Django

The Django template language ¶

This document explains the language syntax of the Django template system. If you're looking for a more technical perspective on how it works and how to extend it, see The Django template language: for Python programmers.

Django's template language is designed to strike a balance between power and ease. It's designed to feel comfortable to those used to working with HTML. If you have any exposure to other text-based template languages, such as Smarty or Jinja2, you should feel right at home with Django's templates.

> **Philosophy**
>
> If you have a background in programming, or if you're used to languages which mix programming code directly into HTML, you'll want to bear in mind that the Django template system is not simply Python embedded into HTML. This is by design: the template system is meant to express presentation, not program logic.
>
> The Django template system provides tags which function similarly to some programming constructs – an `if` tag for boolean tests, a `for` tag for looping, etc. – but these are not simply executed as the corresponding Python code, and the template system will not execute arbitrary Python expressions. Only the tags, filters and syntax listed below are supported by default (although you can add your own extensions to the template language as needed).

Templates ¶

A template is a text file. It can generate any text-based format (HTML, XML, CSV, etc.).

A template contains **variables**, which get replaced with values when the template is evaluated, and **tags**, which control the logic of the template.

Below is a minimal template that illustrates a few basics. Each element will be explained later in this document.

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

{% block title %}{{ section.title }}{% endblock %}

{% block content %}
<h1>{{ section.title }}</h1>

{% for story in story_list %}
<h2>
  <a href="{{ story.get_absolute_url }}">
    {{ story.headline|upper }}
  </a>
</h2>
<p>{{ story.tease|truncatewords:"100" }}</p>
{% endfor %}
{% endblock %}
```

Getting Help

> **Philosophy**
>
> Why use a text-based template instead of an XML-based one (like Zope's TAL)? We wanted Django's template language to be usable for more than just XML/HTML templates. You can use the template language for any text-based format such as emails, JavaScript and CSV.

Language: **en**

Documentation version: **5.1**

^

Variables ¶

Variables look like this: `{{ variable }}`. When the template engine encounters a variable, it evaluates that variable and replaces it with the result. Variable names consist of any combination of alphanumeric characters and the underscore (`"_"`) but may not start with an underscore, and may not be a number. The dot (`"."`) also appears in variable sections, although that has a special meaning, as indicated below. Importantly, *you cannot have spaces or punctuation characters in variable names.*

Use a dot (`.`) to access attributes of a variable.

Behind the scenes

Technically, when the template system encounters a dot, it tries the following lookups, in this order:

- Dictionary lookup

- Attribute or method lookup

- Numeric index lookup

If the resulting value is callable, it is called with no arguments. The result of the call becomes the template value.

This lookup order can cause some unexpected behavior with objects that override dictionary lookup. For example, consider the following code snippet that attempts to loop over a `collections.defaultdict`:

```
{% for k, v in defaultdict.items %}
    Do something with k and v here...
{% endfor %}
```

Because dictionary lookup happens first, that behavior kicks in and provides a default value instead of using the intended `.items()` method. In this case, consider converting to a dictionary first.

In the above example, `{{ section.title }}` will be replaced with the `title` attribute of the `section` object.

If you use a variable that doesn't exist, the template system will insert the value of the `string_if_invalid` option, which is set to `''` (the empty string) by default.

Note that "bar" in a template expression like `{{ foo.bar }}` will be interpreted as a literal string and not using the value of the variable "bar", if one exists in the template context.

Variable attributes that begin with an underscore may not be accessed as they're generally considered private.

Filters ¶

You can modify variables for display by using **filters**.

Filters look like this: `{{ name|lower }}`. This displays the value of the `{{ name }}` variable after being filtered through the `lower` filter, which converts text to lowercase. Use a pipe (`|`) to apply a filter.

Filters can be "chained." The output of one filter is applied to the next. `{{ text|escape|linebreaks }}` is a common idiom for escaping text contents, then converting line breaks to `<p>` tags.

Some filters take arguments. A filter argument looks like this: `{{ bio|truncatewords:30 }}`. This will display the first 30 words of the `bio` variable.

Filter arguments that contain spaces must be quoted; for example, to join a list with commas and spaces you'd use `{{ list|join:", " }}`.

Django provides about sixty built-in template filters. You can read all about them in the built-in filter reference. To give you a taste of what's available, here are some of the more commonly used template filters:

`default`

If a variable is false or empty, use given default. Otherwise, use the value of the variable. For example:

