig
{% for user in users|sort %}
  ...
{% endfor %}
```

Internally, Twig uses the PHP `asort` function to maintain index association. It supports Traversable objects by transforming those to arrays.

---

1. [https://secure.php.net/asort](https://secure.php.net/asort)
New in version 1.38: The `spaceless` filter was added in Twig 1.38.

Use the `spaceless` filter to remove whitespace between HTML tags, not whitespace within HTML tags or whitespace in plain text:

```twig
{# output will be <div><strong>foo</strong></div> #}
```

You can combine `spaceless` with the `apply` tag to apply the transformation on large amounts of HTML:

```twig
{# output will be <div><strong>foo</strong></div> #}
```

The `apply` tag was introduced in Twig 1.40; use the `filter` tag with previous versions.

This tag is not meant to "optimize" the size of the generated HTML content but merely to avoid extra whitespace between HTML tags to avoid browser rendering quirks under some circumstances.
As the filter uses a regular expression behind the scenes, its performance is directly related to the text size you are working on (remember that filters are executed at runtime).

If you want to optimize the size of the generated HTML content, gzip compress the output instead.

If you want to create a tag that actually removes all extra whitespace in an HTML string, be warned that this is not as easy as it seems to be (think of `textarea` or `pre` tags for instance). Using a third-party library like Tidy is probably a better idea.

For more information on whitespace control, read the dedicated section of the documentation and learn how you can also use the whitespace control modifier on your tags.
New in version 1.10.3: The \texttt{split} filter was added in Twig 1.10.3.

The \texttt{split} filter splits a string by the given delimiter and returns a list of strings:

\begin{Verbatim}
\texttt{% set foo = \textquotesingle one,two,three\textquotesingle|split(\textquotesingle,\textquotesingle) %}
\texttt{# foo contains \{\textquotesingle one\textquotesingle, \textquotesingle two\textquotesingle, \textquotesingle three\textquotesingle\} #}
\end{Verbatim}

You can also pass a \texttt{limit} argument:

- If \texttt{limit} is positive, the returned array will contain a maximum of \texttt{limit} elements with the last element containing the rest of string;
- If \texttt{limit} is negative, all components except the last \texttt{-limit} are returned;
- If \texttt{limit} is zero, then this is treated as 1.

\begin{Verbatim}
\texttt{% set foo = \textquotesingle one,two,three,four,five\textquotesingle|split(\textquotesingle,\textquotesingle,3) %}
\texttt{# foo contains \{\textquotesingle one\textquotesingle, \textquotesingle two\textquotesingle, \textquotesingle three,four,five\textquotesingle\} #}
\end{Verbatim}

If the \texttt{delimiter} is an empty string, then value will be split by equal chunks. Length is set by the \texttt{limit} argument (one character by default).

\begin{Verbatim}
\texttt{% set foo = \textquotesingle 123\textquotesingle|split\textquotesingle\textquotesingle %}
\texttt{# foo contains \{\textquotesingle 1\textquotesingle, \textquotesingle 2\textquotesingle, \textquotesingle 3\textquotesingle\} #}
\texttt{% set bar = \textquotesingle aabbcc\textquotesingle|split\textquotesingle\textquotesingle,2\textquotesingle %}
\texttt{# bar contains \{\textquotesingle aa\textquotesingle, \textquotesingle bb\textquotesingle, \textquotesingle cc\textquotesingle\} #}
\end{Verbatim}

Internally, Twig uses the PHP \texttt{explode} or \texttt{str_split} (if delimiter is empty) functions for string splitting.
Arguments

- delimiter: The delimiter
- limit: The limit argument
The `striptags` filter strips SGML/XML tags and replace adjacent whitespace by one space:

```
Listing 61-1
1  {{ some_html|striptags }}
```

You can also provide tags which should not be stripped:

```
Listing 61-2
1  {{ some_html|striptags('<br><p>') }}
```

In this example, the `<br/>`, `<br>`, `<p>`, and `</p>` tags won't be removed from the string.

Internally, Twig uses the PHP `strip_tags`¹ function.

**Arguments**

- `allowable_tags`: Tags which should not be stripped

---

¹ https://secure.php.net/strip_tags
The **title** filter returns a titlecased version of the value. Words will start with uppercase letters, all remaining characters are lowercase:

```
{{ 'my first car'|title }}
```

(# outputs 'My First Car')
New in version 1.32: The side argument was added in Twig 1.32.

New in version 1.6.2: The trim filter was added in Twig 1.6.2.

The **trim** filter strips whitespace (or other characters) from the beginning and end of a string:

```python
{{ '  I like Twig.  ' | trim }}
# outputs 'I like Twig.'

