MODULE-2

CHAPTER 4

The Django Template System

In the previous chapter, you may have noticed something peculiar in how we returned the text in our example views. Namely, the HTML was hard-coded directly in our Python code.

This arrangement leads to several problems:

- Any change to the design of the page requires a change to the Python code. The design of a site tends to change far more frequently than the underlying Python code, so it would be convenient if the design could change without needing to modify the Python code.
- Writing Python code and designing HTML are two different disciplines, and most professional Web development environments split these responsibilities between separate people (or even separate departments). Designers and HTML/CSS coders shouldn't have to edit Python code to get their job done; they should deal with HTML.
- Similarly, it's most efficient if programmers can work on Python code and designers can work on templates at the same time, rather than one person waiting for the other to finish editing a single file that contains both Python and HTML.

For these reasons, it's much cleaner and more maintainable to separate the design of the page from the Python code itself. We can do this with Django's *template system*, which we discuss in this chapter.

Template System Basics

A Django template is a string of text that is intended to separate the presentation of a document from its data. A template defines placeholders and various bits of basic logic (i.e., template tags) that regulate how the document should be displayed. Usually, templates are used for producing HTML, but Django templates are equally capable of generating any text-based format.

Let's dive in with a simple example template. This template describes an HTML page that thanks a person for placing an order with a company. Think of it as a form letter:

```
<html>
<head><title>Ordering notice</title></head>

<body>

<p>Dear {{ person_name }},</p>
```

```
<p>Thanks for placing an order from {{ company }}. It's scheduled to
ship on {{ ship_date|date:"F j, Y" }}.</p>

<p>Here are the items you've ordered:</p>

<ul>
{% for item in item_list %}
<li>{{ item }}</li>
{% endfor %}
</ul>

{% if ordered_warranty %}
<p>Your warranty information will be included in the packaging.</p>
{% endif %}

<p>Sincerely,<br />{{ company }}</p>

</body>
</html>
```

This template is basic HTML with some variables and template tags thrown in. Let's step through it:

- Any text surrounded by a pair of braces (e.g., `{{ person_name }}`) is a *variable*. This means "insert the value of the variable with the given name." How do we specify the values of the variables? We'll get to that in a moment.

- Any text that's surrounded by curly braces and percent signs (e.g., `{% if ordered_warranty %}`) is a *template tag*. The definition of a tag is quite broad: a tag just tells the template system to "do something."

  This example template contains two tags: the `{% for item in item_list %}` tag (a `for` tag) and the `{% if ordered_warranty %}` tag (an `if` tag). A `for` tag acts as a simple loop construct, letting you loop over each item in a sequence. An `if` tag, as you may expect, acts as a logical "if" statement. In this particular case, the tag checks whether the value of the `ordered_warranty` variable evaluates to `True`. If it does, the template system will display everything between the `{% if ordered_warranty %}` and `{% endif %}`. If not, the template system won't display it. The template system also supports `{% else %}` and other various logic statements.

- Finally, the second paragraph of this template has an example of a *filter*, with which you can alter the display of a variable. In this example, `{{ ship_date|date:"F j, Y" }}`, we're passing the `ship_date` variable to the `date` filter, giving the `date` filter the argument `"F j, Y"`. The `date` filter formats dates in a given format, as specified by that argument. Filters are attached using a pipe character (`|`), as a reference to Unix pipes.

Each Django template has access to several built-in tags and filters, many of which are discussed in the sections that follow. Appendix F contains the full list of tags and filters, and it's a good idea to familiarize yourself with that list so you know what's possible. It's also possible to create your own filters and tags, which we cover in Chapter 10.

Using the Template System

To use the template system in Python code, just follow these two steps:

1. Create a `Template` object by providing the raw template code as a string. Django also offers a way to create `Template` objects by designating the path to a template file on the filesystem; we'll examine that in a bit.
2. Call the `render()` method of the `Template` object with a given set of variables (i.e., the context). This returns a fully rendered template as a string, with all of the variables and block tags evaluated according to the context.

The following sections describe each step in more detail.

Creating Template Objects

The easiest way to create a `Template` object is to instantiate it directly. The `Template` class lives in the `django.template` module, and the constructor takes one argument, the raw template code. Let's dip into the Python interactive interpreter to see how this works in code.

INTERACTIVE INTERPRETER EXAMPLES

Throughout this book, we feature example Python interactive interpreter sessions. You can recognize these examples by the triple >> greater-than signs (Python prompt)>>>)>greater-than signs (>>>), which designate the interpreter's prompt. If you're copying examples from this book, don't copy those greater-than signs.

Multiline statements in the interactive interpreter are padded with three dots (`...`), for example:

```
>>> print """This is a
... string that spans
... three lines."""
This is a
string that spans
three lines.
>>> def my_function(value):
...     print value
>>> my_function('hello')
hello
```

Those three dots at the start of the additional lines are inserted by the Python shell—they're not part of our input. We include them here to be faithful to the actual output of the interpreter. If you copy our examples to follow along, don't copy those dots.

From within the project directory created by `django-admin.py startproject` (as covered in Chapter 2), type `python manage.py shell` to start the interactive interpreter. Here's a basic walk-through:

```
>>> from django.template import Template
>>> t = Template("My name is {{ name }}.")
>>> print t
```

If you're following along interactively, you'll see something like this:

```
<django.template.Template object at 0xb7d5f24c>
```

That `0xb7d5f24c` part will be different every time, and it doesn't really matter; it's simply the Python "identity" of the `Template` object.

■ **Note** When using Django, you need to tell Django which settings to use. Interactively, this is typically done using `python manage.py shell`, but you've got a few other options, as described in Appendix E.

When you create a `Template` object, the template system compiles the raw template code into an internal, optimized form, ready for rendering. But if your template code includes any syntax errors, the call to `Template()` will cause a `TemplateSyntaxError` exception:

```
>>> from django.template import Template
>>> t = Template('{% notatag %} ')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  ...
  django.template.TemplateSyntaxError: Invalid block tag: 'notatag'
```

The system raises a `TemplateSyntaxError` exception for any of the following cases:

- Invalid block tags
- Invalid arguments to valid block tags
- Invalid filters
- Invalid arguments to valid filters
- Invalid template syntax
- Unclosed block tags (for block tags that require closing tags)

Rendering a Template

Once you have a `Template` object, you can pass it data by giving it a *context*. A context is simply a set of variables and their associated values. A template uses this to populate its variable tags and evaluate its block tags.

A context is represented in Django by the `Context` class, which lives in the `django.template` module. Its constructor takes one optional argument: a dictionary mapping variable names to variable values. Call the `Template` object's `render()` method with the context to "fill" the template:

```
>>> from django.template import Context, Template
>>> t = Template("My name is {{ name }}.")
>>> c = Context({"name": "Stephane"})
>>> t.render(c)
'My name is Stephane.'
```

■ **Note** A Python dictionary is a mapping between known keys and variable values. A `Context` is similar to a dictionary, but a `Context` provides additional functionality, as covered in Chapter 10.

Variable names must begin with a letter (A–Z or a–z) and may contain digits, underscores, and dots. (Dots are a special case we'll get to in a moment.) Variable names are case sensitive.

Here's an example of template compilation and rendering, using the sample template from the beginning of this chapter:

```
>>> from django.template import Template, Context
>>> raw_template = """<p>Dear {{ person_name }},</p>
...
... <p>Thanks for ordering {{ product }} from {{ company }}. It's scheduled
... to ship on {{ ship_date|date:"F j, Y" }}.</p>
...
... {% if ordered_warranty %}
... <p>Your warranty information will be included in the packaging.</p>
... {% endif %}
...
... <p>Sincerely,<br />{{ company }}</p>"""
>>> t = Template(raw_template)
>>> import datetime
>>> c = Context({'person_name': 'John Smith',
...     'product': 'Super Lawn Mower',
...     'company': 'Outdoor Equipment',
...     'ship_date': datetime.date(2009, 4, 2),
...     'ordered_warranty': True})
>>> t.render(c)
"<p>Dear John Smith,</p>\n\n<p>Thanks for ordering Super Lawn Mower from
Outdoor Equipment. It's scheduled \nto ship on April 2, 2009.</p>\n\n\n<p>Your
warranty information will be included in the
packaging.</p>\n\n\n<p>Sincerely,<br />Outdoor Equipment</p>"
```

Let's step through this code one statement at a time:

- First, we import the classes `Template` and `Context`, which both live in the module `django.template`.
- We save the raw text of our template into the variable `raw_template`. Note that we use triple quote marks to designate the string, because it wraps over multiple lines; in Python code, strings designated with single quote marks cannot be wrapped over multiple lines.
- Next, we create a template object, `t`, by passing `raw_template` to the `Template` class constructor.
- We import the `datetime` module from Python's standard library, because we'll need it in the following statement.

- Then, we create a `Context` object, `c`. The `Context` constructor takes a Python dictionary, which maps variable names to values. Here, for example, we specify that the `person_name` is `'John Smith'`, `product` is `'Super Lawn Mower'`, and so forth.
- Finally, we call the `render()` method on our template object, passing it the context. This returns the rendered template—that is, it replaces template variables with the actual values of the variables, and it executes any block tags.

  Note that the warranty paragraph was displayed because the `ordered_warranty` variable evaluated to `True`. Also note the date, `April 2, 2009`, which is displayed according to the format string `'F j, Y'`. (We explain format strings for the `date` filter shortly.)

  If you're new to Python, you may wonder why this output includes newline characters (`'\n'`) rather than displaying the line breaks. That's happening because of a subtlety in the Python interactive interpreter: the call to `t.render(c)` returns a string, and by default the interactive interpreter displays the *representation* of the string, rather than the printed value of the string. If you want to see the string with line breaks displayed as true line breaks rather than `'\n'` characters, use the `print` statement: `print t.render(c)`.

Those are the fundamentals of using the Django template system: just write a template, create a `Template` object, create a `Context`, and call the `render()` method.

Multiple Contexts, Same Template

Once you have a `Template` object, you can render multiple contexts through it, for example:

```
>>> from django.template import Template, Context
>>> t = Template('Hello, {{ name }}')
>>> print t.render(Context({'name': 'John'}))
Hello, John
>>> print t.render(Context({'name': 'Julie'}))
Hello, Julie
>>> print t.render(Context({'name': 'Pat'}))
Hello, Pat
```

Whenever you're using the same template source to render multiple contexts like this, it's more efficient to create the `Template` object *once*, and then call `render()` on it multiple times:

```
# Bad
for name in ('John', 'Julie', 'Pat'):
    t = Template('Hello, {{ name }}')
    print t.render(Context({'name': name}))

# Good
t = Template('Hello, {{ name }}')
for name in ('John', 'Julie', 'Pat'):
    print t.render(Context({'name': name}))
```

Django's template parsing is quite fast. Behind the scenes, most of the parsing happens via a single call to a short regular expression. This is in stark contrast to XML-based template

engines, which incur the overhead of an XML parser and tend to be orders of magnitude slower than Django's template rendering engine.

Context Variable Lookup

In the examples so far, we've passed simple values in the contexts—mostly strings, plus a `datetime.date` example. However, the template system elegantly handles more complex data structures, such as lists, dictionaries, and custom objects.

The key to traversing complex data structures in Django templates is the dot character (`.`). Use a dot to access dictionary keys, attributes, indices, or methods of an object.