```
{{ value|default:"nothing" }}
```

If `value` isn't provided or is empty, the above will display "`nothing`".

`length`

Returns the length of the value. This works for both strings and lists. For example:

```
{{ value|length }}
```

If **value** is **['a', 'b', 'c', 'd']**, the output will be **4**.

filesizeformat

Formats the value like a "human-readable" file size (i.e. **'13 KB'**, **'4.1 MB'**, **'102 bytes'**, etc.). For example:

```
{{ value|filesizeformat }}
```

If **value** is 123456789, the output would be **117.7 MB**.

Again, these are just a few examples; see the built-in filter reference for the complete list.

You can also create your own custom template filters; see How to create custom template tags and filters.

> **See also**
>
> Django's admin interface can include a complete reference of all template tags and filters available for a given site. See The Django admin documentation generator.

Tags ¶

Tags look like this: **{% tag %}**. Tags are more complex than variables: Some create text in the output, some control flow by performing loops or logic, and some load external information into the template to be used by later variables.

Some tags require beginning and ending tags (i.e. **{% tag %} ... tag contents ... {% endtag %}**).

Django ships with about two dozen built-in template tags. You can read all about them in the built-in tag reference. To give you a taste of what's available, here are some of the more commonly used tags:

for

Loop over each item in an array. For example, to display a list of athletes provided in **athlete_list**:

```
<ul>
{% for athlete in athlete_list %}
    <li>{{ athlete.name }}</li>
{% endfor %}
</ul>
```

if, elif, and else

Evaluates a variable, and if that variable is "true" the contents of the block are displayed:

```
{% if athlete_list %}
    Number of athletes: {{ athlete_list|length }}
{% elif athlete_in_locker_room_list %}
    Athletes should be out of the locker room soon!
{% else %}
    No athletes.
{% endif %}
```

In the above, if **athlete_list** is not empty, the number of athletes will be displayed by the **{{ athlete_list|length }}** variable. Otherwise, if **athlete_in_locker_room_list** is not empty, the message "Athletes should be out…" will be displayed. If both lists are empty, "No athletes." will be displayed.

You can also use filters and various operators in the **if** tag:

```
{% if athlete_list|length > 1 %}
    Team: {% for athlete in athlete_list %} ... {% endfor %}
{% else %}
    Athlete: {{ athlete_list.0.name }}
{% endif %}
```

While the above example works, be aware that most template filters return strings, so mathematical comparisons using filters will generally not work as you expect. **length** is an exception.

block and **extends**

Set up template inheritance (see below), a powerful way of cutting down on "boilerplate" in templates.

Again, the above is only a selection of the whole list; see the built-in tag reference for the complete list.

You can also create your own custom template tags; see How to create custom template tags and filters.

See also

Django's admin interface can include a complete reference of all template tags and filters available for a given site. See The Django admin documentation generator.

Comments ¶

To comment-out part of a line in a template, use the comment syntax: **{# #}**.

For example, this template would render as **'hello'**:

```
{# greeting #}hello
```

A comment can contain any template code, invalid or not. For example:

```
{# {% if foo %}bar{% else %} #}
```

This syntax can only be used for single-line comments (no newlines are permitted between the **{#** and **#}** delimiters). If you need to comment out a multiline portion of the template, see the **comment** tag.

Template inheritance ¶

The most powerful – and thus the most complex – part of Django's template engine 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.

Let's look at template inheritance by starting with an example:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <link rel="stylesheet" href="style.css">
    <title>{% block title %}My amazing site{% endblock %}</title>
</head>

<body>
    <div id="sidebar">
        {% block sidebar %}
        <ul>
            <li><a href="/">Home</a></li>
            <li><a href="/blog/">Blog</a></li>
        </ul>
        {% endblock %}
    </div>

    <div id="content">
        {% block content %}{% endblock %}
    </div>
</body>
</html>
```

This template, which we'll call **base.html**, defines an HTML skeleton document that you might use for a two-column page. It's the job of "child" templates to fill the empty blocks with content.

