

Chapter 9: Advanced Templates

Although most of your interactions with Django’s template language will be in the role of template author, you may want to customize and extend the template engine – either to make it do something it doesn’t already do, or to make your job easier in some other way.

This chapter delves deep into the guts of Django’s template system. It covers what you need to know if you plan to extend the system or if you’re just curious about how it works. It also covers the auto-escaping feature, a security measure you’ll no doubt notice over time as you continue to use Django.

If you’re looking to use the Django template system as part of another application (i.e., without the rest of the framework), make sure to read the “Configuring the Template System in Standalone Mode” section later in the chapter.

Template Language Review

First, let’s quickly review a number of terms introduced in Chapter 4:

- A *template* is a text document, or a normal Python string, that is marked up using the Django template language. A template can contain template tags and variables.
- A *template tag* is a symbol within a template that does something. This definition is deliberately vague. For example, a template tag can produce content, serve as a control structure (an `if` statement or `for` loop), grab content from a database, or enable access to other template tags.

Template tags are surrounded by `{%` and `%}`:

```
{% if is_logged_in %}
    Thanks for logging in!
{% else %}
    Please log in.
{% endif %}
```

- A *variable* is a symbol within a template that outputs a value.

Variable tags are surrounded by `{{` and `}}`:

```
My first name is {{ first_name }}. My last name is {{ last_name }}.
```

- A *context* is a name -> value mapping (similar to a Python dictionary) that is passed to a template.
- A template *renders* a context by replacing the variable “holes” with values from the context and executing all template tags.

For more details about the basics of these terms, refer back to Chapter 4.

The rest of this chapter discusses ways of extending the template engine. First, though, let’s take a quick look at a few internals left out of Chapter 4 for simplicity.

RequestContext and Context Processors

When rendering a template, you need a context. This can be an instance of `django.template.Context`, but Django also comes with a subclass, `django.template.RequestContext`, that acts slightly differently. `RequestContext` adds a bunch of variables to your template context by default – things like the `HttpRequest` object or information about the currently logged-in user. The `render()` shortcut creates a `RequestContext` unless it is passed a different context instance explicitly.

Use `RequestContext` when you don't want to have to specify the same set of variables in a series of templates. For example, consider these two views:

```
from django.template import loader, Context

def view_1(request):
    # ...
    t = loader.get_template('template1.html')
    c = Context({
        'app': 'My app',
        'user': request.user,
        'ip_address': request.META['REMOTE_ADDR'],
        'message': 'I am view 1.'
    })
    return t.render(c)

def view_2(request):
    # ...
    t = loader.get_template('template2.html')
    c = Context({
        'app': 'My app',
        'user': request.user,
        'ip_address': request.META['REMOTE_ADDR'],
        'message': 'I am the second view.'
    })
    return t.render(c)
```

(Note that we're deliberately *not* using the `render()` shortcut in these examples – we're manually loading the templates, constructing the context objects and rendering the templates. We're "spelling out" all of the steps for the purpose of clarity.)

Each view passes the same three variables – `app`, `user` and `ip_address` – to its template. Wouldn't it be nice if we could remove that redundancy?

`RequestContext` and **context processors** were created to solve this problem. Context processors let you specify a number of variables that get set in each context automatically – without you having to specify the variables in each `render()` call. The catch is that you have to use `RequestContext` instead of `Context` when you render a template.

The most low-level way of using context processors is to create some processors and pass them to `RequestContext`. Here's how the above example could be written with context processors:

```
from django.template import loader, RequestContext

def custom_proc(request):
    "A context processor that provides 'app', 'user' and 'ip_address'."
```

```

    return {
        'app': 'My app',
        'user': request.user,
        'ip_address': request.META['REMOTE_ADDR']
    }

def view_1(request):
    # ...
    t = loader.get_template('template1.html')
    c = RequestContext(request, {'message': 'I am view 1.'},
                       processors=[custom_proc])
    return t.render(c)

def view_2(request):
    # ...
    t = loader.get_template('template2.html')
    c = RequestContext(request, {'message': 'I am the second view.'},
                       processors=[custom_proc])
    return t.render(c)

```

Let's step through this code:

- First, we define a function `custom_proc`. This is a context processor – it takes an `HttpRequest` object and returns a dictionary of variables to use in the template context. That's all it does.
- We've changed the two view functions to use `RequestContext` instead of `Context`. There are two differences in how the context is constructed. One, `RequestContext` requires the first argument to be an `HttpRequest` object – the one that was passed into the view function in the first place (`request`). Two, `RequestContext` takes an optional `processors` argument, which is a list or tuple of context processor functions to use. Here, we pass in `custom_proc`, the custom processor we defined above.
- Each view no longer has to include `app`, `user` or `ip_address` in its context construction, because those are provided by `custom_proc`.
- Each view *still* has the flexibility to introduce any custom template variables it might need. In this example, the `message` template variable is set differently in each view.