This is best illustrated with a few examples. For instance, suppose you're passing a Python dictionary to a template. To access the values of that dictionary by dictionary key, use a dot:

```
>>> from django.template import Template, Context
>>> person = {'name': 'Sally', 'age': '43'}
>>> t = Template('{{ person.name }} is {{ person.age }} years old.')
>>> c = Context({'person': person})
>>> t.render(c)
'Sally is 43 years old.'
```

Similarly, dots also allow access of object attributes. For example, a Python `datetime.date` object has `year`, `month`, and `day` attributes, and you can use a dot to access those attributes in a Django template:

```
>>> from django.template import Template, Context
>>> import datetime
>>> d = datetime.date(1993, 5, 2)
>>> d.year
1993
>>> d.month
5
>>> d.day
2
>>> t = Template('The month is {{ date.month }} and the year is {{ date.year }}.')
>>> c = Context({'date': d})
>>> t.render(c)
'The month is 5 and the year is 1993.'
```

This example uses a custom class:

```
>>> from django.template import Template, Context
>>> class Person(object):
...     def __init__(self, first_name, last_name):
...         self.first_name, self.last_name = first_name, last_name
>>> t = Template('Hello, {{ person.first_name }} {{ person.last_name }}.')
>>> c = Context({'person': Person('John', 'Smith')})
>>> t.render(c)
'Hello, John Smith.'
```

Dots are also used to call methods on objects. For example, each Python string has the methods `upper()` and `isdigit()`, and you can call those in Django templates using the same dot syntax:

```
>>> from django.template import Template, Context
>>> t = Template('{{ var }} -- {{ var.upper }} -- {{ var.isdigit }}')
>>> t.render(Context({'var': 'hello'}))
'hello -- HELLO -- False'
>>> t.render(Context({'var': '123'}))
'123 -- 123 -- True'
```

Note that you don't include parentheses in the method calls. Also, it's not possible to pass arguments to the methods; you can only call methods that have no required arguments. (We explain this philosophy later in this chapter.)

Finally, dots are also used to access list indices, for example:

```
>>> from django.template import Template, Context
>>> t = Template('Item 2 is {{ items.2 }}.')
>>> c = Context({'items': ['apples', 'bananas', 'carrots']})
>>> t.render(c)
'Item 2 is carrots.'
```

Negative list indices are not allowed. For example, the template variable `{{ items.-1 }}` would cause a `TemplateSyntaxError`.

■ **Note** Python lists have 0-based indices so that the first item is at index 0, the second is at index 1, and so on.

The dot lookups can be summarized like this: when the template system encounters a dot in a variable name, it tries the following lookups, in this order:

- Dictionary lookup (e.g., `foo["bar"]`)
- Attribute lookup (e.g., `foo.bar`)
- Method call (e.g., `foo.bar()`)
- List-index lookup (e.g., `foo[bar]`)

The system uses the first lookup type that works. It's short-circuit logic.

Dot lookups can be nested multiple levels deep. For instance, the following example uses `{{ person.name.upper }}`, which translates into a dictionary lookup (`person['name']`) and then a method call (`upper()`):

```
>>> from django.template import Template, Context
>>> person = {'name': 'Sally', 'age': '43'}
>>> t = Template('{{ person.name.upper }} is {{ person.age }} years old.')
>>> c = Context({'person': person})
```

```
>>> t.render(c)
'SALLY is 43 years old.'
```

Method Call Behavior

Method calls are slightly more complex than the other lookup types. Here are some things to keep in mind:

- If, during the method lookup, a method raises an exception, the exception will be propagated, unless the exception has a `silent_variable_failure` attribute whose value is `True`. If the exception *does* have a `silent_variable_failure` attribute, the variable will render as an empty string, for example:

```
>>> t = Template("My name is {{ person.first_name }}.")
>>> class PersonClass3:
...     def first_name(self):
...         raise AssertionError, "foo"
>>> p = PersonClass3()
>>> t.render(Context({"person": p}))
Traceback (most recent call last):
...
AssertionError: foo

>>> class SilentAssertionError(AssertionError):
...     silent_variable_failure = True
>>> class PersonClass4:
...     def first_name(self):
...         raise SilentAssertionError
>>> p = PersonClass4()
>>> t.render(Context({"person": p}))
"My name is ."
```

- A method call will work only if the method has no required arguments. Otherwise, the system will move to the next lookup type (list-index lookup).

- Obviously, some methods have side effects, and it would be foolish at best, and possibly even a security hole, to allow the template system to access them.

  Say, for instance, you have a `BankAccount` object that has a `delete()` method. A template shouldn't be allowed to include something like `{{ account.delete }}`. To prevent this, set the function attribute `alters_data` on the method:

```
def delete(self):
    # Delete the account
delete.alters_data = True
```

  The template system won't execute any method marked in this way. In other words, if a template includes `{{ account.delete }}`, that tag will not execute the `delete()` method. It will fail silently.

How Invalid Variables Are Handled

By default, if a variable doesn't exist, the template system renders it as an empty string, failing silently, for example:

```
>>> from django.template import Template, Context
>>> t = Template('Your name is {{ name }}.')
>>> t.render(Context())
'Your name is .'
>>> t.render(Context({'var': 'hello'}))
'Your name is .'
>>> t.render(Context({'NAME': 'hello'}))
'Your name is .'
>>> t.render(Context({'Name': 'hello'}))
'Your name is .'
```

The system fails silently rather than raising an exception because it's intended to be resilient to human error. In this case, all of the lookups failed because variable names have the wrong case or name. In the real world, it's unacceptable for a Web site to become inaccessible due to a small template syntax error.

Note that it's possible to change Django's default behavior in this regard, by tweaking a setting in your Django configuration. We discuss this further in Chapter 10.

Playing with Context Objects

Most of the time, you'll instantiate `Context` objects by passing in a fully populated dictionary to `Context()`. But you can add and delete items from a `Context` object once it's been instantiated, too, using standard Python dictionary syntax:

```
>>> from django.template import Context
>>> c = Context({"foo": "bar"})
>>> c['foo']
'bar'
>>> del c['foo']
>>> c['foo']
''
>>> c['newvariable']  = 'hello'
>>> c['newvariable']
'hello'
```

Basic Template Tags and Filters

As we've mentioned already, the template system ships with built-in tags and filters. The sections that follow provide a rundown of the most common tags and filters.

Tags

The following sections outline the common Django tags.

if/else

The {% if %} tag evaluates a variable, and if that variable is "true" (i.e., it exists, is not empty, and is not a false Boolean value), the system will display everything between {% if %} and {% endif %}, for example:

```
{% if today_is_weekend %}
    <p>Welcome to the weekend!</p>
{% endif %}
```

An {% else %} tag is optional:

```
{% if today_is_weekend %}
    <p>Welcome to the weekend!</p>
{% else %}
    <p>Get back to work.</p>
{% endif %}
```

PYTHON "TRUTHINESS"

In Python, the empty list ([ ]), tuple (( )), dictionary ({}), string (' '), zero (0), and the special object None are False in a Boolean context. Everything else is True.

The {% if %} tag accepts and, or, or not for testing multiple variables, or to negate a given variable. Here's an example:

```
{% if athlete_list and coach_list %}
    Both athletes and coaches are available.
{% endif %}

{% if not athlete_list %}
    There are no athletes.
{% endif %}

{% if athlete_list or coach_list %}
    There are some athletes or some coaches.
{% endif %}

{% if not athlete_list or coach_list %}
    There are no athletes or there are some coaches. (OK, so
    writing English translations of Boolean logic sounds
    stupid; it's not our fault.)
{% endif %}

{% if athlete_list and not coach_list %}
    There are some athletes and absolutely no coaches.
{% endif %}
```

{% if %} tags don't allow and and or clauses within the same tag, because the order of logic would be ambiguous. For example, this is invalid:

```
{% if athlete_list and coach_list or cheerleader_list %}
```

The use of parentheses for controlling order of operations is not supported. If you find yourself needing parentheses, consider performing logic in the view code in order to simplify the templates. Even so, if you need to combine and and or to do advanced logic, just use nested {% if %} tags, for example:

```
{% if athlete_list %}
    {% if coach_list or cheerleader_list %}
        We have athletes, and either coaches or cheerleaders!
    {% endif %}
{% endif %}
```

Multiple uses of the same logical operator are fine, but you can't combine different operators. For example, this is valid:

```
{% if athlete_list or coach_list or parent_list or teacher_list %}
```

There is no {% elif %} tag. Use nested {% if %} tags to accomplish the same thing:

```
{% if athlete_list %}
    <p>Here are the athletes: {{ athlete_list }}.</p>
{% else %}
    <p>No athletes are available.</p>
    {% if coach_list %}
        <p>Here are the coaches: {{ coach_list }}.</p>
    {% endif %}
{% endif %}
```

Make sure to close each {% if %} with an {% endif %}. Otherwise, Django will throw a TemplateSyntaxError.

for

The {% for %} tag allows you to loop over each item in a sequence. As in Python's for statement, the syntax is for X in Y, where Y is the sequence to loop over and X is the name of the variable to use for a particular cycle of the loop. Each time through the loop, the template system will render everything between {% for %} and {% endfor %}.

For example, you could use the following to display a list of athletes given a variable athlete_list:

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

Add `reversed` to the tag to loop over the list in reverse:

```
{% for athlete in athlete_list reversed %}
...
{% endfor %}
```

It's possible to nest `{% for %}` tags:

```
{% for country in countries %}
    <h1>{{ country.name }}</h1>
    <ul>
    {% for city in country.city_list %}
        <li>{{ city }}</li>
    {% endfor %}
    </ul>
{% endfor %}
```

There is no support for "breaking out" of a loop before the loop is finished. If you want to accomplish this, change the variable you're looping over so that it includes only the values you want to loop over. Similarly, there is no support for a "continue" statement that would instruct the loop processor to return immediately to the front of the loop. (See the section "Philosophies and Limitations" later in this chapter for the reasoning behind this design decision.)

The `{% for %}` tag sets a magic `forloop` template variable within the loop. This variable has a few attributes that give you information about the progress of the loop:

- `forloop.counter` is always set to an integer representing the number of times the loop has been entered. This is one-indexed, so the first time through the loop, `forloop.counter` will be set to `1`.

  Here's an example:

  ```
  {% for item in todo_list %}
      <p>{{ forloop.counter }}: {{ item }}</p>
  {% endfor %}
  ```

- `forloop.counter0` is like `forloop.counter`, except it's zero-indexed. Its value will be set to `0` the first time through the loop.

- `forloop.revcounter` is always set to an integer representing the number of remaining items in the loop. The first time through the loop, `forloop.revcounter` will be set to the total number of items in the sequence you're traversing. The last time through the loop, `forloop.revcounter` will be set to `1`.

- `forloop.revcounter0` is like `forloop.revcounter`, except it's zero-indexed. The first time through the loop, `forloop.revcounter0` will be set to the number of elements in the sequence minus 1. The last time through the loop, it will be set to `0`.

- `forloop.first` is a Boolean value set to `True` if this is the first time through the loop. This is convenient for special casing:

  ```
  {% for object in objects %}
      {% if forloop.first %}<li class="first">{% else %}<li>{% endif %}
  ```

```
    {{ object }}
    </li>
{% endfor %}
```

- `forloop.last` is a Boolean value set to `True` if this is the last time through the loop. A common use for this is to put pipe characters between a list of links:

```
{% for link in links %}{{ link }}{% if not forloop.last %} | {% endif %}➥
{%  endfor %}
```

The preceding template code might output something like this:

```
Link1 | Link2 | Link3 | Link4
```

- `forloop.parentloop` is a reference to the `forloop` object for the *parent* loop, in case of nested loops. Here's an example:

```
{% for country in countries %}
    <table>
    {% for city in country.city_list %}
        <tr>
        <td>Country #{{ forloop.parentloop.counter }}</td>
        <td>City #{{ forloop.counter }}</td>
        <td>{{ city }}</td>
        </tr>
    {% endfor %}
    </table>
{% endfor %}
```

The magic `forloop` variable is only available within loops. After the template parser has reached `{% endfor %}`, `forloop` disappears.

