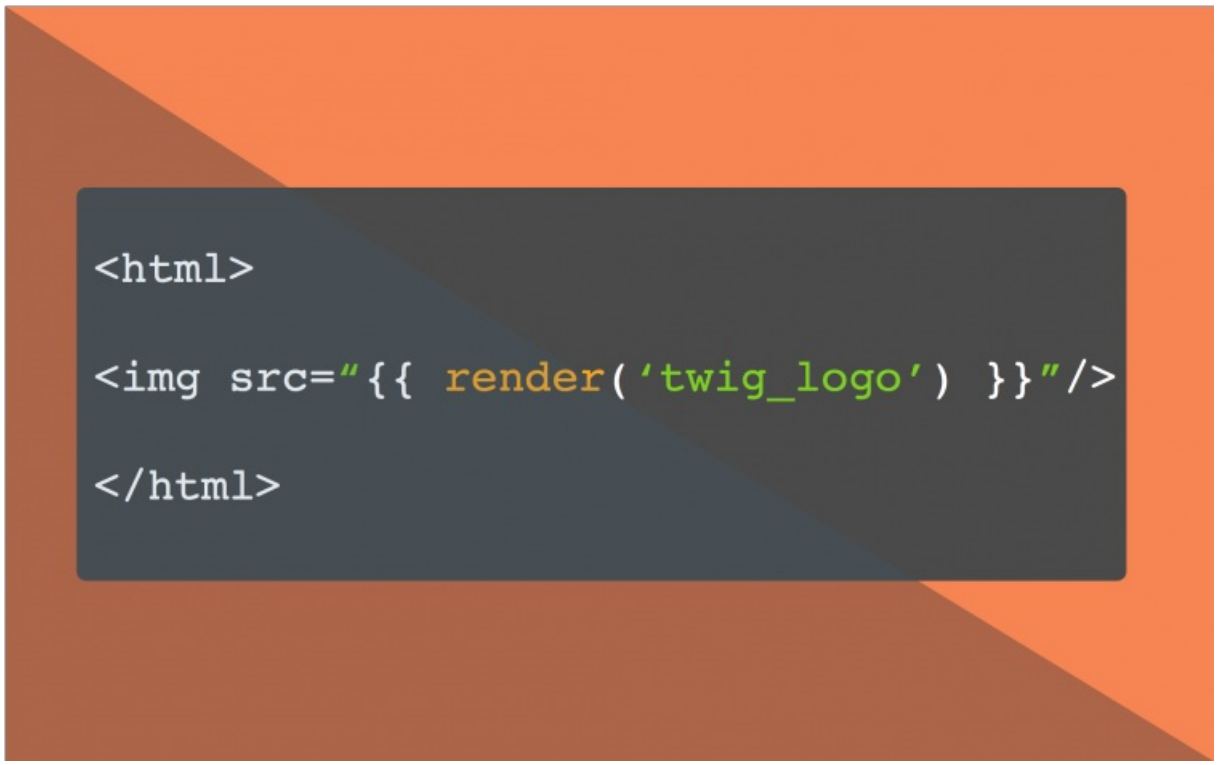
Twig Templating for Friendly Frontend Devs



With <3 from SymfonyCasts

Chapter 1: Twig: The Basics

Welcome to the world of Twig! Twig is a templating language for PHP, which is a boring way of saying that it's a tool used to output variables inside HTML. If a project you're working on uses Twig, then you're in luck: it's easy to learn, powerful and a joy to work with.

To make this interesting, let's build something useful with Twig like a penguin clothing store! Actually, I've already got us started. I have a small website setup under my web server's document root and a test page called `test.php`.

Tip

Want to run this code locally? Download the code from this page, then following the directions in the `README.md` file inside.

Right now, this file prepares some `pageTitle` and `products` variables and then includes another file:

```
// test.php

// setup some variables
$pageTitle = 'Suit Up!';
$products = array(
    new Product('Serious Businessman', 'formal.png'),
    new Product('Penguin Dress', 'dress.png'),
    new Product('Sportstar Penguin', 'sports.png'),
    new Product('Angel Costume', 'angel-costume.png'),
    new Product('Penguin Accessories', 'swatter.png'),
    new Product('Super Cool Penguin', 'super-cool.png'),
);

// render out PHP template
include __DIR__.'/templates/homepage.php';
```

The `homepage.php` file is the actual template. It has all the HTML and we use `foreach` to loop through them and then `echo` to print out some variables:

```
<!-- templates/homepage.php -->

<!-- ... the rest of the HTML page ... -->
<?php foreach ($products as $product) : ?>
    <div class="span4">
        <h2><?php echo $product->getName() ?></h2>
        <!-- ... -->
    </div>
<?php endforeach; ?>
```

Instead of using PHP, let's write our template using Twig! The goal of the template is still the same: to print out variables. The only thing that will change is the syntax.

Setting up Twig

In a separate file, I've setup all the behind-the-scenes work to use Twig. Let's start by rendering a `homepage.twig` file and once again passing it `pageTitle` and `products` variables:

```
// index.php
// ... code that sets up Twig, and says to look for templates in template/

echo $twig->render('homepage.twig', array(
    'pageTitle' => 'Welcome to Penguins R Us!',
    'products' => array(
        'Tuxedo',
        'Bow tie',
        'Black Boxers',
    ),
));
```

> If you're curious how you actually setup Twig, check out the code download and see the Twig Installation documentation.

If you're a frontend developer, then you don't need to worry about this step: all you need to know is where a Twig template is located and what variables you have access to.