In Chapter 4, we introduced the `render()` shortcut, which saves you from having to call `loader.get_template()`, then create a `Context`, then call the `render()` method on the template. In order to demonstrate the lower-level workings of context processors, the above examples didn't use `render()`, . But it's possible – and preferable – to use context processors with `render()`. Do this with the `context_instance` argument, like so:

```

from django.shortcuts import render
from django.template import RequestContext

def custom_proc(request):
    "A context processor that provides 'app', 'user' and 'ip_address'."
    return {
        'app': 'My app',
        'user': request.user,
        'ip_address': request.META['REMOTE_ADDR']
    }

def view_1(request):
    # ...

```

```

return render(request, 'template1.html',
               {'message': 'I am view 1.'},
               context_instance=RequestContext(request, processors=[custom_proc]))

def view_2(request):
    # ...
    return render(request, 'template2.html',
                  {'message': 'I am the second view.'},
                  context_instance=RequestContext(request, processors=[custom_proc]))

```

Here, we've trimmed down each view's template rendering code to a single (wrapped) line.

This is an improvement, but, evaluating the conciseness of this code, we have to admit we're now almost overdosing on the *other* end of the spectrum. We've removed redundancy in data (our template variables) at the cost of adding redundancy in code (in the `processors` call). Using context processors doesn't save you much typing if you have to type `processors` all the time.

For that reason, Django provides support for *global* context processors. The `TEMPLATE_CONTEXT_PROCESSORS` setting (in your `settings.py`) designates which context processors should *always* be applied to `RequestContext`. This removes the need to specify `processors` each time you use `RequestContext`.

By default, `TEMPLATE_CONTEXT_PROCESSORS` is set to the following:

```

TEMPLATE_CONTEXT_PROCESSORS = (
    'django.core.context_processors.auth',
    'django.core.context_processors.debug',
    'django.core.context_processors.i18n',
    'django.core.context_processors.media',
)

```

This setting is a tuple of callables that use the same interface as our `custom_proc` function above – functions that take a request object as their argument and return a dictionary of items to be merged into the context. Note that the values in `TEMPLATE_CONTEXT_PROCESSORS` are specified as *strings*, which means the processors are required to be somewhere on your Python path (so you can refer to them from the setting).

Each processor is applied in order. That is, if one processor adds a variable to the context and a second processor adds a variable with the same name, the second will override the first.

Django provides a number of simple context processors, including the ones that are enabled by default:

`django.core.context_processors.auth`

If `TEMPLATE_CONTEXT_PROCESSORS` contains this processor, every `RequestContext` will contain these variables:

- `user`: A `django.contrib.auth.models.User` instance representing the current logged-in user (or an `AnonymousUser` instance, if the client isn't logged in).
- `messages`: A list of messages (as strings) for the current logged-in user. Behind the scenes, this variable calls `request.user.get_and_delete_messages()` for every request. That method collects the user's messages and deletes them from the database.
- `perms`: An instance of `django.core.context_processors.PermWrapper`, which represents the permissions the current logged-in user has.

See Chapter 14 for more information on users, permissions, and messages.

`django.core.context_processors.debug`

This processor pushes debugging information down to the template layer. If `TEMPLATE_CONTEXT_PROCESSORS` contains this processor, every `RequestContext` will contain these variables:

- `debug`: The value of your `DEBUG` setting (either `True` or `False`). You can use this variable in templates to test whether you're in debug mode.
- `sql_queries`: A list of `{'sql': ..., 'time': ...}` dictionaries representing every SQL query that has happened so far during the request and how long it took. The list is in the order in which the queries were issued.

Because debugging information is sensitive, this context processor will only add variables to the context if both of the following conditions are true:

- The `DEBUG` setting is `True`.
- The request came from an IP address in the `INTERNAL_IPS` setting.