CONTEXT AND THE FORLOOP VARIABLE

Inside the `{% for %}` block, the existing variables are moved out of the way to avoid overwriting the magic `forloop` variable. Django exposes this moved context in `forloop.parentloop`. You generally don't need to worry about this, but if you supply a template variable named `forloop` (though we advise against it), it will be named `forloop.parentloop` while inside the `{% for %}` block.

ifequal/ifnotequal

The Django template system deliberately is not a full-fledged programming language and thus does not allow you to execute arbitrary Python statements (more on this idea in the section "Philosophies and Limitations"). However, it's quite a common template requirement to compare two values and display something if they're equal—and Django provides an `{% ifequal %}` tag for that purpose.

The {% ifequal %} tag compares two values and displays everything between {% ifequal %} and {% endifequal %} if the values are equal.

This example compares the template variables user and currentuser:

```
{% ifequal user currentuser %}
    <h1>Welcome!</h1>
{% endifequal %}
```

The arguments can be hard-coded strings, with either single or double quotes, so the following is valid:

```
{% ifequal section 'sitenews' %}
    <h1>Site News</h1>
{% endifequal %}

{% ifequal section "community" %}
    <h1>Community</h1>
{% endifequal %}
```

Just like {% if %}, the {% ifequal %} tag supports an optional {% else %}:

```
{% ifequal section 'sitenews' %}
    <h1>Site News</h1>
{% else %}
    <h1>No News Here</h1>
{% endifequal %}
```

Only template variables, strings, integers, and decimal numbers are allowed as arguments to {% ifequal %}. These are valid examples:

```
{% ifequal variable 1 %}
{% ifequal variable 1.23 %}
{% ifequal variable 'foo' %}
{% ifequal variable "foo" %}
```

Any other types of variables, such as Python dictionaries, lists, or Booleans, can't be hard-coded in {% ifequal %}. These are invalid examples:

```
{% ifequal variable True %}
{% ifequal variable [1, 2, 3] %}
{% ifequal variable {'key': 'value'} %}
```

If you need to test whether something is true or false, use the {% if %} tags instead of {% ifequal %}.

Comments

Just as in HTML or in a programming language such as Python, the Django template language allows for comments. To designate a comment, use {# #}:

```
{# This is a comment #}
```

The comment will not be output when the template is rendered.

A comment cannot span multiple lines. This limitation improves template parsing performance. In the following template, the rendered output will look exactly the same as the template (i.e., the comment tag will not be parsed as a comment):

```
This is a {# this is not
a comment #}
test.
```

Filters

As explained earlier in this chapter, template filters are simple ways of altering the value of variables before they're displayed. 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*—that is, the output of one filter is applied to the next. Here's a common idiom for escaping text contents and then converting line breaks to `<p>` tags:

```
{{ my_text|escape|linebreaks }}
```

Some filters take arguments. A filter argument looks like this:

```
{{ bio|truncatewords:"30" }}
```

This displays the first 30 words of the `bio` variable. Filter arguments are always in double quotes.

The following are a few of the most important filters; Appendix F covers the rest.

- `addslashes`: Adds a backslash before any backslash, single quote, or double quote. This is useful if the produced text is included in a JavaScript string.
- `date`: Formats a `date` or `datetime` object according to a format string given in the parameter, for example:

  ```
  {{ pub_date|date:"F j, Y" }}
  ```

  Format strings are defined in Appendix F.
- `escape`: Escapes ampersands, quotes, and angle brackets in the given string. This is useful for sanitizing user-submitted data and for ensuring data is valid XML or XHTML. Specifically, `escape` makes these conversions:
  - Converts `&` to `&`
  - Converts `<` to `<`
  - Converts `>` to `>`
  - Converts `"` (double quote) to `&quot;`
  - Converts `'` (single quote) to `&#39;`

- `length`: Returns the length of the value. You can use this on a list or a string, or any Python object that knows how to determine its length (i.e., any object that has a `__len__()` method).

Philosophies and Limitations

Now that you've gotten a feel for the Django template language, we should point out some of its intentional limitations, along with some philosophies behind why it works the way it does.

More than any other component of Web applications, programmer opinions on template systems vary wildly. The fact that Python alone has dozens, if not hundreds, of open source template-language implementations supports this point. Each was likely created because its developer deemed all existing template languages inadequate. (In fact, it is said to be a rite of passage for a Python developer to write his or her own template language! If you haven't done this yet, consider it. It's a fun exercise.)

With that in mind, you might be interested to know that Django doesn't require that you use its template language. Because Django is intended to be a full-stack Web framework that provides all the pieces necessary for Web developers to be productive, many times it's *more convenient* to use Django's template system than other Python template libraries, but it's not a strict requirement in any sense. As you'll see in the upcoming section "Using Templates in Views," it's very easy to use another template language with Django.

Still, it's clear we have a strong preference for the way Django's template language works. The template system has roots in how Web development is done at World Online and the combined experience of Django's creators. Here are a few of those philosophies:

- *Business logic should be separated from presentation logic*. We see a template system as a tool that controls presentation and presentation-related logic—and that's it. The template system shouldn't support functionality that goes beyond this basic goal.

  For that reason, it's impossible to call Python code directly within Django templates. All "programming" is fundamentally limited to the scope of what template tags can do. It *is* possible to write custom template tags that do arbitrary things, but the out-of-the-box Django template tags intentionally do not allow for arbitrary Python code execution.

- *Syntax should be decoupled from HTML/XML*. Although Django's template system is used primarily to produce HTML, it's intended to be just as usable for non-HTML formats, such as plain text. Some other template languages are XML based, placing all template logic within XML tags or attributes, but Django deliberately avoids this limitation. Requiring valid XML to write templates introduces a world of human mistakes and hard-to-understand error messages, and using an XML engine to parse templates incurs an unacceptable level of overhead in template processing.

- *Designers are assumed to be comfortable with HTML code*. The template system isn't designed so that templates necessarily are displayed nicely in WYSIWYG editors such as Dreamweaver. That is too severe a limitation and wouldn't allow the syntax to be as nice as it is. Django expects template authors to be comfortable editing HTML directly.

- *Designers are assumed not to be Python programmers*. The template system authors recognize that Web page templates are most often written by *designers*, not *programmers*, and therefore should not assume Python knowledge.

However, the system also intends to accommodate small teams in which the templates *are* created by Python programmers. It offers a way to extend the system's syntax by writing raw Python code (more on this in Chapter 10).

- *The goal is not to invent a programming language.* The goal is to offer just enough programming-esque functionality, such as branching and looping, that is essential for making presentation-related decisions.

As a result of these design philosophies, the Django template language has the following limitations:

- *A template cannot set a variable or change the value of a variable.* It's possible to write custom template tags that accomplish these goals (see Chapter 10), but the stock Django template tags do not allow it.
- *A template cannot call raw Python code.* There's no way to "drop into Python mode" or use raw Python constructs. Again, it's possible to write custom template tags to do this, but the stock Django template tags don't allow it.

Using Templates in Views

You've learned the basics of using the template system; now let's use this knowledge to create a view. Recall the `current_datetime` view in `mysite.views`, which we started in the previous chapter. Here's what it looks like:

```
from django.http import HttpResponse
import datetime

def current_datetime(request):
    now = datetime.datetime.now()
    html = "<html><body>It is now %s.</body></html>" % now
    return HttpResponse(html)
```

Let's change this view to use Django's template system. At first, you might think to do something like this:

```
from django.template import Template, Context
from django.http import HttpResponse
import datetime

def current_datetime(request):
    now = datetime.datetime.now()
    t = Template("<html><body>It is now {{ current_date }}.</body></html>")
    html = t.render(Context({'current_date': now}))
    return HttpResponse(html)
```

Sure, that uses the template system, but it doesn't solve the problems we pointed out in the introduction of this chapter. Namely, the template is still embedded in the Python code. Let's fix that by putting the template in a *separate file*, which this view will load.

You might first consider saving your template somewhere on your filesystem and using Python's built-in file-opening functionality to read the contents of the template. Here's what that might look like, assuming the template was saved as the file `/home/djangouser/templates/mytemplate.html`:

```
from django.template import Template, Context
from django.http import HttpResponse
import datetime

def current_datetime(request):
    now = datetime.datetime.now()
    # Simple way of using templates from the filesystem.
    # This doesn't account for missing files!
    fp = open('/home/djangouser/templates/mytemplate.html')
    t = Template(fp.read())
    fp.close()
    html = t.render(Context({'current_date': now}))
    return HttpResponse(html)
```

This approach, however, is inelegant for these reasons:

- It doesn't handle the case of a missing file. If the file `mytemplate.html` doesn't exist or isn't readable, the `open()` call will raise an `IOError` exception.
- It hard-codes your template location. If you were to use this technique for every view function, you'd be duplicating the template locations. Not to mention it involves a lot of typing!
- It includes a lot of boring boilerplate code. You've got better things to do than write calls to `open()`, `fp.read()`, and `fp.close()` each time you load a template.

To solve these issues, we'll use *template loading* and *template directories*, both of which are described in the sections that follow.

Template Loading

Django provides a convenient and powerful API for loading templates from disk, with the goal of removing redundancy both in your template-loading calls and in your templates themselves.

In order to use this template-loading API, first you'll need to tell the framework where you store your templates. The place to do this is in your *settings file*.

A Django settings file is the place to put configuration for your Django instance (aka your Django project). It's a simple Python module with module-level variables, one for each setting.

When you ran `django-admin.py startproject mysite` in Chapter 2, the script created a default settings file for you, aptly named `settings.py`. Have a look at the file's contents. It contains variables that look like this (though not necessarily in this order):

```
DEBUG = True
TIME_ZONE = 'America/Chicago'
USE_I18N = True
ROOT_URLCONF = 'mysite.urls'
```

This is pretty self-explanatory; the settings and their respective values are simple Python variables. And because the settings file is just a plain Python module, you can do dynamic things such as checking the value of one variable before setting another. (This also means that you should avoid Python syntax errors in your settings file.)

We'll cover settings files in depth in Appendix E, but for now, have a look at the `TEMPLATE_DIRS` setting. This setting tells Django's template-loading mechanism where to look for templates. By default, it's an empty tuple. Pick a directory where you'd like to store your templates and add it to `TEMPLATE_DIRS`, like so:

```
TEMPLATE_DIRS = (
    '/home/django/mysite/templates',
)
```

There are a few things to note:

- You can specify any directory you want, as long as the directory and templates within that directory are readable by the user account under which your Web server runs. If you can't think of an appropriate place to put your templates, we recommend creating a `templates` directory within your Django project (i.e., within the `mysite` directory you created in Chapter 2, if you've been following along with this book's examples).

- Don't forget the comma at the end of the template directory string! Python requires commas within single-element tuples to disambiguate the tuple from a parenthetical expression. This is a common newbie gotcha.

  If you want to avoid this error, you can make `TEMPLATE_DIRS` a list instead of a tuple, because single-element lists don't require a trailing comma:

  ```
  TEMPLATE_DIRS = [
      '/home/django/mysite/templates'
  ]
  ```

  A tuple is slightly more semantically correct than a list (tuples cannot be changed after being created, and nothing should be changing settings once they've been read), so we recommend using a tuple for your `TEMPLATE_DIRS` setting.

