Formsets - - - -

# class BaseFormSet

A formset is a layer of abstraction to work with multiple forms on the same page. It can be best compared

to a data grid. Let’s say you have the following form:

>>> from django import forms

>>> class ArticleForm(forms.Form):

... title = forms.CharField()

... pub\_date = forms.DateField()

...

You might want to allow the user to create several articles at once. To create a formset out of an ArticleForm

you would do:

>>> from django.forms import formset\_factory

>>> ArticleFormSet = formset\_factory(ArticleForm)

You now have created a formset class named ArticleFormSet. Instantiating the formset gives you the ability

to iterate over the forms in the formset and display them as you would with a regular form:

>>> formset = ArticleFormSet()

>>> for form in formset:

... print(form)

...

<div><label for="id\_form-0-title">Title:</label><input type="text" name="form-0-title" id="id\_form-

,→0-title"></div>

<div><label for="id\_form-0-pub\_date">Pub date:</label><input type="text" name="form-0-pub\_date" id=

,→"id\_form-0-pub\_date"></div>

As you can see it only displayed one empty form. The number of empty forms that is displayed is controlled

by the extra parameter. By default, formset\_factory() defines one extra form; the following example will

create a formset class to display two blank forms:

>>> ArticleFormSet = formset\_factory(ArticleForm, extra=2)

Iterating over a formset will render the forms in the order they were created. You can change this order by

providing an alternate implementation for the \_\_iter\_\_() method.

**Formsets** can also be indexed into, which returns the corresponding form. If you override \_\_iter\_\_, you will

need to also override \_\_getitem\_\_ to have matching behavior.

# Using initial data with a formset

Initial data is what drives the main usability of a formset. As shown above you can define the number of

extra forms. What this means is that you are telling the formset how many additional forms to show in

addition to the number of forms it generates from the initial data. Let’s take a look at an example:

>>> import datetime

>>> from django.forms import formset\_factory

>>> from myapp.forms import ArticleForm

>>> ArticleFormSet = formset\_factory(ArticleForm, extra=2)

>>> formset = ArticleFormSet(

... initial=[

... {

... "title": "Django is now open source",

... "pub\_date": datetime.date.today(),

... }

... ]

... )

>>> for form in formset:

... print(form)

...

<div><label for="id\_form-0-title">Title:</label><input type="text" name="form-0-title" value=

,→"Django is now open source" id="id\_form-0-title"></div>

<div><label for="id\_form-0-pub\_date">Pub date:</label><input type="text" name="form-0-pub\_date"␣

,→value="2023-02-11" id="id\_form-0-pub\_date"></div>

<div><label for="id\_form-1-title">Title:</label><input type="text" name="form-1-title" id="id\_form-

,→1-title"></div>

<div><label for="id\_form-1-pub\_date">Pub date:</label><input type="text" name="form-1-pub\_date" id=

,→"id\_form-1-pub\_date"></div>

<div><label for="id\_form-2-title">Title:</label><input type="text" name="form-2-title" id="id\_form-

,→2-title"></div>

<div><label for="id\_form-2-pub\_date">Pub date:</label><input type="text" name="form-2-pub\_date" id=

,→"id\_form-2-pub\_date"></div>

There are now a total of three forms showing above. One for the initial data that was passed in and two

extra forms. Also note that we are passing in a list of dictionaries as the initial data.

If you use an initial for displaying a **formset**, you should pass the same initial when processing that

formset’s submission so that the **formset** can detect which forms were changed by the user. For example,

you might have something like: ArticleFormSet(request.POST, initial=[...]).

# Limiting the maximum number of forms

The **max\_num** parameter to **formset\_factory()** gives you the ability to limit the number of forms the **formset**

will display:

>>> from django.forms import formset\_factory

>>> from myapp.forms import ArticleForm

>>> ArticleFormSet = formset\_factory(ArticleForm, extra=2, max\_num=1)

>>> formset = ArticleFormSet()

>>> for form in formset:

... print(form)

...

<div><label for="id\_form-0-title">Title:</label><input type="text" name="form-0-title" id="id\_form-

,→0-title"></div>

<div><label for="id\_form-0-pub\_date">Pub date:</label><input type="text" name="form-0-pub\_date" id=

,→"id\_form-0-pub\_date"></div>

If the value of **max\_num** is greater than the number of existing items in the initial data, up to extra additional

blank forms will be added to the **formset**, so long as the total number of forms does not exceed **max\_num**. For

example, if **extra=2** and **max\_num=2** and the **formset** is initialized with one initial item, a form for the initial

item and one blank form will be displayed.