Astute readers will notice that the `debug` template variable will never have the value `False` because, if `DEBUG` is `False`, then the `debug` template variable won't be populated in the first place.

`django.core.context_processors.i18n`

If this processor is enabled, every `RequestContext` will contain these variables:

- `LANGUAGES`: The value of the `LANGUAGES` setting.
- `LANGUAGE_CODE`: `request.LANGUAGE_CODE` if it exists; otherwise, the value of the `LANGUAGE_CODE` setting.

Appendix D provides more information about these two settings.

`django.core.context_processors.request`

If this processor is enabled, every `RequestContext` will contain a variable `request`, which is the current `HttpRequest` object. Note that this processor is not enabled by default; you have to activate it.

You might want to use this if you find your templates needing to access attributes of the current `HttpRequest` such as the IP address:

```
{{ request.REMOTE_ADDR }}
```

Guidelines for Writing Your Own Context Processors

Here are a few tips for rolling your own:

- Make each context processor responsible for the smallest subset of functionality possible. It's easy to use multiple processors, so you might as well split functionality into logical pieces for future reuse.
- Keep in mind that any context processor in `TEMPLATE_CONTEXT_PROCESSORS` will be available in every template powered by that settings file, so try to pick variable names that are unlikely to conflict with variable names your templates might be using independently. As variable names are case-sensitive, it's not a bad idea to use all caps for variables that a processor provides.
- It doesn't matter where on the filesystem they live, as long as they're on your Python path so you can point to them from the `TEMPLATE_CONTEXT_PROCESSORS` setting. With that said, the convention is to save them in a file called `context_processors.py` within your app or project.

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 his 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 bolded!

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. (For more on security, see Chapter 20.)

To avoid this problem, you have two options:

- One, you can make sure to run each untrusted variable through the `escape` filter, which converts potentially harmful HTML characters to unarmful 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 trusted 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 e-mail 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 '``', 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 just 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 on to 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 & 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 & 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 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.

Automatic Escaping of String Literals in Filter Arguments

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.

Inside Template Loading

Generally, you'll store templates in files on your filesystem, but you can use custom *template loaders* to load templates from other sources.

Django has two ways to load templates:

- `django.template.loader.get_template(template_name)`: `get_template` returns the compiled template (a `Template` object) for the template with the given name. If the template doesn't exist, a `TemplateDoesNotExist` exception will be raised.
- `django.template.loader.select_template(template_name_list)`: `select_template` is just like `get_template`, except it takes a list of template names. Of the list, it returns the first template that exists. If none of the templates exist, a `TemplateDoesNotExist` exception will be raised.

As covered in Chapter 4, each of these functions by default uses your `TEMPLATE_DIRS` setting to load templates.

Internally, however, these functions actually delegate to a template loader for the heavy lifting.

Some of loaders are disabled by default, but you can activate them by editing the `TEMPLATE_LOADERS` setting. `TEMPLATE_LOADERS` should be a tuple of strings, where each string represents a template loader. These template loaders ship with Django:

- `django.template.loaders.filesystem.load_template_source`: This loader loads templates from the filesystem, according to `TEMPLATE_DIRS`. It is enabled by default.
- `django.template.loaders.app_directories.load_template_source`: This loader loads templates from Django applications on the filesystem. For each application in `INSTALLED_APPS`, the loader looks for a `templates` subdirectory. If the directory exists, Django looks for templates there.

This means you can store templates with your individual applications, making it easy to distribute Django applications with default templates. For example, if `INSTALLED_APPS` contains `('myproject.polls', 'myproject.music')`, then `get_template('foo.html')` will look for templates in this order:

- `/path/to/myproject/polls/templates/foo.html`
- `/path/to/myproject/music/templates/foo.html`

Note that the loader performs an optimization when it is first imported: it caches a list of which `INSTALLED_APPS` packages have a `templates` subdirectory.

This loader is enabled by default.

- `django.template.loaders.eggs.load_template_source`: This loader is just like `app_directories`, except it loads templates from Python eggs rather than from the filesystem. This loader is disabled by default; you'll need to enable it if you're using eggs to distribute your application. (Python eggs are a way of compressing Python code into a single file.)

Django uses the template loaders in order according to the `TEMPLATE_LOADERS` setting. It uses each loader until a loader finds a match.

