Web Development with Django

Instructor: Jorge Mendez

Logistics

- Homework 6 has been graded on Canvas
- Homework 8 is out, due this Friday
- Final project is out, due April 29th (in practice, May 9th)

Overview

- Free, open-source web framework
- Model-template-view architecture
  - Model — database
  - View — endpoint function
  - Template — HTML template
- Designed to make common web dev tasks fast and easy
- Primarily thought for database-driven websites
  - Create and manage databases directly in Python code
- Used in Instagram, YouTube, Spotify…

Installation

- `pip install Django`
- `conda install –c anaconda django`
- Python includes SQLite, supported by Django
- Django also officially supports PostgreSQL, MySQL, and Oracle
  - Necessary for large-scale production websites

Projects and apps

Creating a project

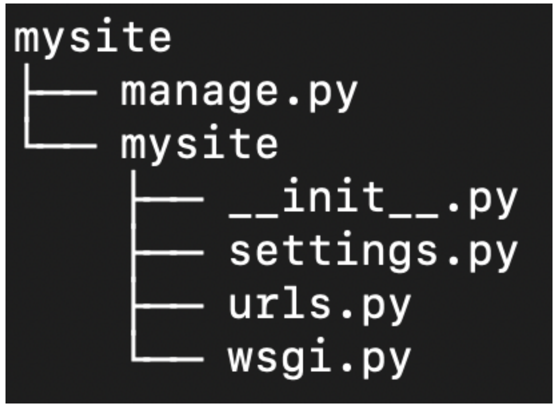
- Command line
  - `django-admin startproject mysite`
- Creates auto-generated code for project setup
- Project: package with database config, Django options, and application settings
- We'll (partially) follow the tutorial from the Django website

Live Example

PY

Creating a project

- Directory structure automatically created
- `mysite/` — outer directory, container for project (name does not matter)
- `manage.py` — command line utility for interacting with Django
- `mysite/` — inner directory, actual Python package (name matters!)
    - `settings.py` — Django project settings
    - `urls.py` — declarations for URLs in your project. Like a table of contents
    - `wsgi.py` — entry point for WSGI servers



```
mysite
├── manage.py
└── mysite
    ├── __init__.py
    ├── settings.py
    ├── urls.py
    └── wsgi.py
```

Running development server

- `python manage.py runserver`
- Runs lightweight development server
  - By default, `DEBUG=True` (in `settings.py`)
  - Provides auto-reloading and error traces
- Django’s server is designed to be used only during development (not production)

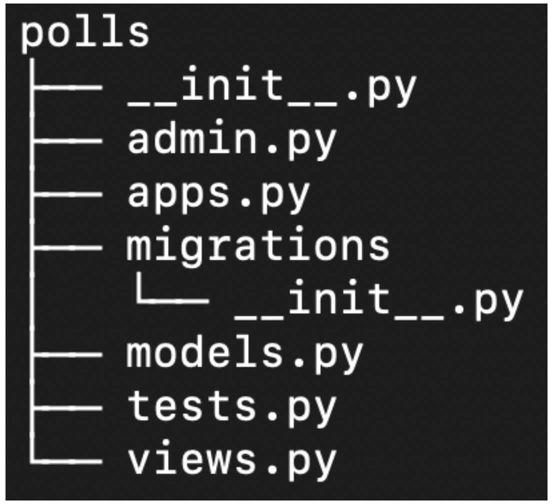
Live Example

Creating an app

- Applications are also Python packages
- What’s the difference between a project and an app?
  - Apps are web apps that do something concrete
  - Projects are a collection of apps and configurations
  - Projects contain multiple apps, and apps can be in various projects
- App path can be anywhere

Example: Creating an app

- Will create it in `mysite/` outer directory
  - This way it can be imported as a top-level package, not a sub-package of `mysite` (inner)
- From `mysite/`: `python manage.py startapp polls`
  - Creates app directory structure



```
polls
├── __init__.py
├── admin.py
├── apps.py
├── migrations
│   └── __init__.py
├── models.py
├── tests.py
└── views.py
```

