بسم الله

django

PART 2

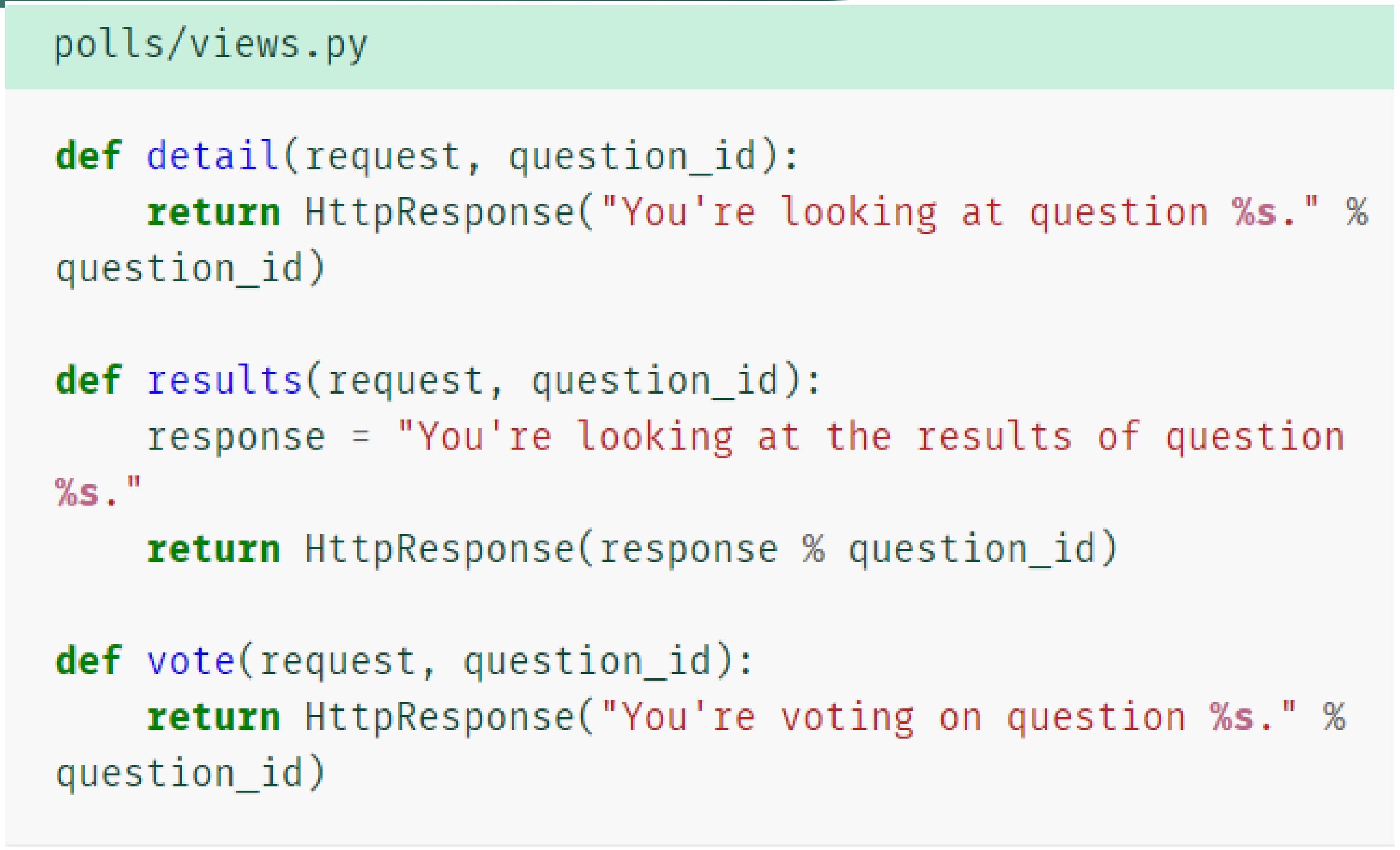
HOSSEIN FORGHANI

MAKTAB SHARIF

Contents

Writing More Views

- For example these views get an argument



```
polls/views.py
```

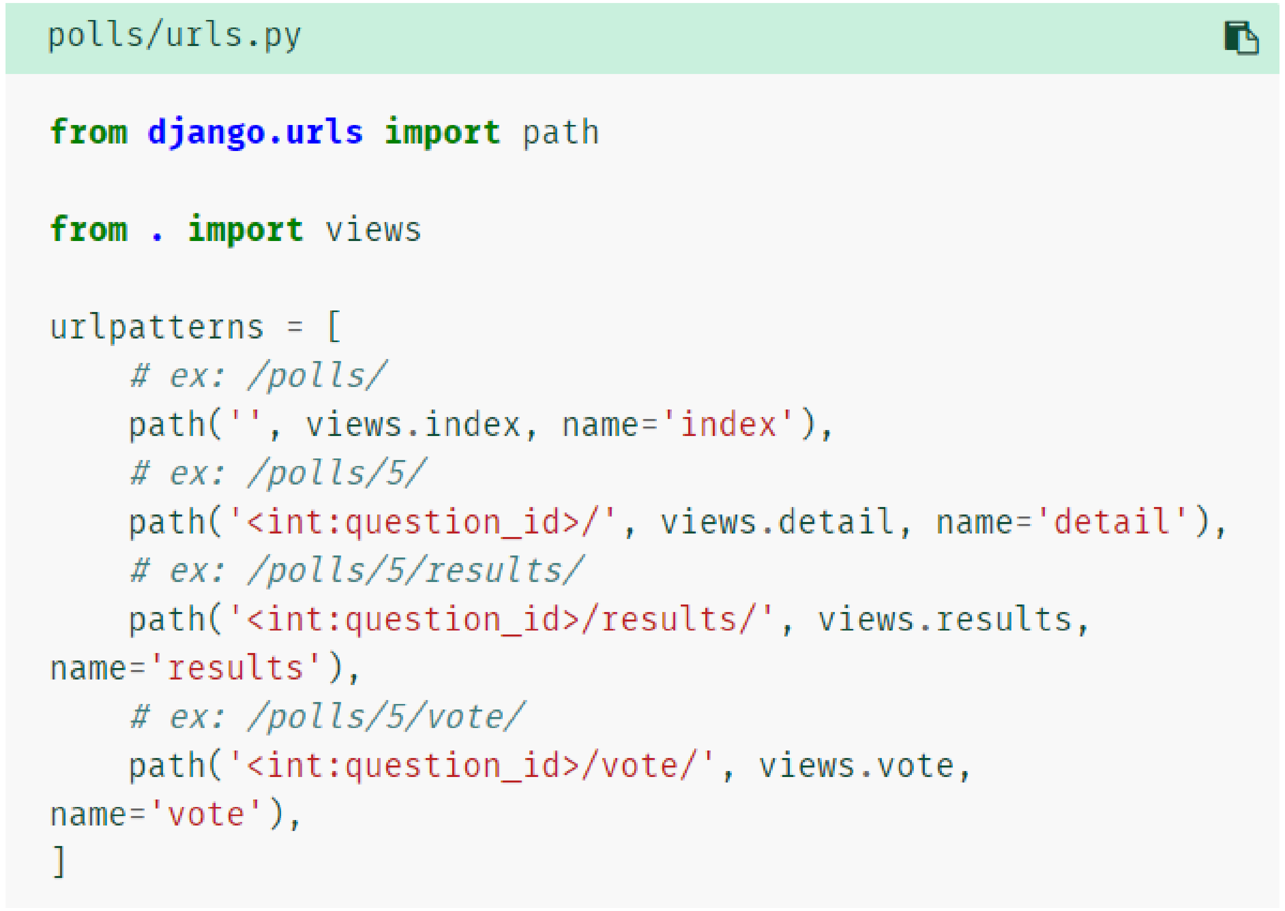
```
def detail(request, question_id):
    return HttpResponse("You're looking at question %s." %
question_id)

def results(request, question_id):
    response = "You're looking at the results of question
%s."
    return HttpResponse(response % question_id)

def vote(request, question_id):
    return HttpResponse("You're voting on question %s." %
question_id)
```

Changing polls.urls

- Wire these new views into the polls.urls module:



polls/urls.py

```
from django.urls import path

from . import views

urlpatterns = [
    # ex: /polls/
    path('', views.index, name='index'),
    # ex: /polls/5/
    path('<int:question_id>/', views.detail, name='detail'),
    # ex: /polls/5/results/
    path('<int:question_id>/results/', views.results,
name='results'),
    # ex: /polls/5/vote/