{{ '  I like Twig.' | trim('.', 'left') }}
# outputs '  I like Twig'

{{ '  I like Twig.  ' | trim(' ', 'right') }}
# outputs '  I like Twig.'
```

Internally, Twig uses the PHP `trim`, `ltrim`, and `rtrim` functions.

---

1. https://secure.php.net/trim
2. https://secure.php.net/ltrim
3. https://secure.php.net/rtrim
Arguments

- `character_mask`: The characters to strip
- `side`: The default is to strip from the left and the right (*both*) sides, but `left` and `right` will strip from either the left side or right side only
Chapter 64

upper

The `upper` filter converts a value to uppercase:

```plaintext
Listing 64-1
1  {{ 'welcome'|upper }}
2  (# outputs 'WELCOME' #)
```
New in version 1.12.3: Support for encoding an array as query string was added in Twig 1.12.3.

New in version 1.16.0: The `raw` argument was removed in Twig 1.16.0. Twig now always encodes according to RFC 3986.

The `url_encode` filter percent encodes a given string as URL segment or an array as query string:

```twig
{# Listing 65-1 #}
{
  "path-segment"|url_encode
}

{# outputs "path-seg%2Ament" #}
{
  "string with spaces"|url_encode
}

{# outputs "string%20with%20spaces" #}
{
  {'param': 'value', 'foo': 'bar'}|url_encode
}

{# outputs "param=value&foo=bar" #}
```

Internally, Twig uses the PHP `urlencode`¹ (or `rawurlencode`² if you pass `true` as the first parameter) or the `http_build_query` function. Note that as of Twig 1.16.0, `urlencode` always uses `rawurlencode` (the `raw` argument was removed.)

---

1. https://secure.php.net/urlencode
2. https://secure.php.net/rawurlencode
Chapter 66

attribute

New in version 1.2: The attribute function was added in Twig 1.2.

The attribute function can be used to access a "dynamic" attribute of a variable:

```twig
1 {{ attribute(object, method) }}
2 {{ attribute(object, method, arguments) }}
3 {{ attribute(array, item) }}
```

In addition, the defined test can check for the existence of a dynamic attribute:

```twig
1 {{ attribute(object, method) is defined ? 'Method exists' : 'Method does not exist' }}
```

The resolution algorithm is the same as the one used for the . notation, except that the item can be any valid expression.
Chapter 67

block

New in version 1.28: Using **block** with the **defined** test was added in Twig 1.28. New in version 1.28: Support for the template argument was added in Twig 1.28.

When a template uses inheritance and if you want to print a block multiple times, use the **block** function:

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

The **block** function can also be used to display one block from another template:

```
{{ block("title", "common_blocks.twig") }}
```

Use the **defined** test to check if a block exists in the context of the current template:

```
{% if block("footer") is defined %}
    ...
{% endif %}
{% if block("footer", "common_blocks.twig") is defined %}
    ...
{% endif %}
```

*extends, parent*
New in version 1.12.1: constant now accepts object instances as the second argument.

New in version 1.28: Using `constant` with the `defined` test was added in Twig 1.28.

`constant` returns the constant value for a given string:

```
1 {{ some_date|date(constant('DATE_W3C')) }}
2 {{ constant('Namespace\Classname::CONSTANT_NAME') }}
```

As of 1.12.1 you can read constants from object instances as well:

```
1 {{ constant('RSS', date) }}
```

Use the `defined` test to check if a constant is defined:

```
1 (% if constant('SOME_CONST') is defined %)
2     ...
3 (% endif %)
The `cycle` function cycles on an array of values:

```
Listing 69.1
1 (% set start_year = date() | date('Y') %)
2 (% set end_year = start_year + 5 %)
3 (% for year in start_year..end_year %)
4   {% cycle(['odd', 'even'], loop.index0) %}
5 (% endfor %)
```

The array can contain any number of values:

```
Listing 69.2
1 (% set fruits = ['apple', 'orange', 'citrus'] %)
2 (% for i in 0..10 %)
3   {% cycle(fruits, i) %}
4 (% endfor %)
```

**Arguments**

- **position**: The cycle position
New in version 1.6: The date function has been added in Twig 1.6.