Live Example

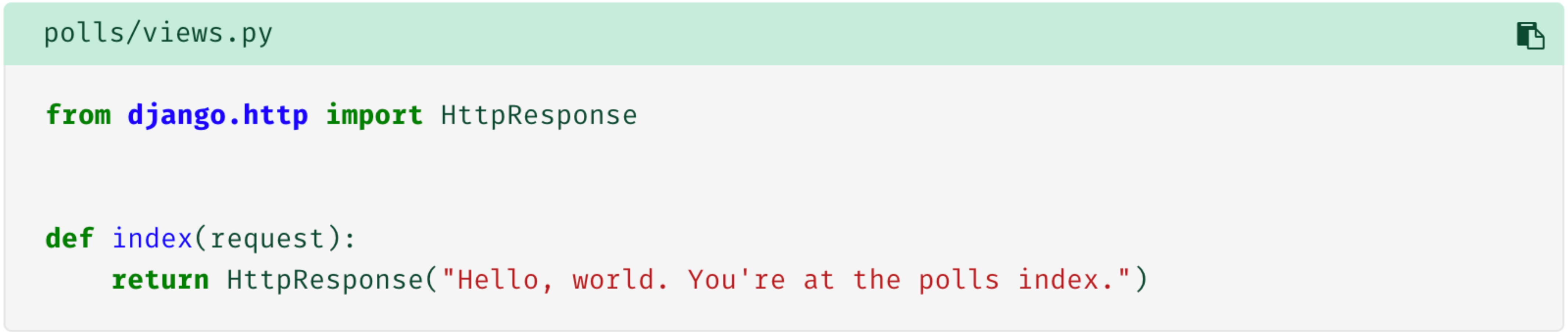
Creating a view

- A view is a function that renders a page
  - Like functions in a Flask app

1. Create function in `views.py` that returns an HTTP response
2. Add a `urls.py` (table of contents) to the app directory
3. Add the URL to `urlpatterns` list in `urls.py`
4. Add the `urls.py` from the app to the project’s `urlpatterns` in `urls.py`

Example: Creating a view

1. Create function in `views.py` that returns an HTTP response

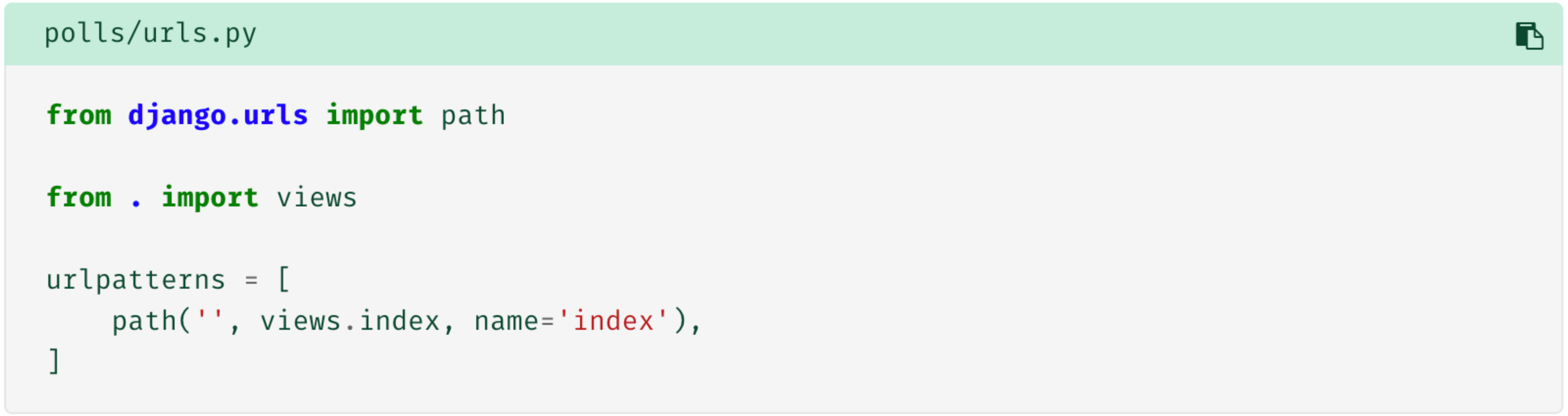


polls/views.py

```python
from django.http import HttpResponse


def index(request):
    return HttpResponse("Hello, world. You're at the polls index.")
```

