Django

Documentation

Form and field validation

Form validation happens when the data is cleaned. If you want to customize this process, there are various places you can change, each one serving a different purpose. Three types of cleaning methods are run during form processing. These are normally executed when you call the **`is_valid()`** method on a form. There are other things that can trigger cleaning and validation (accessing the **`errors`** attribute or calling **`full_clean()`** directly), but normally they won't be needed.

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. See below for the best practice in raising **`ValidationError`**. If no **`ValidationError`** is raised, the method should return the cleaned (normalized) data as a Python object.

Most validation can be done using validators - simple helpers that can be reused easily. Validators are simple functions (or callables) that take a single argument and raise **`ValidationError`** on invalid input. Validators are run after the field's **`to_python`** and **`validate`** methods have been called.

Validation of a Form is split into several steps, which can be customized or overridden:

- The **`to_python()`** method on a Field is the first step in every validation. It coerces the value to correct datatype and raises **`ValidationError`** if that is not possible. This method accepts the raw value from the widget and returns the converted value. For example, a FloatField will turn the data into a Python **`float`** or raise a **`ValidationError`**.
- The **`validate()`** method on a Field handles field-specific validation that is not suitable for a validator, It takes a value that has been coerced to correct datatype and raises **`ValidationError`** on any error. This method does not return anything and shouldn't alter the value. You should override it to handle validation logic that you can't or don't want to put in a validator.
- The **`run_validators()`** method on a Field runs all of the field's validators and aggregates all the errors into a single **`ValidationError`**. You shouldn't need to override this method.
- The **`clean()`** method on a Field subclass. This is responsible for running **`to_python`**, **`validate`** and **`run_validators`** in the correct order and propagating their errors. If, at any time, any of the methods raise **`ValidationError`**, the validation stops and that error is raised. This method returns the clean data, which is then inserted into the **`cleaned_data`** dictionary of the form.
- The **`clean_<fieldname>()`** method in a form subclass – where **`<fieldname>`** is replaced with the name of the form field attribute. This method does any cleaning that is specific to that particular attribute, unrelated to the type of field that it is. This method is not passed any parameters. You will need to look up the value of the field in **`self.cleaned_data`** and remember that it will be a Python object at this point, not the original string submitted in the form (it will be in **`cleaned_data`** because the general field **`clean()`** method, above, has already cleaned the data once).

  For example, if you wanted to validate that the contents of a **`CharField`** called **`serialnumber`** was unique, **`clean_serialnumber()`** would be the right place to do this. You don't need a specific field (it's just a **`CharField`**), but you want a formfield-specific piece of validation and, possibly, cleaning/normalizing the data.

  This method should return the cleaned value obtained from cleaned_data, regardless of whether it changed anything or not.
- The Form subclass's **`clean()`** method. This method can perform any validation that requires access to multiple fields from the form at once. This is where you might put in things to check that if field **`A`** is supplied, field **`B`** must contain a valid email address and the like. This method can return a completely different dictionary if it wishes, which will be used as the **`cleaned_data`**.

Since the field validation methods have been run by the time **clean()** is called, you also have access to the form's **errors** attribute which contains all the errors raised by cleaning of individual fields.

Note that any errors raised by your **Form.clean()** override will not be associated with any field in particular. They go into a special "field" (called **__all__**), which you can access via the **non_field_errors()** method if you need to. If you want to attach errors to a specific field in the form, you need to call **add_error()**.

Also note that there are special considerations when overriding the **clean()** method of a **ModelForm** subclass. (see the ModelForm documentation for more information)

These methods are run in the order given above, one field at a time. That is, for each field in the form (in the order they are declared in the form definition), the **Field.clean()** method (or its override) is run, then **clean_<fieldname>()**. Finally, once those two methods are run for every field, the **Form.clean()** method, or its override, is executed whether or not the previous methods have raised errors.

Examples of each of these methods are provided below.

As mentioned, any of these methods can raise a **ValidationError**. For any field, if the **Field.clean()** method raises a **ValidationError**, any field-specific cleaning method is not called. However, the cleaning methods for all remaining fields are still executed.

Raising `ValidationError`

Changed in Django 1.6.

In order to make error messages flexible and easy to override, consider the following guidelines:

- Provide a descriptive error **code** to the constructor:

```
# Good
ValidationError(_('Invalid value'), code='invalid')

# Bad
ValidationError(_('Invalid value'))
```

- Don't coerce variables into the message; use placeholders and the **params** argument of the constructor:

```
# Good
ValidationError(
    _('Invalid value: %(value)s'),
    params={'value': '42'},
)

# Bad
ValidationError(_('Invalid value: %s') % value)
```

- Use mapping keys instead of positional formatting. This enables putting the variables in any order or omitting them altogether when rewriting the message:

```
# Good
ValidationError(
    _('Invalid value: %(value)s'),
    params={'value': '42'},
)

# Bad
ValidationError(
    _('Invalid value: %s'),
    params=('42',),
)
```

- Wrap the message with **gettext** to enable translation:

```
# Good
ValidationError(_('Invalid value'))

# Bad
ValidationError('Invalid value')
```

Putting it all together:

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

Following these guidelines is particularly necessary if you write reusable forms, form fields, and model fields.

While not recommended, if you are at the end of the validation chain (i.e. your form **clean()** method) and you know you will *never* need to override your error message you can still opt for the less verbose:

```
ValidationError(_('Invalid value: %s') % value)
```

New in Django 1.7.