New in version 1.6.1: The default timezone support has been added in Twig 1.6.1.

Converts an argument to a date to allow date comparison:

```
{% if date(user.created_at) < date('-2days') %}
  # do something #
{% endif %}
```

The argument must be in one of PHP’s supported date and time formats\(^1\).

You can pass a timezone as the second argument:

```
{% if date(user.created_at) < date('-2days', 'Europe/Paris') %}
  # do something #
{% endif %}
```

If no argument is passed, the function returns the current date:

```
{% if date(user.created_at) < date() %}
  # always! #
{% endif %}
```

---

You can set the default timezone globally by calling `setTimezone()` on the `core` extension instance:

```php
Listing 70-4
$twig = new Twig\Environment($loader);
$twig->getExtension('Twig\Extension\CoreExtension')->setTimezone('Europe/Paris');

// before Twig 1.26
$twig->getExtension('core')->setTimezone('Europe/Paris');
```

Arguments

- **date**: The date
- **timezone**: The timezone
Chapter 71

dump

New in version 1.5: The `dump` function was added in Twig 1.5.

The `dump` function dumps information about a template variable. This is mostly useful to debug a
template that does not behave as expected by introspecting its variables:

```twig
{{ dump(user) }}
```

The `dump` function is not available by default. You must add the `\Twig\Extension\DebugExtension` extension explicitly when creating your Twig environment:

```php
$twig = new \Twig\Environment($loader,
    ['debug' => true,
    // ...])
$twig->addExtension(new \Twig\Extension\DebugExtension());
```

Even when enabled, the `dump` function won't display anything if the `debug` option on the
environment is not enabled (to avoid leaking debug information on a production server).

In an HTML context, wrap the output with a `pre` tag to make it easier to read:

```twig
<pre>
{{ dump(user) }}
</pre>
```
Using a `pre` tag is not needed when XDebug\(^1\) is enabled and `html_errors` is on; as a bonus, the output is also nicer with XDebug enabled.

You can debug several variables by passing them as additional arguments:

```
Listing 71-4
1  {{ dump(user, categories) }}
```

If you don't pass any value, all variables from the current context are dumped:

```
Listing 71-5
1  {{ dump() }}
```

Internally, Twig uses the PHP `var_dump`\(^2\) function.

**Arguments**

- `context`: The context to dump

---

1. https://xdebug.org/docs/display
2. https://secure.php.net/var_dump
Chapter 72

**include**

*New in version 1.12:* The `include` function was added in Twig 1.12.

The `include` function returns the rendered content of a template:

```twig
{{ include('template.html') }}
{{ include(some_var) }}
```

Included templates have access to the variables of the active context.

If you are using the filesystem loader, the templates are looked for in the paths defined by it.

The context is passed by default to the template but you can also pass additional variables:

```twig
# template.html will have access to the variables from the current context and the additional ones provided
{{ include('template.html', {foo: 'bar'}) }}
```

You can disable access to the context by setting `with_context` to `false`:

```twig
# only the foo variable will be accessible
{{ include('template.html', {foo: 'bar'}, with_context = false) }}

# no variables will be accessible
{{ include('template.html', with_context = false) }}
```

And if the expression evaluates to a `Twig\Template` or a `Twig\TemplateWrapper` instance, Twig will use it directly:

```twig
// {{ include(\Twig\Template) }}
```

When you set the `ignore_missing` flag, Twig will return an empty string if the template does not exist:
You can also provide a list of templates that are checked for existence before inclusion. The first template that exists will be rendered:

```
{{ include(['page_detailed.html', 'page.html']) }}
```

If `ignore_missing` is set, it will fall back to rendering nothing if none of the templates exist, otherwise it will throw an exception.