Example: Creating a view

2. Add a `urls.py` (table of contents) to the app directory
3. Add the URL to `urlpatterns` list in `urls.py`



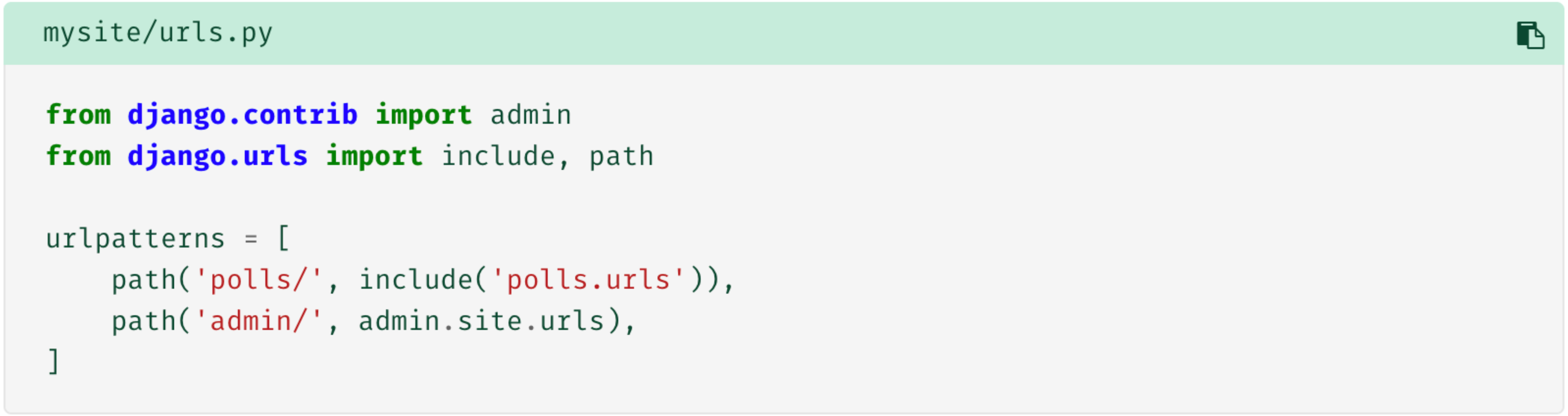
polls/urls.py

```python
from django.urls import path

from . import views

urlpatterns = [
    path('', views.index, name='index'),
]
```

- The `path()` function creates a URL object specifically for `urlpatterns`

Example: Creating a view

4. Add the `urls.py` from the app to the project's `urlpatterns` in `urls.py`



mysite/urls.py

```python
from django.contrib import admin
from django.urls import include, path

urlpatterns = [
    path('polls/', include('polls.urls')),
    path('admin/', admin.site.urls),
]
```

- `include()` references all URLs in another URL config file

The `path()` function

- `path(route, view, name, kwargs)`
  - `route` — string with the URL pattern to match
    - Can include variables like `<varname>` or `<type:varname>` like in Flask
  - `view` — either a function or an `include()`
    - Functions are called with the HTTP request as the first argument and any variable from the URL as keyword arguments
  - `name` — string for reverse matching (must be unique to avoid clashes)
    - Allows us to do `reverse(name)` instead of `reverse(view)`, both of which are valid, to retrieve the URL of a given view.
  - `kwargs` — additional keyword arguments sent to the `view`

Database setup

Default settings

- `settings.py` contains database settings
  - Defaults are for using SQLite, so we'll leave them untouched
- INSTALLED_APPS — list of apps for project
  - `django.contrib.admin` — The admin site
  - `django.contrib.auth` — An authentication system
  - `django.contrib.contenttypes` — A framework for content types
  - `django.contrib.sessions` — A session framework
  - `django.contrib.messages` — A messaging framework
  - `django.contrib.staticfiles` — A framework for managing static files

