Learning Objectives

- **Identify why custom template tags can be better than filters**
- **Create a simple tag**
- **Add context to a custom template tag**
- **Use a template inside another template with inclusion tags**
- **Identify when to use an advanced template tag**

Clone Blango Repo

Clone Blango Repo

Before we continue, you need to clone the `blango` repo so you have all of your code. You will need the SSH information for your repo.

In the Terminal

- Clone the repo. Your command should look something like this:

```
git clone git@github.com:<your_github_username>/blango.git
```

- You should see a `blango` directory appear in the file tree.

You are now ready for the next assignment.

Custom Template Tags

Custom Template Tags

If filters aren't powerful enough to achieve the desired output in your template, the next step is to use template tags. Template tags are much more powerful and flexible. You have used a lot of template tags already:

- `extends`, as in `{% extends "base.html" }`
- `block`, like `{% block content %}`
- `if` and `for` template tags, for flow control
- and probably many more!

Django allows you to write your own custom template tags too. Template tags can accept no arguments, or as many as you need, including positional, and optional keyword arguments. You use them to output values, render a template, or even parse content between two tags that you specify.

Before we start on some custom template tags, we'll add a new Post detail page to Blango, so we can view a Post's content. This will involve:

- Creating a new template to display the post
  - And, a small refactoring of the byline rendering to remove duplicated code
- Linking to the Post detail page in the `index.html` template
- Adding a view to fetch and render the new template
- Adding a URL mapping to the new view

Let's get started.

Try It Out

Create a new file called `post-detail.html` inside the `blango/templates/blog` directory. As with `index.html`, it will extend the `base.html` template and override the `content` block.

Copy and paste this content inside `post-detail.html`:

```
{% extends "base.html" %}
{% block content %}
<h2>{{ post.title }}</h2>
<div class="row">
    <div class="col">
        {% include "blog/post-byline.html" %}
    </div>
</div>
<div class="row">
    <div class="col">
        {{ post.content|safe }}
    </div>
</div>
{% endblock %}
```

Two things to note:
- We are including `post-byline.html` which doesn't exist yet, but we will create it soon.
- The `safe` filter is being applied to `post.content`. As we discussed in the previous section on HTML safe filters, this is definitely **not** best practice for a production site as we couldn't trust the user generated input. As was already mentioned, a library like Bleach should be used to sanitise the HTML output before being rendered. For the sake of our example site, where we are probably the one and only user entering content, it is good enough.

Create another new file in the same directory, called `post-byline.html`. We want to display the byline (author detail and published date) on both the Post list and Post detail pages, so we'll move it into its own template so it can be included.

Inside `post-byline.html`, you can paste this content:

Open post-byline.html

```
{% load blog_extras %}
<small>By {{ post.author|author_details:request.user }} on {{
        post.published_at|date:"M, d Y" }}</small>
```

Next we'll do a couple of changes to `index.html`:

Open index.html

- Remove the `{% load blog_extras %}` template tag, as we no longer need to use the `author_details` filter in this template.
- Replace the byline HTML with an `include` of the `post-byline.html` template.
- Change the *Read More* link to go to the new URL that we'll create.

You can copy and paste this HTML into `index.html`:

```
{% extends "base.html" %}
{% block content %}
    <h2>Blog Posts</h2>
    {% for post in posts %}
    <div class="row">
        <div class="col">
            <h3>{{ post.title }}</h3>
            {% include "blog/post-byline.html" %}
            <p>{{ post.summary }}</p>
            <p>
                ({{ post.content|wordcount }} words)
                <a href="{% url "blog-post-detail" post.slug
        %}">Read More</a
            </p>
        </div>
    </div>
    {% endfor %}
{% endblock %}
```

Notice the use of the `url` template tag to link to a URL with the name `blog-post-detail`, providing the `post.slug` as an argument.

Next, we have to create the view to render this template. Open the `blog` app's `views.py` file. The `post_detail` view is quite simple, just fetching a Post using its `slug`. We make use of the `get_object_or_404` shortcut which will automatically return a 404 Not Found response if the requested object isn't found in the database:

Open views.py

```
def post_detail(request, slug):
    post = get_object_or_404(Post, slug=slug)
    return render(request, "blog/post-detail.html", {"post":
        post})
```

Make sure to import the `get_object_or_404` function at the stop of the file.

```
from django.shortcuts import render, get_object_or_404
```

For the final step, we need to add the URL mapping in `urls.py`.