In this example, the **block** tag defines three 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:

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

{% block title %}My amazing blog{% endblock %}

{% block content %}
{% for entry in blog_entries %}
    <h2>{{ entry.title }}</h2>
    <p>{{ entry.body }}</p>
{% endfor %}
{% 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 – in this case, "base.html".

At that point, the template engine will notice the three **block** tags in **base.html** and replace those blocks with the contents of the child template. Depending on the value of **blog_entries**, the output might look like:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <link rel="stylesheet" href="style.css">
    <title>My amazing blog</title>
</head>

<body>
    <div id="sidebar">
        <ul>
            <li><a href="/">Home</a></li>
            <li><a href="/blog/">Blog</a></li>
        </ul>
    </div>

    <div id="content">
        <h2>Entry one</h2>
        <p>This is my first entry.</p>

        <h2>Entry two</h2>
        <p>This is my second entry.</p>
    </div>
</body>
</html>
```

Note that since the child template didn't define the **sidebar** block, the value from the parent template is used instead. Content within a **{% block %}** tag in a parent template is always used as a fallback.

You can use as many levels of inheritance as needed. One common way of using inheritance is the following three-level approach:

- Create a **base.html** template that holds the main look-and-feel of your site.

- Create a **base_SECTIONNAME.html** template for each "section" of your site. For example, **base_news.html**, **base_sports.html**. These templates all extend **base.html** and include section-specific styles/design.

- Create individual templates for each type of page, such as a news article or blog entry. These templates extend the appropriate section template.

This approach maximizes code reuse and helps to add items to shared content areas, such as section-wide navigation.

Here are some tips for working with inheritance:

- If you use **{% extends %}** in a template, it must be the first template tag in that template. Template inheritance won't work, otherwise.

- More **{% block %}** tags in your base templates are better. Remember, child templates don't have to define all parent blocks, so you can fill in reasonable defaults in a number of blocks, then only define the ones you need later. It's better to have more hooks than fewer hooks.

- If you find yourself duplicating content in a number of templates, it probably means you should move that content to a **{% block %}** in a parent template.

- If you need to get the content of the block from the parent template, the **{{ block.super }}** variable will do the trick. This is useful if you want to add to the contents of a parent block instead of completely overriding it. Data inserted using **{{ block.super }}** will not be automatically escaped (see the next section), since it was already escaped, if necessary, in the parent template.

- By using the same template name as you are inheriting from, **{% extends %}** can be used to inherit a template at the same time as overriding it. Combined with **{{ block.super }}**, this can be a powerful way to make small customizations. See Extending an overridden template in the *Overriding templates* How-to for a full example.

- Variables created outside of a **{% block %}** using the template tag **as** syntax can't be used inside the block. For example, this template doesn't render anything:

```
{% translate "Title" as title %}
{% block content %}{{ title }}{% endblock %}
```

- For extra readability, you can optionally give a *name* to your **{% endblock %}** tag. For example:

```
{% block content %}
...
{% endblock content %}
```

In larger templates, this technique helps you see which **{% block %}** tags are being closed.

- **{% block %}** tags are evaluated first. That's why the content of a block is always overridden, regardless of the truthiness of surrounding tags. For example, this template will *always* override the content of the **title** block:

```
{% if change_title %}
    {% block title %}Hello!{% endblock title %}
{% endif %}
```

Finally, note that 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.

Automatic HTML escaping ¶

When generating HTML from templates, there's always a risk that a variable will include characters that affect the resulting HTML. For example, consider this template fragment:

```
Hello, {{ name }}
```

At first, this seems like a harmless way to display a user's name, but consider what would happen if the user entered their name as this:

```
<script>alert('hello')</script>
```

With this name value, the template would be rendered as:

```
Hello, <script>alert('hello')</script>
```

...which means the browser would pop-up a JavaScript alert box!

Similarly, what if the name contained a **'<'** symbol, like this?

