Chapter 2

Using Template System
Basic Template Tags and Filters
Template Loading
Inheritance

Issues in existing condition

- Any change to the design of the page requires a change to 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
- Similarly, it’s most efficient if programmers can work on Python code and designers can work on templates at the same time, rather than working serially.

Using the Template System

- Two steps
- Create a Template object by providing the raw template code as a string. Or create Template objects by designating the path to a template file on the filesystem
- Call the render() method of the Template object with a given set of variables

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

Rendering a Template

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

Template using a variable

- ch2_slide6.py
- >>> 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

Alternatives

```
# 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}))
  ```

Context Variable Lookup

```
>>> 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)`
-

Using 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)
```

Dot Lookup

- >>> t = Template('{{ var }} -- {{ var.upper }} -- {{ var.isdigit }}')
- >>> t.render(Context({'var': 'hello'}))
- >>> t = Template('Item 2 is {{ items.2 }}.')
- >>> c = Context({'items': ['apples', 'bananas', 'carrots']})
- >>> t.render(c)
  - Negative list indices are not allowed

Dot Lookup

- 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])
-

Method Call Behavior

- 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
- >>> t = Template("My name is {{ person.first_name }}.")
- >>> class PersonClass3:
- … def first_name(self):
- … raise AssertionError, "foo"
- >>> p = PersonClass3()
- >>> t.render(Context({"person": p}))

Solution

- >>> t = Template("My name is {{ person.first_name }}.")
- >>> class SilentAssertionError(AssertionError):
- ... silent_variable_failure = True
- >>> class PersonClass4:
- ... def first_name(self):
- ... raise SilentAssertionError
- >>> p = PersonClass4()
- >>> t.render(Context({"person": p}))
- How are invalid variables handled?

Syntax

```
{% if today_is_weekend %}
        <p>Welcome to the weekend!</p>
{% else %}
        <p>Get back to work.</p>
{% endif %}
{% if athlete_list and coach_list %}
         Both athletes and coaches are available.
{% endif %}
```

If/else(cont...)

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

Further conditions

- {% ifequal user currentuser %}
- <h1>Welcome!</h1>
- {% endifequal %}
- {% ifequal section 'sitenews' %}
- <h1>Site News</h1>
- {% else %}
- <h1>No News Here</h1>
- {% endifequal %}

Loop

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

- `forloop.counter`: a variable always set to an integer

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

Design philosophies and Limitations

- Business logic should be separated from presentation logic
- Syntax should be decoupled from HTML/XML
- Designers are assumed to be comfortable with HTML code
- Designers are assumed not to be Python programmers.
- A template cannot set a variable or change the value of a variable.
- A template cannot call raw Python code.

Using a template in view

- Solution 1:
- 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)
```

- Solution 2:
- assuming the template was saved as the file /home/djangouser/templates/mytemplate.html:

```
fp=
open('/home/djangouser/templates/mytemplate.html'
)
t = Template(fp.read())
fp.close()
html = t.render(Context({'current_date': now}))
```

Template Loading

- Change the variable TEMPLATE_DIRS to some specified path
- Do not forget the comma at the end.
- Else you can use []

The code:views.py

```
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
```

render_to_response()

- 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
- This shortcut is a function called render_to_response(), which lives in the module django.shortcuts
- Rather than loading templates and creating Context and HttpResponse objects manually.use render_to_response()

The code:urls.py

```
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})
```

The magic of locals

- It returns a dictionary mapping all local variable names to their values
- def current_datetime(request):

  current_date = datetime.datetime.now()

  return render_to_response('current_datetime.html',

  locals())
- doesn’t offer a huge improvement,
- can save you some typing if you have several template variables to define

The include Template Tag

- For header/footer
- {% include 'nav.html' %}
- If it does not exist the tag will fail silently, displaying nothing in the place of the tag.
- Inheritance
- extends