Django form processing: Initialization, field access, validation and error handling.

Because Django form processing can have a great deal of variations, I'll start the discussion from [the same form and view method explained in the previous section](https://www.webforefront.com/django/setupdjangoforms.html#formstructure), now consolidated in listing 6-8.

Listing 6-8 Django form class with backing processing view method

from django import forms

from django.shortcuts import render

class ContactForm(forms.Form):

name = forms.CharField(required=False)

email = forms.EmailField(label='Your email')

comment = forms.CharField(widget=forms.Textarea)

def contact(request):

if request.method == 'POST':

# POST, generate form with data from the request

form = ContactForm(request.POST)

# Reference is now a bound instance with user data sent in POST

# process data, insert into DB, generate email, redirect to a new URL,etc

else:

# GET, generate blank form

form = ContactForm()

# Reference is now an unbound (empty) form

# Reference form instance (bound/unbound) is sent to template for rendering

return render(request,'about/contact.html',{'form':form})

Initialize forms : initial for fields & forms, \_\_init\_\_ method, label\_suffix, auto\_id, field\_order and use\_required\_attribute

When users request a page backed by a Django form they're sent an empty form (a.k.a. *unbound* form) -- represented by the GET section andContactForm() instance in [listing 6-8](https://www.webforefront.com/django/formprocessing.html#listing-6-8). Although the form is always empty from a user data perspective, the form can contain data from as part of its initialization sequence (e.g. a pre-filled form with a user's email or name).

To perform this kind of Django form initialization you have three options. The first technique consists of initializing the form with a dictionary of values via the initial argument directly on the form instance declared in the view method. Listing 6-9 shows this technique.

Listing 6-9 Django form instance with initial argument declared in view method

def contact(request):

....

....

else:

# GET, generate blank form

form = ContactForm(initial={'email':'johndoe@coffeehouse.com','name':'John Doe'})

# Form is now initialized for first presentation to display these values

# Reference form instance (bound/unbound) is sent to template for rendering

return render(request,'about/contact.html',{'form':form})

The ContactForm() now uses initial={'email':'johndoe@coffeehouse.com','name':'John Doe'} to generate a pre-filled form instance with values for the email and name fields so end users don't have to go to the trouble of typing these values from scratch themselves.

The technique in listing 6-9 is intended for one-time or instance specific initialization values. If you want to constantly provide the same value for a given form field, a more suitable technique is to use the same initial argument directly on a form field, as show in listing 6-10.

Listing 6-10 Django form fields with initial argument

from django import forms

class ContactForm(forms.Form):

name = forms.CharField(required=False,initial='Please provide your name')

email = forms.EmailField(label='Your email', initial='We need your email')

comment = forms.CharField(widget=forms.Textarea)

def contact(request):

....

....

else:

# GET, generate blank form

form = ContactForm()

# Form is now initialized for first presentation and is filled with initial values in form definition

# Reference form instance (bound/unbound) is sent to template for rendering

return render(request,'about/contact.html',{'form':form})

Notice in listing 6-10 how it's now the form fields in the form class equipped with the initial argument. It's worth mentioning that if you use the initial argument on both the form instance and form fields, form instance values take precedence over any form field (e.g. if you combine the statements from [listing 6-9](https://www.webforefront.com/django/formprocessing.html#listing-6-9) and [listing 6-10](https://www.webforefront.com/django/formprocessing.html#listing-6-10), the form will always be pre-filled with email='johndoe@coffeehouse.com' and name='John Doe' from [listing 6-9](https://www.webforefront.com/django/formprocessing.html#listing-6-9)).

One of the drawbacks of using the initial argument in form fields like [listing 6-10](https://www.webforefront.com/django/formprocessing.html#listing-6-10) is that values can't change dynamically at run-time (i.e. you can't personalize the initial values based on who calls the form like [listing 6-9](https://www.webforefront.com/django/formprocessing.html#listing-6-9)). To solve this and providing the utmost flexibility for the most complex form initialization scenarios is a third technique using the \_\_init\_\_ method of a form class illustrated in listing 6-11.