The **Form.errors.as_data()** and **Form.errors.as_json()** methods greatly benefit from fully featured **ValidationError**s (with a **code** name and a **params** dictionary).

Raising multiple errors

If you detect multiple errors during a cleaning method and wish to signal all of them to the form submitter, it is possible to pass a list of errors to the **ValidationError** constructor.

As above, it is recommended to pass a list of **`ValidationError`** instances with **`code`**s and **`params`** but a list of strings will also work:

```
# Good
raise ValidationError([
    ValidationError(_('Error 1'), code='error1'),
    ValidationError(_('Error 2'), code='error2'),
])

# Bad
raise ValidationError([
    _('Error 1'),
    _('Error 2'),
])
```

Using validation in practice

The previous sections explained how validation works in general for forms. Since it can sometimes be easier to put things into place by seeing each feature in use, here are a series of small examples that use each of the previous features.

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. These can be passed to a field's constructor, via the field's **`validators`** argument, or defined on the **`Field`** class itself with the **`default_validators`** attribute.

Simple validators can be used to validate values inside the field, let's have a look at Django's **`SlugField`**:

```
from django.forms import CharField
from django.core import validators

class SlugField(CharField):
    default_validators = [validators.validate_slug]
```

As you can see, **`SlugField`** is just a **`CharField`** with a customized validator that validates that submitted text obeys to some character rules. This can also be done on field definition so:

```
slug = forms.SlugField()
```

is equivalent to:

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

Common cases such as validating against an email or a regular expression can be handled using existing validator classes available in Django. For example, **`validators.validate_slug`** is an instance of a **`RegexValidator`** constructed with the first argument being the pattern: **`^[-a-zA-Z0-9_]+$`**. See the section on writing validators to see a list of what is already available and for an example of how to write a validator.

Form field default cleaning

Let's firstly create a custom form field that validates its input is a string containing comma-separated email addresses. The full class looks like this:

```
from django import forms
from django.core.validators import validate_email

class MultiEmailField(forms.Field):
    def to_python(self, value):
        "Normalize data to a list of strings."

        # Return an empty list if no input was given.
        if not value:
            return []
        return value.split(',')

    def validate(self, value):
        "Check if value consists only of valid emails."

        # Use the parent's handling of required fields, etc.
        super(MultiEmailField, self).validate(value)

        for email in value:
            validate_email(email)
```

Every form that uses this field will have these methods run before anything else can be done with the field's data. This is cleaning that is specific to this type of field, regardless of how it is subsequently used.

Let's create a simple **`ContactForm`** to demonstrate how you'd use this field:

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

Simply use **`MultiEmailField`** like any other form field. When the **`is_valid()`** method is called on the form, the **`MultiEmailField.clean()`** method will be run as part of the cleaning process and it will, in turn, call the custom **`to_python()`** and **`validate()`** methods.

Cleaning a specific field attribute

Continuing on from the previous example, suppose that in our **`ContactForm`**, we want to make sure that the **`recipients`** field always contains the address **`"fred@example.com"`**. This is validation that is specific to our form, so we don't want to put it into the general **`MultiEmailField`** class. Instead, we write a cleaning method that operates on the **`recipients`** field, like so:

```
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 the cleaned data, whether you have changed it or
        # not.
        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"`**. We are performing validation on more than one field at a time, so the form's **`clean()`** method is a good spot to do this. Notice that we are talking about the **`clean()`** method on the form here, whereas earlier we were writing a **`clean()`** method on a field. It's important to keep the field and form difference clear when working out where to validate things. Fields are single data points, forms are a collection of fields.

By the time the form's **`clean()`** method is called, all the individual field clean methods will have been run (the previous two sections), so **`self.cleaned_data`** will be populated with any data that has survived so far. So you also need to remember to allow for the fact that the fields you are wanting to validate might not have survived the initial individual field checks.

There are two ways to report any errors from this step. Probably the most common method is to display the error at the top of the form. To create such an error, you can raise a **`ValidationError`** from the **`clean()`** method. For example:

```
from django import forms

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

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

        if cc_myself and subject:
            # Only do something if both fields are valid so far.
            if "help" not in subject:
                raise forms.ValidationError("Did not send for 'help' in "
                        "the subject despite CC'ing yourself.")
```

Changed in Django 1.7:

In previous versions of Django, **`form.clean()`** was required to return a dictionary of **`cleaned_data`**. This method may still return a dictionary of data to be used, but it's no longer required.

In this code, if the validation error is raised, the form will display an error message at the top of the form (normally) describing the problem.

Note that the call to **`super(ContactForm, self).clean()`** in the example code ensures that any validation logic in parent classes is maintained.

The second approach might involve assigning the error message to one of the fields. In this case, let's assign an error message to both the "subject" and "cc_myself" rows in the form display. Be careful when doing this in practice, since it can lead to confusing form output. We're showing what is possible here and leaving it up to you and your designers to work out what works effectively in your particular situation. Our new code (replacing the previous sample) looks like this:

```
from django import forms

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

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

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

The second argument of **`add_error()`** can be a simple string, or preferably an instance of **`ValidationError`**. See Raising ValidationError for more details. Note that **`add_error()`** automatically removes the field from **`cleaned_data`**.

Widgets **Middleware**

Django Software Foundation

Code of Conduct

Get Involved

Join a Group

Contribute to Django

Submit a Bug

Report a Security Issue

Follow Us

GitHub

Twitter

News RSS

Django Users Mailing List