DJANGO FORMS

06016322 - WEB PROGRAMMING

HTML FORMS

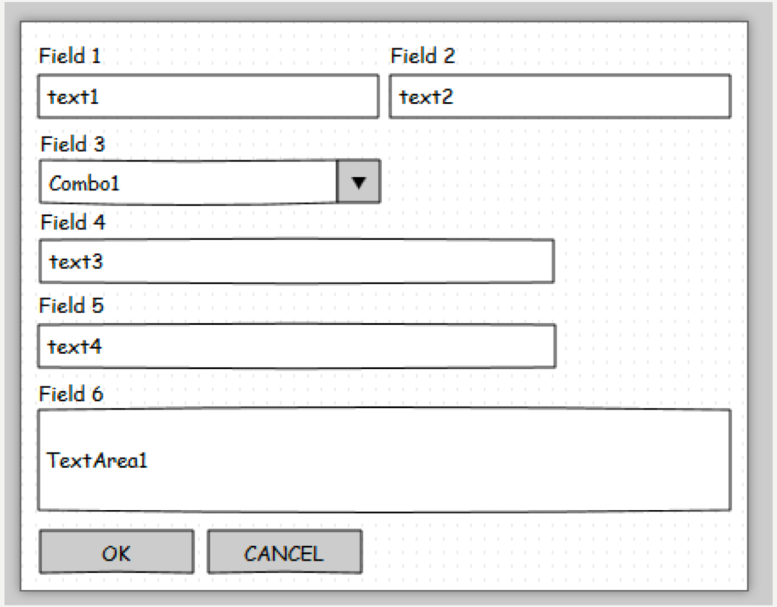
- In HTML, a form is a collection of elements inside <form>...</form> that allow a visitor to do things like enter text, select options, manipulate objects or controls, and so on, and then send that information back to the server.

- As well as its <input> elements, a form must specify two things:
  - *where*: the URL to which the data corresponding to the user's input should be returned
  - *how*: the HTTP method the data should be returned by

- **GET** and **POST** are the only HTTP methods to use when dealing with forms.

DJANGO'S ROLE IN FORMS

- Handling forms is a complex business. You may need to:
  - Render the appropriate HTML inputs
  - Submit form data to server
  - Validation and cleaned up
  - Save to database



DJANGO’S ROLE IN FORMS

- Django’s form functionality can simplify and automate vast portions of this work, and can also do it more securely than most programmers would be able to do in code they wrote themselves.
- Django handles three distinct parts of the work involved in forms:
  - reparing and restructuring data to make it ready for rendering
  - creating HTML forms for the data
  - receiving and processing submitted forms and data from the client

THE DJANGO FORM CLASS

- In much the same way that a Django model describes the logical structure of an object, its behavior, and the way its parts are represented to us, a Form class describes a form and determines how it works and appears.
  - A form class's fields map to HTML form <input>elements.
  - A form's fields are themselves classes; they manage form data and perform validation when a form is submitted.
    - DateField
    - IntegerField
    - FileField
  - A form field is represented to a user in the browser as an HTML "widget"

BUILDING A FORM

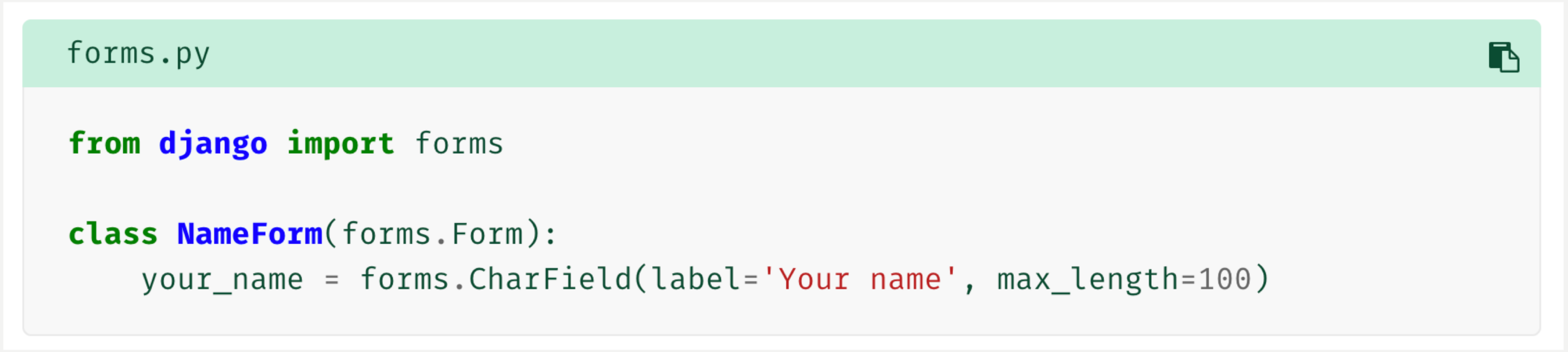
- Suppose you want to create a simple form on your website, in order to obtain the user's name. You'd need something like this in your template:

```html
<form action="/your-name/" method="post">
    <label for="your_name">Your name: </label>
    <input id="your_name" type="text" name="your_name" value="{{ current_name }}">
    <input type="submit" value="OK">
</form>
```