When including a template created by an end user, you should consider sandboxing it:

```
{{ include('page.html', sandboxed = true) }}
```

**Arguments**

- **template**: The template to render
- **variables**: The variables to pass to the template
- **with_context**: Whether to pass the current context variables or not
- **ignore_missing**: Whether to ignore missing templates or not
- **sandboxed**: Whether to sandbox the template or not
Chapter 73

max

New in version 1.15: The max function was added in Twig 1.15.

max returns the biggest value of a sequence or a set of values:

```
Listing 73-1
1 {{ max(1, 3, 2) }}
2 {{ max([1, 3, 2]) }}

When called with a mapping, max ignores keys and only compares values:

Listing 73-2
1 {{ max({2: "e", 1: "a", 3: "b", 5: "d", 4: "c"}) }}
2 (# returns "e" #)
```
New in version 1.15: The \texttt{min} function was added in Twig 1.15. \texttt{min} returns the lowest value of a sequence or a set of values:

\begin{verbatim}
Listing 74.1
1  {{ min(1, 3, 2) }}
2  {{ min([1, 3, 2]) }}
\end{verbatim}

When called with a mapping, \texttt{min} ignores keys and only compares values:

\begin{verbatim}
Listing 74.2
1  {{ min({2: "e", 3: "a", 1: "b", 5: "d", 4: "c"}) }}
2  (# returns "a" #)
\end{verbatim}
When a template uses inheritance, it's possible to render the contents of the parent block when overriding a block by using the `parent` function:

```plaintext
e{{ parent() }}
```

The `parent()` call will return the content of the `sidebar` block as defined in the `base.html` template.

`extends, block, block`
Chapter 76

random

New in version 1.5: The random function was added in Twig 1.5.
New in version 1.6: String and integer handling was added in Twig 1.6.
New in version 1.38: The "max" argument was added in Twig 1.38.

The random function returns a random value depending on the supplied parameter type:

- a random item from a sequence;
- a random character from a string;
- a random integer between 0 and the integer parameter (inclusive).
- a random integer between the integer parameter (when negative) and 0 (inclusive).
- a random integer between the first integer and the second integer parameter (inclusive).

```twig
{{ random(['apple', 'orange', 'citrus']) }}  # example output: orange
{{ random('ABC') }}  # example output: C
{{ random() }}  # example output: 15386094 (works as the native PHP mt_rand function)
{{ random(5) }}  # example output: 3
{{ random(50, 100) }}  # example output: 63
```

Arguments

- values: The values
- max: The max value when values is an integer
Returns a list containing an arithmetic progression of integers:

```
Listing 77-1 1 (% for i in range(0, 3) %)
2  {{ 1 }},
3 (% endfor %)
4 5 (# outputs 0, 1, 2, 3, #)
```

When step is given (as the third parameter), it specifies the increment (or decrement for negative values):

```
Listing 77-2 1 (% for i in range(0, 6, 2) %)
2  {{ 1 }},
3 (% endfor %)
4 5 (# outputs 0, 2, 4, 6, #)
```

Note that if the start is greater than the end, `range` assumes a step of -1:

```
Listing 77-3 1 (% for i in range(3, 0) %)
2  {{ 1 }},
3 (% endfor %)
4 5 (# outputs 3, 2, 1, 0, #)
```

The Twig built-in .. operator is just syntactic sugar for the `range` function (with a step of 1, or -1 if the start is greater than the end):

```
Listing 77-4 1 (% for i in 0..3 %)
2  {{ 1 }},
3 (% endfor %)
```
The `range` function works as the native PHP `range` function.

Arguments

- `low`: The first value of the sequence.
- `high`: The highest possible value of the sequence.
- `step`: The increment between elements of the sequence.

1. https://secure.php.net/range
Chapter 78

source

New in version 1.15: The source function was added in Twig 1.15.

New in version 1.18.3: The ignore_missing flag was added in Twig 1.18.3.