If the number of items in the initial data exceeds **max\_num**, all initial data forms will be displayed regardless

of the value of **max\_num** and no extra forms will be displayed. For example, if **extra=3** and **max\_num=1** and the **formset** is initialized with two initial items, two forms with the initial data will be displayed.

A **max\_num** value of None (the default) puts a high limit on the number of forms displayed (1000). In practice this is equivalent to no limit.

By default, **max\_num** only affects how many forms are displayed and does not affect validation. If

**validate\_max=True** is passed to the **formset\_factory(),** then **max\_num** will affect validation.

# Limiting the maximum number of instantiated forms

The absolute\_max parameter to formset\_factory() allows limiting the number of forms that can be instantiated when supplying POST data. This protects against memory exhaustion attacks using forged POST

requests:

>>> from django.forms.formsets import formset\_factory

>>> from myapp.forms import ArticleForm

>>> ArticleFormSet = formset\_factory(ArticleForm, absolute\_max=1500)

>>> data = {

... "form-TOTAL\_FORMS": "1501",... "form-INITIAL\_FORMS": "0",

... }

>>> formset = ArticleFormSet(data)

>>> len(formset.forms)

1500

>>> formset.is\_valid()

False

>>> formset.non\_form\_errors()

['Please submit at most 1000 forms.']

When **absolute\_max** is None, it defaults to **max\_num + 1000**. (If **max\_num** is None, it defaults to **2000**).

If **absolute\_max** is less than **max\_num**, a **ValueError** will be raised.

# Formset validation

Validation with a formset is almost identical to a regular Form. There is an is\_valid method on the formset

to provide a convenient way to validate all forms in the formset:

>>> from django.forms import formset\_factory

>>> from myapp.forms import ArticleForm

>>> ArticleFormSet = formset\_factory(ArticleForm)

>>> data = {

... "form-TOTAL\_FORMS": "1",

... "form-INITIAL\_FORMS": "0",

... }

>>> formset = ArticleFormSet(data)

>>> formset.is\_valid()

True

We passed in no data to the formset which is resulting in a valid form. The formset is smart enough to ignore

extra forms that were not changed. If we provide an invalid article:

>>> data = {

... "form-TOTAL\_FORMS": "2",

... "form-INITIAL\_FORMS": "0",

... "form-0-title": "Test",

... "form-0-pub\_date": "1904-06-16",

... "form-1-title": "Test",

... "form-1-pub\_date": "", # <-- this date is missing but required

... }

>>> formset = ArticleFormSet(data)

>>> formset.is\_valid()False

>>> formset.errors

[{}, {'pub\_date': ['This field is required.']}]

As we can see, **formset.errors** is a list whose entries correspond to the forms in the formset. Validation was

performed for each of the two forms, and the expected error message appears for the second item.

Just like when using a normal Form, each field in a formset’s forms may include **HTML** attributes such as

**maxlength** for browser validation. However, form fields of **formsets** won’t include the required attribute as

that validation may be incorrect when adding and deleting forms.

**BaseFormSet.total\_error\_count()**

To check how many errors there are in the formset, we can use the **total\_error\_count** method:

>>> # Using the previous example

>>> formset.errors

[{}, {'pub\_date': ['This field is required.']}]

>>> len(formset.errors)

2

>>> formset.total\_error\_count()

1

We can also check if form data differs from the initial data (i.e. the form was sent without any data):

>>> data = {

... "form-TOTAL\_FORMS": "1",

... "form-INITIAL\_FORMS": "0",

... "form-0-title": "",

... "form-0-pub\_date": "",

... }

>>> formset = ArticleFormSet(data)

>>> formset.has\_changed()

False

# Understanding the ManagementForm

You may have noticed the additional data **(form-TOTAL\_FORMS, form-INITIAL\_FORMS)** that was required in

the formset’s data above. This data is required for the **ManagementForm**. This form is used by the formset

to manage the collection of forms contained in the formset. If you don’t provide this management data, the

formset will be invalid:

>>> data = {

... "form-0-title": "Test",... "form-0-pub\_date": "",

... }

>>> formset = ArticleFormSet(data)

>>> formset.is\_valid()

False

It is used to keep track of how many form instances are being displayed. If you are adding new forms via

JavaScript, you should increment the count fields in this form as well. On the other hand, if you are using

JavaScript to allow deletion of existing objects, then you need to ensure the ones being removed are properly

marked for deletion by including **form-#-DELETE** in the **POST** data. It is expected that all forms are present

in the POST data regardless.