[Source: https://docs.djangoproject.com/en/2.1/topics/forms/](https://docs.djangoproject.com/en/2.1/topics/forms/)

BUILDING A FORM IN DJANGO - THE FORM CLASS

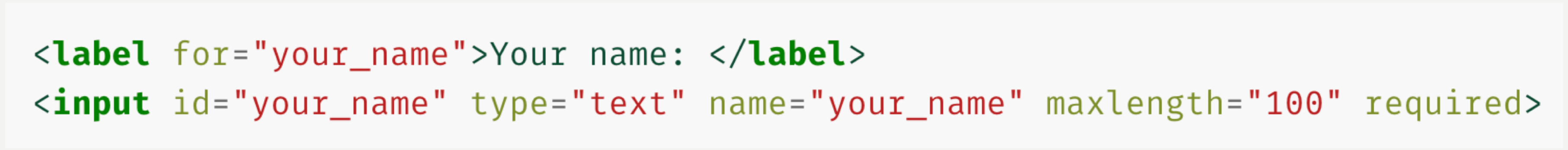
- We already know what we want our HTML form to look like. Our starting point for it in Django is this:



forms.py

```python
from django import forms

class NameForm(forms.Form):
    your_name = forms.CharField(label='Your name', max_length=100)
```



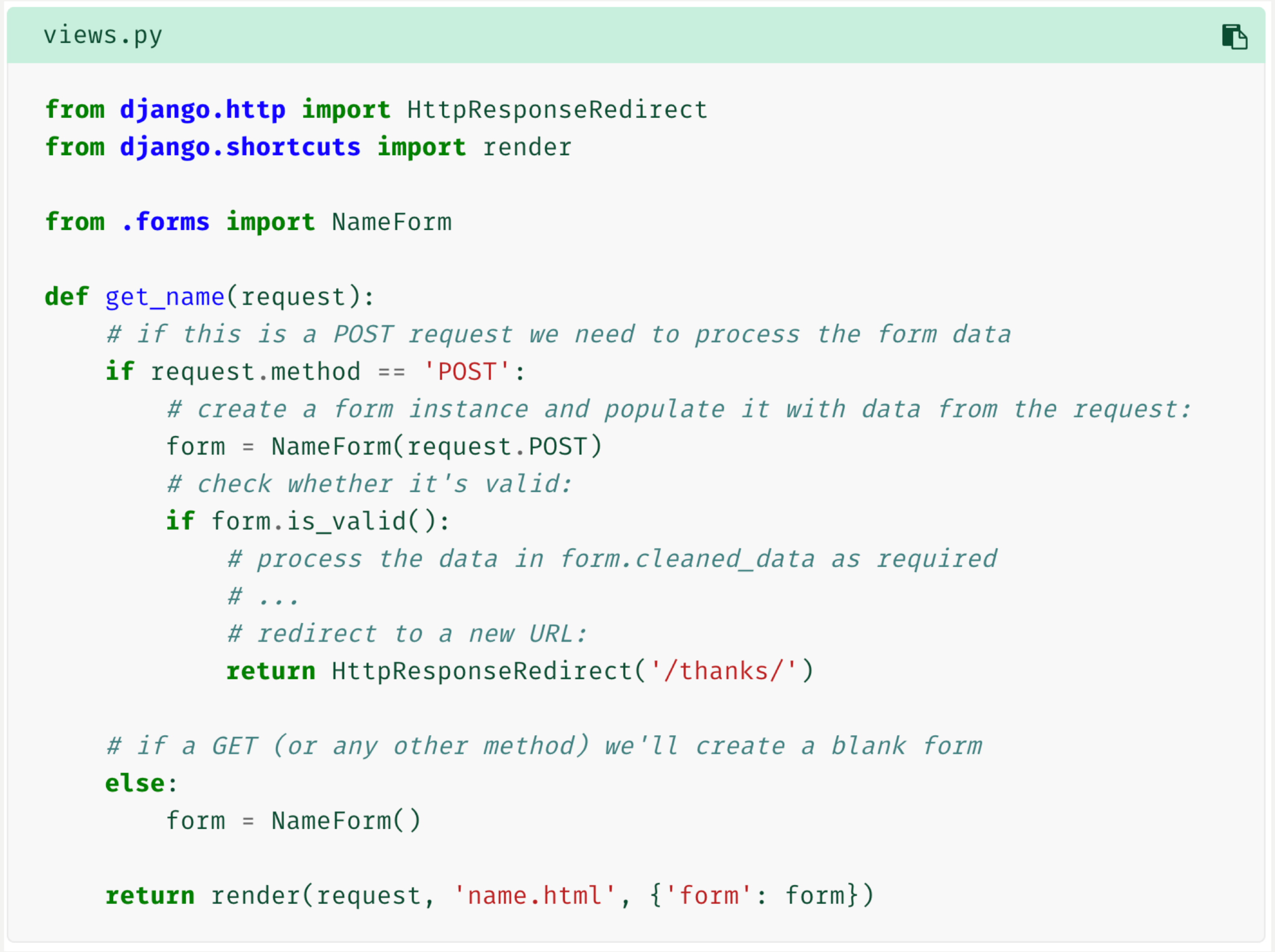
```html
<label for="your_name">Your name: </label>
<input id="your_name" type="text" name="your_name" maxlength="100" required>
```