- If you're on Windows, include your drive letter and use Unix-style forward slashes rather than backslashes, as follows:

- It's simplest to use absolute paths (i.e., directory paths that start at the root of the filesystem). If you want to be a bit more flexible and decoupled, though, you can take advantage of the fact that Django settings files are just Python code by constructing the contents of `TEMPLATE_DIRS` dynamically, for example:

  ```
  import os.path

  TEMPLATE_DIRS = (
      os.path.join(os.path.dirname(__file__), 'templates').replace('\\', '/'),
  )
  ```

  This example uses the "magic" Python variable `__file__`, which is automatically set to the file name of the Python module in which the code lives.

```
    TEMPLATE_DIRS = (
        'C:/www/django/templates',
    )
```

With TEMPLATE_DIRS set, the next step is to change the view code to use Django's template-loading functionality rather than hard-coding the template paths. Returning to our current_datetime view, let's change it like so:

```
from django.template.loader import get_template
from django.template import Context
from django.http import HttpResponse
import datetime

def current_datetime(request):
    now = datetime.datetime.now()
    t = get_template('current_datetime.html')
    html = t.render(Context({'current_date': now}))
    return HttpResponse(html)
```

In this example, we're using the function django.template.loader.get_template() rather than loading the template from the filesystem manually. The get_template() function takes a template name as its argument, figures out where the template lives on the filesystem, opens that file, and returns a compiled Template object.

If get_template() cannot find the template with the given name, it raises a TemplateDoesNotExist exception. To see what that looks like, fire up the Django development server again, as in Chapter 3, by running python manage.py runserver within your Django project's directory. Then, point your browser at the page that activates the current_datetime view (e.g., http://127.0.0.1:8000/time/). Assuming your DEBUG setting is set to True and you haven't yet created a current_datetime.html template, you should see a Django error page highlighting the TemplateDoesNotExist error, as shown in Figure 4-1.

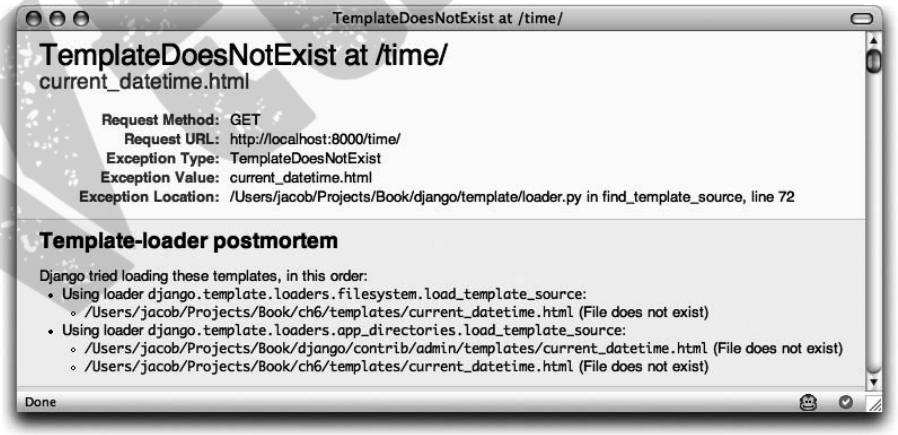

Figure 4-1. *The error page shown when a template cannot be found*

This error page is similar to the one we explained in Chapter 3, with one additional piece of debugging information: a "Template-loader postmortem" section. This section tells you which templates Django tried to load, along with the reason each attempt failed (e.g., "File does not exist"). This information is invaluable when you're trying to debug template-loading errors.

As you can probably tell from the error messages found in the Figure 4-1, Django attempted to find the template by combining the directory in the `TEMPLATE_DIRS` setting with the template name passed to `get_template()`. So if your `TEMPLATE_DIRS` contains `'/home/django/templates'`, Django looks for the file `'/home/django/templates/current_datetime.html'`. If `TEMPLATE_DIRS` contains more than one directory, each is checked until the template is found or they've all been checked.

Moving along, create the `current_datetime.html` file within your template directory using the following template code:

```
<html><body>It is now {{ current_date }}.</body></html>
```

Refresh the page in your Web browser, and you should see the fully rendered page.

render_to_response()

Because it's such a common idiom to load a template, fill a `Context`, and return an `HttpResponse` object with the result of the rendered template, Django provides a shortcut that lets you do those things in one line of code. This shortcut is a function called `render_to_response()`, which lives in the module `django.shortcuts`. Most of the time, you'll be using `render_to_response()` rather than loading templates and creating `Context` and `HttpResponse` objects manually.

Here's the ongoing `current_datetime` example rewritten to use `render_to_response()`:

```
from django.shortcuts import render_to_response
import datetime

def current_datetime(request):
    now = datetime.datetime.now()
    return render_to_response('current_datetime.html', {'current_date': now})
```

What a difference! Let's step through the code changes:

- We no longer have to import `get_template`, `Template`, `Context`, or `HttpResponse`. Instead, we import `django.shortcuts.render_to_response`. The `import datetime` remains.
- Within the `current_datetime` function, we still calculate `now`, but the template loading, context creation, template rendering, and `HttpResponse` creation is all taken care of by the `render_to_response()` call. Because `render_to_response()` returns an `HttpResponse` object, we can simply `return` that value in the view.

The first argument to `render_to_response()` should be the name of the template to use. The second argument, if given, should be a dictionary to use in creating a `Context` for that template. If you don't provide a second argument, `render_to_response()` will use an empty dictionary.

The locals() Trick

Consider our latest incarnation of `current_datetime`:

```
def current_datetime(request):
    now = datetime.datetime.now()
    return render_to_response('current_datetime.html', {'current_date': now})
```

Many times, as in this example, you'll find yourself calculating some values, storing them in variables (e.g., `now` in the preceding code), and sending those variables to the template. Particularly lazy programmers should note that it's slightly redundant to have to give names for temporary variables *and* give names for the template variables. Not only is it redundant, but also it's extra typing.

So if you're one of those lazy programmers and you like keeping code particularly concise, you can take advantage of a built-in Python function called `locals()`. It returns a dictionary mapping all local variable names to their values. Thus, the preceding view could be rewritten like so:

```
def current_datetime(request):
    current_date = datetime.datetime.now()
    return render_to_response('current_datetime.html', locals())
```

Here, instead of manually specifying the context dictionary as before, we pass the value of `locals()`, which will include all variables defined at that point in the function's execution. As a consequence, we've renamed the `now` variable to `current_date`, because that's the variable name that the template expects. In this example, `locals()` doesn't offer a *huge* improvement, but this technique can save you some typing if you have several template variables to define—or if you're lazy.

One thing to watch out for when using `locals()` is that it includes *every* local variable, which may comprise more variables than you actually want your template to have access to. In the previous example, `locals()` will also include `request`. Whether this matters to you depends on your application.

A final thing to consider is that `locals()` incurs a small bit of overhead, because when you call it, Python has to create the dictionary dynamically. If you specify the context dictionary manually, you avoid this overhead.

Subdirectories in get_template()

It can get unwieldy to store all of your templates in a single directory. You might like to store templates in subdirectories of your template directory, and that's fine. In fact, we recommend doing so; some more advanced Django features (such as the generic views system, which we cover in Chapter 9) expect this template layout as a default convention.

Storing templates in subdirectories of your template directory is easy. In your calls to `get_template()`, just include the subdirectory name and a slash before the template name, like so:

```
t = get_template('dateapp/current_datetime.html')
```

Because `render_to_response()` is a small wrapper around `get_template()`, you can do the same thing with the first argument to `render_to_response()`.

There's no limit to the depth of your subdirectory tree. Feel free to use as many as you like.

■ **Note** Windows users, be sure to use forward slashes rather than backslashes. `get_template()` assumes a Unix-style file name designation.

The include Template Tag

Now that we've covered the template-loading mechanism, we can introduce a built-in template tag that takes advantage of it: `{% include %}`. This tag allows you to include the contents of another template. The argument to the tag should be the name of the template to include, and the template name can be either a variable or a hard-coded (quoted) string, in either single or double quotes. Anytime you have the same code in multiple templates, consider using an `{% include %}` to remove the duplication.

These two examples include the contents of the template `nav.html`. The examples are equivalent and illustrate that either single or double quotes are allowed:

```
{% include 'nav.html' %}
{% include "nav.html" %}
```

This example includes the contents of the template `includes/nav.html`:

```
{% include 'includes/nav.html' %}
```

This example includes the contents of the template whose name is contained in the variable `template_name`:

```
{% include template_name %}
```

As in `get_template()`, the file name of the template is determined by adding the template directory from `TEMPLATE_DIRS` to the requested template name.

Included templates are evaluated with the context of the template that's including them.

If a template with the given name isn't found, Django will do one of two things:

- If `DEBUG` is set to `True`, you'll see the `TemplateDoesNotExist` exception on a Django error page.
- If `DEBUG` is set to `False`, the tag will fail silently, displaying nothing in the place of the tag.

Template Inheritance

Our template examples so far have been tiny HTML snippets, but in the real world, you'll be using Django's template system to create entire HTML pages. This leads to a common Web development problem: across a Web site, how do you reduce the duplication and redundancy of common page areas, such as sitewide navigation?

A classic way of solving this problem is to use *server-side includes*, directives you can embed within your HTML pages to "include" one Web page inside another. Indeed, Django supports that approach, with the `{% include %}` template tag just described. But the preferred way of solving this problem with Django is to use a more elegant strategy called *template inheritance*.

In essence, template inheritance lets you build a base "skeleton" template that contains all the common parts of your site and defines "blocks" that child templates can override.

Let's see an example of this by creating a more complete template for our `current_datetime` view, by editing the `current_datetime.html` file:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html lang="en">
<head>
    <title>The current time</title>
</head>
<body>
    <h1>My helpful timestamp site</h1>
    <p>It is now {{ current_date }}.</p>

    <hr>
    <p>Thanks for visiting my site.</p>
</body>
</html>
```

That looks just fine, but what happens when we want to create a template for another view—say, the `hours_ahead` view from Chapter 3? If we want to again make a nice, valid, full HTML template, we'd create something like this:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html lang="en">
<head>
    <title>Future time</title>
</head>
<body>
    <h1>My helpful timestamp site</h1>
    <p>In {{ hour_offset }} hour(s), it will be {{ next_time }}.</p>

    <hr>
    <p>Thanks for visiting my site.</p>
</body>
</html>
```

Clearly, we've just duplicated a lot of HTML. Imagine if we had a more typical site, including a navigation bar, a few style sheets, perhaps some JavaScript—we'd end up putting all sorts of redundant HTML into each template.

The server-side include solution to this problem is to factor out the common bits in both templates and save them in separate template snippets, which are then included in each template. Perhaps you'd store the top bit of the template in a file called `header.html`:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html lang="en">
<head>
```

And perhaps you'd store the bottom bit in a file called `footer.html`:

```
    <hr>
    <p>Thanks for visiting my site.</p>
</body>
</html>
```

With an include-based strategy, headers and footers are easy. It's the middle ground that's messy. In this example, both pages feature a title—`<h1>My helpful timestamp site</h1>`—but that title can't fit into `header.html` because the `<title>` on both pages is different. If we included the `<h1>` in the header, we'd have to include the `<title>`, which wouldn't allow us to customize it per page. See where this is going?

Django's template inheritance system solves these problems. You can think of it as an "inside-out" version of server-side includes. Instead of defining the snippets that are *common*, you define the snippets that are *different*.

The first step is to define a *base template*—a skeleton of your page that *child templates* will later fill in. Here's a base template for our ongoing example:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html lang="en">
<head>
    <title>{% block title %}{% endblock %}</title>
</head>
<body>
    <h1>My helpful timestamp site</h1>
    {% block content %}{% endblock %}
    {% block footer %}
    <hr>
    <p>Thanks for visiting my site.</p>
    {% endblock %}
</body>
</html>
```

This template, which we'll call `base.html`, defines a simple HTML skeleton document that we'll use for all the pages on the site. It's the job of child templates to override, or add to, or leave alone the contents of the blocks. (If you're following along at home, save this file to your template directory.)

We're using a template tag here that you haven't seen before: the `{% block %}` tag. All the `{% block %}` tags do is tell the template engine that a child template may override those portions of the template.