Listing 6-11. Django form initialized with \_\_init\_\_ method

from django import forms

class ContactForm(forms.Form):

name = forms.CharField(required=False)

email = forms.EmailField(label='Your email')

comment = forms.CharField(widget=forms.Textarea)

def \_\_init\_\_(self, \*args, \*\*kwargs):

# Get 'initial' argument if any

initial\_arguments = kwargs.get('initial', None)

updated\_initial = {}

if initial\_arguments:

# We have initial arguments, fetch 'user' placeholder variable if any

user = initial\_arguments.get('user',None)

# Now update the form's initial values if user

if user:

updated\_initial['name'] = getattr(user, 'first\_name', None)

updated\_initial['email'] = getattr(user, 'email', None)

# You can also initialize form fields with hardcoded values

# or perform complex DB logic here to then perform initialization

updated\_initial['comment'] = 'Please provide a comment'

# Finally update the kwargs initial reference

kwargs.update(initial=updated\_initial)

super(ContactForm, self).\_\_init\_\_(\*args, \*\*kwargs)

def contact(request):

....

....

else:

# GET, generate blank form

form = ContactForm(initial={'user':request.user,'otherstuff':'otherstuff'})

# Form is now initialized via the form's \_\_init\_\_ method

# Reference form instance (bound/unbound) is sent to template for rendering

return render(request,'about/contact.html',{'form':form})

The first important aspect in listing 6-11 is how the form is initialized in the view method, notice it uses the same initial argument but the dictionary values are now {'user':request.user,'otherstuff':'otherstuff'}. Do these values look strange ? The form doesn't even have fields named user or otherstuff, so what's happening ?

These last values are perfectly valid and in other circumstances would be ignored because the Django form does indeed have no fields by these names, but since we'll be manipulating the guts of form initialization process in the \_\_init\_\_ method, we can access these placeholder values for indirect initialization purposes. More importantly, using these placeholder values illustrates how it's possible to use context data or unrelated form data to initialize Django form fields.

Next, let's turn our attention to the Django form's \_\_init\_\_ method, which is invoked when you create a form instance. The \_\_init\_\_ method's arguments \*args, \*\*kwargs are standard Python syntax -- if you've never seen this last syntax check out the appendix on [Python basics: Methods: Default, optional, \*args & \*\*kwargs arguments](https://www.webforefront.com/django/pythonbasics-methods.html).

The first step in \_\_init\_\_ checks for an initial value and creates a reference to hold the new values to initialize the form. If an initial value is present, a check is made for a user value to use these values for the form's actual name and email fields. Next, irrespective of any passed in values, a direct assignment is made on the form's comment field.

Finally, the form's initial reference is updated with a new set of values that reflect the form's actual fields, leading to the form's initialization using data outside the context of a form (e.g. request data, database query,etc). As a last step in the \_\_init\_\_ method, a call is made to the super() method so that the base/parent class initialization process takes place.

Always use the initial argument or \_\_init\_\_ method to populate forms with initialization data and keep them unbound

It's important to note all the previous initialization techniques keep a form unbound which is a term used to describe form instances that haven't been populated with user data. The term bound is reserved for when a form's field values are populated with user data. This subtle difference between bound and unbound is really important when you enter the [validation phase of Django forms](https://www.webforefront.com/django/formprocessing.html#validation) described in the next section.

This also means the syntax ContactForm(initial={'email':'johndoe@coffeehouse.com','name':'John Doe'}) is not equivalent to ContactForm({'email':'johndoe@coffeehouse.com','name':'John Doe'}). The first variation uses the initial argument to create an unbound form instance, while the second variation creates a bound form instance by passing the values directly without any argument.

In addition to initializing the first set of data loaded on a Django form, there are four other initialization options for Django forms that influence a form's layout in templates: label\_suffix, auto\_id, field\_order and use\_required\_attribute.