    path('<int:question_id>/vote/', views.vote,
name='vote'),
]
```

What to do in View?

- Each view is responsible for:
  - returning an HttpResponse object
  - or raising an exception such as Http404
- Use Django's template system to separate the design from Python

Django Templates

- First, create a directory called templates in your polls directory

By convention DjangoTemplates looks for a “templates” subdirectory in each of the INSTALLED_APPS.

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
            ],
        },
    },
]
```

Django Templates – cont.

- create another directory called polls, and within that create a file called index.html
- Your template should be at:

```
polls/templates/polls/index.html
```

- Then you can refer to this template within Django as:

```
polls/index.html
```

Your first Template

- Put the following code in that template:

- But use complete HTML documents:

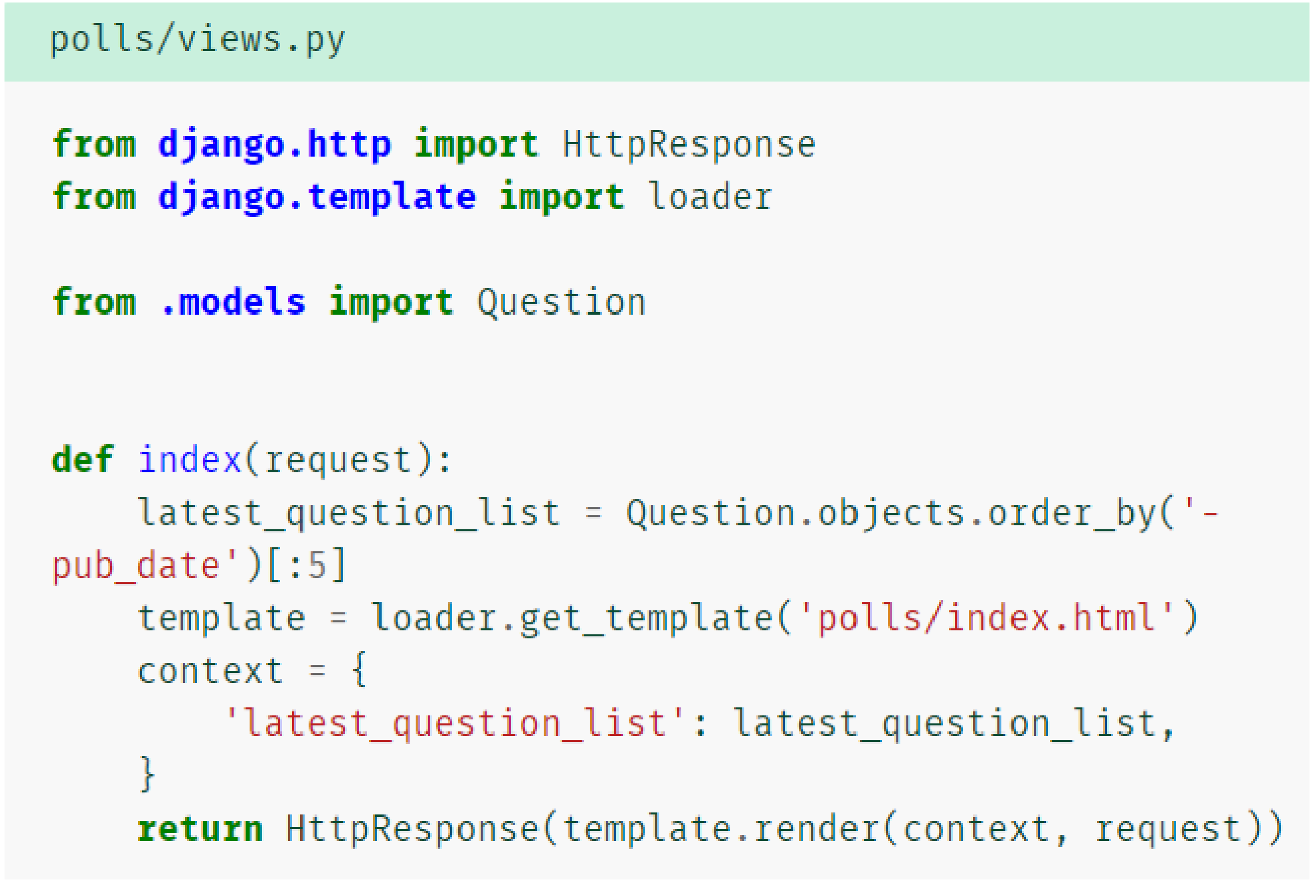
```
<!DOCTYPE html>
<html>
    <head>…</head>
    <body>…</body>
</html>
```

polls/templates/polls/index.html