The source function returns the content of a template without rendering it:

```twig
{{ source('template.html') }}
{{ source(some_var) }}
```

When you set the ignore_missing flag, Twig will return an empty string if the template does not exist:

```twig
{{ source('template.html', ignore_missing=true) }}
```

The function uses the same template loaders as the ones used to include templates. So, if you are using the filesystem loader, the templates are looked for in the paths defined by it.

Arguments

- name: The name of the template to read
- ignore_missing: Whether to ignore missing templates or not
New in version 1.11: The `template_from_string` function was added in Twig 1.11.

New in version 1.39: The name argument was added in Twig 1.39.

The `template_from_string` function loads a template from a string:

```twig
{% include template_from_string("Hello {{ name }}") %}
{% include template_from_string(page.template) %}
```

To ease debugging, you can also give the template a name that will be part of any related error message:

```twig
{% include template_from_string(page.template, "template for page ~ page.name") %}
```

The `template_from_string` function is not available by default. You must add the `Twig\Extension\StringLoaderExtension` extension explicitly when creating your Twig environment:

```php
$twig = new Twig\Environment(...);
$twig->addExtension(new Twig\Extension\StringLoaderExtension());
```

Even if you will probably always use the `template_from_string` function with the `include` function, you can use it with any tag or function that takes a template as an argument (like the `embed` or `extends` tags).
Arguments

- **template**: The template
- **name**: A name for the template
Chapter 80

constant

New in version 1.13.1: constant now accepts object instances as the second argument.

constant checks if a variable has the exact same value as a constant. You can use either global constants or class constants:

Listing 80-1

```%
if post.status is constant('Post::PUBLISHED') %]
  the status attribute is exactly the same as Post::PUBLISHED
%
```

You can test constants from object instances as well:

Listing 80-2

```%
if post.status is constant('PUBLISHED', post) %]
  the status attribute is exactly the same as Post::PUBLISHED
%
```
defined checks if a variable is defined in the current context. This is very useful if you use the `strict_variables` option:

```
{# defined works with variable names #}
{% if foo is defined %}
...
{% endif %}

{# and attributes on variables names #}
{% if foo.bar is defined %}
...
{% endif %}

{% if foo['bar'] is defined %}
...
{% endif %}
```

When using the `defined` test on an expression that uses variables in some method calls, be sure that they are all defined first:

```
{% if var is defined and foo.method(var) is defined %}
...
{% endif %}
```
Chapter 82

divisible by

New in version 1.14.2: The divisible by test was added in Twig 1.14.2 as an alias for divisibleby. divisible by checks if a variable is divisible by a number:

```
Listing 82-1 1  (% if loop.index is divisible by(1) %)
2  ...
3  (% endif %)
```
New in version 1.33: Support for the \_\_toString() magic method has been added in Twig 1.33.

empty checks if a variable is an empty string, an empty array, an empty hash, exactly \texttt{false}, or exactly \texttt{null}.

For objects that implement the \texttt{Countable} interface, \texttt{empty} will check the return value of the \texttt{count()} method.

For objects that implement the \_\_toString() magic method (and not \texttt{Countable}), it will check if an empty string is returned.

```twig
{% if foo is empty %}
...
{% endif %}
```
even returns true if the given number is even:

```
Listing 84-1 1  {{ var is even }}
```

odd
New in version 1.7: The iterable test was added in Twig 1.7.

`iterable` checks if a variable is an array or a traversable object:

```
{% # evaluates to true if the foo variable is iterable # %
{% if users is iterable %}
    {% for user in users %}
        Hello {{ user }}!
    {% endfor %}
{% else %}
    {# users is probably a string #}
    Hello {{ users }}!
{% endif %}
```
null returns `true` if the variable is `null`:

```java
{{ var is null }}
```

`none` is an alias for `null`. 
odd returns true if the given number is odd:

```markdown
Listing 87.1 1 {{ var is odd }}

   even
```
New in version 1.14.2: The `same as` test was added in Twig 1.14.2 as an alias for `sameas`.

`same as` checks if a variable is the same as another variable. This is equivalent to `===` in PHP:

```twig
{% if foo.attribute is same as(false) %}
the foo attribute really is the 'false' PHP value
{% endif %}
```