Now that we have this base template, we can modify our existing `current_datetime.html` template to use it:

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

{% block title %}The current time{% endblock %}

{% block content %}
<p>It is now {{ current_date }}.</p>
{% endblock %}
```

While we're at it, let's create a template for the `hours_ahead` view from Chapter 3. (If you're following along with code, we'll leave it up to you to change `hours_ahead` to use the template system.) Here's what that would look like:

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

{% block title %}Future time{% endblock %}

{% block content %}
<p>In {{ hour_offset }} hour(s), it will be {{ next_time }}.</p>
{% endblock %}
```

Isn't this beautiful? Each template contains only the code that's *unique* to that template. No redundancy needed. If you need to make a sitewide design change, just make the change to `base.html`, and all of the other templates will immediately reflect the change.

Here's how it works. When you load the template `current_datetime.html`, the template engine sees the `{% extends %}` tag, noting that this template is a child template. The engine immediately loads the parent template—in this case, `base.html`.

At that point, the template engine notices the three `{% block %}` tags in `base.html` and replaces those blocks with the contents of the child template. So, the title we've defined in `{% block title %}` will be used, as will the `{% block content %}`.

Note that since the child template doesn't define the `footer` block, the template system uses the value from the parent template instead. Content within a `{% block %}` tag in a parent template is always used as a fallback.

Inheritance doesn't affect the way the context works, and you can use as many levels of inheritance as needed. One common way of using inheritance is the following three-level approach:

1. Create a `base.html` template that holds the main look and feel of your site. This is the stuff that rarely, if ever, changes.

2. Create a `base_SECTION.html` template for each "section" of your site (e.g., `base_photos.html` and `base_forum.html`). These templates extend `base.html` and include section-specific styles/design.

3. Create individual templates for each type of page, such as a forum page or a photo gallery. These templates extend the appropriate section template.

This approach maximizes code reuse and makes it easy to add items to shared areas, such as sectionwide navigation.

Here are some tips for working with template inheritance:

- If you use `{% extends %}` in a template, it must be the first template tag in that template. Otherwise, template inheritance won't work.
- Generally, the more `{% block %}` tags in your base templates, the better. Remember, child templates don't have to define all parent blocks, so you can fill in reasonable defaults in a number of blocks, and then define only the ones you need in the child templates. It's better to have more hooks than fewer hooks.
- If you find yourself duplicating code in a number of templates, it probably means you should move that code 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.
- You may not 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.
- The template name you pass to `{% extends %}` is loaded using the same method that `get_template()` uses. That is, the template name is appended to your `TEMPLATE_DIRS` setting.
- In most cases, the argument to `{% extends %}` will be a string, but it can also be a variable, if you don't know the name of the parent template until runtime. This lets you do some cool, dynamic stuff.

What's Next?

Most modern Web sites are *database driven*, meaning the site content is stored in a relational database. This allows a clean separation of data and logic (in the same way views and templates allow the separation of logic and display). The next chapter covers the tools Django provides to interact with a database.

CHAPTER 5

Interacting with a Database: Models

In Chapter 3, we covered the fundamentals of building dynamic Web sites with Django: setting up views and URLconfs. As we explained, a view is responsible for doing *some arbitrary logic* and then returning a response. In the example, our arbitrary logic was to calculate the current date and time.

In modern Web applications, the arbitrary logic often involves interacting with a database. Behind the scenes, a *database-driven Web site* connects to a database server, retrieves some data out of it, and displays that data, nicely formatted, on a Web page. Or, similarly, the site could provide functionality that lets site visitors populate the database on their own.

Many complex Web sites provide some combination of the two. Amazon.com, for instance, is a great example of a database-driven site. Each product page is essentially a query into Amazon's product database formatted as HTML, and when you post a customer review, it gets inserted into the database of reviews.

Django is well suited for making database-driven Web sites, as it comes with easy yet powerful ways of performing database queries using Python. This chapter explains that functionality: Django's database layer.

Note While it's not strictly necessary to know basic database theory and SQL in order to use Django's database layer, it's highly recommended. An introduction to those concepts is beyond the scope of this book, but keep reading even if you're a database newbie. You'll probably be able to follow along and grasp concepts based on the context.

The "Dumb" Way to Do Database Queries in Views

Just as Chapter 3 detailed a "dumb" way to produce output within a view (by hard-coding the text directly within the view), there's a "dumb" way to retrieve data from a database in a view. It's simple: just use any existing Python library to execute an SQL query and do something with the results.

In this example view, we use the `MySQLdb` library (available at `http://www.djangoproject.com/r/python-mysql/`) to connect to a MySQL database, retrieve some records, and feed them to a template for display as a Web page:

```
from django.shortcuts import render_to_response
import MySQLdb

def book_list(request):
    db = MySQLdb.connect(user='me', db='mydb', passwd='secret', host='localhost')
    cursor = db.cursor()
    cursor.execute('SELECT name FROM books ORDER BY name')
    names = [row[0] for row in cursor.fetchall()]
    db.close()
    return render_to_response('book_list.html', {'names': names})
```

This approach works, but some problems should jump out at you immediately:

- We're hard-coding the database connection parameters. Ideally, these parameters would be stored in the Django configuration.
- We're having to write a fair bit of boilerplate code: creating a connection, creating a cursor, executing a statement, and closing the connection. Ideally, all we'd have to do is specify which results we wanted.
- It ties us to MySQL. If, down the road, we switch from MySQL to PostgreSQL, we'll have to use a different database adapter (e.g., `psycopg` rather than `MySQLdb`), alter the connection parameters, and—depending on the nature of the SQL statement—possibly rewrite the SQL. Ideally, the database server we're using would be abstracted, so that a database server change could be made in a single place.

As you might expect, Django's database layer aims to solve these problems. Here's a sneak preview of how the previous view can be rewritten using Django's database API:

```
from django.shortcuts import render_to_response
from mysite.books.models import Book

def book_list(request):
    books = Book.objects.order_by('name')
    return render_to_response('book_list.html', {'books': books})
```

We'll explain this code a little later in the chapter. For now, just get a feel for how it looks.

The MTV Development Pattern

Before we delve into any more code, let's take a moment to consider the overall design of a database-driven Django Web application.

As we mentioned in previous chapters, Django is designed to encourage loose coupling and strict separation between pieces of an application. If you follow this philosophy, it's easy to make changes to a particular piece of the application without affecting the other pieces. In view functions, for instance, we discussed the importance of separating the business logic from the presentation logic by using a template system. With the database layer, we're applying that same philosophy to data access logic.

Those three pieces together—data access logic, business logic, and presentation logic—comprise a concept that's sometimes called the *Model-View-Controller* (MVC) pattern of

software architecture. In this pattern, "Model" refers to the data access layer, "View" refers to the part of the system that selects what to display and how to display it, and "Controller" refers to the part of the system that decides which view to use, depending on user input, accessing the model as needed.

WHY THE ACRONYM?

The goal of explicitly defining patterns such as MVC is mostly to streamline communication among developers. Instead of having to tell your coworkers, "Let's make an abstraction of the data access, then let's have a separate layer that handles data display, and let's put a layer in the middle that regulates this," you can take advantage of a shared vocabulary and say, "Let's use the MVC pattern here."

Django follows this MVC pattern closely enough that it can be called an MVC framework. Here's roughly how the M, V, and C break down in Django:

- *M*, the data-access portion, is handled by Django's database layer, which is described in this chapter.
- *V*, the portion that selects which data to display and how to display it, is handled by views and templates.
- *C*, the portion that delegates to a view depending on user input, is handled by the framework itself by following your URLconf and calling the appropriate Python function for the given URL.

Because the "C" is handled by the framework itself and most of the excitement in Django happens in models, templates, and views, Django has been referred to as an *MTV framework*. In the MTV development pattern,

- *M* stands for "Model," the data access layer. This layer contains anything and everything about the data: how to access it, how to validate it, which behaviors it has, and the relationships between the data.
- *T* stands for "Template," the presentation layer. This layer contains presentation-related decisions: how something should be displayed on a Web page or other type of document.
- *V* stands for "View," the business logic layer. This layer contains the logic that accesses the model and defers to the appropriate template(s). You can think of it as the bridge between models and templates.

If you're familiar with other MVC Web-development frameworks, such as Ruby on Rails, you may consider Django views to be the "controllers" and Django templates to be the "views." This is an unfortunate confusion brought about by differing interpretations of MVC. In Django's interpretation of MVC, the "view" describes the data that gets presented to the user; it's not necessarily just *how* the data looks, but *which* data is presented. In contrast, Ruby on Rails and similar frameworks suggest that the controller's job includes deciding which data gets presented to the user, whereas the view is strictly *how* the data looks, not *which* data is presented.

Neither interpretation is more "correct" than the other. The important thing is to understand the underlying concepts.

Configuring the Database

With all of that philosophy in mind, let's start exploring Django's database layer. First, we need to take care of some initial configuration: we need to tell Django which database server to use and how to connect to it.

We'll assume you've set up a database server, activated it, and created a database within it (e.g., using a `CREATE DATABASE` statement). SQLite is a special case; in that case, there's no database to create, because SQLite uses standalone files on the filesystem to store its data.

As with `TEMPLATE_DIRS` in the previous chapter, database configuration lives in the Django settings file, called `settings.py` by default. Edit that file and look for the database settings:

```
DATABASE_ENGINE = ''
DATABASE_NAME = ''
DATABASE_USER = ''
DATABASE_PASSWORD = ''
DATABASE_HOST = ''
DATABASE_PORT = ''
```

Here's a rundown of each setting.

- `DATABASE_ENGINE` tells Django which database engine to use. If you're using a database with Django, `DATABASE_ENGINE` must be set to one of the strings shown in Table 5-1.

Table 5-1. *Database Engine Settings*

Setting	Database	Required Adapter
`postgresql`	PostgreSQL	`psycopg` version 1.x, `http://www.djangoproject.com/r/python-pgsql/1/`.
`postgresql_psycopg2`	PostgreSQL	`psycopg` version 2.x, `http://www.djangoproject.com/r/python-pgsql/`.
`mysql`	MySQL	`MySQLdb`, `http://www.djangoproject.com/r/python-mysql/`.
`sqlite3`	SQLite	No adapter needed if using Python 2.5+. Otherwise, `pysqlite`, `http://www.djangoproject.com/r/python-sqlite/`.
`oracle`	Oracle	`cx_Oracle`, `http://www.djangoproject.com/r/python-oracle/`.

Note that for whichever database back-end you use, you'll need to download and install the appropriate database adapter. Each one is available for free on the Web; just follow the links in the "Required Adapter" column in Table 5-1.

- DATABASE_NAME tells Django the name of your database. If you're using SQLite, specify the full filesystem path to the database file on your filesystem (e.g., '/home/django/mydata.db').
- DATABASE_USER tells Django which username to use when connecting to your database. If you're using SQLite, leave this blank.
- DATABASE_PASSWORD tells Django which password to use when connecting to your database. If you're using SQLite or have an empty password, leave this blank.
- DATABASE_HOST tells Django which host to use when connecting to your database. If your database is on the same computer as your Django installation (i.e., localhost), leave this blank. If you're using SQLite, leave this blank.

  MySQL is a special case here. If this value starts with a forward slash (/) and you're using MySQL, MySQL will connect via a Unix socket to the specified socket, for example:

  ```
  DATABASE_HOST = '/var/run/mysql'
  ```

  If you're using MySQL and this value *doesn't* start with a forward slash, then this value is assumed to be the host.
- DATABASE_PORT tells Django which port to use when connecting to your database. If you're using SQLite, leave this blank. Otherwise, if you leave this blank, the underlying database adapter will use whichever port is default for your given database server. In most cases, the default port is fine, so you can leave this blank.