Your first Twig Template

In our project, Twig is looking for the template files in a `templates/` directory, so let's create our `homepage.twig` there!

Just like in PHP, you can write anything and it'll just be displayed as HTML on the page:

```
<!-- templates/homepage.twig -->
Hello Twig Viewers!
```

To see this amazing message, go to the `index.php` file in your browser. This works because we made the `index.php` file render the `homepage.twig` template. Whenever you're creating or editing a page, you'll need to figure out which Twig template is being used for that page. There's no exact science to this and it depends on how your application is built.

Rendering a Variable

Remember that we're passing a `pageTitle` variable to our template. To render it, write two opening curly braces, the name of the variable without a dollar sign, then two closing curly braces:

```
<!-- templates/homepage.twig -->
<h1>{{ pageTitle }}</h1>
```

When we refresh the page, it works! We've just written our first line of Twig! Whenever you want to print something, just open Twig with two curly braces, write the variable name, then close Twig. We'll get fancier in a little while with some things called functions and filters, but this is the most fundamental syntax in Twig.

Looping over Variables

Next, the `products` variable is an array that we need to loop through. Twig comes with a for tag that is able to loop through items just like PHP's `foreach`.

Remember that anything we type here will be printed out raw on the page until we "open up" Twig. This time, open Twig by typing `{%`. Now that we're in Twig, use the `for` tag to loop over `products`. `product` will be the variable name we use for each item as we loop. Close Twig by adding an identical `%}`. Unlike when we echo'ed the `pageTitle` variable, the `for` tag needs an `endfor`:

```
<!-- templates/homepage.twig -->
<h1>{{ pageTitle }}</h1>

<div class="row">
    {% for product in products %}

    {% endfor %}
</div>
```

Twig will loop over each item in `products` and execute each line between `for` and `endfor`. Each item in `products` is just a string, so let's print it out:

```
<!-- templates/homepage.twig -->
<h1>{{ pageTitle }}</h1>

<div class="row">
    {% for product in products %}
        <div class="span4">
            <h2>{{ product }}</h2>
        </div>
    {% endfor %}
</div>
```

This works exactly like before. We have a `product` variable, so we can print it by placing it inside two opening curly braces and two closing curly braces.

And when we refresh, another Twig success! Before long, we'll have these penguins looking fly.

The 2 Syntaxes of Twig: `{{` and `{%`

So we've seen how to print a variable and how to loop over a variable that's an array or collection. This may not seem like much, but you've already seen pretty much all of Twig's syntaxes! To start writing Twig code in your HTML, there are only two different syntaxes:

- `{{ }}` The "say something" syntax
- `{% %}` The "do something" syntax

The "Say Something" Syntax: `{{ ... }}`

The double-curly-brace (`{{`) is always used to print something. If whatever you need to do will result in something being printed to the screen, then you'll use this syntax. I call this the "say something" tag, ya know, because it's how you "speak" in Twig.

The "Do Something" Syntax: `{% ... %}`

The curly-percent (`{%`) is the other syntax, which I call the "do something" syntax. It's used for things like if and for tags as well as other things that "do" something. The `{%` is really easy because there are only a handful of things that can be used inside of it. If you go to Twig's website click Documentation, and scroll down, you can see a full list of everything in Twig. The "tags" header shows you everything that can be used inside of a "do something" tag, with more details about how each of these works. The only ones you need to worry about now are if and for. We'll talk about a bunch more of these later.

And that's it! Use the `{{` "say something" syntax to print and the `{%` "do something" when you want to do one of the things on this list. These are the only two Twig syntaxes and we'll learn more tools that can be used inside of each of these.

The Comment Syntax: `{# ... #}`

Actually, we've lied a little. There is a third syntax, used for comments: `{#`. Just like with the "say something" and "do something" syntaxes, write the opening `{#` and also the closing `#}` at the end of your comments:

```
{# This template is really starting to get interesting ... #}
{# ... #}
```

Tip

We'll use the `{# ... #}` syntax in the rest of this tutorial whenever we're hiding some parts of a Twig template.

Whitespace inside Twig

Inside Twig, whitespace doesn't matter. this means that we can add or remove spaces whenever we want:

```
{%for product    in      products%}
    <div class="span4">
        <h2>{{product}}</h2>
    </div>
{% endfor %}
```

Of course, this looks a bit uglier, so we usually keep just one space between everything. Outside of Twig (in the final HTML), all the whitespace is kept just like it appears. There are ways to make Twig control the whitespace of your file, which we'll talk about later.

Chapter 2: Functions, Filters and Debugging with dump

Just like PHP or JavaScript, Twig has functions that can be used once we're inside either a say something or do something Twig delimiter. To see the built-in functions, once again check out the bottom of the Twig Documentation page. In your application - especially if you're using Twig inside something like Symfony or Drupal - you may have even more functions available to you. Fortunately, the syntax to use a function is always the same: just check out your project's documentation to see what other goodies you have.

Using a Function

Let's check out the random() function... which gives us, shockingly, a random number! If we want to print out that random number, we can do it using the "say something" syntax:

```
{# templates/homepage.twig #}

<div class="price">
    {{ random() }}
</div>
```

You can also use a function inside a "do something" tag, like checking to see if the random number is less than 5:

```
{# templates/homepage.twig #}

<div class="price">
    {{ random() }}

    {# random(10) returns a number from 0 to 10 #}
    {% if random(10) < 5 %}
        Really cheap!
    {% endif %}
</div>
```

This works exactly the same, except that it lives inside the `if` statement. Like we've done here, a function can have zero or many arguments, each separated by a comma.