When you generate the output for a Django form in a template -- a topic that's described in detail later in this chapter in ['Set up the layout for Django forms in templates'](https://www.webforefront.com/django/formtemplatelayout.html) -- form fields are generally accompanied by what's called a field label, which is another name for a human-friendly descriptor. For example, if you have fields called name and email, their default labels are Your name and Your email, respectively. To separate field labels from a field's HTML markup (e.g. <input type="text">) Django defines a label suffix, which defaults to : (the colon symbol) to produce output with the pattern <field\_label>:<input type="<field\_type>">. Through the label\_suffix you can define a custom label suffix symbol to separate every form field from its label. So for example, the ContactForm(label\_suffix='...') syntax outputs every form field label separated by ... (e.g. Your email...<input type="text">).

**Tip** Individual form fields can also use the [label\_suffix attribute](https://www.webforefront.com/django/formfieldtypesandvalidation.html#fieldlayout), see the [Django form field types section](https://www.webforefront.com/django/formfieldtypesandvalidation.html). If label\_suffix is declared in both a form field and form initialization, the former takes precedence.

Another initialization option for Django forms is auto\_id which automatically generates an id and label for every form field. By default, a Django form is always set to auto\_id=True so you'll always get auto generated HTML ids and labels when outputting a form with form.as\_table(), as illustrated in the first part of listing 6-12:

Listing 6-12 Django form with automatic ids (default auto\_id=True option) and no automatic ids auto\_id=False option

<!-- Option 1, default auto\_id=True -->

<tr>

<th><label for="id\_name">Name:</label></th>

<td><input id="id\_name" name="name" type="text" /></td>

</tr>

<tr>

<th><label for="id\_email">Your email:</label></th>

<td><input id="id\_email" name="email" type="email" /></td>

</tr>

<tr>

<th><label for="id\_comment">Comment:</label></th>

<td><textarea cols="40" id="id\_comment" name="comment" rows="10"></textarea></td>

</tr>

<!-- Option 2 auto\_id=False -->

<tr>

<th>Name:</th>

<td><input name="name" type="text" /></td>

</tr>

<tr>

<th>Your email:</th>

<td><input name="email" type="email" /></td>

</tr>

<tr>

<th>Comment:</th>

<td><textarea cols="40" name="comment" rows="10">\r\n</textarea></td>

</tr>

Notice how the field labels in the top output of listing 6-12 are wrapped around <label for="id\_field\_name"> <label> and the fields HTML tags includes the id="id\_field\_name" attribute. In most cases this is a desirable output as it allows fields to be easily referenced for purposes of attaching JavaScript events or CSS classes. However, for other circumstances auto\_id=True can produce very verbose output and inclusively conflicting HTML tags (e.g. if you have two form instances of the same type on the same page, there will be two identical ids).

To turn off the auto generation of ids and labels you can initialize a form with the auto\_id=False option. For example, the ContactForm(auto\_id=False)syntax generates the output presented in the second half of listing 6-12.

Another initialization option for unbound form instances that influences a form's layout is the field\_order option. By default, form fields are output in the order they're declared, so the form definition in [listing 6-10](https://www.webforefront.com/django/formprocessing.html#listing-6-10) follows the output order: name, email, comment. You can override this default field output by using the field\_order option which accepts a list of field names with the desired output order. The field\_order option can be declared as part of the initialization process or inclusively as if it were a form field.

For example, the ContactForm(field\_order=['email','comment','name']) syntax ensures the email field is output first, followed by comment and name. It's worth mentioning the field\_order option can accept an incomplete field list, such as ContactForm(field\_order=['email']) which outputs theemail field followed by the remaining form fields in their declared order, in this case name and then comment. If you're constantly setting field\_orderto initialize a form's field order, a quicker solution is to set a default field\_order as part of the form itself:

class ContactForm(forms.Form):

name = forms.CharField(required=False)

email = forms.EmailField(label='Your email')

comment = forms.CharField(widget=forms.Textarea)

field\_order = ['email','comment','name'] # Sets order email,comment,name

If you declare field\_order as part of the form itself and on initialization, the initialization field\_order takes precedence. The upcoming section ['Set up the layout for Django forms in templates'](https://www.webforefront.com/django/formtemplatelayout.html) contains more details on the practical use of field\_order in templates.