Once you've entered those settings, test your configuration. First, from within the mysite project directory you created in Chapter 2, run the command python manage.py shell.

You'll notice this starts a Python interactive interpreter. Looks can be deceiving, though! There's an important difference between running the command python manage.py shell within your Django project directory and the more generic python. The latter is the basic Python shell, but the former tells Django which settings file to use before it starts the shell. This is a key requirement for doing database queries: Django needs to know which settings file to use in order to get your database connection information.

Behind the scenes, python manage.py shell simply assumes that your settings file is in the same directory as manage.py. There are other ways to tell Django which settings module to use; we'll cover those options later. For now, use python manage.py shell whenever you need to drop into the Python interpreter to do Django-specific tinkering.

Once you've entered the shell, type these commands to test your database configuration:

```
>>> from django.db import connection
>>> cursor = connection.cursor()
```

If nothing happens, then your database is configured properly. Otherwise, check the error message for clues about what's wrong. Table 5-2 shows some common errors.

Table 5-2. *Database Configuration Error Messages*

Error Message	Solution
You haven't set the DATABASE_ENGINE setting yet.	Set the `DATABASE_ENGINE` setting to something other than an empty string.
Environment variable DJANGO_SETTINGS_MODULE is undefined.	Run the command `python manage.py shell` rather than `python`.
Error loading _____ module: No module named _____.	You haven't installed the appropriate database-specific adapter (e.g., `psycopg` or `MySQLdb`).
_____ isn't an available database backend.	Set your `DATABASE_ENGINE` setting to one of the valid engine settings described previously. Perhaps you made a typo?
database _____ does not exist	Change the `DATABASE_NAME` setting to point to a database that exists, or execute the appropriate `CREATE DATABASE` statement in order to create it.
role _____ does not exist	Change the `DATABASE_USER` setting to point to a user that exists, or create the user in your database.
could not connect to server	Make sure `DATABASE_HOST` and `DATABASE_PORT` are set correctly, and make sure the server is running.

Your First App

Now that you've verified the connection is working, it's time to create a *Django app*—a bundle of Django code, including models and views, that lives together in a single Python package and represents a full Django application.

It's worth explaining the terminology here, because this tends to trip up beginners. We already created a *project*, in Chapter 2, so what's the difference between a *project* and an *app*? The difference is that of configuration vs. code:

- A project is an instance of a certain set of Django apps, plus the configuration for those apps. Technically, the only requirement of a project is that it supplies a settings file, which defines the database connection information, the list of installed apps, the `TEMPLATE_DIRS`, and so forth.

- An app is a portable set of Django functionality, usually including models and views, that lives together in a single Python package. For example, Django comes with a number of apps, such as a commenting system and an automatic admin interface. A key thing to note about these apps is that they're portable and reusable across multiple projects.

There are very few hard-and-fast rules about how you fit your Django code into this scheme; it's flexible. If you're building a simple Web site, you may use only a single app. If you're building a complex Web site with several unrelated pieces such as an e-commerce system and a message board, you'll probably want to split those into separate apps so that you'll be able to reuse them individually in the future.

Indeed, you don't necessarily need to create apps at all, as evidenced by the example view functions we've created so far in this book. In those cases, we simply created a file called

views.py, filled it with view functions, and pointed our URLconf at those functions. No "apps" were needed.

However, there's one requirement regarding the app convention: if you're using Django's database layer (models), you must create a Django app. Models must live within apps. Thus, in order to start writing our models, we'll need to create a new app.

Within the mysite project directory you created in Chapter 2, type this command to create a new app named books:

```
python manage.py startapp books
```

This command does not produce any output, but it does create a books directory within the mysite directory. Let's look at the contents of that directory:

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

These files will contain the models and views for this app.

Have a look at models.py and views.py in your favorite text editor. Both files are empty, except for an import in models.py. This is the blank slate for your Django app.

Defining Models in Python

As we discussed earlier in this chapter, the "M" in "MTV" stands for "Model." A Django model is a description of the data in your database, represented as Python code. It's your data layout—the equivalent of your SQL CREATE TABLE statements—except it's in Python instead of SQL, and it includes more than just database column definitions. Django uses a model to execute SQL code behind the scenes and return convenient Python data structures representing the rows in your database tables. Django also uses models to represent higher-level concepts that SQL can't necessarily handle.

If you're familiar with databases, your immediate thought might be, "Isn't it redundant to define data models in Python *and* in SQL?" Django works the way it does for several reasons:

- Introspection requires overhead and is imperfect. In order to provide convenient data-access APIs, Django needs to know the database layout *somehow*, and there are two ways of accomplishing this. The first way is to explicitly describe the data in Python, and the second way is to introspect the database at runtime to determine the data models.

  This second way seems cleaner, because the metadata about your tables lives in only one place, but it introduces a few problems. First, introspecting a database at runtime obviously requires overhead. If the framework had to introspect the database each time it processed a request, or even when the Web server was initialized, this would incur an unacceptable level of overhead. (While some believe that level of overhead is acceptable, Django's developers aim to trim as much framework overhead as possible, and this approach has succeeded in making Django faster than its high-level framework competitors in benchmarks.) Second, some databases, notably older versions of MySQL, do not store sufficient metadata for accurate and complete introspection.

- Writing Python is fun, and keeping everything in Python limits the number of times your brain has to do a "context switch." It helps productivity if you keep yourself in a single programming environment/mentality for as long as possible. Having to write SQL, then Python, and then SQL again is disruptive.
- Having data models stored as code rather than in your database makes it easier to keep your models under version control. This way, you can easily keep track of changes to your data layouts.
- SQL allows for only a certain level of metadata about a data layout. Most database systems, for example, do not provide a specialized data type for representing email addresses or URLs. Django models do. The advantage of higher-level data types is higher productivity and more reusable code.
- SQL is inconsistent across database platforms. If you're distributing a Web application, for example, it's much more pragmatic to distribute a Python module that describes your data layout than separate sets of `CREATE TABLE` statements for MySQL, PostgreSQL, and SQLite.

A drawback of this approach, however, is that it's possible for the Python code to get out of sync with what's actually in the database. If you make changes to a Django model, you'll need to make the same changes inside your database to keep your database consistent with the model. We'll detail some strategies for handling this problem later in this chapter.

Finally, we should note that Django includes a utility that can generate models by introspecting an existing database. This is useful for quickly getting up and running with legacy data.

Your First Model

As an ongoing example in this chapter and the next chapter, we'll focus on a basic book/author/publisher data layout. We use this as our example because the conceptual relationships between books, authors, and publishers are well known, and this is a common data layout used in introductory SQL textbooks. You're also reading a book that was written by authors and produced by a publisher!

We'll suppose the following concepts, fields, and relationships:

- An author has a salutation (e.g., Mr. or Mrs.), a first name, a last name, an email address, and a headshot photo.
- A publisher has a name, a street address, a city, a state/province, a country, and a Web site.
- A book has a title and a publication date. It also has one or more authors (a many-to-many relationship with authors) and a single publisher (a one-to-many relationship—aka foreign key—to publishers).

The first step in using this database layout with Django is to express it as Python code. In the `models.py` file that was created by the `startapp` command, enter the following:

```
from django.db import models

class Publisher(models.Model):
    name = models.CharField(maxlength=30)
    address = models.CharField(maxlength=50)
    city = models.CharField(maxlength=60)
    state_province = models.CharField(maxlength=30)
    country = models.CharField(maxlength=50)
    website = models.URLField()

class Author(models.Model):
    salutation = models.CharField(maxlength=10)
    first_name = models.CharField(maxlength=30)
    last_name = models.CharField(maxlength=40)
    email = models.EmailField()
    headshot = models.ImageField(upload_to='/tmp')

class Book(models.Model):
    title = models.CharField(maxlength=100)
    authors = models.ManyToManyField(Author)
    publisher = models.ForeignKey(Publisher)
    publication_date = models.DateField()
```

Let's quickly examine this code to cover the basics. The first thing to notice is that each model is represented by a Python class that is a subclass of `django.db.models.Model`. The parent class, `Model`, contains all the machinery necessary to make these objects capable of interacting with a database—and that leaves our models responsible solely for defining their fields, in a nice and compact syntax. Believe it or not, this is all the code we need to write to have basic data access with Django.

Each model generally corresponds to a single database table, and each attribute on a model generally corresponds to a column in that database table. The attribute name corresponds to the column's name, and the type of field (e.g., `CharField`) corresponds to the database column type (e.g., `varchar`). For example, the `Publisher` model is equivalent to the following table (assuming PostgreSQL `CREATE TABLE` syntax):

```
CREATE TABLE "books_publisher" (
    "id" serial NOT NULL PRIMARY KEY,
    "name" varchar(30) NOT NULL,
    "address" varchar(50) NOT NULL,
    "city" varchar(60) NOT NULL,
    "state_province" varchar(30) NOT NULL,
    "country" varchar(50) NOT NULL,
    "website" varchar(200) NOT NULL
);
```

Indeed, Django can generate that `CREATE TABLE` statement automatically, as we'll show in a moment.

The exception to the one-class-per-database-table rule is the case of many-to-many relationships. In our example models, `Book` has a `ManyToManyField` called `authors`. This designates that a book has one or many authors, but the `Book` database table doesn't get an `authors` column. Rather, Django creates an additional table—a many-to-many "join table"—that handles the mapping of books to authors.

■ **Note** For a full list of field types and model syntax options, see Appendix B.

Finally, note we haven't explicitly defined a primary key in any of these models. Unless you instruct it otherwise, Django automatically gives every model an integer primary key field called `id`. Each Django model is required to have a single-column primary key.

Installing the Model

We've written the code; now let's create the tables in our database. In order to do that, the first step is to *activate* these models in our Django project. We do that by adding the `books` app to the list of installed apps in the settings file.

Edit the `settings.py` file again, and look for the `INSTALLED_APPS` setting. `INSTALLED_APPS` tells Django which apps are activated for a given project. By default, it looks something like this:

```
INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
)
```

Temporarily comment out all four of those strings by putting a hash character (`#`) in front of them. (They're included by default as a common-case convenience, but we'll activate and discuss them later.) While you're at it, modify the default `MIDDLEWARE_CLASSES` and `TEMPLATE_CONTEXT_PROCESSORS` settings. These depend on some of the apps we just commented out. Then, add `'mysite.books'` to the `INSTALLED_APPS` list, so the setting ends up looking like this:

```
MIDDLEWARE_CLASSES = []
TEMPLATE_CONTEXT_PROCESSORS = []
INSTALLED_APPS = (
    #'django.contrib.auth',
    #'django.contrib.contenttypes',
    #'django.contrib.sessions',
    #'django.contrib.sites',
    'mysite.books',
)
```

Note As we're dealing with a single-element tuple here, don't forget the trailing comma. By the way, this book's authors prefer to put a comma after *every* element of a tuple, regardless of whether the tuple has only a single element. This avoids the issue of forgetting commas, and there's no penalty for using that extra comma.

`'mysite.books'` refers to the `books` app we're working on. Each app in `INSTALLED_APPS` is represented by its full Python path—that is, the path of packages, separated by dots, leading to the app package.

Now that the Django app has been activated in the settings file, we can create the database tables in our database. First, let's validate the models by running this command:

```
python manage.py validate
```

The `validate` command checks whether your models' syntax and logic are correct. If all is well, you'll see the message `0 errors found`. If you don't, make sure you typed in the model code correctly. The error output should give you helpful information about what was wrong with the code.

Anytime you think you have problems with your models, run `python manage.py validate`. It tends to catch all the common model problems.