Using Filters

This is nice, but Twig has something even cooler: filters! Like with everything, you can find a filter list in the Twig Documentation, though you may have even more in your project. Let's check out the upper filter. We can use the upper filter to uppercase each product by placing a pipe (`|`) to the right of the variable and using the filter:

```
{# ... #}
<h2>{{ product|upper }}</h2>
```

This works just like a function, except with a different syntax. Whatever you have to the left of the pipe is passed to the filter, which in this example, upper-cases everything.

Heck, you can use as many filters as you want!

```
{# ... #}
<h2>{{ product|upper|reverse }}</h2>
```

Now we're upper-casing the name and then reversing the text.

Using Functions, Filters and Math

Filters can be used after functions too. Instead of printing out the random number, let's divide it by 100 to get a decimal, then use the number_format to show only one decimal place:

```
{{ (random() / 100)|number_format(1) }}
```

Getting the Length of a Collection

In fact, functions and filters can be used anywhere. Let's use the length filter to print a message if there are no penguin products for sale:

```
{% if products|length == 0 %}
    <div class="alert alert-error span12">
        It looks like we're out of really awesome-looking penguin clothes :/.
    </div>
{% endif %}
```

This filter takes an array or collection to the left and transforms it into a number, which represents the number of items in the collection. We can use this to see if there are no products. I'll temporarily pass in zero so we can check this out.

Filters and Arguments using "date"

Just like functions, sometimes a filter has one or more arguments. A really common filter is date. This takes a date string or PHP DateTime object and changes it into a string. We can go to date() to look up the letters used in the date format. To try this out, we can just hardcode a string to start:

```
{# templates/homepage.twig #}

<div class="sale-ends-at">
    {{ 'tomorrow noon'|date('D M jS ga') }}
</div>
```

The "tomorrow noon" part is just a valid input to PHP's strtotime() function, which accepts all sorts of interesting strings as valid dates. The Twig date filter takes that string and renders it in the new format that we want. Of course, we can also send a variable through the date filter. Let's pass in a `saleEndsAt` variable into the template and render it the same way:

```
// index.php
// ...

echo $twig->render('homepage.twig', array(
    // ...
    'saleEndsAt' => new \DateTime('+1 month')
));
```

```
{# templates/homepage.twig #}

<div class="sale-ends-at">
    {{ saleEndsAt|date('D M jS ga') }}
</div>
```

We can even use the date filter to print out the current year. For the value to the left of the filter, I'll use `now`. I'll use the `Y` string to print out the 4-digit year. Sweet!

```
{{ 'now'|date('Y') }}
```

Use functions and especially filters to do cool stuff in Twig, and look at the documentation for each to see if what you're using has any arguments.

The dump Function for Debugging

Before we move on, let's talk about the dump() function. If you don't know what a variable looks like, use the `dump()` function to see its details:

```
{{ dump(products) }}
```

Even better, use the `dump()` function with no arguments to see all the variables that you have access to:

```
{{ dump() }}
```

With this function, there's not much you won't be able to do!

We experimented a lot in this section. I'll use the `{#` syntax to comment out some of the things we've done so that our page makes a bit more sense.

> To clean things up, we removed the `upper` and `reverse` filters, the entire spot where we print the random numbers, and the printing of the current year.

Chapter 3: Using Objects and Array Keys

Until now, we've been working with simple values like `pageTitle` or `products`, which is an array that contains simple values where we loop over and print each out. Now, let's make things a bit more interesting!

Using data from an Array

I'm going to pass in a new variable called `pageData`:

```
// index.php
// ...

echo $twig->render('homepage.twig', array(
    'pageData' => array(
        'title'     => 'Suit Up!',
        'summary'   => "You're hip, you're cool, you're a penguin! Now, start dressing like one! Find the latest suits, bow-ties,
swim shorts and other outfits here!",
        'hasSale'   => true,
    ),
    // ...
));
```

If you hadn't seen the PHP code I just used to create this variable, you could use the handy dump function to see that it's an array with a `title`, `summary` and `hasSale` keys:

```
{{ dump(pageData) }}
```

So how can we get to the data on the keys of the array?

The answer is with the almighty period (`.`). To print the `title`, just say `pageData.title`. To print the summary, use the same trick!

```
<div class="hero-unit">
    <h1>{{ pageData.title }}</h1>
    <p>
        {{ pageData.summary }}
    </p>
</div>
```

This can be used anywhere, like in an `if` statement in exactly the same way:

```
{% if pageData.hasSale %}
    <div>We're having a sale!</div>
{% endif %}
```

So if you ever need data from an array, the `.` operator is your answer!

The much-rarer `[]` Syntax

The `products` variable is also an array, but since it's a collection of items, we loop over it with the for tag instead. But if we did need to manually get the first item, or "zero" key from the array, we can do that. If you're thinking that you would say `products.0`, you're right!

```
{{ products.0 }}
```

You may sometimes see another syntax for getting items from an array:

```
{{ products[0] }}
```

Don't let this confuse you - you almost *always* want to use the period. The square bracket syntax is only needed in some uncommon cases when you need to use a variable as the key:

```
{{ products[random(5)] }}
```

Tip

The `[]` is used if you want force getting the attribute off of an object like an array, instead of trying to access the property. That's a very rare case, so don't worry about it.