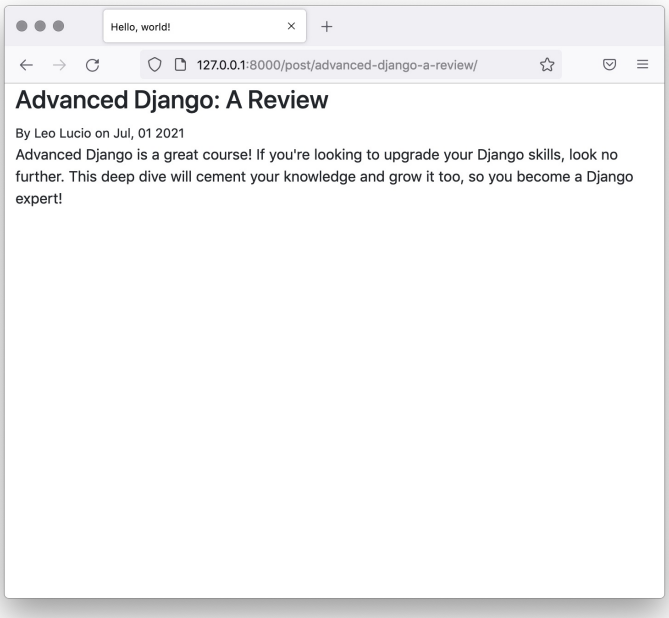
Open urls.py

Add a new url mapping to `urlpatterns`, like this:

```
    path("post/<slug>/", blog.views.post_detail, name="blog-
        post-detail")
```

After editing this file, start the Django dev server if it's not already running, then load (or refresh) the main page. You should see that the post list page doesn't look any different, but you should be able to click on a `Read More` link to go to a Post detail page, which will look similar to this (depending on your post content, of course):

View Blog



Post Detail

Now on to template tags, starting with simple tags.

Simple Tags

Simple Tags

As you might have guessed, a simple tag is the simplest way of creating a custom template tag. A simple tag is built with a function that can take any number of arguments (even 0). Like a custom filter, the function is created in a Python file inside the `templatetags` directory of a Django app. We will reuse the `blog_extra.py` file for our custom templates as well. Once the function is defined, it must be registered, with the `Library.simple_tag` function. There are a few ways to use this function, but we'll stick with using it as a decorator.

For our first simple tag, let's look at how we can shorten the amount of HTML we have to write, by implementing a tag that outputs the HTML for a Bootstrap row.

As we know, creating a row is done using a `div` with the `row` class:

```
<div class="row">
<!-- row content -->
</div>
```

We'll create two simple tags, so we can build a Bootstrap row like this:

```
{% row %}
<!-- row content -->
{% endrow %}
```

This will cut down on code a little bit, and will make it easier to understand where a row is ended, as sometimes it can be confusing to look through lots of closing `</div>` tags one after another.

Try It Out

Open `blog_extras.py`, and define a `row` function that returns the HTML for opening a row. Make sure the function is decorated with `@register.simple_tag`. Similarly, define an `endrow` function that returns HTML for closing a row.

```
@register.simple_tag
def row():
    return '<div class="row">'


@register.simple_tag
def endrow():
    return "</div>"
```

By default, the `simple_tag` function uses the name of the function as the template tag's name, but like the `filter` registration function, it can accept a `name` argument to customize the name of the `tag` in templates.