Extending the Template System

Now that you understand a bit more about the internals of the template system, let's look at how to extend the system with custom code.

Most template customization comes in the form of custom template tags and/or filters. Although the Django template language comes with many built-in tags and filters, you'll probably assemble your own libraries of tags and filters that fit your own needs. Fortunately, it's quite easy to define your own functionality.

Creating a Template Library

Whether you're writing custom tags or filters, the first thing to do is to create a **template library** – a small bit of infrastructure Django can hook into.

Creating a template library is a two-step process:

- First, decide which Django application should house the template library. If you've created an app via `manage.py startapp`, you can put it in there, or you can create another app solely for the template library. We'd recommend the latter, because your filters might be useful to you in future projects.

Whichever route you take, make sure to add the app to your `INSTALLED_APPS` setting. We'll explain this shortly.

- Second, create a `templatetags` directory in the appropriate Django application's package. It should be on the same level as `models.py`, `views.py`, and so forth. For example:

```
books/  
    __init__.py  
    models.py  
    templatetags/  
    views.py
```

Create two empty files in the `templatetags` directory: an `__init__.py` file (to indicate to Python that this is a package containing Python code) and a file that will contain your custom tag/filter definitions. The name of the latter file is what you'll use to load the tags later. For example, if your custom tags/filters are in a file called `poll_extras.py`, you'd write the following in a template:

```
{% load poll_extras %}
```

The `{% load %}` tag looks at your `INSTALLED_APPS` setting and only allows the loading of template libraries within installed Django applications. This is a security feature; it allows you to host Python code for many template libraries on a single computer without enabling access to all of them for every Django installation.

If you write a template library that isn't tied to any particular models/views, it's valid and quite normal to have a Django application package that contains only a `templatetags` package. There's no limit on how many modules you put in the `templatetags` package. Just keep in mind that a `{% load %}` statement will load tags/filters for the given Python module name, not the name of the application.

Once you've created that Python module, you'll just have to write a bit of Python code, depending on whether you're writing filters or tags.

To be a valid tag library, the module must contain a module-level variable named `register` that is an instance of `template.Library`. This is the data structure in which all the tags and filters are registered. So, near the top of your module, insert the following:

```
from django import template  
  
register = template.Library()
```

Note

For a fine selection of examples, read the source code for Django's default filters and tags. They're in `django/template/defaultfilters.py` and `django/template/defaulttags.py`, respectively. Some applications in `django.contrib` also contain template libraries.

Once you've created this `register` variable, you'll use it to create template filters and tags.

Writing Custom Template Filters

Custom filters are just Python functions that take one or two arguments:

- The value of the variable (input)
- The value of the argument, which can have a default value or be left out altogether

For example, in the filter `{{ var|foo:"bar" }}`, the filter `foo` would be passed the contents of the variable `var` and the argument `"bar"`.

Filter functions should always return something. They shouldn't raise exceptions, and they should fail silently. If there's an error, they should return either the original input or an empty string, whichever makes more sense.

Here's an example filter definition:

```
def cut(value, arg):  
    "Removes all values of arg from the given string"  
    return value.replace(arg, '')
```

And here's an example of how that filter would be used to cut spaces from a variable's value:

```
{{ somevariable|cut:" " }}
```

Most filters don't take arguments. In this case, just leave the argument out of your function:

```
def lower(value): # Only one argument.  
    "Converts a string into all lowercase"  
    return value.lower()
```

When you've written your filter definition, you need to register it with your `Library` instance, to make it available to Django's template language:

```
register.filter('cut', cut)  
register.filter('lower', lower)
```

The `Library.filter()` method takes two arguments:

- The name of the filter (a string)
- The filter function itself

If you're using Python 2.4 or above, you can use `register.filter()` as a decorator instead:

```
@register.filter(name='cut')  
def cut(value, arg):  
    return value.replace(arg, '')  
  
@register.filter  
def lower(value):  
    return value.lower()
```

If you leave off the `name` argument, as in the second example, Django will use the function's name as the filter name.

Here, then, is a complete template library example, supplying the `cut` filter:

```
from django import template  
  
register = template.Library()  
  
@register.filter(name='cut')  
def cut(value, arg):  
    return value.replace(arg, '')
```

Writing Custom Template Tags

Tags are more complex than filters, because tags can do nearly anything.

Chapter 4 describes how the template system works in a two-step process: compiling and rendering. To define a custom template tag, you need to tell Django how to manage *both* of these steps when it gets to your tag.