Getting Data from an Object

I'm going to complicate things again by changing what the `products` variable looks like. But first, use our friend the dump function to see that `products` is just a collection of strings right now:

```
{{ dump(products) }}
```

Now, I'll change the `products` variable:

```
// index.php
// ...

echo $twig->render('homepage.twig', array(
    // ...
    'products' => array(
        new Product('Serious Businessman', 'formal.png'),
        new Product('Penguin Dress', 'dress.png'),
        new Product('Sportstar Penguin', 'sports.png'),
        new Product('Angel Costume', 'angel-costume.png'),
        new Product('Penguin Accessories', 'swatter.png'),
        new Product('Super Cool Penguin', 'super-cool.png'),
    ),
));
```

After my change, refresh the page to see that `products` is now a collection of `Product` objects. Each `Product` object has a `name` and `imagePath` property.

If we don't change anything inside Twig, we'll get an error:

> Catchable fatal error: Object of class Product could not be converted to string in twig/vendor/twig/twig/lib/Twig/Environment.php(320) : eval()'d code on line 30

This means that we can print a string, but not an object. That makes sense. Each Product object has `name` and `imagePath` properties, and we really want to print those individually.

Tip

If an object has a `__toString()` method, then it actually *can* be printed.

And guess what?! We can use the period character once again to do this! Even though `pageData` is an array and each `product` is an object, getting data off each is exactly the same:

```
{% for product in products %}
    <div class="span4">
        <h2>{{ product.name }}</h2>
        <div class="product-img">
            <img src="assets/images/{{ product.imagePath }}" class="img-rounded" />
        </div>
    </div>
{% endfor %}
```

Refresh the page to see that our products have more details!

Tip

In your project, you'll likely have a Twig function or variable that you use when referring to static images, CSS or JS files. Check your documentation to see.

Alright! By using the `dump()` function, we can see what a variable looks like. We can print it, loop over it, or print a child key or property for it. We're dangerous like a killer whale!

For the more technical folk, behind the scenes, Twig checks to see if the Product class has a public `name` property. If the property doesn't exist or isn't public, it looks for a `getName` method and calls it to get the value. This lets us say `product.name` without really caring how the PHP code for the class looks.

Tip

You can also call a method on an object if you need to:

```
{{ product.getName() }}
```

Chapter 4: Using a Layout: Template Inheritance

If we view the HTML source of our project so far, we'll see *just* the HTML tags and printed variables from our `homepage.twig` file. So far, there's no HTML layout, head or body tags. but since our project has been ugly long enough, let's add these.

To add the layout, there's nothing technically wrong with including it right in `homepage.twig`. This is perfectly straightforward and has nothing to do with Twig. But let's setup a second page: `/contact`. I'll tweak our setup script to make this work:

```
// index.php
// ...

case '/contact':
    echo $twig->render('contact.twig', array(
        'pageData' => array(
            'title' => 'Find us in the south pole!',
        )
    ));
    break;
```

Create the new `contact.twig` file in the `templates/` directory and add `/contact` to the end of your URL:

```
{# templates/contact.twig #}

<h1>{{ pageData.title }}</h1>

<p>Make some penguin noises, we're listening...</p>
```

```
http://localhost/twig/index.php/contact
```

Ok, so now that we have two pages, if we put the layout code in `homepage.twig`, it wouldn't be available on the contact page. Since that would be a bummer, let's learn how to use a layout across multiple templates.

Template Inheritance

The key is template inheritance, which is a lot less scary than it sounds. Before we get into the details, let's create a new Twig template that will hold the base layout. I'll paste in some starting HTML, which you can get from the code download. I'm including some CSS and JS files, but this is pretty boring overall.

Actually, there's only one small piece of Twig code: a block tag. This defines a block called "body" and its purpose will make more sense in a moment:

```
{# templates/layout.twig #}
<!DOCTYPE html>
<html lang="en">
{# ... more stuff ... #}

{% block body %}{% endblock %}

{# ... the rest ... #}
```

Start by adding another Twig tag via the "do something" syntax called extends. After `extends`, type `layout.twig`.

```
{# templates/homepage.twig #}
{% extends 'layout.twig' %}

{# ... the rest of the template #}
```

This tells Twig that we want to be "dressed" in `layout.twig`, or said differently, that we want to use `layout.twig` as the layout for the homepage. We also need to surround all of our content with a `{% block body %}` tag and a closing `{% endblock %}`.

This looks just like what we have in the layout file, except with our content inside.

```
{# templates/homepage.twig #}
{% extends 'layout.twig' %}

{% block body %}
    <div class="hero-unit">
        {# ... #}
    </div>

    <div class="row">
        {# ... #}
    </div>
{% endblock %}
```

When we refresh the page, it works! By viewing the source, we can see the HTML layout with the content of the `homepage.twig` file right in the middle where we expect it.

Teamwork: extends and block

This works because of a great team effort between the extends and block tags. When we use `extends`, it says that this template should be placed inside of `layout.twig`. But Twig is a bit dumb: it doesn't really know *where* to put the content from the homepage. The `block` tag fixes that. By putting a block `body` in the layout *and* a block `body` around our homepage content, Twig knows exactly where the content should live in the layout.

Using Multiple Blocks

We can even use multiple blocks. Let's add a `title` block to the layout:

```
{# templates/layout.twig #}

<title>{% block title %}{% endblock %}</title>

{# ... #}
```

If we refresh, the title is blank. But now, we can add a `title` block to our homepage:

```
{# templates/homepage.twig #}
{% extends 'layout.twig' %}

{% block title %}50% off of Bow Ties{% endblock %}

{% block body %}
    {# ... #}
{% endblock %}
```

The order of the blocks doesn't matter, whatever lives in the `title` block will be placed in the `title` block of the layout. The same is true of any block. Even the names `title` and `body` aren't special. If we rename `body` to `content`, we just need to also rename the block in any other templates.