Finally, the use\_required\_attribute option allows you to set the overall use of the HTML 5 required attribute. By default use\_required\_attribute=True, which means all required form fields are output with the HTML 5 required attribute, ensuring browsers enforces these form fields are always provided. You can disable the use of this HTML 5 client-side validation required attribute by initializing a form with use\_required\_attribute=False. Note that setting use\_required\_attribute=False does not influence the Django server-side validation of a form field (e.g. if a form field is required, Django server-side validation still catches fields that aren't provided, irrespective of the use\_required\_attribute option).

Accessing form values: request.POST and cleaned\_data.

Once a user fills out a Django form, the form is sent back to the server to be processed and perform an action with the user data (e.g. create an order, send an email, save the data to the database) -- a step which is represented in [the POST section in listing 6-8](https://www.webforefront.com/django/formprocessing.html#listing-6-8).

One of the key advantages of Django form processing is you can use the request.POST variable to create a bound form instance. But although therequest.POST variable is the initial access point to populate a Django form with user data, you shouldn't use this data beyond initializing a form instance, as the data in request.POST is too raw for direct access.

For example, in request.POST you still don't know if the user provided data is valid. In addition, the data in request.POST is still treated as strings, so if your Django form happens to have an IntegerField() or DateField() it still needs to be converted manually to the expected data type (e.g. '6' to 6 integer, '2017-01-01' to 2017-01-01 datetime), which is just unnecessary work that another another part of Django forms deals with.

Once you have a bound form generated with the request.POST variable, you can then access each of the form's field values through the cleaned\_datadictionary. For example, if the bound form has a form field called name you can use the syntax form.cleaned\_data['name'] to access the user provided name value. More importantly, if the form field is an IntegerField() named age the syntax form.cleaned\_data['age'] produces an integer value, a formatting behavior that also applies to other form fields with non-string data types (e.g. DateField()).

**Caution** You can't access cleaned\_data until is\_valid() is called on the form.

By design, it isn't possible to access a form instance's cleaned\_data dictionary unless you first call the is\_valid() method. If you try to access cleaned\_data before calling is\_valid() you'll get the error AttributeError: 'form\_reference' object has no attribute 'cleaned\_data'.

If you think about this for a second it's good practice, after all, why would you want to access data that hasn't been validated ? The next section describes theis\_valid() method.

Validating form values : is\_valid(), validators, clean\_<field>() and clean()

The is\_valid() method is one of the more important parts of Django form processing. Once you create a bound form with request.POST, you call the is\_valid() method on the instance to determine if the included values comply with a form's field definitions (e.g. if an EmailField() value is a valid email). Although the is\_valid() method returns a boolean True or False value, it has two important side-effects:

* Calling is\_valid() also creates the cleaned\_data dictionary on the form instance to hold the form field values that passed validation rules.
* Calling is\_valid() also creates the errors dictionary on the form instance to hold the form errors for each of the fields that didn't pass the validation rules.

Listing 6-13 illustrates a modified version of [listing 6-8](https://www.webforefront.com/django/formprocessing.html#listing-6-8) with the is\_valid() method.

Listing 6-13 Django form is\_valid() method for form processing

from django.http import HttpResponseRedirect

def contact(request):

if request.method == 'POST':

# POST, generate form with data from the request

form = ContactForm(request.POST)

# Reference is now a bound instance with user data sent in POST

# Call is\_valid() to validate data and create cleaned\_data and errors dict

if form.is\_valid():

# Form data is valid, you can now access validated values in the cleaned\_data dict

# e.g. form.cleaned\_data['email']

# process data, insert into DB, generate email

# Redirect to a new URL

return HttpResponseRedirect('/about/contact/thankyou')

else:

pass # Not needed

# is\_valid() method created errors dict, so form reference now contains errors

# this form reference drops to the last return statement where errors

# can then be presented accessing form.errors in a template

else:

# GET, generate blank form

form = ContactForm()

# Reference is now an unbound (empty) form

# Reference form instance (bound/unbound) is sent to template for rendering

return render(request,'about/contact.html',{'form':form})

Notice in listing 6-13 that right after a bound form instance is created a call is made to the is\_valid() method. If all the form field values comply against the form field data types, we enter a conditional where it's possible to access the form values through the cleaned\_data dictionary, perform whatever business logic is necessary and relinquish control to another page, which in listing 6-13 is to perform redirect.