Note that it does not include the <form> tags, or a submit button.

We'll have to provide those ourselves in the template.

THE VIEW

- Form data sent back to a Django website is processed by a **view**, generally the same view which published the form. This allows us to reuse some of the same logic.



views.py

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

from .forms import NameForm

def get_name(request):
    # if this is a POST request we need to process the form data
    if request.method == 'POST':
        # create a form instance and populate it with data from the request:
        form = NameForm(request.POST)
        # check whether it's valid:
        if form.is_valid():
            # process the data in form.cleaned_data as required
            # ...
            # redirect to a new URL:
            return HttpResponseRedirect('/thanks/')

    # if a GET (or any other method) we'll create a blank form
    else:
        form = NameForm()

    return render(request, 'name.html', {'form': form})
```

THE TEMPLATE

```
<form action="/your-name/" method="post">
    {% csrf_token %}
    {{ form }}
    <input type="submit" value="Submit">
</form>
```

- There are other output options though for the <label>/<input> pairs:
  - **{{ form.as_table }}** will render them as table cells wrapped in <tr> tags
  - **{{ form.as_p }}** will render them wrapped in <p> tags
  - **{{ form.as_ul }}** will render them wrapped in <li> tags
- Note that you'll have to provide the surrounding <table> or <ul> elements yourself.

```
<p><label for="id_subject">Subject:</label>
    <input id="id_subject" type="text" name="subject" maxlength="100" required></p>
<p><label for="id_message">Message:</label>
    <textarea name="message" id="id_message" required></textarea></p>
<p><label for="id_sender">Sender:</label>
    <input type="email" name="sender" id="id_sender" required></p>
<p><label for="id_cc_myself">Cc myself:</label>
    <input type="checkbox" name="cc_myself" id="id_cc_myself"></p>
```

RENDERING FIELDS MANUALLY

```
{{ form.non_field_errors }}
<div class="fieldWrapper">
    {{ form.subject.errors }}
    <label for="{{ form.subject.id_for_label }}">Email subject:</label>
    {{ form.subject }}
</div>
<div class="fieldWrapper">
    {{ form.message.errors }}
    <label for="{{ form.message.id_for_label }}">Your message:</label>
    {{ form.message }}
</div>
<div class="fieldWrapper">
    {{ form.sender.errors }}
    <label for="{{ form.sender.id_for_label }}">Your email address:</label>
    {{ form.sender }}
</div>
<div class="fieldWrapper">
    {{ form.cc_myself.errors }}
    <label for="{{ form.cc_myself.id_for_label }}">CC yourself?</label>
    {{ form.cc_myself }}
</div>
```

FORM VALIDATION

FORM AND FIELD VALIDATION

- Form validation happens when the data is cleaned. If you want to customize this process, there are various places to make changes, each one serving a different purpose.

- In general, any cleaning method can raise **ValidationError** if there is a problem with the data it is processing, passing the relevant information to the **ValidationError** constructor.

```
raise ValidationError(
    _('Invalid value: %(value)s'),
    code='invalid',
    params={'value': '42'},
)
```

- If no ValidationError is raised, the method should return the cleaned (normalized) data as a Python object.

AN EXAMPLE FORM