Tip

If you want to *add* to the content of the parent block instead of completely replacing it, use the parent() function:

```
{% block title %}
    Contact us | {{ parent() }}
{% endblock %}
```

Common Mistake: Content outside of a Block

Let's try to write something outside of a block in `homepage.twig`:

```
{# templates/homepage.twig #}
{% extends 'layout.twig' %}

Where should this text be placed in the layout?

{% block title %}50% off of Bow Ties{% endblock %}

{% block body %}
  {# ... #}
{% endblock %}
```

When we refresh, we see a nasty error:

> Uncaught exception `Twig_Error_Syntax` with message "A template that extends another one cannot have a body in `homepage.twig` at line 4."

Twig knows that we want it to take the content from the `body` tag of homepage and put it where the `body` tag is in the layout. But when it sees this new text, it doesn't know what to do with that or where to put it! The error is saying that if we extend another template, everything must live in a block so that Twig knows where to put that content in the layout.

Adding the Layout to the Contact Page

Our homepage looks great, but the contact page still needs a layout. To give it one, just add the `extends` tag, then surround the content with a block called `body`, since that's the name of the block in our layout:

```
{# templates/contact.twig #}
{% extends 'layout.twig' %}

{% block body %}
  <h1>{{ pageData.title }}</h1>

  {# ... #}
{% endblock %}
```

And just like that, we have a real page!

Default Content in a block

Of course the contact page doesn't have a title. We could add a `title` block just like we did on the homepage. Instead, in the layout, we can put some content inside of the title block:

```
{# templates/layout.twig #}
{# ... #}

<title>{% block title %}Penguin Swag{% endblock %}</title>
```

This becomes the default page title, which is used on the contact page since we don't have a `title` block in `contact.twig`. But when we go to the homepage, we still see the title from the `title` block in `homepage.twig`.

Template Inheritance, a Summary!

Phew! Let's review everything we just learned:

- Initially, a Twig template doesn't render anything other than what's actually *in* that template file.
- To use a layout, we use the extends tag at the top of the template and then surround *all* of our content in a block. Because the template and the layout have blocks with the same names, Twig takes the content from each block and puts it into the layout to build the whole page.
- In the layout, a block can also have some default content. Because `contact.twig` doesn't have a `title` block, the default text is used.

Chapter 5: Including other Templates

Using a base layout is very common, and we've just mastered that! Now suppose that we have a small "sales" banner that we want to include on both the homepage and the contact page, because penguins care about saving some krill too. If we wanted it on every page, then we should put it in our layout, but pretend for now that we only need it on these 2 pages.

Using the include Function

To avoid duplication, create a new template that will hold the sales banner. The filename doesn't matter, but I often prefix these files with an underscore to show that they only contain a small page fragment, not a whole page:

```
{# templates/_banner.twig #}

<div class="well">
    <h3>Save some Krill!</h3>

    <p>Sale in summer suits all this weekend!</p>
</div>
```

To include this on the homepage we can use the include() function. We use this in a say something syntax because `include()` renders the other template, and we want to print its content:

```
{# templates/homepage.twig #}

{% block body %}
    {{ include('_banner.twig') }}

    {# ... #}
{% endblock %}
```

Let's add the same code to `contact.twig` and refresh to make sure that our big sales banner is showing up. Cool!

Passing Variables

We can also access our variables from within the included template. Since both pages have a `pageData` variable, we can use it from within the included template:

```
{# templates/_banner.twig #}

<div class="well">
    <h3>Save some Krill!</h3>

    <p>Sale in summer suits all this weekend! {{ pageData.title }}</p>
</div>
```

You can also pass extra variables to the template. The include() function takes two arguments: the name of the template to include and a collection of additional variables to pass. These variables are a key-value list of names and their values.

In Twig, a key-value array is called a "hash", and uses a syntax that's just like JavaScript or JSON (i.e. `{"foo": "bar"}`). Let's pass a `backgroundColor` variable into the template and use it.

```
{# templates/homepage.twig #}

{% block body %}
    {{ include('_banner.twig', { 'backgroundColor': 'violet' }) }}

    {# ... #}
{% endblock %}
```

```
{# templates/_banner.twig #}

<div class="well" style="background-color: {{ backgroundColor }};">
    <h3>Save some Krill!</h3>

    <p>Sale in summer suits all this weekend! {{ pageData.title }}</p>
</div>
```

When we refresh, we see a beautiful purple background.

Chapter 6: If Statements with "Tests"

But when we try the contact page, it blows up

> Uncaught exception `Twig_Error_Runtime` with message "Variable `backgroundColor` does not exist in `_banner.twig` at line 5"

We're not passing a `backgroundColor` in the `include` call in `contact.twig`, so Twig gets angry! We can of course pass this variable to the template. But instead, let's check to see if the variable is defined in the `_banner.twig` template, and default to `lightblue` if it is not. To do this, we can use an `if` statement and a special defined test.

```
{# templates/_banner.twig #}

{% if backgroundColor is defined %}
    {# ... do something here #}
{% endif %}
```

Let's also see another new Twig tag called set, which sets a new variable.

```
{# templates/_banner.twig #}

{% if backgroundColor is defined %}
    {% set backgroundColor = backgroundColor %}
{% else %}
    {% set backgroundColor = 'lightblue' %}
{% endif %}

<div class="well" style="background-color: {{ backgroundColor }};">
    <h3>Save some Krill!</h3>

    <p>Sale in summer suits all this weekend! {{ pageData.title }}</p>
</div>
```

Try out both pages in the browser. Awesomesauce!