If any of the form field values fails to pass a rule, then is\_valid() returns False and in the process creates an errors dictionary with details about the values that failed to pass their rules. Because of this last automatic creation of errors, all that's needed after is\_valid() returns False is to return the same form instance in order to display the errors dictionary to an end user so he can correct his mistakes.

But as important as the is\_valid() method is to Django form processing, its validation is just done against a form field's data type. For example, is\_valid() can validate if a value is left empty, if a value matches a given number range or even if a value is a valid date, in essence anything supported by Django form field types.

But what if you want to perform more sophisticated validation after is\_valid() ? Like checking a value against a database before deeming it valid or checking two values against one another (e.g. a provided zip code value against a provided city value). While you can add these validation checks directly after the is\_valid() call, Django offers three more efficient ways to enforce advanced rules by adding them to the form field or form class definition.

If you want a reusable validation mechanism you can use across multiple Django form fields, the best choice is a validator assigned through a form field'svalidators option. The validators form field option expects a list of methods designed to raise a forms.ValidationError error in case a value doesn't comply with expected rules. Listing 6-14 illustrates a Django form with one of its fields using a custom validator method via the validators option.

Listing 6-14 Django form field validators option with custom validator method for form processing

from django import forms

import re

def validate\_comment\_word\_count(value):

count = len(value.split())

if count < 30:

raise forms.ValidationError(('Please provide at least a 30 word message,

%(count)s words is not descriptive enough'), params={'count': count},)

class ContactForm(forms.Form):

name = forms.CharField(required=False)

email = forms.EmailField(label='Your email')

comment = forms.CharField(widget=forms.Textarea,validators=[validate\_comment\_word\_count])

The first section in listing 6-14 shows the custom validate\_command\_word\_count()method, which (rudimentarly) checks if message has at least thirty words. If the method's input is not at least thirty words, Django's forms.ValidationError error is raised to indicate a rule violation.

In the bottom half of listing 6-14 you can see a modified ContactForm where the comment field uses the validators=[validate\_csv] option. This tells Django that after is\_valid() is run and all the form fields have been checked for errors against their data types, it should also run the validate\_comment\_word\_count validator method against the value provided for the comment field. If the comment value does not comply with this rule, then a ValidatioError error is raised which is added to the form errors dictionary -- the same one described in the past section which is used to check field values against their data types.

As you can see from the example in listing 6-14, you can equally reuse the custom validate\_comment\_word\_count() method on any other form field in the same form or in another Django form through the validators option. In addition, you can also apply multiple validators to a field since the validatorsoption accepts a list of validators. Finally, it's worth mentioning the django.core.validators package contains a series of validators you can also reuse[[2]](https://www.webforefront.com/django/formprocessing.html" \l "footnote-2)and which are used behind by the scenes by certain form field data types.

In addition to the form field validators option, it's also possible to add validation form rules through the clean\_<field>() and clean() methods which are created as part of a Django form class -- just like \_\_init\_\_() described earlier. Just like methods specified in the form field validators option,clean\_<field>() and clean() methods are automatically invoked when the is\_valid() method is run. Listing 6-15 illustrates the use of twoclean\_<field>() methods.

Listing 6-15 Django form field validation with clean\_<field>() methods

from django import forms

class ContactForm(forms.Form):

name = forms.CharField(required=False)

email = forms.EmailField(label='Your email')

comment = forms.CharField(widget=forms.Textarea)

def clean\_name(self):

# Get the field value from cleaned\_data dict

value = self.cleaned\_data['name']

# Check if the value is all upper case

if value.isupper():

# Value is all upper case, raise an error

raise forms.ValidationError("""Please don't use all upper

case for your name, use lower case""",code='uppercase')

# Always return value

return value

def clean\_email(self):

# Get the field value from cleaned\_data dict

value = self.cleaned\_data['email']

# Check if the value end in @hotmail.com

if value.endswith('@hotmail.com'):

# Value ends in @hotmail.com, raise an error

raise forms.ValidationError("""Please don't use a hotmail email,

we simply don't like it""",code='hotmail')

# Always return value

return value

In listing 6-15 there are two clean\_<field>() methods to add validation rules for the name and email fields. Django automatically searches for form methods prefixed with clean\_ and attempts to match a form's field names to the remaining name, to enforce validation on the field in question. This means you can have as many clean\_<field>() methods as form fields.