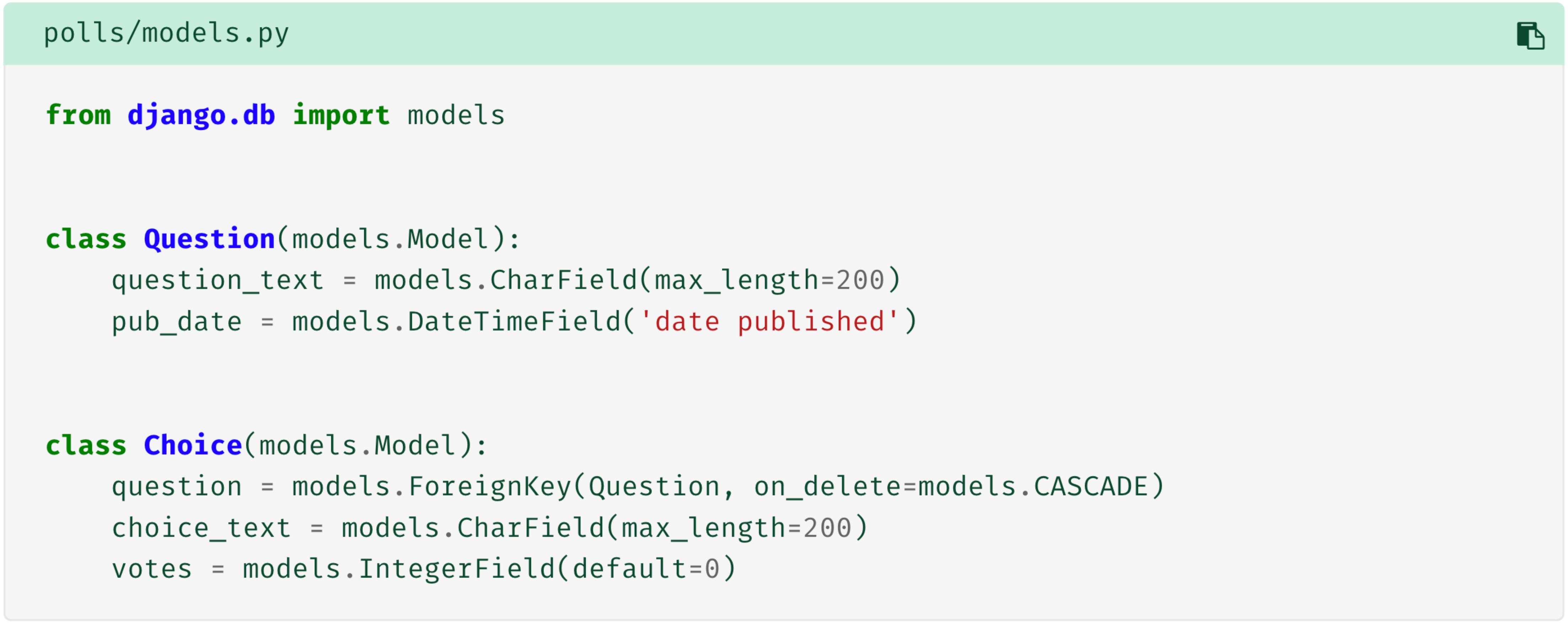
Creating models

- Models are the database layout with some additional metadata
- Each table in the database is represented by a Python class
- Each class variable represents a column in the table
- Classes must subclass `django.db.models.Model`

1. Update models in `models.py`
2. Add the app to the `INSTALLED_APPS` list
3. Run python `manage.py makemigrations app` to track changes
4. Run `python manage.py migrate` to apply changes

Example: Creating models

1. Update models in `models.py`



polls/models.py

```python
from django.db import models


class Question(models.Model):
    question_text = models.CharField(max_length=200)
    pub_date = models.DateTimeField('date published')


class Choice(models.Model):
    question = models.ForeignKey(Question, on_delete=models.CASCADE)
    choice_text = models.CharField(max_length=200)
    votes = models.IntegerField(default=0)
```

Notes on models

- The variable names are used for the column names
- Different tables can be related to others'
  - `ForeignKey` — one-to-many
  - `ManyToManyField` — self-explanatory
  - `OneToOneField` — self-explanatory

2. Add `polls.apps.PollsConfig` to `INSTALLED_APPS` list to include model
   - Defined in `apps.py`

Creating tables

3. `python manage.py makemigrations polls` — include new models
   - Generates *migration*: Python code that is translated into SQLite code for each table
4. `python manage.py migrate` — creates database tables
   - Looks at `INSTALLED_APPS` and the database settings in `settings.py`
   - Runs SQLite code from migrations (i.e., updates database schemas)

- `python manage.py sqlmigrate polls id` — returns SQL code for migration `#id` (to visualize the code)