```
{% if latest_question_list %}
    <ul>
    {% for question in latest_question_list %}
        <li><a href="/polls/{{ question.id }}/">{{ 
question.question_text }}</a></li>
    {% endfor %}
    </ul>
{% else %}
    <p>No polls are available.</p>
{% endif %}
```

Rendering Template

- Update our index view in polls/views.py:
- It loads the template called polls/index.html and passes it a context
- context: a dictionary mapping template variable names to Python objects



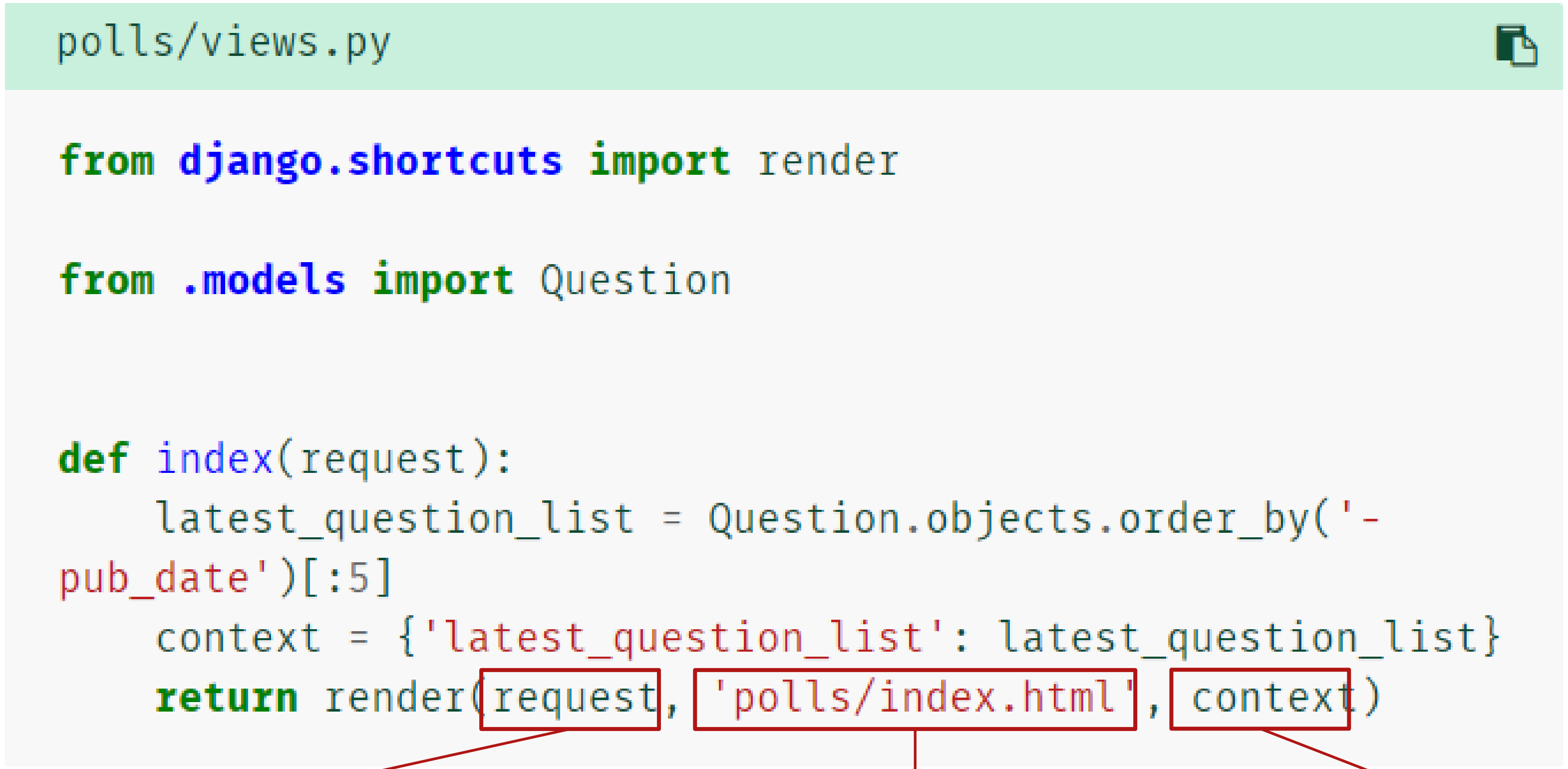
polls/views.py

```
from django.http import HttpResponse
from django.template import loader

from .models import Question


def index(request):
    latest_question_list = Question.objects.order_by('-
pub_date')[:5]
    template = loader.get_template('polls/index.html')
    context = {
        'latest_question_list': latest_question_list,
    }
    return HttpResponse(template.render(context, request))
```

A Shortcut: render()



polls/views.py