The logic inside each clean\_<field>() method follows a similar pattern to validators methods. First you extract a field's value from the form's cleaned\_data dictionary via the self reference which represents the form instance. Next, you run whatever rule or logic you want against the field value, if you deem the value doesn't comply you raise a forms.ValidationError which adds the error to the form instance. Finally, and this is only different tovalidators methods, you must return the field value irrespective of raising an error or changing its value.

Sometimes it's necessary to apply a rule that doesn't necessarily belong to a specific field, in which case the generic clean() method is the preferred approach vs. a clean\_<field>() method. Listing 6-16 illustrates the use of the clean() method.

Listing 6-16. Django form field validation with clean() method

from django import forms

class ContactForm(forms.Form):

name = forms.CharField(required=False)

email = forms.EmailField(label='Your email')

comment = forms.CharField(widget=forms.Textarea)

def clean(self):

# Call clean() method to ensure base class validation

super(ContactForm, self).clean()

# Get the field values from cleaned\_data dict

name = self.cleaned\_data.get('name','')

email = self.cleaned\_data.get('email','')

# Check if the name is part of the email

if name.lower() not in email:

# Name is not in email , raise an error

raise forms.ValidationError("Please provide an email that contains your name, or viceversa")

In listing 6-16 you can see a similar approach to the previous clean\_<field>() methods in [listing 6-15](https://www.webforefront.com/django/formprocessing.html#listing-6-15). But because the clean() method is a class wide method and you're overriding it yourself, it differs from clean\_<field>() methods in that you must first explicitly call the clean() method of the base/parent form class (i.e. super(...).clean()) to ensure the base class validation is applied. Form field value extraction is also done through the cleaned\_data data dictionary, as is the validation logic and raising of forms.ValidationError to indicate a rule violation. Finally, the clean() method differs from clean\_<field>() methods in that it doesn't return a value.

Functionally the clean() method is different because it's called after all the methods in the validators options and clean\_<field>() methods, a behavior that's important because all these methods rely on data in the cleaned\_data dictionary. This means if a validators or clean\_<field>()method raises a ValidationError error it won't return any value and the cleaned\_data dictionary won't contain a value for this field. So by the time theclean() method is run, the cleaned\_data dictionary may not necessarily have all the form field values if one was short-circuited in a validators orclean\_<field>() method, which is the reason why the clean() method uses the safer dictionary access syntax cleaned\_data.get('<field>','') to assign a default value in case the cleaned\_data dictionary doesn't have a given field.

Another important behavioral difference between the clean() method, clean\_<field>() & validators methods is how they treat forms.ValidationError. When a forms.ValidationError is raised in a validators or clean\_<field>() method, the error is assigned to the <field>in question in the form's errors dictionary -- which is important for display purposes. But when a forms.ValidationError is raised in the clean()method, the error is assigned to a special placeholder field named \_\_all\_\_ -- also known as "non-field errors" -- which is also placed in the form's errorsdictionary.

If you want to assign an error in the clean() method to a specific form field you can use the add\_error() method as illustrated in listing 6-17.

Listing 6-17. Django form field error assignment with add\_error() in clean() method

def clean(self):

# Call clean() method to ensure base class validation

super(ContactForm, self).clean()

# Get the field values from cleaned\_data dict

name = self.cleaned\_data.get('name','')

# Check if the name is part of the email

if name.lower() not in email:

# Name is not in email , raise an error

message = "Please provide an email that contains your name, or viceversa"

self.add\_error('name', message)

self.add\_error('email', forms.ValidationError(message))

self.add\_error(None, message)

Notice how listing 6-17 uses the add\_error() method on the form instance instead of the raise forms.ValidationError() syntax. The add\_error()method accepts two arguments, the first one the form field name on which to assign the error and the second can be either an error message string or an instance of the ValidationError class.