```
<b>username
```

That would result in a rendered template like this:

```
Hello, <b>username
```

...which, in turn, would result in the remainder of the web page being in bold!

Clearly, user-submitted data shouldn't be trusted blindly and inserted directly into your web pages, because a malicious user could use this kind of hole to do potentially bad things. This type of security exploit is called a Cross Site Scripting (XSS) attack.

To avoid this problem, you have two options:

- One, you can make sure to run each untrusted variable through the **escape** filter (documented below), which converts potentially harmful HTML characters to unharmful ones. This was the default solution in Django for its first few years, but the problem is that it puts the onus on *you*, the developer / template author, to ensure you're escaping everything. It's easy to forget to escape data.

- Two, you can take advantage of Django's automatic HTML escaping. The remainder of this section describes how auto-escaping works.

By default in Django, every template automatically escapes the output of every variable tag. Specifically, these five characters are escaped:

- **<** is converted to **<**

- **>** is converted to **>**

- **'** (single quote) is converted to **'**

- **"** (double quote) is converted to **"**

- **&** is converted to **&**

Again, we stress that this behavior is on by default. If you're using Django's template system, you're protected.

How to turn it off ¶

If you don't want data to be auto-escaped, on a per-site, per-template level or per-variable level, you can turn it off in several ways.

Why would you want to turn it off? Because sometimes, template variables contain data that you *intend* to be rendered as raw HTML, in which case you don't want their contents to be escaped. For example, you might store a blob of HTML in your database and want to embed that directly into your template. Or, you might be using Django's template system to produce text that is *not* HTML – like an email message, for instance.

For individual variables ¶

To disable auto-escaping for an individual variable, use the **safe** filter:

```
This will be escaped: {{ data }}
This will not be escaped: {{ data|safe }}
```

Think of *safe* as shorthand for *safe from further escaping* or *can be safely interpreted as HTML*. In this example, if **data** contains **'<b>'**, the output will be:

```
This will be escaped: &lt;b&gt;
This will not be escaped: <b>
```

For template blocks ¶

To control auto-escaping for a template, wrap the template (or a particular section of the template) in the **autoescape** tag, like so:

```
{% autoescape off %}
    Hello {{ name }}
{% endautoescape %}
```

The **autoescape** tag takes either **on** or **off** as its argument. At times, you might want to force auto-escaping when it would otherwise be disabled. Here is an example template:

```
Auto-escaping is on by default. Hello {{ name }}

{% autoescape off %}
    This will not be auto-escaped: {{ data }}.

    Nor this: {{ other_data }}
    {% autoescape on %}
        Auto-escaping applies again: {{ name }}
    {% endautoescape %}
{% endautoescape %}
```

The auto-escaping tag passes its effect onto templates that extend the current one as well as templates included via the **include** tag, just like all block tags. For example:

base.html ¶

```
{% autoescape off %}
<h1>{% block title %}{% endblock %}</h1>
{% block content %}
{% endblock %}
{% endautoescape %}
```

child.html ¶

```
{% extends "base.html" %}
{% block title %}This &amp; that{% endblock %}
{% block content %}{{ greeting }}{% endblock %}
```

Because auto-escaping is turned off in the base template, it will also be turned off in the child template, resulting in the following rendered HTML when the `greeting` variable contains the string **Hello!**:

```
<h1>This &amp; that</h1>
<b>Hello!</b>
```

Notes ¶

Generally, template authors don't need to worry about auto-escaping very much. Developers on the Python side (people writing views and custom filters) need to think about the cases in which data shouldn't be escaped, and mark data appropriately, so things Just Work in the template.

If you're creating a template that might be used in situations where you're not sure whether auto-escaping is enabled, then add an `escape` filter to any variable that needs escaping. When auto-escaping is on, there's no danger of the `escape` filter *double-escaping* data — the `escape` filter does not affect auto-escaped variables.

String literals and automatic escaping ¶

As we mentioned earlier, filter arguments can be strings:

```
{{ data|default:"This is a string literal." }}
```

All string literals are inserted **without** any automatic escaping into the template — they act as if they were all passed through the `safe` filter. The reasoning behind this is that the template author is in control of what goes into the string literal, so they can make sure the text is correctly escaped when the template is written.

This means you would write :

```
{{ data|default:"3 &lt; 2" }}
```

...rather than:

```
{{ data|default:"3 < 2" }}   {# Bad! Don't do this. #}
```

This doesn't affect what happens to data coming from the variable itself. The variable's contents are still automatically escaped, if necessary, because they're beyond the control of the template author.

Accessing method calls ¶

Most method calls attached to objects are also available from within templates. This means that templates have access to much more than just class attributes (like field names) and variables passed in from views. For example, the Django ORM provides the "entry_set" syntax for finding a collection of objects related on a foreign key. Therefore, given a model called "comment" with a foreign key relationship to a model called "task" you can loop through all comments attached to a given task like this:

```
{% for comment in task.comment_set.all %}
    {{ comment }}
{% endfor %}
```

Similarly, QuerySets provide a `count()` method to count the number of objects they contain. Therefore, you can obtain a count of all comments related to the current task with:

```
{{ task.comment_set.all.count }}
```

You can also access methods you've explicitly defined on your own models:

models.py ¶

```
class Task(models.Model):
    def foo(self):
        return "bar"
```

template.html ¶

```
{{ task.foo }}
```

Because Django intentionally limits the amount of logic processing available in the template language, it is not possible to pass arguments to method calls accessed from within templates. Data should be calculated in views, then passed to templates for display.

Custom tag and filter libraries ¶

Certain applications provide custom tag and filter libraries. To access them in a template, ensure the application is in `INSTALLED_APPS` (we'd add `'django.contrib.humanize'` for this example), and then use the `load` tag in a template:

```
{% load humanize %}

{{ 45000|intcomma }}
```

In the above, the `load` tag loads the `humanize` tag library, which then makes the `intcomma` filter available for use. If you've enabled `django.contrib.admindocs`, you can consult the documentation area in your admin to find the list of custom libraries in your installation.

The `load` tag can take multiple library names, separated by spaces. Example:

Getting Help

Language: **en**

Documentation version: **5.1**

```
{% load humanize i18n %}
```

See How to create custom template tags and filters for information on writing your own custom template libraries.

Custom libraries and template inheritance ¶

When you load a custom tag or filter library, the tags/filters are only made available to the current template – not any parent or child templates along the template-inheritance path.

For example, if a template `foo.html` has `{% load humanize %}`, a child template (e.g., one that has `{% extends "foo.html" %}`) will *not* have access to the humanize template tags and filters. The child template is responsible for its own `{% load humanize %}`.

This is a feature for the sake of maintainability and sanity.

See also

The Templates Reference
Covers built-in tags, built-in filters, using an alternative template language, and more.

Support Django!

Daniel Johansson donated to the Django Software Foundation to support Django development. Donate today!

Contents

- The Django template language
  - Templates
  - Variables
  - Filters
  - Tags
  - Comments
  - Template inheritance
  - Automatic HTML escaping
    - How to turn it off
      - For individual variables
      - For template blocks
    - Notes
    - String literals and automatic escaping
  - Accessing method calls

Getting Help

Language: **en**

Documentation version: **5.1**

∧

Browse

You are here:

Getting help

FAQ
Try the FAQ — it's got answers to many common questions.

Index, Module Index, or Table of Contents
Handy when looking for specific information.

django-users mailing list
Search for information in the archives of the django-users mailing list, or post a question.

#django IRC channel
Ask a question in the #django IRC channel, or search the IRC logs to see if it's been asked before.

Django Discord Server
Join the Django Discord Community.

Official Django Forum
Join the community on the Django Forum.

Ticket tracker
Report bugs with Django or Django documentation in our ticket tracker.

Getting Help

Language: **en**

Documentation version: **5.1**

Download:

Offline (Django 5.1): HTML | PDF | ePub
Provided by Read the Docs.

Learn More

About Django

Getting Started with Django

Team Organization

Django Software Foundation

Code of Conduct

Diversity Statement

Get Involved

Join a Group

Contribute to Django

Submit a Bug

Report a Security Issue

Get Help

Getting Help FAQ

#django IRC channel

Django Discord

Official Django Forum

Getting Help

Language: **en**

Documentation version: **5.1**

Follow Us

GitHub

Twitter

Fediverse (Mastodon)

News RSS

Django Users Mailing List

Support Us

Sponsor Django

Corporate membership

Official merchandise store

Benevity Workplace Giving Program

Getting Help

Language: **en**

Documentation version: **5.1**