If your models are valid, run the following command for Django to generate `CREATE TABLE` statements for your models in the `books` app (with colorful syntax highlighting available if you're using Unix):

```
python manage.py sqlall books
```

In this command, `books` is the name of the app. It's what you specified when you ran the command `manage.py startapp`. When you run the command, you should see something like this:

```
BEGIN;
CREATE TABLE "books_publisher" (
    "id" serial NOT NULL PRIMARY KEY,
    "name" varchar(30) NOT NULL,
    "address" varchar(50) NOT NULL,
    "city" varchar(60) NOT NULL,
    "state_province" varchar(30) NOT NULL,
    "country" varchar(50) NOT NULL,
    "website" varchar(200) NOT NULL
);
CREATE TABLE "books_book" (
    "id" serial NOT NULL PRIMARY KEY,
    "title" varchar(100) NOT NULL,
    "publisher_id" integer NOT NULL REFERENCES "books_publisher" ("id"),
    "publication_date" date NOT NULL
);
```

```
CREATE TABLE "books_author" (
    "id" serial NOT NULL PRIMARY KEY,
    "salutation" varchar(10) NOT NULL,
    "first_name" varchar(30) NOT NULL,
    "last_name" varchar(40) NOT NULL,
    "email" varchar(75) NOT NULL,
    "headshot" varchar(100) NOT NULL
);
CREATE TABLE "books_book_authors" (
    "id" serial NOT NULL PRIMARY KEY,
    "book_id" integer NOT NULL REFERENCES "books_book" ("id"),
    "author_id" integer NOT NULL REFERENCES "books_author" ("id"),
    UNIQUE ("book_id", "author_id")
);
CREATE INDEX books_book_publisher_id ON "books_book" ("publisher_id");
COMMIT;
```

Note the following:

- Table names are automatically generated by combining the name of the app (`books`) and the lowercased name of the model (`publisher`, `book`, and `author`). You can override this behavior, as detailed in Appendix B.
- As we mentioned earlier, Django adds a primary key for each table automatically—the `id` fields. You can override this, too.
- By convention, Django appends `"_id"` to the foreign key field name. As you might have guessed, you can override this behavior as well.
- The foreign key relationship is made explicit by a `REFERENCES` statement.
- These `CREATE TABLE` statements are tailored to the database you're using, so database-specific field types such as `auto_increment` (MySQL), `serial` (PostgreSQL), or `integer primary key` (SQLite) are handled for you automatically. The same goes for quoting of column names (e.g., using double quotes or single quotes). This example output is in PostgreSQL syntax.

The `sqlall` command doesn't actually create the tables or otherwise touch your database—it just prints output to the screen so you can see what SQL Django would execute if you asked it. If you wanted to, you could copy and paste this SQL into your database client, or use Unix pipes to pass it directly. However, Django provides an easier way of committing the SQL to the database. Run the `syncdb` command, like so:

```
python manage.py syncdb
```

You'll see something like this:

```
Creating table books_publisher
Creating table books_book
Creating table books_author
Installing index for books.Book model
```

The `syncdb` command is a simple "sync" of your models to your database. It looks at all of the models in each app in your `INSTALLED_APPS` setting, checks the database to see whether the appropriate tables exist yet, and creates the tables if they don't yet exist. Note that `syncdb` does *not* sync changes in models or deletions of models; if you make a change to a model or delete a model, and you want to update the database, `syncdb` will not handle that. (More on this later.)

If you run `python manage.py syncdb` again, nothing happens, because you haven't added any models to the `books` app or added any apps to `INSTALLED_APPS`. Ergo, it's always safe to run `python manage.py syncdb`—it won't clobber things.

If you're interested, take a moment to dive into your database server's command-line client and see the database tables Django created. You can manually run the command-line client (e.g., `psql` for PostgreSQL) or you can run the command `python manage.py dbshell`, which will figure out which command-line client to run, depending on your `DATABASE_SERVER` setting. The latter is almost always more convenient.

Basic Data Access

Once you've created a model, Django automatically provides a high-level Python API for working with those models. Try it out by running `python manage.py shell` and typing the following:

```
>>> from books.models import Publisher
>>> p1 = Publisher(name='Addison-Wesley',
        address='75 Arlington St.',
...     city='Boston', state_province='MA', country='U.S.A.',
...     website='http://www.addison-wesley.com/')
>>> p.save()
>>> p2 = Publisher(name="O'Reilly", address='10 Fawcett St.',
...     city='Cambridge', state_province='MA', country='U.S.A.',
...     website='http://www.oreilly.com/')
>>> p2.save()
>>> publisher_list = Publisher.objects.all()
>>> publisher_list
[<Publisher: Publisher object>, <Publisher: Publisher object>]
```

These few lines of code accomplish quite a bit. Here are the highlights:

- To create an object, just import the appropriate model class and instantiate it by passing in values for each field.
- To save the object to the database, call the `save()` method on the object. Behind the scenes, Django executes an SQL `INSERT` statement here.
- To retrieve objects from the database, use the attribute `Publisher.objects`. Fetch a list of all `Publisher` objects in the database with the statement `Publisher.objects.all()`. Behind the scenes, Django executes an SQL `SELECT` statement here.

Naturally, you can do quite a lot with the Django database API—but first, let's take care of a small annoyance.

Adding Model String Representations

When we printed out the list of publishers, all we got was this unhelpful display, which makes it difficult to tell the `Publisher` objects apart:

```
[<Publisher: Publisher object>, <Publisher: Publisher object>]
```

We can fix this easily by adding a method called `__str__()` to our `Publisher` object. A `__str__()` method tells Python how to display the "string" representation of an object. You can see this in action by adding a `__str__()` method to the three models:

```
from django.db import models
class Publisher(models.Model):
    name = models.CharField(maxlength=30)
    address = models.CharField(maxlength=50)
    city = models.CharField(maxlength=60)
    state_province = models.CharField(maxlength=30)
    country = models.CharField(maxlength=50)
    website = models.URLField()

    def __str__(self):
        return self.name

class Author(models.Model):
    salutation = models.CharField(maxlength=10)
    first_name = models.CharField(maxlength=30)
    last_name = models.CharField(maxlength=40)
    email = models.EmailField()
    headshot = models.ImageField(upload_to='/tmp')

    def __str__(self):
        return '%s %s' % (self.first_name, self.last_name)

class Book(models.Model):
    title = models.CharField(maxlength=100)
    authors = models.ManyToManyField(Author)
    publisher = models.ForeignKey(Publisher)
    publication_date = models.DateField()

    def __str__(self):
        return self.title
```

As you can see, a `__str__()` method can do whatever it needs to do in order to return a string representation. Here, the `__str__()` methods for `Publisher` and `Book` simply return the object's name and title, respectively, but the `__str__()` for `Author` is slightly more complex: it pieces together the `first_name` and `last_name` fields. The only requirement for `__str__()` is that it return a string. If `__str__()` doesn't return a string—if it returns, say, an integer—then Python will raise a `TypeError` with a message like "`__str__ returned non-string`".

For the changes to take effect, exit out of the Python shell and enter it again with `python manage.py shell`. (This is the simplest way to make code changes take effect.) Now the list of `Publisher` objects is much easier to understand:

```
>>> from books.models import Publisher
>>> publisher_list = Publisher.objects.all()
>>> publisher_list
[<Publisher: Addison-Wesley>, <Publisher: O'Reilly>]
```

Make sure any model you define has a `__str__()` method—not only for your own convenience when using the interactive interpreter, but also because Django uses the output of `__str__()` in several places when it needs to display objects.

Finally, note that `__str__()` is a good example of adding *behavior* to models. A Django model describes more than the database table layout for an object; it also describes any functionality that object knows how to do. `__str__()` is one example of such functionality—a model knows how to display itself.

Inserting and Updating Data

You've already seen this done—to insert a row into your database, first create an instance of your model using keyword arguments, like so:

```
>>> p = Publisher(name='Apress',
...         address='2855 Telegraph Ave.',
...         city='Berkeley',
...         state_province='CA',
...         country='U.S.A.',
...         website='http://www.apress.com/')
```

This act of instantiating a model class does *not* touch the database.

To save the record into the database (i.e., to perform the SQL `INSERT` statement), call the object's `save()` method:

```
>>> p.save()
```

In SQL, this can roughly be translated into the following:

```
INSERT INTO book_publisher
    (name, address, city, state_province, country, website)
VALUES
    ('Apress', '2855 Telegraph Ave.', 'Berkeley', 'CA',
     'U.S.A.', 'http://www.apress.com/');
```

Because the `Publisher` model uses an autoincrementing primary key `id`, the initial call to `save()` does one more thing: it calculates the primary key value for the record and sets it to the `id` attribute on the instance:

```
>>> p.id
52    # this will differ based on your own data
```

Subsequent calls to `save()` will save the record in place, without creating a new record (i.e., performing an SQL `UPDATE` statement instead of an `INSERT`):

```
>>> p.name = 'Apress Publishing'
>>> p.save()
```

The preceding `save()` statement will result in roughly the following SQL:

```
UPDATE book_publisher SET
    name = 'Apress Publishing',
    address = '2855 Telegraph Ave.',
    city = 'Berkeley',
    state_province = 'CA',
    country = 'U.S.A.',
    website = 'http://www.apress.com'
WHERE id = 52;
```

Selecting Objects

Creating and updating data sure is fun, but it is also useless without a way to sift through that data. We've already seen a way to look up all the data for a certain model:

```
>>> Publisher.objects.all()
[<Publisher: Addison-Wesley>, <Publisher: O'Reilly>, <Publisher: Apress Publishing>
```

This roughly translates to the following SQL:

```
SELECT
    id, name, address, city, state_province, country, website
FROM book_publisher;
```

Notice that Django doesn't use `SELECT *` when looking up data and instead lists all fields explicitly. This is by design, as in certain circumstances `SELECT *` can be slower, and (more important) listing fields more closely follows one tenet of the Zen of Python: "Explicit is better than implicit." For more on the Zen of Python, try typing `import this` at a Python prompt.

Let's take a close look at each part of this `Publisher.objects.all()` line:

- First, we have the model we defined, `Publisher`. No surprise here: when you want to look up data, you use the model for that data.
- Next, we have this `objects` business. Technically, this is a *manager*. Managers are discussed in detail in Appendix B. For now, all you need to know is that managers take care of all "table-level" operations on data including, most important, data lookup. All objects automatically get an `objects` manager; you'll use it any time you want to look up model instances.
- Finally, we have `all()`. This is a method on the `objects` manager that returns all the rows in the database. Though this object *looks* like a list, it's actually a `QuerySet`—an object that represents some set of rows from the database. Appendix C deals with `QuerySets` in detail. For the rest of this chapter, we'll just treat them like the lists they emulate.

Any database lookup is going to follow this general pattern—we'll call methods on the manager attached to the model we want to query against.

Filtering Data

While fetching all objects certainly has its uses, most of the time we're going to want to deal with a subset of the data. We do this with the `filter()` method:

```
>>> Publisher.objects.filter(name="Apress Publishing")
[<Publisher: Apress Publishing>]
```

`filter()` takes keyword arguments that get translated into the appropriate SQL `WHERE` clauses. The preceding example would get translated into something like this:

```
SELECT
    id, name, address, city, state_province, country, website
FROM book_publisher
WHERE name = 'Apress Publishing';
```

You can pass multiple arguments into `filter()` to narrow down things further:

```
>>> Publisher.objects.filter(country="U.S.A.", state_province="CA")
[<Publisher: Apress Publishing>]
```

Those multiple arguments get translated into SQL `AND` clauses. Thus, the example in the code snippet translates into the following:

```
SELECT
    id, name, address, city, state_province, country, website
FROM book_publisher
WHERE country = 'U.S.A.' AND state_province = 'CA';
```

Notice that by default the lookups use the SQL `=` operator to do exact match lookups. Other lookup types are available:

```
>>> Publisher.objects.filter(name__contains="press")
[<Publisher: Apress Publishing>]
```

That's a double underscore there between `name` and `contains`. Like Python itself, Django uses the double underscore to signal that something "magic" is happening—here, the `__contains` part gets translated by Django into an SQL `LIKE` statement:

```
SELECT
    id, name, address, city, state_province, country, website
FROM book_publisher
WHERE name LIKE '%press%';
```

Many other types of lookups are available, including `icontains` (case-insensitive `LIKE`), `startswith` and `endswith`, and `range` (SQL `BETWEEN` queries). Appendix C describes all of these lookup types in detail.

