Module-2: Django Templates and Models

Template System Basics (P31-32), Using Django Template System (P33-40), Basic Template Tags and Filters (P40-47), Template Loading (P49-54), Template Inheritance (P54-58)

MVT Development Pattern (P59-62). Configuring Databases (P62-65), Defining and Implementing Models (P65-71), Basic Data Access (P71), Adding Model String Representations (P72-73), Inserting/Updating data (P73-74), Selecting and deleting objects (P74-79), Schema Evolution (P79-82)

Template System Basics (P31-32)

- 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.

Using Django Template System (P33-40)

- 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.

Topics covered: **Creating Template Objects - Rendering a Template - Multiple Contexts, Same Template - Context Variable Lookup - Playing with Context Objects**

1. 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.
- From within the **project directory** created by **django-admin.py startproject**, 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

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.

different every time, and it doesn't really matter; it's simply the Python **"identity" of the Template object**.

- 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**:
  1. Invalid block tags
  2. Invalid arguments to valid block tags
  3. Invalid filters
  4. Invalid arguments to valid filters
  5. Invalid template syntax
  6. Unclosed block tags (for block tags that require closing tags)

2. 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": "Sarathy"})
>>> t.render(c)
'My name is Sarathy.'
```

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.

- 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 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, 2024, which is displayed according to the format string 'F j, Y'.

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 (Dictionary lookup, **Attribute** lookup, **Method** call, **List-index** 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': 'Shalini', 'age': '23'}
>>> t = Template('{{ person.name }} is {{ person.age }} years old.')
>>> c = Context({'person': person})
>>> t.render(c)
'Shalini is 23 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(2024, 5, 2)
>>> d.year 2024
>>> 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 2024.'
```

- 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: 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**.

- 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**.
- **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': 'Shalini', 'age': '23'}
>>> t = Template('{{ person.name.upper }} is {{ person.age }} years old.')
>>> c = Context({'person': person})
>>> t.render(c)
'SHALINI is 23 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 our Django configuration. We discuss this **further in Chapter "Generating Non-HTML Content".**

Playing with Context Objects

- Most of the time, we'll instantiate Context objects by passing in a fully populated dictionary to Context().
- But we 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 (P40-47)

- The template system ships with built-in tags and filters. The following sections outline the common Django tags.

Django tags : *if/else – for - ifequal/ifnotequal – Comments*

" 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 %}
```

NOTE: *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.
- 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.

****

" 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**.
- 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 hardcoded 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

- 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.

- **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" }}.
- **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 "
  - Converts ' (single quote) to '
- 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).

****

Using Templates in Views

- We'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.**
- We might first consider saving our template somewhere on our 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 we 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**.

Template Loading (P49-54)

- **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 we'll need to tell the framework where we store our templates. **The place to do this is in our settings file.**
- **A Django settings file is the place to put configuration for our Django instance**. It's a simple Python module with module-level variables, one for each setting. When we ran **django-admin.py startproject mysite**, **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, **we can do dynamic things such as checking the value of one variable before setting another**.
- 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:

- We can specify any directory we want, as long as the directory and templates within that directory are readable by the user account under which your Web server runs. If

we can't think of an appropriate place to put our templates, we recommend creating a templates directory within our Django project (i.e., within the mysite directory you created).

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

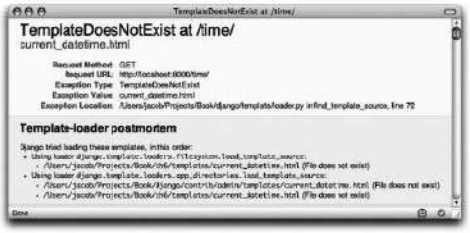
```
TEMPLATE_DIRS = [
'/home/django/mysite/templates'
]
```

- 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).
- With **TEMPLATE_DIRS** set, the next step is to change the view code to use Django's **templateloading** 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, 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:



- As we can probably **tell from the error messages** found in the above Figure, **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 our **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 our template directory using the following template code:

  <html><body>It is now **{{ current_date }}**.</body></html>
- **Refresh the page** in our Web browser, and we should **see the fully rendered page.**

****

Topics covered: render_to_response() - The locals() Trick - Subdirectories in get_template() - The include Template Tag

****

"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 us 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, we'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})
```

- 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, we'll **find ourself 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.
- One thing to watch out for when using ***locals()*** is that it includes every ***local*** variable, which may comprise more variables than we actually want our template to have access to. In the previous example, ***locals()*** will also include ***request***. Whether this matters to us depends on our 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 we specify the context dictionary manually, we avoid this overhead.

****

"Subdirectories in get_template()"

- Storing templates in subdirectories of our template directory is easy. In our 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.

****

"The include Template Tag"

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

Template Inheritance (P54-58)

- In the real world, we'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 does one 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. However, 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>
```

What happens when we want to create a template for another view —say, the ***hours_ahead*** view full HTML template, we'd create something like:

```
<!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>
```

- 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 we'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.
- Django's template inheritance system solves these problems. We can think of it as an "inside-out" version of server-side includes. Instead of defining the ***snippets*** that are ***common***, we 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.***
- 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 %}
```

- Let's create a template for the ***hours_ahead*** view 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 %}
```

- Each template contains only the code that's unique to that template. No redundancy needed. If you need to make a site-wide design change, **just make the change to *base.html***, and **all of the other templates will immediately reflect the change**.
- 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: 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***:
  - Create a ***base.html* template** that holds the main look and feel of your site. This stuff rarely, if ever, changes.
  - 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.***
  - 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 section-wide navigation.

- Here are some tips for working with template inheritance:
  - If we 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 our base templates, ***the better***.
  - If we find ourself ***duplicating code in a number of templates***, it probably means we should ***move that code to a {% block %}*** in a ***parent template***.
  - If we 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.
  - We 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 we pass to ***{% extends %}*** is loaded using the same method that ***get_template()*** uses. That is, the template name is appended to our ***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.