Let's look again at the `if background is defined`. Normally an `if` statement says something like `if foo != 'bar'` or `if isPublished == true`. That all works totally fine in Twig.

But in addition to `==` and `!=` and the others, you can say `is` and follow that by a word like defined, even, odd, empty or several other words. These are called tests, and they're listed once again right back on the main Documentation page of the Twig website.

Go Deeper!

A list of all of the operators (e.g. `/`, `*`, `==`, etc) can be found on the Twig Documentation.

For example, instead of using the `length` filter and seeing if the number of items in the `products` collection is zero, we could say `if products is empty`:

```
{# template/homepage.twig #}
{# ... #}

{% if products is empty %}
    <div class="alert alert-error span12">
        It looks like we're out of really awesome-looking penguin clothes :/.
    </div>
{% endif %}
```

If for some reason we wanted to know if the total number of products were even, we could use the length filter to

get the number, then check that the number is "even" by using the even test:

```
{% if products|length is even %}
    <p>
        There are an even number of products
    </p>
{% endif %}
```

The tests are easy to use and can shorten the code needed to do some things, so don't forget about them!

Negating a Test

You can also negate a test by using the not keyword. We can use this to simplify our code from earlier:

```
{# templates/_banner.twig #}

{% if backgroundColor is not defined %}
    {% set backgroundColor = 'lightblue' %}
{% endif %}

<div class="well" style="background-color: {{ backgroundColor }};">
    <h3>Save some Krill!</h3>

    <p>Sale in summer suits all this weekend! {{ pageData.title }}</p>
</div>
```

Awesome! At this point, you know a lot of tools in Twig. Let's keep going and learn some more.

Chapter 7: The for "loop" and inline "if" Syntax

Let's give ourselves a challenge! Our products are printing out a bit weird right now because they're floating but not breaking correctly. To fix this, we need to wrap every three products in their very own `row` div.

To do this, we can use a divisible by() test to see if the item number we're on is divisible by three:

```
<div class="row">
    {% for product in products %}
        <div class="span4">
            {# ... #}
        </div>

        {% if loopNumber is divisible by(3) %}
            </div><div class="row">
        {% endif %}
    {% endfor %}
</div>
```

> Just like functions and filters, sometimes a "test" also takes one or more arguments.

When we refresh, Twig is sad because the `loopNumber` variable is undefined. Yep, that's totally my fault, I made up that variable out of thin air. So how *can* we figure out how many items into the loop we are?

The Magical loop Variable

Twig comes to the rescue here and lets us say `loop.index`.

```
{% for product in products %}
    {# ... #}

    {% if loop.index is divisible by(3) %}
        </div><div class="row">
    {% endif %}
{% endfor %}
```

When we refresh, things work awesomely! So where did this magical loop variable come from? Normally in Twig, we have access to a few variables that were passed to us and that's it. If we use an undefined variable, we see an error.

This is all 100% true. But when we're inside a for tag, we magically have access to a new variable called loop. `loop.first` and `loop.last` tell us if this is the first or last item in the collection while `loop.index` counts up 1, 2, 3, 4 and so on for each item. Twig has a lot of really slick features like this, which you can find out by reading further into its docs.

In fact, to avoid an extra row being added if we have exactly 3, 6 or 9 objects, let's *not* print a new `row` if we're on the last item:

```
{% for product in products %}
    {# ... #}

    {% if loop.index is divisible by(3) and not loop.last %}
        </div><div class="row">
    {% endif %}
{% endfor %}
```

And not that it matters for Twig, but let's also move our "even" products message into its own row where it belongs.

```
{# templates/homepage.twig #}
{# ... after the for loop #}

{% if products|length is even %}
    <div class="row">
        <div class="span12">
            There is an even number of products! OMG!
        </div>
    </div>
{% endif %}
```

When we refresh, everything looks good and clean!

The for-else tag

While we're talking about cool `for` loop features, let's see another one: the `for-else` trick. Instead of seeing if `products` is empty, we can add an `else` tag inside of the `for` loop.

```
{% for product in products %}
    {# ... #}
{% else %}
    <div class="alert alert-error span12">
        It looks like we're out of really awesome-looking penguin clothes :/.
    </div>
{% endfor %}
```

If `products` is empty, it skips the `for` loop and calls the `else` section instead. When we try it, it still works great.

The inline if Syntax

Finally, let's see a really short syntax you can choose to use instead of the classic `if` tag. Head back to the banner template where we're setting the `backgroundColor` variable if it's not set and then printing it. Let's remove all of this and instead put all the logic in the "say something" block:

```
<div class="well" style="background-color: {{ backgroundColor is defined ? backgroundColor : 'lightBlue' }};">
    {# ... #}
</div>
```

You may be familiar with this syntax from another language, but if you're not, don't worry! It looks odd, but is really easy. The first part is a condition that returns true or false, just like an if statement. If it's true, the first variable `backgroundColor` is printed. If it's false, the second string `lightblue` is printed. The result is identical to before.

Chapter 8: Mistakes and Macros

Twig is a lot of fun to work with, and if you've made it this far, you know a lot about it. In this chapter, let's make some mistakes and debug them. We'll also talk about macros, which are like functions you create in Twig.

Making Mistakes!

Like with anything, there are a lot of ways to make mistakes in Twig. Fortunately, Twig usually gives you very clear errors and the line number the error is on. We saw an error earlier when we put some content outside of a block tag.

Let's a see a few more common errors. Hopefully, some of these will seem pretty obvious to you. First, putting different Twig syntaxes inside of each other is definitely not allowed:

```
{% if {{ pageTitle }} == 'Hello' %}

{% endif %}
```

When we refresh, Twig yells at us!