Now, let's make use of the `row` helper in the templates. Open `index.html` and make sure to load `blog_extra` after the `extends` template tags. Then, replace the row `<div>`s with our new template tag. When finished, the HTML should look like this:

Open index.html

```
{% extends "base.html" %}
{% load blog_extras %}
{% block content %}
    <h2>Blog Posts</h2>
    {% for post in posts %}
    {% row %}
        <div class="col">
            <h3>{{ post.title }}</h3>
            {% include "blog/post-byline.html" %}
            <p>{{ post.summary }}</p>
            <p>
                ({{ post.content|wordcount }} words)
                <a href="{% url "blog-post-detail" post.slug
        %}">Read More</a>
            </p>
        </div>
    {% endrow %}
    {% endfor %}
{% endblock %}
```

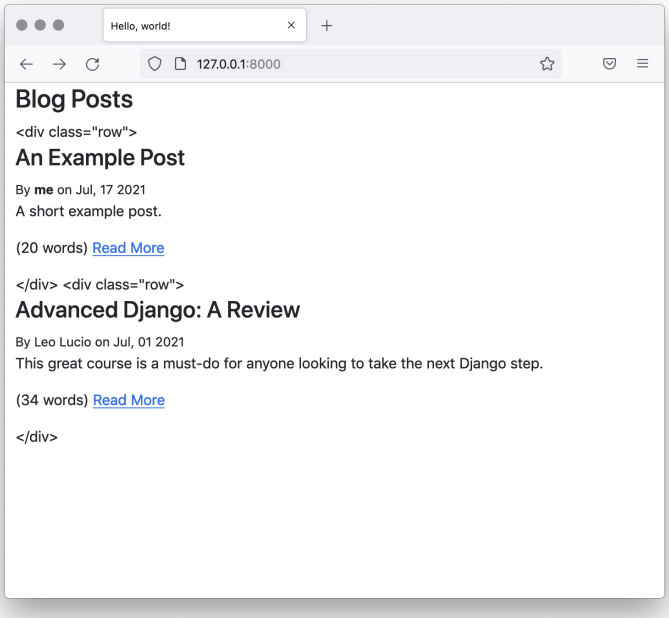
Perform the same changes in `post-detail.html`, the code afterward will look like this:

Open post-detail.html

```
{% extends "base.html" %}
{% load blog_extras %}
{% block content %}
<h2>{{ post.title }}</h2>
{% row %}
    <div class="col">
        {% include "blog/post-byline.html" %}
    </div>
{% endrow %}
{% row %}
    <div class="col">
        {{ post.content|safe }}
    </div>
{% endrow %}
{% endblock %}
```

Now load up either page in your browser, and you should notice a problem.

View Blog



Simple Tag Output Problem

To be safe, simple tags automatically escape their output, so the tags are not being rendered properly. We first need to mark their return values as safe, which can be done with the `format_html` function. Wrap the return values of your template tag functions with this function, like this:

Open blog_extras.py

```
@register.simple_tag
def row():
    return format_html('<div class="row">')


@register.simple_tag
def endrow():
    return format_html("</div>")
```

Then refresh the page and it should look like it did originally.

▼ **Refresh the Website**

Click on the blue, circular arrows to refresh the website.

blue, circular arrows

Next we'll look at adding arguments to the template tag. It's as simple as adding the arguments to the function, and then passing them in to the template. Let's make it so that the `class` of the row can optionally be passed in. First you'll need to add it as an argument to the function, making sure to set a default argument so that it's optional. Then, just add the class inside the HTML. Something like this:

```
@register.simple_tag
def row(extra_classes=""):
    return format_html('<div class="row {}">', extra_classes)
```

Remember that you can't name the argument `class` as it's a Python keyword.

To see our change in action, we'll add a some borders on the bottom of each Post summary in the `index.html` page. This is done with the Bootstrap utility class `border-bottom`.

Open `index.html` and change the `row` template tag from:

Open index.html

```
{% row %}
```

to

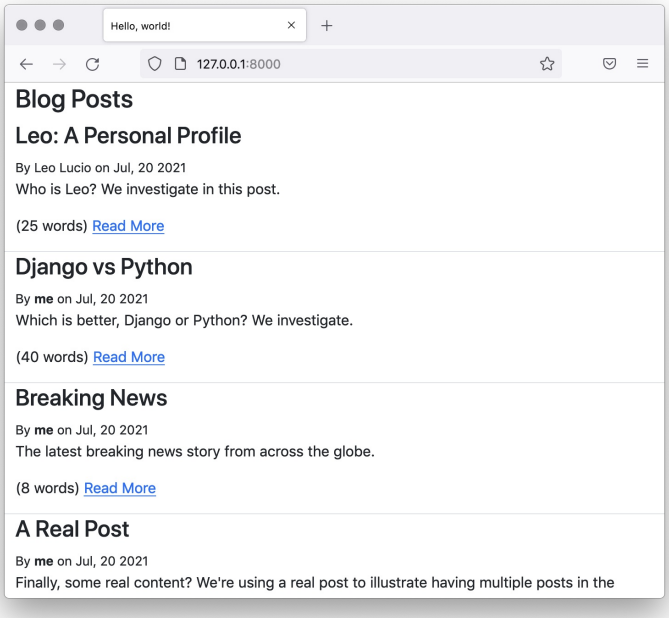
```
{% row "border-bottom" %}
```

Refresh the posts list page in your browser and you'll see lines separating each post now.

▼ **Refresh the Website**

Click on the blue, circular arrows to refresh the website.

blue, circular arrows



Separator Lines

challenge

Try this variation:

Try creating `col` and `endcol` tags that work in the same manner, but instead output the opening and closing tags for a Bootstrap column. Note that the `endcol` function will output identical markup as the `endrow` function.

You can add as many arguments as you like to the template tag function, and pass in variables from the template using both positional and named arguments.

▼ **Solution**

Here is one possible solution:

```
@register.simple_tag
def col(extra_classes=""):
    return format_html('<div class="col {}">',
        extra_classes)


@register.simple_tag
def endcol():
    return format_html("</div>")
```

Next we will look at how to access context variables inside template tag functions.

Access Template Context

Accessing the Template Context in Template Tags

So far we've only seen how to access variables or values that have been explicitly passed to a template tag function. With some minor changes to how the tag is registered, it can also have access to all the same context variables as the template in which it's used. One example might be to always be able to access the `request` variable without having to remember to pass it into the template tag all the time.

To give access to the context, you need to make two changes to the template tag function:

1. When registering, pass `takes_context=True` to the `register.simple_tag` decorator.
2. Add `context` as the first argument to the template tag function.

For example, here's how you could reimplement `author_details` filter as a template tag that takes no arguments.

```
@register.simple_tag(takes_context=True)
def author_details_tag(context):
    request = context["request"]
    current_user = request.user
    post = context["post"]
    author = post.author

    if author == current_user:
        return format_html("<strong>me</strong>")

    if author.first_name and author.last_name:
        name = f"{author.first_name} {author.last_name}"
    else:
        name = f"{author.username}"

    if author.email:
        prefix = format_html('<a href="mailto:{}">',
        author.email)
        suffix = format_html("</a>")
    else:
        prefix = ""
        suffix = ""

    return format_html("{}{}{}", prefix, name, suffix)
```

And we would use it in the byline like this:

Open post-byline.html

```
<small>By {% author_details_tag %} on {{
        post.published_at|date:"M, d Y" }}</small>
```

You can see that we don't need to pass in any variables (although we could pass in arbitrary variables too). We have access to the template context and can access any variables we need by using it.

challenge

Try this variation:

While we won't be using this template tag in the Blango project, you can try implementing it yourself and testing it out.

View Blog

Once you're done, revert the `post-byline.html` and `blog_extras.py` to the way they were when you started.

▼ **Reverting your work**

Here is the original version of the `author_details` filter:

```
@register.filter
def author_details(author, current_user):
    if not isinstance(author, user_model):
        # return empty string as safe default
        return ""

    if author == current_user:
        return format_html("<strong>me</strong>")

    if author.first_name and author.last_name:
        name = f"{author.first_name} {author.last_name}"
    else:
        name = f"{author.username}"

    if author.email:
        prefix = format_html('<a href="mailto:{}">',
        author.email)
        suffix = format_html("</a>")
    else:
        prefix = ""
        suffix = ""

    return format_html('{}{}{}', prefix, name, suffix)
```

Here is the original version of the byline:

```
{% load blog_extras %}
<small>By {{ post.author|author_details:request.user }} on
        {{ post.published_at|date:"M, d Y" }}</small>
```

Next we'll look at inclusion tags, and how you can use them to render a template inside another template.