When Django compiles a template, it splits the raw template text into *nodes*. Each node is an instance of `django.template.Node` and has a `render()` method. Thus, a compiled template is simply a list of `Node` objects. For example, consider this template:

```
Hello, {{ person.name }}.

{% ifequal name.birthday today %}
    Happy birthday!
{% else %}
    Be sure to come back on your birthday
    for a splendid surprise message.
{% endifequal %}
```

In compiled template form, this template is represented as this list of nodes:

- Text node: "Hello, "
- Variable node: `person.name`
- Text node: ".\n\n"
- IfEqual node: `name.birthday` and `today`

When you call `render()` on a compiled template, the template calls `render()` on each `Node` in its node list, with the given context. The results are all concatenated together to form the output of the template. Thus, to define a custom template tag, you specify how the raw template tag is converted into a `Node` (the compilation function) and what the node's `render()` method does.

In the sections that follow, we cover all the steps in writing a custom tag.

Writing the Compilation Function

For each template tag the parser encounters, it calls a Python function with the tag contents and the parser object itself. This function is responsible for returning a `Node` instance based on the contents of the tag.

For example, let's write a template tag, `{% current_time %}`, that displays the current date/time, formatted according to a parameter given in the tag, in `strftime` syntax (see <http://www.djangoproject.com/r/python/strftime/>). It's a good idea to decide the tag syntax before anything else. In our case, let's say the tag should be used like this:

```
<p>The time is {% current_time "%Y-%m-%d %I:%M %p" %}.</p>
```

Note

Yes, this template tag is redundant—Django's default `{% now %}` tag does the same task with simpler syntax. This template tag is presented here just for example purposes.

The parser for this function should grab the parameter and create a `Node` object:

```
from django import template
```

```

register = template.Library()

def do_current_time(parser, token):
    try:
        # split_contents() knows not to split quoted strings.
        tag_name, format_string = token.split_contents()
    except ValueError:
        msg = '%r tag requires a single argument' % token.split_contents()[0]
        raise template.TemplateSyntaxError(msg)
    return CurrentTimeNode(format_string[1:-1])

```

There's a lot going on here:

- Each template tag compilation function takes two arguments, `parser` and `token`. `parser` is the template parser object. We don't use it in this example. `token` is the token currently being parsed by the parser.
- `token.contents` is a string of the raw contents of the tag. In our example, it's `'current_time "%Y-%m-%d %I:%M %p"'`.
- The `token.split_contents()` method separates the arguments on spaces while keeping quoted strings together. Avoid using `token.contents.split()` (which just uses Python's standard string-splitting semantics). It's not as robust, as it naively splits on *all* spaces, including those within quoted strings.
- This function is responsible for raising `django.template.TemplateSyntaxError`, with helpful messages, for any syntax error.
- Don't hard-code the tag's name in your error messages, because that couples the tag's name to your function. `token.split_contents()[0]` will *always* be the name of your tag – even when the tag has no arguments.
- The function returns a `CurrentTimeNode` (which we'll create shortly) containing everything the node needs to know about this tag. In this case, it just passes the argument `"%Y-%m-%d %I:%M %p"`. The leading and trailing quotes from the template tag are removed with `format_string[1:-1]`.
- Template tag compilation functions *must* return a `Node` subclass; any other return value is an error.

Writing the Template Node

The second step in writing custom tags is to define a `Node` subclass that has a `render()` method. Continuing the preceding example, we need to define `CurrentTimeNode`:

```

import datetime

class CurrentTimeNode(template.Node):
    def __init__(self, format_string):
        self.format_string = str(format_string)

    def render(self, context):
        now = datetime.datetime.now()
        return now.strftime(self.format_string)

```

These two functions (`__init__()` and `render()`) map directly to the two steps in template processing (compilation and rendering). Thus, the initialization function only needs to store the format string for later use, and the `render()` function does the real work.

Like template filters, these rendering functions should fail silently instead of raising errors. The only time that template tags are allowed to raise errors is at compilation time.

Registering the Tag

Finally, you need to register the tag with your module's `Library` instance. Registering custom tags is very similar to registering custom filters (as explained above). Just instantiate a `template.Library` instance and call its `tag()` method. For example:

```
register.tag('current_time', do_current_time)
```

The `tag()` method takes two arguments:

- The name of the template tag (string).
- The compilation function.