```
from django.shortcuts import render

from .models import Question


def index(request):
    latest_question_list = Question.objects.order_by('-
pub_date')[:5]
    context = {'latest_question_list': latest_question_list}
    return render(request, 'polls/index.html', context)
```

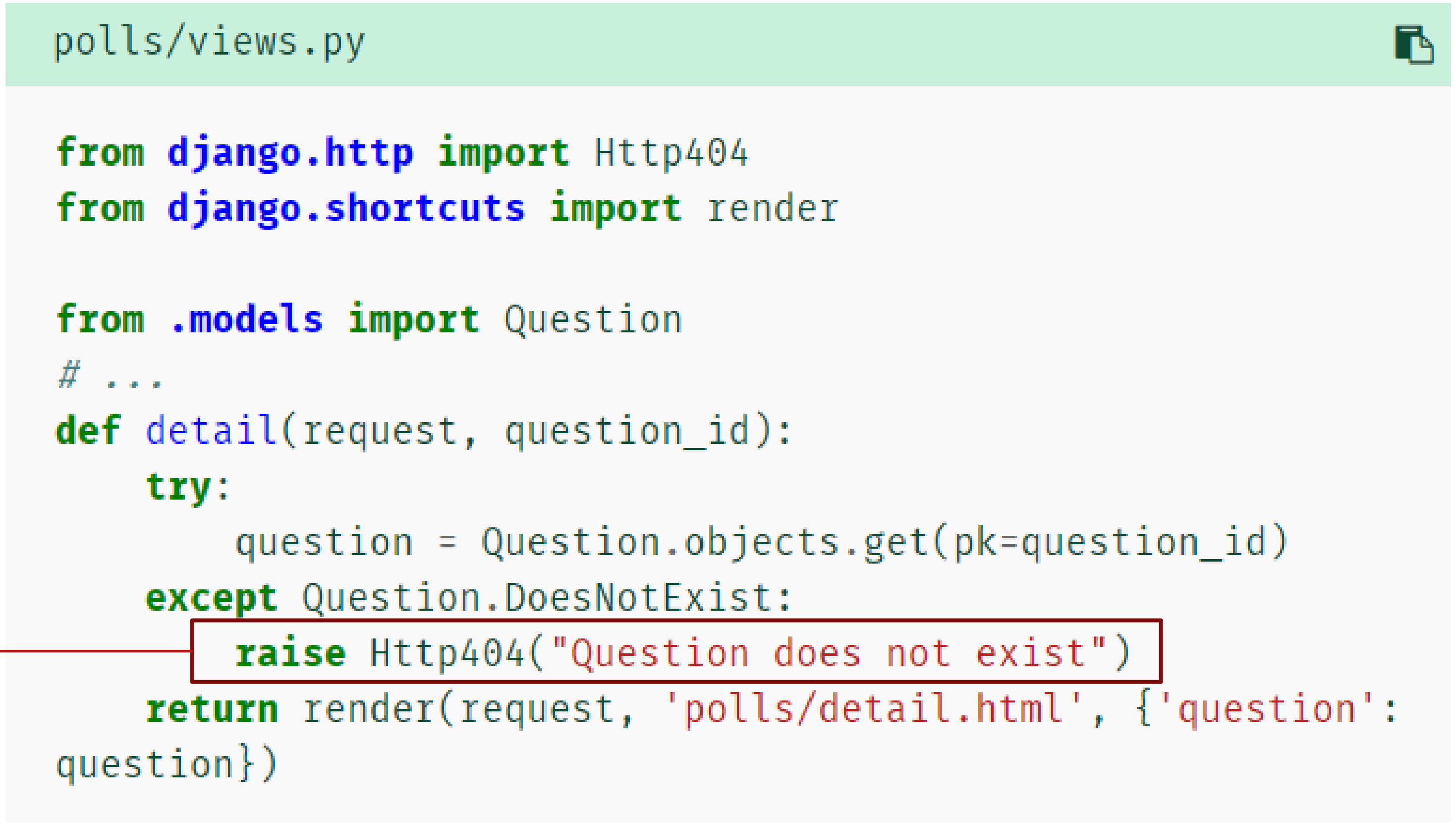
Request object

Template name

A dictionary (optional)

Raising a 404 Error

- Now implement question detail view:



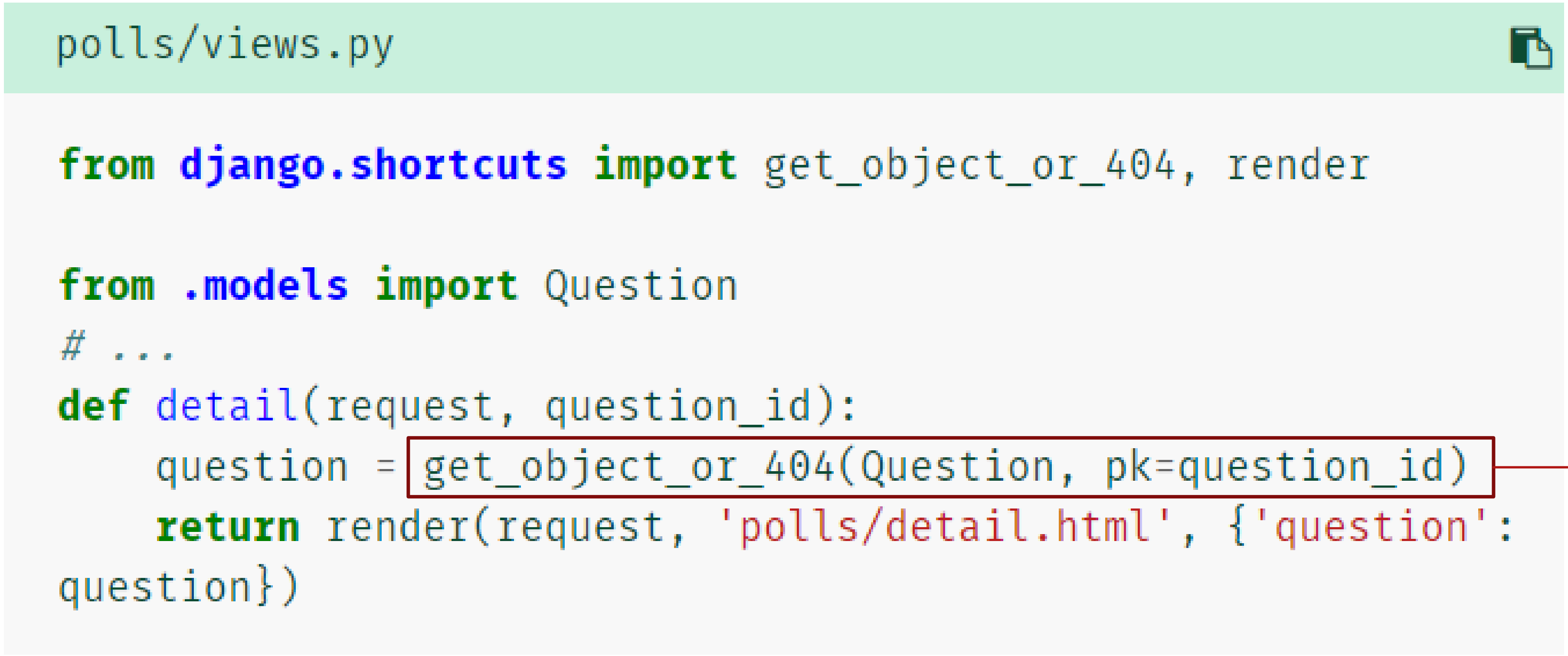
polls/views.py