Retrieving Single Objects

Sometimes you want to fetch only a single object. That's what the `get()` method is for:

```
>>> Publisher.objects.get(name="Apress Publishing")
<Publisher: Apress Publishing>
```

Instead of a list (rather, `QuerySet`), only a single object is returned. Because of that, a query resulting in multiple objects will cause an exception:

```
>>> Publisher.objects.get(country="U.S.A.")
Traceback (most recent call last):
    ...
AssertionError: get() returned more than one Publisher—it returned 2!
```

A query that returns no objects also causes an exception:

```
>>> Publisher.objects.get(name="Penguin")
Traceback (most recent call last):
    ...
DoesNotExist: Publisher matching query does not exist.
```

Ordering Data

As you play around with the previous examples, you might discover that the objects are being returned in a seemingly random order. You aren't imagining things—so far we haven't told the database how to order its results, so we're simply getting back data in some arbitrary order chosen by the database.

That's obviously a bit silly; we wouldn't want a Web page listing publishers to be ordered randomly. So, in practice, we'll probably want to use `order_by()` to reorder our data into a useful list:

```
>>> Publisher.objects.order_by("name")
[<Publisher: Apress Publishing>, <Publisher: Addison-Wesley>, <Publisher: O'Reilly>]
```

This doesn't look much different from the earlier `all()` example, but the SQL now includes a specific ordering:

```
SELECT
    id, name, address, city, state_province, country, website
FROM book_publisher
ORDER BY name;
```

We can order by any field we like:

```
>>> Publisher.objects.order_by("address")
[<Publisher: O'Reilly>, <Publisher: Apress Publishing>, <Publisher: Addison-Wesley>]

>>> Publisher.objects.order_by("state_province")
[<Publisher: Apress Publishing>, <Publisher: Addison-Wesley>, <Publisher: O'Reilly>]
```

and by multiple fields:

```
>>> Publisher.objects.order_by("country", "address")
[<Publisher: Apress Publishing>, <Publisher: O'Reilly>, <Publisher: Addison-Wesley>]
```

We can also specify reverse ordering by prefixing the field name with a – (a minus sign character):

```
>>> Publisher.objects.order_by("-name")
[<Publisher: O'Reilly>, <Publisher: Apress Publishing>, <Publisher: Addison-Wesley>]
```

While this flexibility is useful, using `order_by()` all the time can be quite repetitive. Most of the time you'll have a particular field you usually want to order by. In these cases, Django lets you attach a default ordering to the model:

```
class Publisher(models.Model):
    name = models.CharField(maxlength=30)
    address = models.CharField(maxlength=50)
    city = models.CharField(maxlength=60)
    state_province = models.CharField(maxlength=30)
    country = models.CharField(maxlength=50)
    website = models.URLField()

    def __str__(self):
        return self.name

    class Meta:
        ordering = ["name"]
```

This `ordering = ["name"]` bit tells Django that unless an ordering is given explicitly with `order_by()`, all publishers should be ordered by name.

■ **Note** Django uses the internal `class Meta` as a place to specify additional metadata about a model. It's completely optional, but it can do some very useful things. See Appendix B for the options you can put under `Meta`.

Chaining Lookups

You've seen how you can filter data, and you've seen how you can order it. At times, of course, you're going to want to do both. In these cases, you simply "chain" the lookups together:

```
>>> Publisher.objects.filter(country="U.S.A.").order_by("-name")
[<Publisher: O'Reilly>, <Publisher: Apress Publishing>, <Publisher: Addison-Wesley>]
```

As you might expect, this translates to an SQL query with both a `WHERE` and an `ORDER BY`:

```
SELECT
    id, name, address, city, state_province, country, website
FROM book_publisher
WHERE country = 'U.S.A'
ORDER BY name DESC;
```

You can keep chaining queries as long as you like. There's no limit.

Slicing Data

Another common need is to look up only a fixed number of rows. Imagine you have thousands of publishers in your database, but you want to display only the first one. You can do this using Python's standard list slicing syntax:

```
>>> Publisher.objects.all()[0]
<Publisher: Addison-Wesley>
```

This translates roughly to the following:

```
SELECT
    id, name, address, city, state_province, country, website
FROM book_publisher
ORDER BY name
LIMIT 1;
```

■ **Note** We've just scratched the surface of dealing with models, but you should now know enough to understand all the examples in the rest of this book. When you're ready to learn the complete details behind object lookups, turn to Appendix C.

Deleting Objects

To delete objects, simply call the `delete()` method on your object:

```
>>> apress = Publisher.objects.get(name="Addison-Wesley")
>>> apress.delete()
>>> Publisher.objects.all()
[<Publisher: Apress Publishing>, <Publisher: O'Reilly>]
```

You can also delete objects in bulk by calling `delete()` on the result of some lookup:

```
>>> publishers = Publisher.objects.all()
>>> publishers.delete()
>>> Publisher.objects.all()
[]
```

Deletions are *permanent*, so be careful! In fact, it's usually a good idea to avoid deleting objects unless you absolutely have to—relational databases don't do "undo" so well, and restoring from backups is painful.

It's often a good idea to add "active" flags to your data models. You can look up only "active" objects, and simply set the active field to `False` instead of deleting the object. Then, if you realize you've made a mistake, you can simply flip the flag back.

Making Changes to a Database Schema

When we introduced the `syncdb` command earlier in this chapter, we noted that `syncdb` merely creates tables that don't yet exist in your database—it does *not* sync changes in models or perform deletions of models. If you add or change a model's field, or if you delete a model, you'll need to make the change in your database manually. This section explains how to do that.

When dealing with schema changes, it's important to keep a few things in mind about how Django's database layer works:

- Django will complain loudly if a model contains a field that has not yet been created in the database table. This will cause an error the first time you use the Django database API to query the given table (i.e., it will happen at code execution time, not at compilation time).
- Django does *not* care if a database table contains columns that are not defined in the model.
- Django does *not* care if a database contains a table that is not represented by a model.

Making schema changes is a matter of changing the various pieces—the Python code and the database itself—in the right order.

Adding Fields

When adding a field to a table/model in a production setting, the trick is to take advantage of the fact that Django doesn't care if a table contains columns that aren't defined in the model. The strategy is to add the column in the database and then update the Django model to include the new field.

However, there's a bit of a chicken-and-egg problem here, because in order to know how the new database column should be expressed in SQL, you need to look at the output of Django's `manage.py sqlall` command, which requires that the field exist in the model. (Note that you're not *required* to create your column with exactly the same SQL that Django would, but it's a good idea to do so, just to be sure everything's in sync.)

The solution to the chicken-and-egg problem is to use a development environment instead of making the changes on a production server. (You *are* using a testing/development environment, right?) Here are the detailed steps to take.

First, take these steps in the development environment (i.e., not on the production server):

1. Add the field to your model.
2. Run `manage.py sqlall [yourapp]` to see the new `CREATE TABLE` statement for the model. Note the column definition for the new field.
3. Start your database's interactive shell (e.g., `psql` or `mysql`, or you can use `manage.py dbshell`). Execute an `ALTER TABLE` statement that adds your new column.

4. (Optional.) Launch the Python interactive shell with `manage.py shell` and verify that the new field was added properly by importing the model and selecting from the table (e.g., `MyModel.objects.all()[:5]`).

Then on the production server perform these steps:

1. Start your database's interactive shell.
2. Execute the `ALTER TABLE` statement you used in step 3 of the development environment steps.
3. Add the field to your model. If you're using source-code revision control and you checked in your change in development environment step 1, now is the time to update the code (e.g., `svn update`, with Subversion) on the production server.
4. Restart the Web server for the code changes to take effect.

For example, let's walk through what we'd do if we added a `num_pages` field to the `Book` model described earlier in this chapter. First, we'd alter the model in our development environment to look like this:

```
class Book(models.Model):
    title = models.CharField(maxlength=100)
    authors = models.ManyToManyField(Author)
    publisher = models.ForeignKey(Publisher)
    publication_date = models.DateField()
    num_pages = models.IntegerField(blank=True, null=True)

    def __str__(self):
        return self.title
```

■ **Note** Read the "Adding NOT NULL Columns" sidebar for important details on why we included `blank=True` and `null=True`.

Then we'd run the command `manage.py sqlall books` to see the `CREATE TABLE` statement. It would look something like this:

```
CREATE TABLE "books_book" (
    "id" serial NOT NULL PRIMARY KEY,
    "title" varchar(100) NOT NULL,
    "publisher_id" integer NOT NULL REFERENCES "books_publisher" ("id"),
    "publication_date" date NOT NULL,
    "num_pages" integer NULL
);
```

The new column is represented like this:

```
"num_pages" integer NULL
```

Next, we'd start the database's interactive shell for our development database by typing `psql` (for PostgreSQL), and we'd execute the following statements:

```
ALTER TABLE books_book ADD COLUMN num_pages integer;
```

ADDING NOT NULL COLUMNS

There's a subtlety here that deserves mention. When we added the `num_pages` field to our model, we included the `blank=True` and `null=True` options. We did this because a database column will contain NULL values when you first create it.

However, it's also possible to add columns that cannot contain NULL values. To do this, you have to create the column as `NULL`, then populate the column's values using some default(s), and then alter the column to set the `NOT NULL` modifier. For example:

```
BEGIN;
ALTER TABLE books_book ADD COLUMN num_pages integer;
UPDATE books_book SET num_pages=0;
ALTER TABLE books_book ALTER COLUMN num_pages SET NOT NULL;
COMMIT;
```

If you go down this path, remember that you should leave off `blank=True` and `null=True` in your model.

After the `ALTER TABLE` statement, we'd verify that the change worked properly by starting the Python shell and running this code:

```
>>> from mysite.books.models import Book
>>> Book.objects.all()[:5]
```

If that code didn't cause errors, we'd switch to our production server and execute the `ALTER TABLE` statement on the production database. Then, we'd update the model in the production environment and restart the Web server.

Removing Fields

Removing a field from a model is a lot easier than adding one. To remove a field, just follow these steps:

1. Remove the field from your model and restart the Web server.
2. Remove the column from your database, using a command like this:

   ```
   ALTER TABLE books_book DROP COLUMN num_pages;
   ```

Removing Many-to-Many Fields

Because many-to-many fields are different from normal fields, the removal process is different:

1. Remove the `ManyToManyField` from your model and restart the Web server.
2. Remove the many-to-many table from your database, using a command like this:

```
DROP TABLE books_books_publishers;
```

Removing Models

Removing a model entirely is as easy as removing a field. To remove a model, just follow these steps:

1. Remove the model from your `models.py` file and restart the Web server.
2. Remove the table from your database, using a command like this:

```
DROP TABLE books_book;
```

What's Next?

Once you've defined your models, the next step is to populate your database with data. You might have legacy data, in which case Chapter 16 will give you advice about integrating with legacy databases. You might rely on site users to supply your data, in which case Chapter 7 will teach you how to process user-submitted form data.

But in some cases, you or your team might need to enter data manually, in which case it would be helpful to have a Web-based interface for entering and managing data. The next chapter covers Django's admin interface, which exists precisely for that reason.