As with filter registration, it is also possible to use `register.tag` as a decorator in Python 2.4 and above:

```
@register.tag(name="current_time")
def do_current_time(parser, token):
    # ...

@register.tag
def shout(parser, token):
    # ...
```

If you leave off the `name` argument, as in the second example, Django will use the function's name as the tag name.

Setting a Variable in the Context

The previous section's example simply returned a value. Often it's useful to set template variables instead of returning values. That way, template authors can just use the variables that your template tags set.

To set a variable in the context, use dictionary assignment on the context object in the `render()` method. Here's an updated version of `CurrentTimeNode` that sets a template variable, `current_time`, instead of returning it:

```
class CurrentTimeNode2(template.Node):
    def __init__(self, format_string):
        self.format_string = str(format_string)

    def render(self, context):
        now = datetime.datetime.now()
        context['current_time'] = now.strftime(self.format_string)
        return ''
```

(We'll leave the creation of a `do_current_time2` function, plus the registration of that function to a `current_time2` template tag, as exercises for the reader.)

Note that `render()` returns an empty string. `render()` should always return a string, so if all the template tag does is set a variable, `render()` should return an empty string.

Here's how you'd use this new version of the tag:

```
{% current_time2 "%Y-%M-%d %I:%M %p" %}
<p>The time is {{ current_time }}.</p>
```

But there's a problem with `CurrentTimeNode2`: the variable name `current_time` is hard-coded. This means you'll

need to make sure your template doesn't use `{{ current_time }}` anywhere else, because `{% current_time2 %}` will blindly overwrite that variable's value.

A cleaner solution is to make the template tag specify the name of the variable to be set, like so:

```
{% get_current_time "%Y-%M-%d %I:%M %p" as my_current_time %}
<p>The current time is {{ my_current_time }}.</p>
```

To do so, you'll need to refactor both the compilation function and the `Node` class, as follows:

```
import re

class CurrentTimeNode3(template.Node):
    def __init__(self, format_string, var_name):
        self.format_string = str(format_string)
        self.var_name = var_name

    def render(self, context):
        now = datetime.datetime.now()
        context[self.var_name] = now.strftime(self.format_string)
        return ''

def do_current_time(parser, token):
    # This version uses a regular expression to parse tag contents.
    try:
        # Splitting by None == splitting by spaces.
        tag_name, arg = token.contents.split(None, 1)
    except ValueError:
        msg = '%r tag requires arguments' % token.contents[0]
        raise template.TemplateSyntaxError(msg)

    m = re.search(r'(.*) as (\w+)', arg)
    if m:
        fmt, var_name = m.groups()
    else:
        msg = '%r tag had invalid arguments' % tag_name
        raise template.TemplateSyntaxError(msg)

    if not (fmt[0] == fmt[-1] and fmt[0] in ('"', "'")):
        msg = "%r tag's argument should be in quotes" % tag_name
        raise template.TemplateSyntaxError(msg)

    return CurrentTimeNode3(fmt[1:-1], var_name)
```

Now `do_current_time()` passes the format string and the variable name to `CurrentTimeNode3`.

Parsing Until Another Template Tag

Template tags can work as blocks containing other tags (like `{% if %}`, `{% for %}`, etc.). To create a template tag like this, use `parser.parse()` in your compilation function.

Here's how the standard `{% comment %}` tag is implemented:

```
def do_comment(parser, token):
    nodelist = parser.parse(('endcomment',))
    parser.delete_first_token()
    return CommentNode()

class CommentNode(template.Node):
    def render(self, context):
        return ''
```

`parser.parse()` takes a tuple of names of template tags to parse until. It returns an instance of `django.template.NodeList`, which is a list of all `Node` objects that the parser encountered *before* it encountered any of the tags named in the tuple.

So in the preceding example, `nodelist` is a list of all nodes between `{% comment %}` and `{% endcomment %}`, not counting `{% comment %}` and `{% endcomment %}` themselves.

After `parser.parse()` is called, the parser hasn't yet "consumed" the `{% endcomment %}` tag, so the code needs to explicitly call `parser.delete_first_token()` to prevent that tag from being processed twice.

Then `CommentNode.render()` simply returns an empty string. Anything between `{% comment %}` and `{% endcomment %}` is ignored.