```python
from django.http import Http404
from django.shortcuts import render

from .models import Question
# ...
def detail(request, question_id):
    try:
        question = Question.objects.get(pk=question_id)
    except Question.DoesNotExist:
        raise Http404("Question does not exist")
    return render(request, 'polls/detail.html', {'question':
question})
```

raises the Http404 exception if a question with the requested ID doesn't exist

A Shortcut: get_object_or_404()


polls/views.py

```
from django.shortcuts import get_object_or_404, render

from .models import Question
# ...
def detail(request, question_id):
    question = get_object_or_404(Question, pk=question_id)
    return render(request, 'polls/detail.html', {'question':
question})
```

Takes a Django model as its first argument and an arbitrary number of keyword arguments, which it passes to the get() function

get_list_or_404()

- works just as get_object_or_404() – except using filter() instead of get(). It raises Http404 if the list is empty.

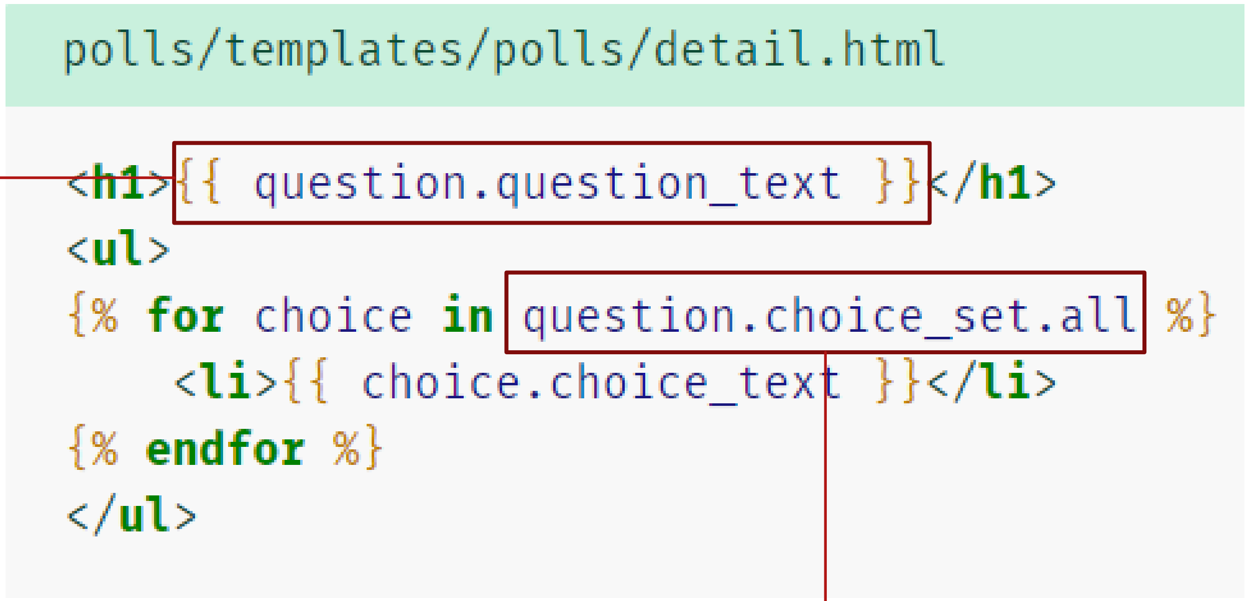
```
get_list_or_404(Question, question_text__startswith="What")
```

Template for Detail Page

May be:

1. question['question_text']
2. question.question_text ✓

Also you can try mylist.0 for mylist[0]



polls/templates/polls/detail.html

```
<h1>{{ question.question_text }}</h1>
<ul>
{% for choice in question.choice_set.all %}
    <li>{{ choice.choice_text }}</li>
{% endfor %}
</ul>
```

question.choice_set.all()

Removing hardcoded URLs