Live Example

PY

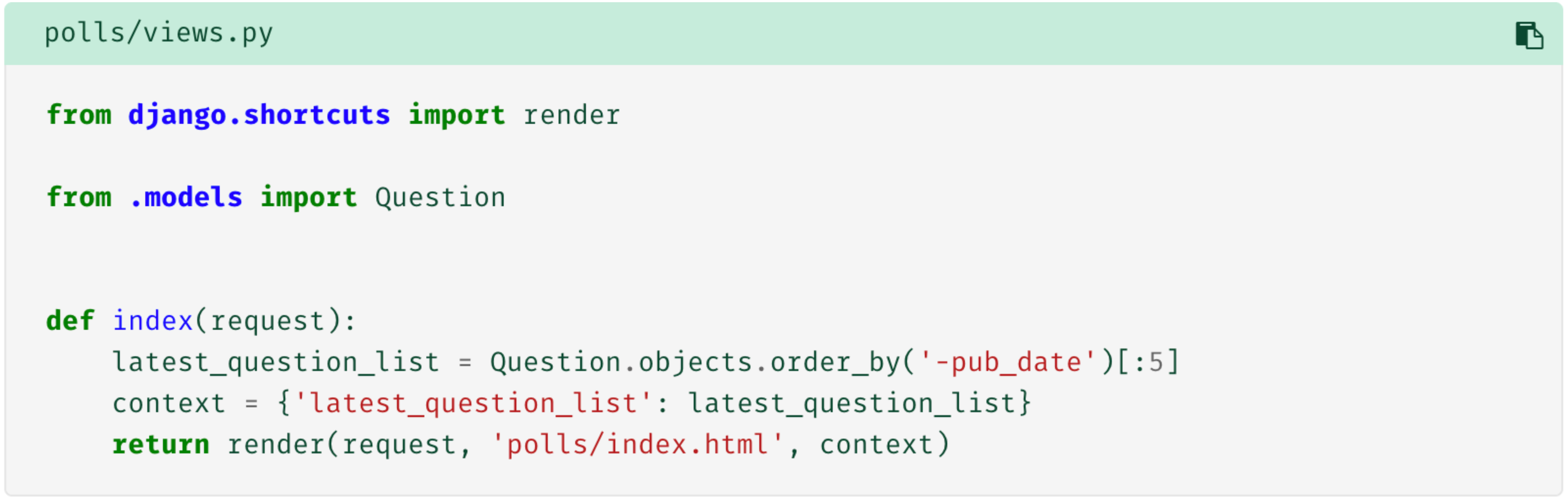
Using the model objects in Python

```
from django.utils import timezone
q = Question(question_text="What's new?", pub_date=timezone.now())
q.save()     # Add to the database
```

- You can add custom methods to your class
- It is important to add `__str__` method

Some more of the polls app

The `index` view

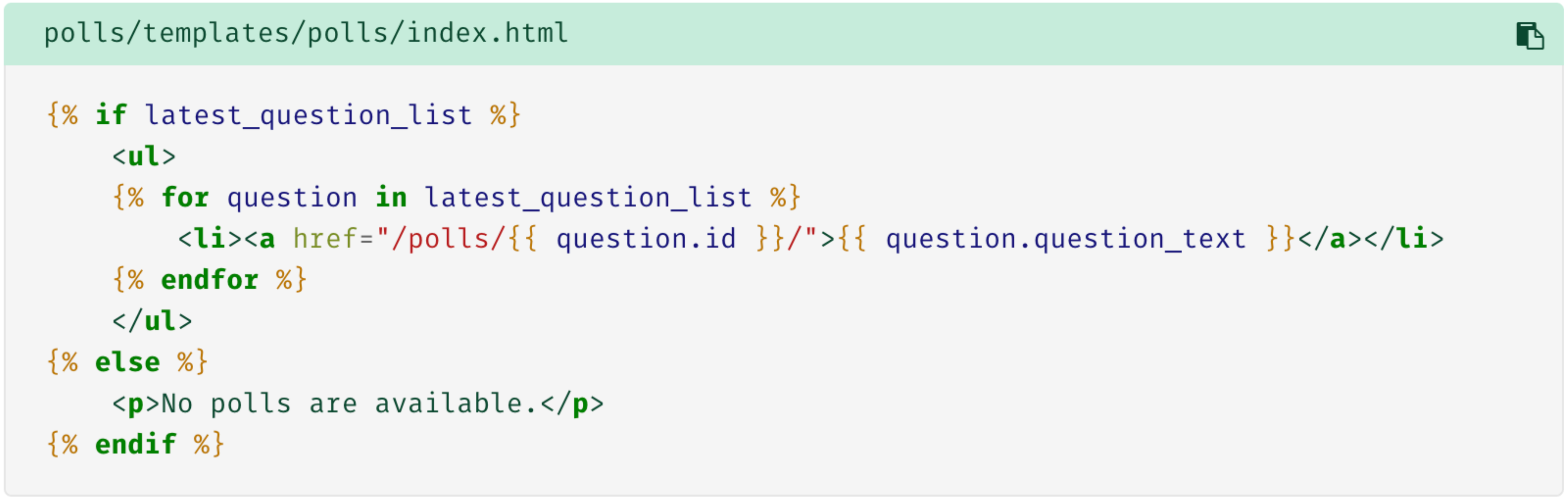


polls/views.py

```python
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)
```