Parsing Until Another Template Tag and Saving Contents

In the previous example, `do_comment()` discarded everything between `{% comment %}` and `{% endcomment %}`. It's also possible to do something with the code between template tags instead.

For example, here's a custom template tag, `{% upper %}`, that capitalizes everything between itself and `{% endupper %}`:

```
{% upper %}
    This will appear in uppercase, {{ user_name }}.
{% endupper %}
```

As in the previous example, we'll use `parser.parse()`. This time, we pass the resulting `nodelist` to `Node`:

```
def do_upper(parser, token):
    nodelist = parser.parse(('endupper',))
    parser.delete_first_token()
    return UpperNode(nodelist)

class UpperNode(template.Node):
    def __init__(self, nodelist):
        self.nodelist = nodelist

    def render(self, context):
        output = self.nodelist.render(context)
        return output.upper()
```

The only new concept here is `self.nodelist.render(context)` in `UpperNode.render()`. This simply calls `render()` on each `Node` in the node list.

For more examples of complex rendering, see the source code for `{% if %}`, `{% for %}`, `{% ifequal %}`, and `{% ifchanged %}`. They live in `django/template/defaulttags.py`.

Shortcut for Simple Tags

Many template tags take a single argument – a string or a template variable reference – and return a string after doing some processing based solely on the input argument and some external information. For example, the `current_time` tag we wrote earlier is of this variety. We give it a format string, and it returns the time as a string.

To ease the creation of these types of tags, Django provides a helper function, `simple_tag`. This function, which is a method of `django.template.Library`, takes a function that accepts one argument, wraps it in a `render` function and the other necessary bits mentioned previously, and registers it with the template system.

Our earlier `current_time` function could thus be written like this:

```
def current_time(format_string):
    try:
        return datetime.datetime.now().strftime(str(format_string))
    except UnicodeEncodeError:
        return ''

register.simple_tag(current_time)
```

In Python 2.4, the decorator syntax also works:

```
@register.simple_tag
def current_time(token):
    # ...
```

Notice a couple of things to notice about the `simple_tag` helper function:

- Only the (single) argument is passed into our function.
- Checking for the required number of arguments has already been done by the time our function is called, so we don't need to do that.
- The quotes around the argument (if any) have already been stripped away, so we receive a plain Unicode string.

Inclusion Tags

Another common template tag is the type that displays some data by rendering *another* template. For example, Django's admin interface uses custom template tags to display the buttons along the bottom of the "add/change" form pages. Those buttons always look the same, but the link targets change depending on the object being edited. They're a perfect case for using a small template that is filled with details from the current object.

These sorts of tags are called *inclusion tags*. Writing inclusion tags is probably best demonstrated by example. Let's write a tag that produces a list of books for a given `Author` object. We'll use the tag like this:

```
{% books_for_author author %}
```

The result will be something like this:

```
<ul>
  <li>The Cat In The Hat</li>
  <li>Hop On Pop</li>
  <li>Green Eggs And Ham</li>
</ul>
```

First, we define the function that takes the argument and produces a dictionary of data for the result. Notice that we need to return only a dictionary, not anything more complex. This will be used as the context for the template fragment:

```
def books_for_author(author):
    books = Book.objects.filter(authors__id=author.id)
    return {'books': books}
```

Next, we create the template used to render the tag's output. Following our example, the template is very simple:

```
<ul>
{% for book in books %}
    <li>{{ book.title }}</li>
{% endfor %}
</ul>
```

Finally, we create and register the inclusion tag by calling the `inclusion_tag()` method on a `Library` object.

Following our example, if the preceding template is in a file called `book_snippet.html`, we register the tag like this:

```
register.inclusion_tag('book_snippet.html')(books_for_author)
```

Python 2.4 decorator syntax works as well, so we could have written this, instead:

```
@register.inclusion_tag('book_snippet.html')
def books_for_author(author):
    # ...
```

Sometimes, your inclusion tags need access to values from the parent template's context. To solve this, Django provides a `takes_context` option for inclusion tags. If you specify `takes_context` in creating an inclusion tag, the tag will have no required arguments, and the underlying Python function will have one argument: the template context as of when the tag was called.

For example, say you're writing an inclusion tag that will always be used in a context that contains `home_link` and `home_title` variables that point back to the main page. Here's what the Python function would look like:

```
@register.inclusion_tag('link.html', takes_context=True)
def jump_link(context):
    return {
        'link': context['home_link'],
        'title': context['home_title'],
    }
```

(Note that the first parameter to the function *must* be called `context`.)