The first two add\_error() methods in listing 6-17 assign the error to the name and email fields, respectively. And the third add\_error() method with the None key assigns the error to the \_\_all\_\_ placeholder making it equivalent to raise forms.ValidationError() from [listing 6-16](https://www.webforefront.com/django/formprocessing.html#listing-6-16).

Error form values: errors

Form errors as its been described in the previous sections are automatically added to a form instance in the errors dictionary after calling the is\_valid()method. Inclusively, it's possible to access the errors dictionary directly without calling the is\_valid() method, unlike the cleaned\_data dictionary.

The errors dictionary is important because all form errors end up on it. Whether the errors are raised because a value doesn't comply with a form field data type or with raise forms.ValidationError() in the clean() method, clean\_<field>() methods, validators methods or the add\_error() method in the clean() method, all errors end up in a form's errors dictionary.

The errors dictionary follows the pattern {'<field\_name>':'<error\_message>'} which makes it easy to identify form errors in either a view method or in a template for display purposes. The only exception to this last pattern occurs when a form error isn't field specific (e.g. such as those created in the clean() method) in which case a form error is assigned to a special key named \_\_all\_\_, errors which are also called non-field errors.

Although you can access the errors dictionary as any other Python dictionary, Django provides a series of methods described in table 6-1 to make working with errors much easier:

Table 6-1 Django form errors methods

|  |  |
| --- | --- |
| **Method** | **Description** |
| form.errors | Gives you access to the raw errors dictionary. |
| form.errors.as\_data() | Outputs a dictionary with the original ValidationError instances. For example, if errors outputs {'email':['This field is required']}, then errors.as\_data() outputs {'email':[ValidationError(['This field is required'])]}. |
| form.errors.as\_json(escape\_html=False) | Outputs a JSON structure with the contents of the errors dictionary. For example, if errors outputs {'email':['This field is required']}, then errors.as\_json() outputs {'email':[{'message':'This field is required','code':'required'}]}. Note that by default as\_json() does not escape its output, if you want errors to be escaped use the escape\_html flag (e.g. as\_json(escape\_html=True)). |
| form.add\_error(field,message) | Associates an error message to a given form field. Although typically used in the clean() method, it can be used in a view method if necessary. Note that if field is not specified the error message goes on to form part of the \_\_all\_\_ placeholder key in errors which are deemed non-field errors. |
| form.has\_error(field, code=None) | Returns a True or False value if a given field has an error. Note that by default has\_error returns True if any error type is associated with a field. To perform the evaluation against a particular error type you can use the code keyword (e.g. form.has\_error('email',code='required')). To check if a form has non-field errors you can use NON\_FIELD\_ERRORS as the field value. |
| form.non\_field\_errors() | Returns a list of non-form errors associated with a form (i.e. the \_\_all\_\_ placeholder key). These errors are typically created in the clean() clean method via ValidationError or add\_error(None,'message'). |

You may have noticed the ValidationError class instances created in all the previous examples use different arguments, meaning there are multiple ways to create ValidationError instances. For example, some ValidationError instances use a simple string, but it's also possible to create aValidationError instance with a list of ValidationError instances, as well as specify a code attribute to further classify an error type. Listing 6-18 illustrates a series of ValidationError class instances using these variations.

Listing 6-18.- Django form ValidationError instance creation

from django import forms

# Placed inside def clean\_email(self):

raise forms.ValidationError("""Please don't use a hotmail email,

we simply don't like it""",code='hotmail')

# Placed inside def clean(self):

raise forms.ValidationError([

forms.ValidationError(""""Please provide an email that matches

your name, or viceversa""",code='custom'),

forms.ValidationError("""Please provide your professional email,

%(value)s doesn't look professional """,code='required',

params={'value':self.cleaned\_data.get('email') })