```
<li><a href="/polls/{{ question.id }}/">{{ question.question_text }}</a></li>
```

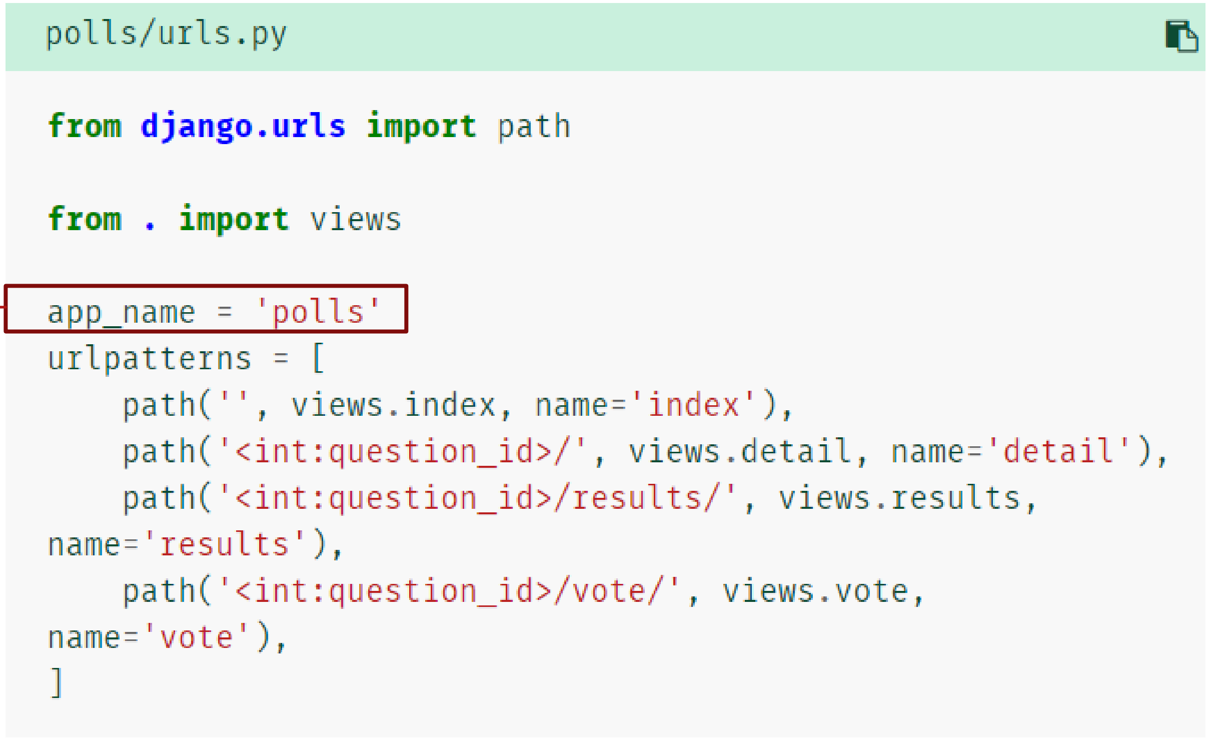
Since you defined the name argument

```
<li><a href="{% url 'detail' question.id %}">{{ question.question_text }}</a></li>
```

Namespacing URL names

- But how to distinguish between the detail url in polls app and detail url in blog app?

By adding namespaces to your URLconf



polls/urls.py

```python
from django.urls import path

from . import views

app_name = 'polls'
urlpatterns = [
    path('', views.index, name='index'),
    path('<int:question_id>/', views.detail, name='detail'),
    path('<int:question_id>/results/', views.results,
name='results'),
    path('<int:question_id>/vote/', views.vote,
name='vote'),
]
```

Namespacing URL names – cont.

- Now change from

```
<li><a href="{% url 'detail' question.id %}">{{ question.question_text }}</a></li>
```

To:

```
<li><a href="{% url 'polls:detail' question.id %}">{{ question.question_text }}</a></li>
```

Template Language

Django Template Language (DTL)

- Very similar to Jinja2
- Variables:

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

- If you use a variable that doesn't exist, by default empty string ('') is inserted

Filters

- Modifies variables for display:

```
{{ name|lower }}
```

→ Makes the string lower-case

- Filters can be chained:

```
{{ text|escape|linebreaks }}
```

→ escaping text contents, then converting line breaks to <p> tags

- Some filters take arguments:

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

→ first 30 words of the bio variable

- Filter arguments that contain spaces must be quoted:

```
{{ list|join:", " }}
```

→ join a list with commas and spaces

Tags

- Tags look like this:

```
{% tag %}
```

- Some create text in the output, some control flow by performing loops or logic, and some load external information into the template
- Some tags require beginning and ending tags:

```
{% tag %} ... tag contents ... {% endtag %}
```

Built-in Tags: for

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

Variables Inside “for”

Variable	**Description**
forloop.counter	The current iteration of the loop (1-indexed)
forloop.counter0	The current iteration of the loop (0-indexed)
forloop.revcounter	The number of iterations from the end of the loop (1-indexed)
forloop.revcounter0	The number of iterations from the end of the loop (0-indexed)
forloop.first	True if this is the first time through the loop
forloop.last	True if this is the last time through the loop
forloop.parentloop	For nested loops, this is the loop surrounding the current one

Built-in Tags: if, elif, and else

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

Built-in Tags: if, elif, and else

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

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

Built-in Tags: comment

- Ignores everything between `{% comment %}` and `{% endcomment %}`

```
<p>Rendered text with {{ pub_date|date:"c" }}</p>
{% comment "Optional note" %}
<p>Commented out text with {{ create_date|date:"c" }}</p>
{% endcomment %}
```

Built-in Tags: autoescape

- A variable will include characters that affect the resulting HTML
- By default, automatically escapes the output of every variable tag
- You can disable auto-escaping by:

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

- Or:

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

Built-in Tags: cycle

- Produces one of its arguments each time this tag is encountered

```
{% for o in some_list %}
      <tr class="{% cycle 'row1' 'row2' %}">
      ...
      </tr>
{% endfor %}
```

- In some cases you might want to refer to the current value of a cycle:

```
{% cycle 'row1' 'row2' as rowcolors %}
```

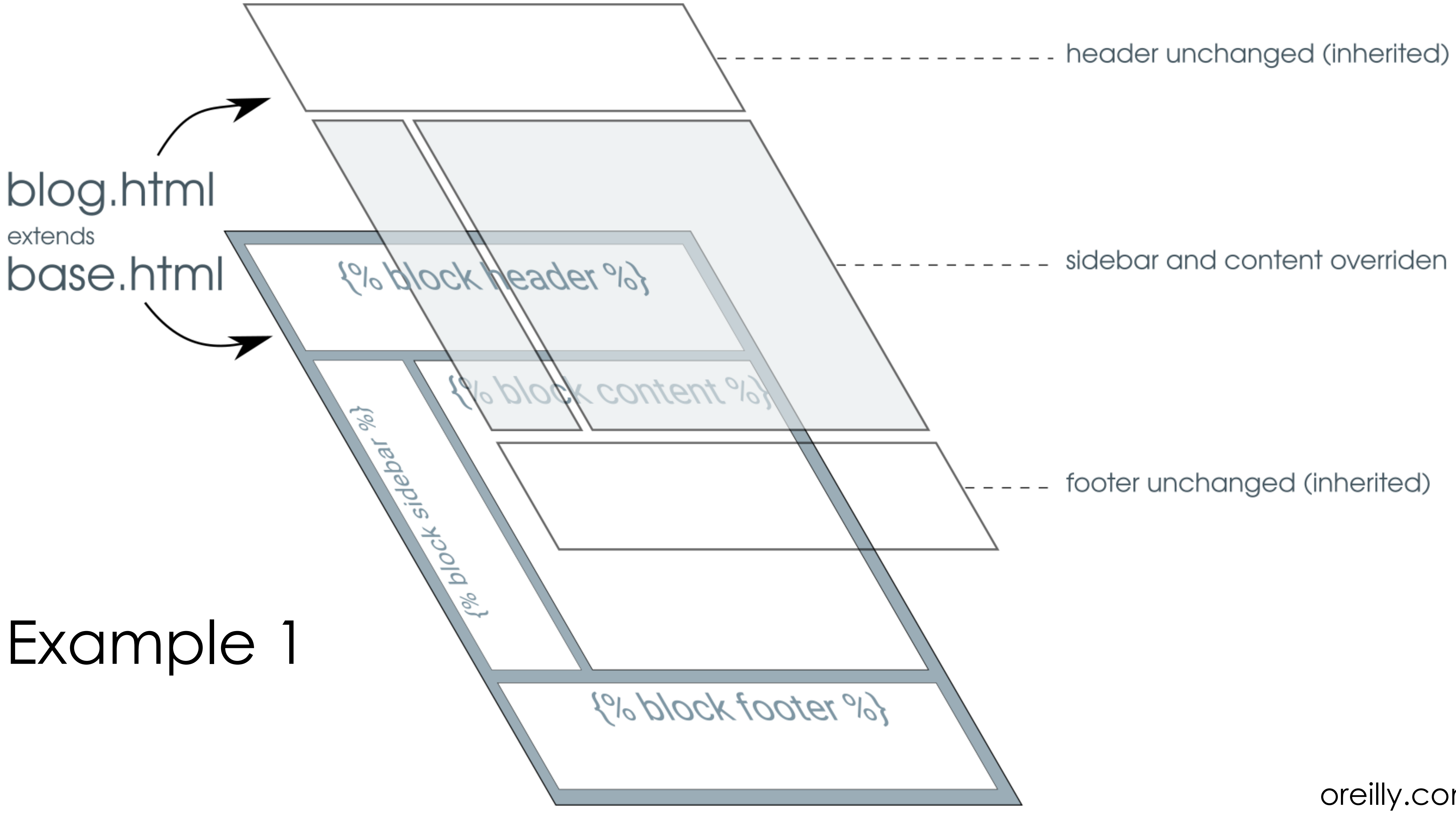
Built-in Tags: now

- Displays the current date and/or time