The template `link.html` might contain the following:

```
Jump directly to <a href="{{ link }}">{{ title }}</a>.
```

Then, anytime you want to use that custom tag, load its library and call it without any arguments, like so:

```
{% jump_link %}
```

Writing Custom Template Loaders

Django's built-in template loaders (described in the "Inside Template Loading" section above) will usually cover all your template-loading needs, but it's pretty easy to write your own if you need special loading logic. For example, you could load templates from a database, or directly from a Subversion repository using Subversion's Python bindings, or (as shown shortly) from a ZIP archive.

A template loader – that is, each entry in the `TEMPLATE_LOADERS` setting – is expected to be a callable object with this interface:

```
load_template_source(template_name, template_dirs=None)
```

The `template_name` argument is the name of the template to load (as passed to `loader.get_template()` or `loader.select_template()`), and `template_dirs` is an optional list of directories to search instead of `TEMPLATE_DIRS`.

If a loader is able to successfully load a template, it should return a tuple: (`template_source`, `template_path`). Here, `template_source` is the template string that will be compiled by the template engine, and `template_path` is the path the template was loaded from. That path might be shown to the user for debugging purposes, so it should quickly identify where the template was loaded from.

If the loader is unable to load a template, it should raise `django.template.TemplateDoesNotExist`.

Each loader function should also have an `is_usable` function attribute. This is a Boolean that informs the template engine whether this loader is available in the current Python installation. For example, the eggs loader (which is capable of loading templates from Python eggs) sets `is_usable` to `False` if the `pkg_resources` module isn't installed, because `pkg_resources` is necessary to read data from eggs.

An example should help clarify all of this. Here's a template loader function that can load templates from a ZIP file. It uses a custom setting, `TEMPLATE_ZIP_FILES`, as a search path instead of `TEMPLATE_DIRS`, and it expects each item on that path to be a ZIP file containing templates:

```
from django.conf import settings
from django.template import TemplateDoesNotExist
import zipfile

def load_template_source(template_name, template_dirs=None):
    "Template loader that loads templates from a ZIP file."

    template_zipfiles = getattr(settings, "TEMPLATE_ZIP_FILES", [])

    # Try each ZIP file in TEMPLATE_ZIP_FILES.
    for fname in template_zipfiles:
        try:
            z = zipfile.ZipFile(fname)
            source = z.read(template_name)
        except (IOError, KeyError):
            continue
        z.close()
        # We found a template, so return the source.
        template_path = "%s:%s" % (fname, template_name)
        return (source, template_path)

    # If we reach here, the template couldn't be loaded
    raise TemplateDoesNotExist(template_name)
```

```
# This loader is always usable (since zipfile is included with Python)
load_template_source.is_usable = True
```

The only step left if we want to use this loader is to add it to the `TEMPLATE_LOADERS` setting. If we put this code in a package called `mysite.zip_loader`, then we add `mysite.zip_loader.load_template_source` to `TEMPLATE_LOADERS`.

Configuring the Template System in Standalone Mode

Note

This section is only of interest to people trying to use the template system as an output component in another application. If you are using the template system as part of a Django application, the information presented here doesn't apply to you.

Normally, Django loads all the configuration information it needs from its own default configuration file, combined with the settings in the module given in the `DJANGO_SETTINGS_MODULE` environment variable. (This was explained in "A special Python prompt" in Chapter 4.) But if you're using the template system independently of the rest of Django, the environment variable approach isn't very convenient, because you probably want to configure the template system in line with the rest of your application rather than dealing with settings files and pointing to them via environment variables.

To solve this problem, you need to use the manual configuration option described fully in Appendix D. In a nutshell, you need to import the appropriate pieces of the template system and then, *before* you call any of the template functions, call `django.conf.settings.configure()` with any settings you wish to specify.

You might want to consider setting at least `TEMPLATE_DIRS` (if you are going to use template loaders), `DEFAULT_CHARSET` (although the default of `utf-8` is probably fine) and `TEMPLATE_DEBUG`. All available settings are described in Appendix D, and any setting starting with `TEMPLATE_` is of obvious interest.

What's Next

Continuing this section's theme of advanced topics, the [next chapter](#) covers advanced usage of Django models.