> A hash key must be a quoted string, a number, a name, or an expression enclosed in parentheses (unexpected token "punctuation" of value "{" in `homepage.twig` at line 10'

We now know that you open up Twig just once using `{{` or `{%` and then you can write inside of it. What's tricky about this error message is the "a hash key must be a quoted string" part. What the heck is a hash key?

Remember from earlier that when we used the include() function, we passed it a collection, or *hash* of variable names and values. Whenever you're already inside Twig and you write a `{` character, Twig thinks this is a hash. That's only really important because I want you to be able to recognize that when you see a Twig error containing the word "hash", it's talking about a curly-brace character.

Another common error is when we try to get some data off of a blank object. Suppose that there's a `featuredProduct` variable that we have access to. This is supposed to be a Product object, but pretend that someone has made a mistake and `featureProduct` is actually blank!

```
// index.php
// ...

echo $twig->render('homepage.twig', array(
    // ...
    'featuredProduct' => null,
));
```

When we try to print the `name` on it, we get a strange error.

> Item `name` for "" does not exist in `homepage.twig` at line 23

What it really means is that `featureProduct` is blank, so you can't try to get its name of course! The empty double-quotes is a bit deceiving, but when you see it, it means that you're trying to get something from a non-existent object. In this case, we'd need to check our PHP code to see why this variable is missing.

Overall, Twig works hard to give you very clear errors. Your errors will almost *always* contain a line number where the problem is and a decent message. For example, if we leave off certain characters or even add extra things, Twig's messages always tell us exactly what's wrong.

Macros

If you're printing out the same markup over and over again, you may find it useful to write your own Twig functions. For example, let's pretend that we're iterating over 2 different sets of products: `products` and `featureProducts`:

```
// index.php
// ...

echo $twig->render('homepage.twig', array(
    // ...
    'products' => array(
        new Product('Serious Businessman', 'formal.png'),
        new Product('Penguin Dress', 'dress.png'),
        new Product('Sportstar Penguin', 'sports.png'),
    ),
    'featuredProducts' => array(
        new Product('Angel Costume', 'angel-costume.png'),
        new Product('Penguin Accessories', 'swatter.png'),
        new Product('Super Cool Penguin', 'super-cool.png'),
    ),
));
```

In `homepage.twig`, we certainly don't want to duplicate our `for` loop and all the markup inside of it. Instead, let's create a macro, which is just a Twig function!

Start by creating a new "do something" tag called "macro". Let's call our macro `printProducts` and have it accept two arguments: the array of products to print and the message in case there are no products. Add a closing `endmacro` and then copy the `for` loop code from below. The only adjustment we need to make is to print out the `emptyMessage` variable:

```
{% macro printProducts(products, emptyMessage) %}
    {% for product in products %}
        <div class="span4">
            <h2>{{ product.name }}</h2>

            <div class="product-img">
                <img src="../assets/images/{{ product.imagePath }}" alt="{{ product.name }}"/>
            </div>
        </div>

        {% if loop.index is divisible by(3) and not loop.last %}
            </div><div class="row">
        {% endif %}
    {% else %}
        <div class="alert alert-error span12">
            {{ emptyMessage }}
        </div>
    {% endfor %}
{% endmacro %}
```

Ok! To use this, we call it just like any Twig function, except prefixed with `_self.`:

```
<div class="row">
    {{ _self.printProducts(
        products,
        "Oh now! We're all out of super-awesome penguin clothes!")
    }}
</div>
```

When we fresh, the first three products are printed perfectly! Now printing out the featured products is very easy:

```
<div class="row">
    {{ _self.printProducts(
        featuredProducts,
        "Snow storm in the arctic: nothing on sale today :/")
    }}
</div>
```

Macros can be a huge tool when you're building some reusable functionality. In some ways, using a macro is similar to using the include() function. Both allow you to move markup and logic into a separate place and then use

it. The biggest advantage of a macro is that it's very clear what variables you need to pass to it.

But like with the include() function, macros can also live in totally different files. Let's create a new `macros.twig` file and move it there:

```
{# templates/macros.twig #}

{% macro printProducts(products, emptyMessage) %}
    {# ... #}
{% endmacro %}
```

To use the macro in `homepage.twig`, add an `imports` "do something" tag. This tells Twig to "import" the macros from that file and make them available as `myMacros`:

```
{# templates/homepage.twig #}

{% import 'macros.twig' as myMacros %}
```

To use it, just change `_self` to `myMacros`:

```
<div class="row">
    {{ myMacros.printProducts(
        featuredProducts,
        "Snow storm in the arctic: nothing on sale today :/")
    }}
</div>

<div class="row">
    {{ myMacros.printProducts(
        products,
        "Oh now! We're all out of super-awesome penguin clothes!")
    }}
</div>
```

Tip

When we say `_self`, it's a way of referring to this very template.

Chapter 9: Extra Credit Tricks and HTML Escaping

You're a pro now, so let's have a little fun and see some sweet tricks!

Expert Control of Blocks with the block Function

We've learned a lot about template inheritance and blocks. Now, let's make things a bit more interesting. We created a `title` block in our layout so that individual pages could control the page title. If a page has a `title` block, it replaces the page title entirely. If it has *no* `title` block, then the default title is used.

Let me change the `title` block to be a little more interesting:

```
<title>
    {% block title %}
        Penguins Pants Plus! Your source for fancy penguin suits
    {% endblock %} | Penguins Pants Plus!
</title>
```

My goal is to suffix the page title with "Penguins Pants Plus!" so that all of our pages are consistent. The only funny thing here, besides penguin pants, is that if the `title` block isn't overridden, the suffix is a little redundant. Is it possible to *only* add the extra text if the `title` block is overridden?