```
It is {% now "jS F Y H:i" %}
```

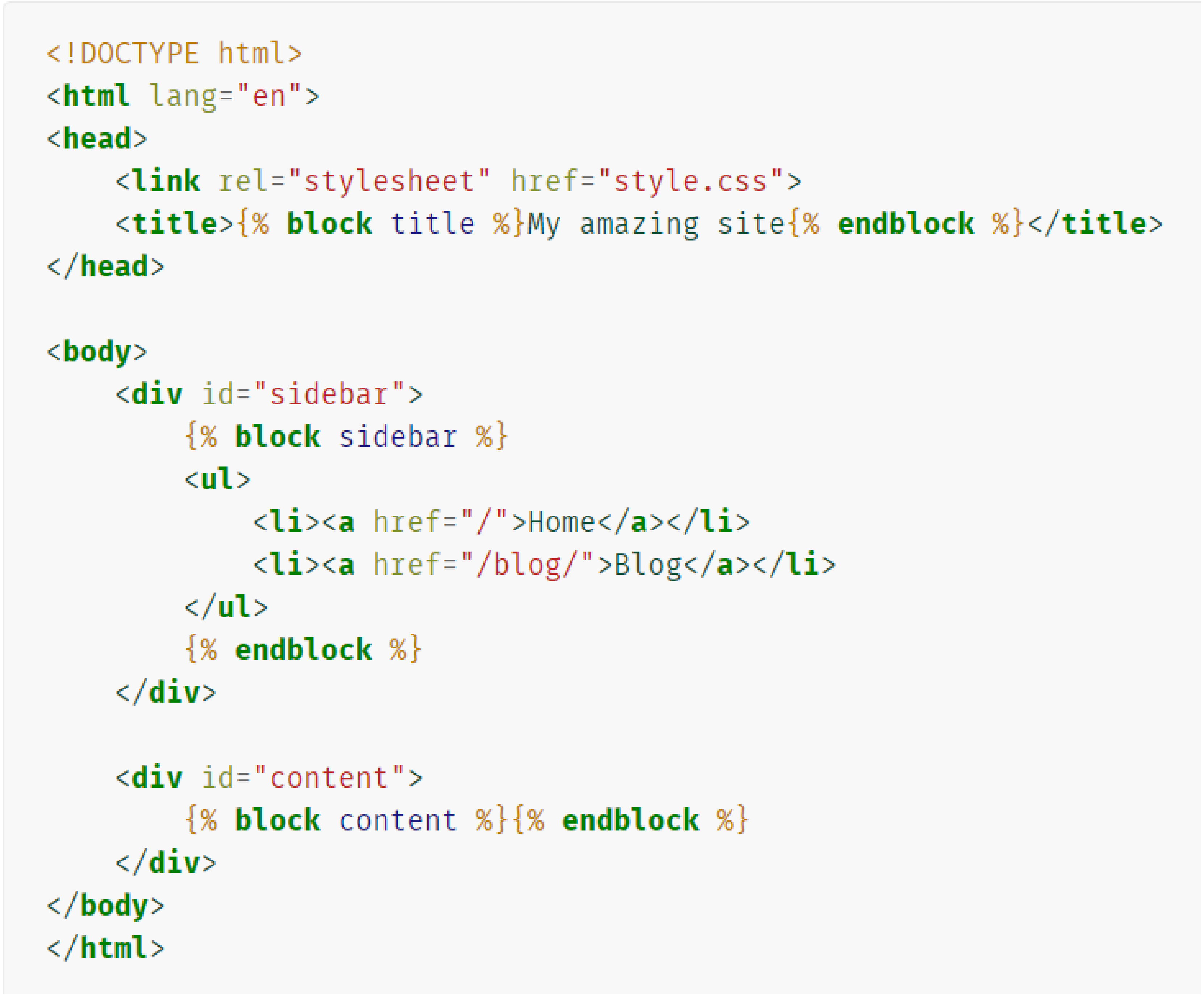
Built-in Tags: block & extends

- Set up template inheritance
- Allows you to build a base “skeleton” template that contains all the common elements of your site and defines blocks that child templates can override.
- You can use as many levels of inheritance as needed


blog.html
extends
base.html
{% block header %}
{% block content %}
{% block sidebar %}
{% block footer %}
header unchanged (inherited)
sidebar and content overriden
footer unchanged (inherited)

Example 1

Example 2



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

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

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

A child template might look like this:

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

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

{% block content %}
{% for entry in blog_entries %}
    <h2>{{ entry.title }}</h2>
    <p>{{ entry.body }}</p>
{% endfor %}
{% endblock %}
```

must be the first template tag

The output might look like:

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

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

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

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

Inheritance Notes

- More {% block %} tags in your base templates are better
- If you need to get the content of the block from the parent template, the {{ block.super }} variable will do the trick
- You can’t define multiple block tags with the same name in the same template

Built-in Tags: include

- Loads a template and renders it with the current context

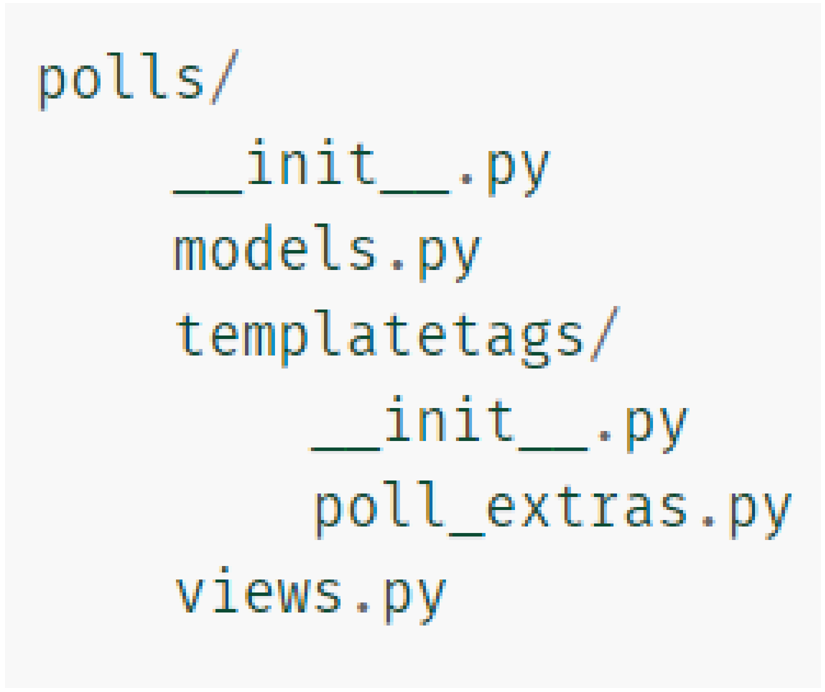
```
{% include "foo/bar.html" %}
```

- You can pass additional context to the template:

```
{% include "name_snippet.html" with person="Jane" greeting="Hello" %}
```

Custom template tags and filters

- The app should contain a templatetags directory
- Don’t forget the __init__.py file
- Don’t forget to add the app to INSTALLED_APPS



```
polls/
    __init__.py
    models.py
    templatetags/
        __init__.py
        poll_extras.py
    views.py
```

Writing custom filters

- Write your filters in ‘poll_extras’ module:

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

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

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

- If the filter does not have argument, keep just ‘value’ in your function

Using custom filters

- And in your template:

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

- And use it as:

```
{{ somevariable|cut:"0" }}
```

Registering custom filters

- You can use register.filter() as a decorator instead:

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

@register.filter
def lower(value):
        return value.lower()
```

Writing custom tags

- Put in ‘poll_extras’ module:

```
import datetime from django
import template

register = template.Library()

@register.simple_tag
def current_time(format_string):
      return datetime.datetime.now().strftime(format_string)
```

References

- https://docs.djangoproject.com/en/3.1/intro/tutorial03/
- https://docs.djangoproject.com/en/3.1/ref/templates/builtins
- https://docs.djangoproject.com/en/3.1/ref/templates/language/
- https://docs.djangoproject.com/en/3.1/howto/custom-template-tags/

Any Question?