- `render()` returns an HTTP response for displaying the template
- The template can be managed directly by Django or by external template engine (e.g., Jinja2)
- arguments from context are passed into template

The index template



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 %}
```

- This is what a typical template looks like

Django template language

Overview

- A template is a text file that generates other text files
- Goal: use code to generate (parts of) HTML, CSS, or XML files
- We’ll follow the Django documentation for the template language

Variables

1. `{{variable}}` — evaluates variable name and replaces with result

- `foo.bar` — searches for dict element → attribute/method → numeric index
  - If `foo.bar` is callable (e.g., a function), it is called with no argument (`foo.bar()`) and the result is the template value
  - `bar` is treated as a string `‘bar’` and not as a variable (even if a variable `bar` exists)
- If `variable` is not found, it is replaced with `string_if_invalid` (`‘’` by default)

Filters

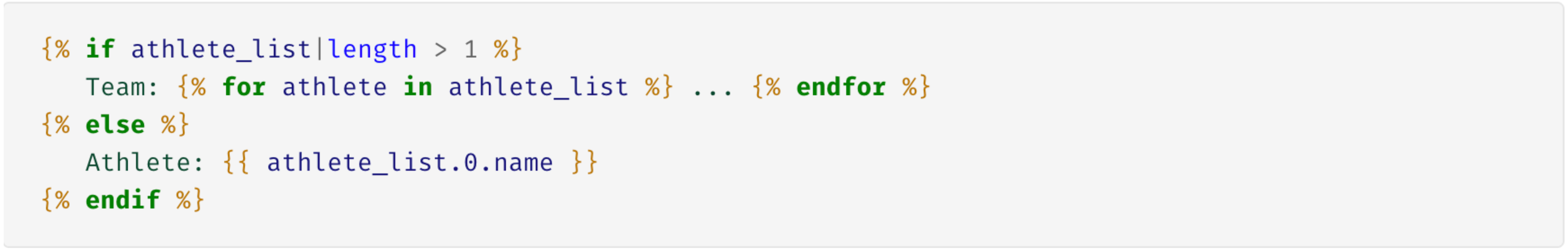
2. `{{var|filter}}` — applies a `filter` to `var`

- `{{name|lower}}` — converts string to lower case
- `{{text|escape|linebreaks}}` — escapes HTML code and changes linebreaks for <p>
- `{{bio|truncatewords:30}}` — display first 30 words of `bio`
- `{{list|join:’, ’}}` — join strings in list, separate by `’, ‘`
- `{{var|default:val}}` — if `var` is missing, replace with `val`
- Django includes about 60 filters

Tags

3. `{% tag %}` or `{% tag %}` … `{% end tag %}` — create text, control flow, load external info…

- `{% for elem in list %} do_stuff {% endfor %}`
- `{% if cond %} do_suff {% elif %} do_other_stuff {% endif %}`



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

Template inheritance

- Most powerful and complex part
- `{% extends %}` — inherit a template
  - Must be the *first* tag in the child template
- `{% block name %} some content {% endblock %}` — replace the parent's content in block `name` with child's content
  - If a block is missing, defaults from parent are used

Template example (parent)

```
<!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>
```

Template example (child)

```
{% 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 %}
```

Template example (result)

```html
<!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>
```

Auto-escaping

- By default, Django automatically escapes the following characters
- `<` is converted to `&lt`
- `>` is converted to `&gt`
- `'` (single quote) is converted to `&#39`
- `"` (double quote) is converted to `&quot`
- `&` is converted to `&amp`

Takeaways

- Django is a complete web framework, as opposed to Flask
- It follows a model-view-template architecture
- Database handling is at the core of Django
- Like in Flask, views handle specific requests
- Django also includes its own template engine
- We barely got into the tutorial…