The secret is the block function, which you can use to return the content of a block at any time. Replace the traditional `{% block title %}` and instead use the `block()` function in a say something tag:

```
{{ block('title') }} | Penguins Pants Plus!
```

When we refresh, this works like before: the `block` function prints out the content from the `title` block. The only thing we're missing is the default page title if the `title` block isn't set.

Now that we know about this `block` function, we can do that easily with an `if` statement:

```
{% if block('title') %}
    {{ block('title') }} | Penguins Pants Plus!
{% else %}
    Penguins Pants Plus! Your source for fancy penguin suits
{% endif %}
```

Tip

The `block()` function throws an exception since Twig 2.0 if there's no block with specified name. At first, be sure that the block is defined:

```
{% if block('title') is defined %}
    {{ block('title') }} | Penguins Pants Plus!
{% else %}
    Penguins Pants Plus! Your source for fancy penguin suits
{% endif %}
```

Success! When we override the `title` block on the homepage, we get the suffix added. But on the contact page, we just get the default page title. I try to use blocks in their traditional fashion as often as possible. But when things get more complicated, use the `block` function to do some really custom things.

Go Deeper!

To get really advanced, you can import blocks from other templates and use them in this way. See the use tag for more details.

The Short block Syntax

And while we're on this topic, we can make the `title` block even shorter in `homepage.twig`:

```
{% block title 'Start looking fly!' %}
```

You're free to choose whatever format you want, but if your block is just a simple string, you'll often see this version used.

Concatenating Strings

One apparent drawback to this is that you can't mix static text and variables like you could before by just writing some text and then using the "say something" syntax.

For example, suppose we wanted to include the `pageData.title` variable in the page title. How can we combine it with the static text? The answer is with the `~` character, which concatenates strings in Twig.

```
{% block title 'Start looking fly! '~pageData.title %}
```

You won't see this too often, but it'll come in handy when you need it.

Whitespace Control

Normally, the whitespace you put in a Twig file is left completely alone. We can see this when we view the source. In fact, we have some extra space around the `title` tag because of the new trick we're using in Twig. Let's see if we can get rid of it!

On any twig starting or ending tag, you can add a minus sign (`-`):

```
<title>
  {%- if block('title') %}
    {{ block('title') }} | Penguins Pants Plus!
  {% else %}
    Penguins Pants Plus! Your source for fancy penguin suits
  {% endif %}
</title>
```

This tells Twig to trim all the whitespace to the left of that tag until it hits a non-whitespace character. When we view the source, we can see a slightly smaller amount of whitespace. If we add enough of these, we'll see all the extra space disappear:

```
<title>
  {%- if block('title') -%}
    {{- block('title') -}} | Penguins Pants Plus!
  {%- else -%}
    Penguins Pants Plus! Your source for fancy penguin suits
  {%- endif -%}
</title>
```

The spaceless Tag

Another way to control whitespace is with the spaceless tag. The point of this tag is a little different: it removes all whitespace between HTML tags, without affecting space inside an HTML tag or inside static text. If we surround the meta tags with this and refresh, we'll see those meta tags all print right next to each other on one line:

```
{% spaceless %}
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<meta name="description" content="">
<meta name="author" content="">
{% endspaceless %}
```

Using Undefined Variables with the default Filter

Let's see one more common trick that may look strange when you first see it. Look back in the `banner.twig` template where we used the single-line if syntax. Actually, there's an easier way to do this by using the default filter:

```
<div class="well" style="background-color: {{ backgroundColor|default('lightblue') }};">
```

Normally, if you reference an undefined variable in Twig, it blows up! But when you use the `default` filter, it avoids that error and instead, returns the default value `lightblue`. You may see this trick quite often when someone is using a variable that may or may not be defined.

Tip

Depending on your settings, Twig may just fail silently if you reference an undefined variable.

Escaping

Ok, one last thing - HTML escaping! Whenever you render content that may have been filled in by the user, you need to escape it. This prevents people from writing HTML tags that you don't want or, worse, JavaScript code that could be used for cross-site scripting attacks. That's scarier than a hungry pack of leopard seals!

Let's try this out by adding some HTML markup to our page summary:

```
// index.php
// ...

echo $twig->render('homepage.twig', array(
    'pageData' => array(
        'summary'   => "You're <strong>hip</strong>, you're cool ...",
    ),
    // ...
));
```

When we refresh, Twig is automatically escaping these characters and printing them out safely. Actually, whether or not Twig automatically does this depends on how it's setup. In your case, try this out and see if Twig is escaping or not escaping automatically. You can try this easily by printing out a static string and seeing what happens.

```
{{ '<strong>hallo</strong>'|upper }}
```

In some cases, you may *need* to actually print out some content unescaped. To do this, just use the handy raw filter:

```
<p>
    {{ pageData.summary|raw }}
</p>
```

Tip

If automatic escaping is *off*, then you need to be quite careful and use the escape filter on any strings you print out to make sure they are escaped.

Happy Trails

Well hello Twig expert! Our time talking about Twig is coming to an end, but the good news is that you have all the tools you need to be successful and your penguins are looking dapper. Remember that all the tags, functions, filters and tests that are available in Twig can be found on the bottom of its documentation page.

Also remember that in your project, you may have even more tags, functions, filters or tests that are specific to you. Your challenge from here is to find out what those are and what secrets each holds.

Good luck, and seeya next time!