Inclusion Tags

Inclusion Tags

One way of including a template inside another is with the `include` template tag. We've used this to include the `post-byline.html` template in our other templates.

```
{% include "blog/post-byline.html" %}
```

This is very simple to implement, but its main drawback is that included templates can only access variables that are already in the including template's context. That means any extra variables need to be passed in from the calling view, so if it's a template that's used in lots of places you'll be repeating the data-loading code in lots of different views.

By using an inclusion tag, you can query for extra data inside your template tag function, which can then be used to render a template.

Inclusion tags are registered with the `Library.inclusion_tag` function. This has one required argument, the name of the template to render. Unlike simple tags, inclusion tags don't return a string to render. They return a context dictionary, which is used to render template used during registration.

A useful feature for a blog site is a way to see recent posts. We'll create an inclusion tag that fetches the five most recent posts, but excludes the current post being viewed. Then it will render a template.

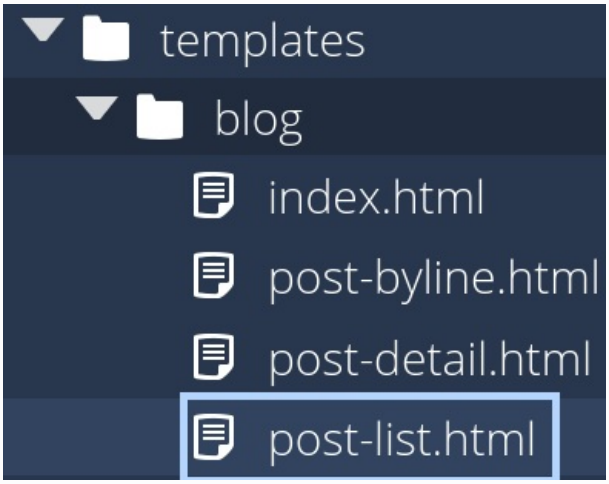
challenge

Try this variation:

Start by creating the template to render, a file named `post-list.html` in the `blog/templates/blog` directory.

▼ **Solution**

After creating the file, the file tree should look like this:



File tree with post-list.html file

We'll have it include a place to put the title, so that it can be generic and show different titles depending on which page it's used. We'll then iterate over a list of `Post` objects and render them in a `<ul>`; with each `<li>` containing a link to the post detail page for that Post.

Open post-list.html

```
<h4>{{ title }}</h4>
<ul>
{% for post in posts %}
    <li><a href="{% url "blog-post-detail" post.slug %}">Read
        More</a></li>
{% endfor %}
</ul>
```

Next, return to `blog_extras.py`. Write a `recent_posts` function that takes a `post` argument. It then fetches the five most recent `Post` objects (ordered by `published_at`), but excludes the `Post` that was passed in (because we want to show recent posts that aren't the `Post` being viewed). The template tag

function returns a dictionary with the posts in the `posts` key and the string *Recent Posts* in the `title` key – this will be the context data that Django uses to render the template.

After writing the function, then decorate it with `register.inclusion_tag`, passing in the path of the `post-list.html` template.

Try it yourself, you should come up with something like this:

Open blog_extras.py

```
@register.inclusion_tag("blog/post-list.html")
def recent_posts(post):
    posts = Post.objects.exclude(pk=post.pk)[:5]
    return {"title": "Recent Posts", "posts": posts}
```

Make sure to also import the `Post` model at the start of the file.

```
from blog.models import Post
```

Finally the template tag needs to be used in the `post-detail.html` template. Create a new row and column at the bottom of the page, and then add your new template tag inside. Remember to pass in the current `Post`.