The management form is available as an attribute of the formset itself. When rendering a formset in a **tem**p**late**, you can include all the management data by rendering **{{ my\_formset.management\_form }}** (**substi**t**uting** the name of your formset as appropriate).

Note: As well as the **form-TOTAL\_FORMS** and **form-INITIAL\_FORMS** fields shown in the examples here, the

management form also includes **form-MIN\_NUM\_FORMS** and **form-MAX\_NUM\_FORMS** fields. They are output

with the rest of the management form, but only for the convenience of client-side code. These fields are not

required and so are not shown in the example POST data.

**total\_form\_count** and **initial\_form\_count**

**BaseFormSet** has a couple of methods that are closely related to the **ManagementForm, total\_form\_count**

and **initial\_form\_count**.

**total\_form\_count** returns the total number of forms in this formset. **initial\_form\_count** returns the

number of forms in the formset that were pre-filled, and is also used to determine how many forms are

required. You will probably never need to override either of these methods, so please be sure you understand

what they do before doing so.

**empty\_form**

**BaseFormSet** provides an additional attribute **empty\_form** which returns a form instance with a prefix of

**\_\_prefix\_\_** for easier use in dynamic forms with JavaScript.

**error\_messages**

The **error\_messages** argument lets you override the default messages that the formset will raise. Pass

in a dictionary with keys matching the error messages you want to override. Error message keys include **'too\_few\_forms'**, **'too\_many\_forms**', and **'missing\_management\_form'**. The **'too\_few\_forms'** and

**'too\_many\_forms'** error messages may contain %(num)d, which will be replaced with **min\_num** and **max\_num**,

respectively.

For example, here is the default error message when the management form is missing:

>>> formset = ArticleFormSet({})

>>> formset.is\_valid()

False

>>> formset.non\_form\_errors()

['ManagementForm data is missing or has been tampered with. Missing fields: form-TOTAL\_FORMS, form-

,→INITIAL\_FORMS. You may need to file a bug report if the issue persists.']

And here is a custom error message:

>>> formset = ArticleFormSet(

... {}, error\_messages={"missing\_management\_form": "Sorry, something went wrong."}

... )

>>> formset.is\_valid()

False

>>> formset.non\_form\_errors()

['Sorry, something went wrong.']

# Custom formset validation

A formset has a clean method similar to the one on a Form class. This is where you define your own validation

that works at the formset level:

>>> from django.core.exceptions import ValidationError

>>> from django.forms import BaseFormSet

>>> from django.forms import formset\_factory

>>> from myapp.forms import ArticleForm

>>> class BaseArticleFormSet(BaseFormSet):

... def clean(self):

... """Checks that no two articles have the same title."""

... if any(self.errors):

... # Don't bother validating the formset unless each form is valid on its own

... return

... titles = set()

... for form in self.forms:

... if self.can\_delete and self.\_should\_delete\_form(form):

... continue

... title = form.cleaned\_data.get("title")

... if title in titles:

... raise ValidationError("Articles in a set must have distinct titles.")

... titles.add(title)>>> ArticleFormSet = formset\_factory(ArticleForm, formset=BaseArticleFormSet)

>>> data = {

... "form-TOTAL\_FORMS": "2",

... "form-INITIAL\_FORMS": "0",

... "form-0-title": "Test",

... "form-0-pub\_date": "1904-06-16",

... "form-1-title": "Test",

... "form-1-pub\_date": "1912-06-23",

... }

>>> formset = ArticleFormSet(data)

>>> formset.is\_valid()

False

>>> formset.errors

[{}, {}]

>>> formset.non\_form\_errors()

['Articles in a set must have distinct titles.']

The formset clean method is called after all the **Form.clean** methods have been called. The errors will be

found using the **non\_form\_errors()** method on the formset.