```
class ContactForm(forms.Form):
    subject = forms.CharField(max_length=100)
    message = forms.CharField()
    sender = forms.EmailField()
    recipients = MultiEmailField()
    cc_myself = forms.BooleanField(required=False)
```

USING VALIDATORS

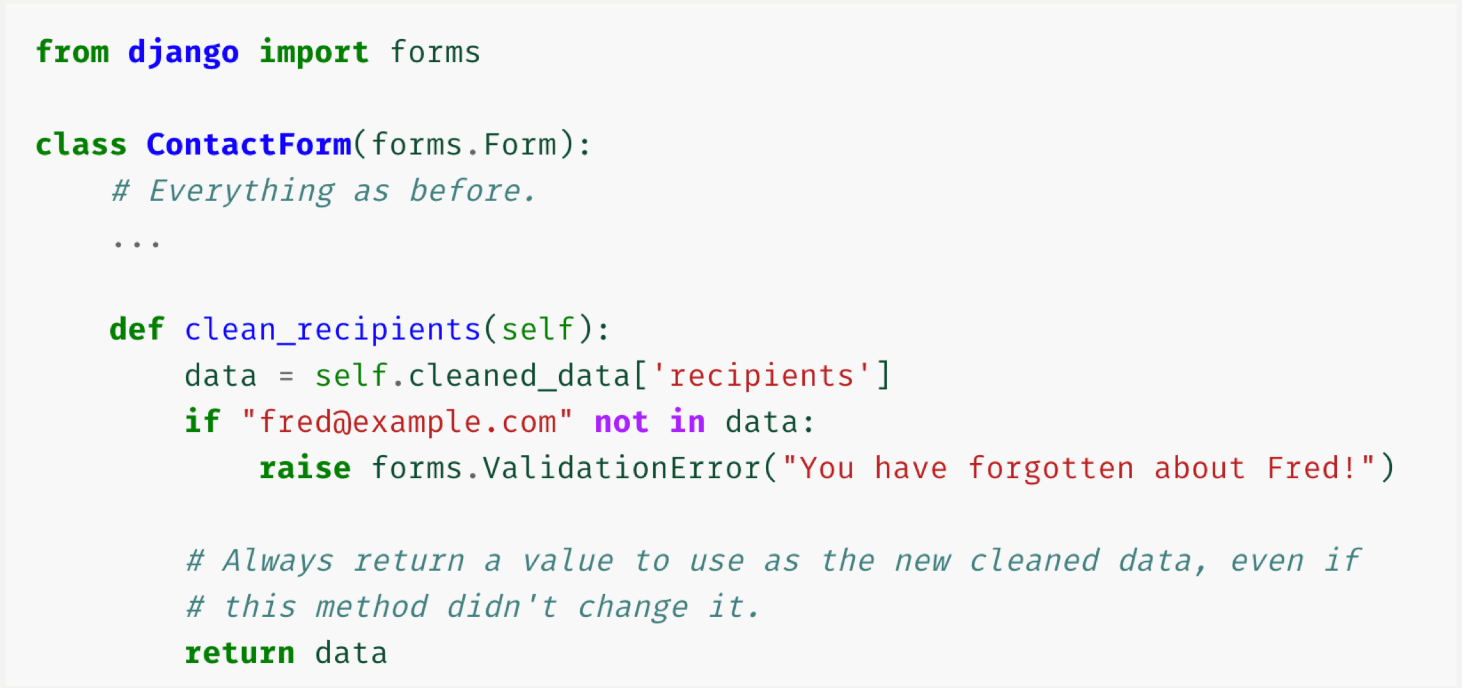
- Django's form (and model) fields support use of simple utility functions and classes known as validators.
- A validator is merely a callable object or function that takes a value and simply returns nothing if the value is valid or raises a **ValidationError** if not.

```
slug = forms.CharField(validators=[validators.validate_slug])
```

- Examples of built-in validators:
  - RegexValidator
  - EmailValidator
  - URLValidator
  - FileExtensionValidator

CLEANING A SPECIFIC FIELD ATTRIBUTE

- Suppose that in our ContactForm, we want to make sure that the recipientsfield always contains the address "fred@example.com".
  - You can do it by writing a cleaning method that operates on the recipients field, like so:



```python
from django import forms

class ContactForm(forms.Form):
    # Everything as before.
    ...

    def clean_recipients(self):
        data = self.cleaned_data['recipients']
        if "fred@example.com" not in data:
            raise forms.ValidationError("You have forgotten about Fred!")

        # Always return a value to use as the new cleaned data, even if
        # this method didn't change it.
        return data
```

CLEANING AND VALIDATING FIELDS THAT DEPEND ON EACH OTHER

- Suppose we add another requirement to our contact form: if the **cc_myself** field is True, the **subject** must contain the word "help".

- The call to super().clean() in the example code ensures that any validation logic in parent classes is maintained.

```python
from django import forms

class ContactForm(forms.Form):
    # Everything as before.
    ...

    def clean(self):
        cleaned_data = super().clean()
        cc_myself = cleaned_data.get("cc_myself")
        subject = cleaned_data.get("subject")

        if cc_myself and subject and "help" not in subject:
            msg = "Must put 'help' in subject when cc'ing yourself."
            self.add_error('cc_myself', msg)
            self.add_error('subject', msg)
```