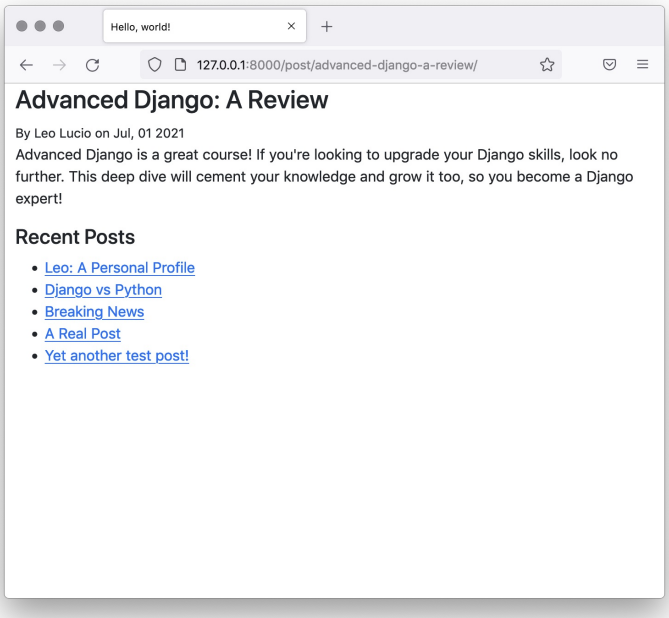
The addition to the `post-detail.html` template should look like this:

Open post-detail.html

```
<!-- existing code here -->
{% row %}
    {% col %}
        {% recent_posts post %}
    {% endcol %}
{% endrow %}
```

Reload a post detail page in your browser, and provided you have more than one `Post` object, you should see something like this:

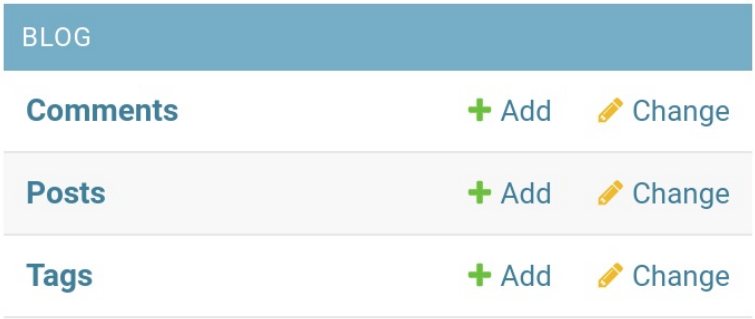
View Blog



Related Posts

▼ **Adding more blog posts**

If you need to add more blog posts to the project:
* Go to the admin panel by adding `/admin` to the URL
* Log in as the admin
* Click the `+ Add` button to add as many posts as you need



Add Post Button

- Be sure to add a date and time to the post, otherwise, they will be filtered out of the website

The current post that's being viewed should not be in the list of *Recent Posts*.

As with a simple tag, we can also pass the context to the inclusion tag function, by adding a `context` argument and adding `takes_context=True` to the decorator call.

Next we'll take a brief look at advanced template tags.

Advanced Template Tags

Advanced Template Tags

Simple template tags and inclusion tags should handle majority of your use cases for custom template tags. However, if you need even more customization, Django offers it. Normally, if you do need to use any advanced template tags it's for quite a specific use case so we won't go into detail here. However we will give a brief overview.

First some background. When Django parses a template, it traverses it and breaks it into nodes (`django.template.Node` subclasses). It will then render each each node in turn, by calling its `render` function.

Advanced template tags consist of two parts:

- A parser function that is called when its template tag is encountered in a template. The parser function will parse the template tag and extract variables from the context. The parser function returns a template `Node` subclass.
- The template `Node` subclass, which has a `render` method. This method is called to return the string that is to be rendered in the template.

The parser function is registered similarly to the other types of tags: by being decorated with `@register.tag`.

The main reasons to use an advanced template tag are:

- To capture the content (a list of `Node`s) between two tags. You can then perform operations on each node, and choose how they're output. For example, you could implement a custom permissions template tag that only outputs the content between them if the current user has the correct permissions.
- To set context variables. A template tag could be used to set a value in the template context that is then accessible further on in the template.

As you can see, these are quite specific situations, and most of the time the simpler template tags will do the job. But if you think you need advanced template tags, then you can check out the full documentation for advanced custom template tags

Pushing to GitHub

Pushing to GitHub

Before continuing, you must push your work to GitHub. In the terminal:

- Commit your changes:

```
git add .
git commit -m "Finish custom template tags"
```

- Push to GitHub:

```
git push
```