Non-form errors will be rendered with an additional class of nonform to help distinguish them from formspecific errors. For example, **{{ formset.non\_form\_errors }}** would look like:

<ul class="errorlist nonform">

<li>Articles in a set must have distinct titles.</li>

</ul>

# Validating the number of forms in a formset

Django provides a couple ways to validate the minimum or maximum number of submitted forms. Applications which need more customizable validation of the number of forms should use custom formset validation.

## validate\_max

If validate\_max=True is passed to formset\_factory(), validation will also check that the number of forms

in the data set, minus those marked for deletion, is less than or equal to max\_num.

>>> from django.forms import formset\_factory

>>> from myapp.forms import ArticleForm

>>> ArticleFormSet = formset\_factory(ArticleForm, max\_num=1, validate\_max=True)

>>> data = {

... "form-TOTAL\_FORMS": "2",

... "form-INITIAL\_FORMS": "0",

... "form-0-title": "Test",

... "form-0-pub\_date": "1904-06-16",

... "form-1-title": "Test 2",

... "form-1-pub\_date": "1912-06-23",

... }

>>> formset = ArticleFormSet(data)

>>> formset.is\_valid()

False

>>> formset.errors

[{}, {}]

>>> formset.non\_form\_errors()

['Please submit at most 1 form.']

validate\_max=True validates against max\_num strictly even if **max\_num** was exceeded because the amount of

initial data supplied was excessive.

The error message can be customized by passing the **'too\_many\_forms'** message to the **error\_messages** argument.

Note: Regardless of validate\_max, if the number of forms in a data set exceeds absolute\_max, then the

form will fail to validate as if validate\_max were set, and additionally only the first absolute\_max forms

will be validated. The remainder will be truncated entirely. This is to protect against memory exhaustion

attacks using forged POST requests.

## validate\_min

If validate\_min=True is passed to formset\_factory(), validation will also check that the number of forms

in the data set, minus those marked for deletion, is greater than or equal to min\_num.

>>> from django.forms import formset\_factory

>>> from myapp.forms import ArticleForm

>>> ArticleFormSet = formset\_factory(ArticleForm, min\_num=3, validate\_min=True)

>>> data = {

... "form-TOTAL\_FORMS": "2",

... "form-INITIAL\_FORMS": "0",

... "form-0-title": "Test",

... "form-0-pub\_date": "1904-06-16",

... "form-1-title": "Test 2",

... "form-1-pub\_date": "1912-06-23",

... }

>>> formset = ArticleFormSet(data)

>>> formset.is\_valid()

False

>>> formset.errors

[{}, {}]

>>> formset.non\_form\_errors()

['Please submit at least 3 forms.']

The error message can be customized by passing the **'too\_few\_forms'** message to the **error\_messages** argument.

Note: Regardless of **validate\_min**, if a formset contains no data, then **extra + min\_num** empty forms will

be displayed.

# Dealing with ordering and deletion of forms

The **formset\_factory()** provides two optional parameters **can\_order** and **can\_delete** to help with ordering

of forms in **formsets** and deletion of forms from a formset.can\_order

BaseFormSet.can\_order

Default: False

Lets you create a formset with the ability to order:

>>> from django.forms import formset\_factory

>>> from myapp.forms import ArticleForm

>>> ArticleFormSet = formset\_factory(ArticleForm, can\_order=True)

>>> formset = ArticleFormSet(

... initial=[

... {"title": "Article #1", "pub\_date": datetime.date(2008, 5, 10)},

... {"title": "Article #2", "pub\_date": datetime.date(2008, 5, 11)},

... ]

... )

>>> for form in formset:

... print(form)

...

<div><label for="id\_form-0-title">Title:</label><input type="text" name="form-0-title" value=

,→"Article #1" id="id\_form-0-title"></div>

<div><label for="id\_form-0-pub\_date">Pub date:</label><input type="text" name="form-0-pub\_date"␣

,→value="2008-05-10" id="id\_form-0-pub\_date"></div>

<div><label for="id\_form-0-ORDER">Order:</label><input type="number" name="form-0-ORDER" value="1"␣

,→id="id\_form-0-ORDER"></div>

<div><label for="id\_form-1-title">Title:</label><input type="text" name="form-1-title" value=

,→"Article #2" id="id\_form-1-title"></div>

<div><label for="id\_form-1-pub\_date">Pub date:</label><input type="text" name="form-1-pub\_date"␣

,→value="2008-05-11" id="id\_form-1-pub\_date"></div>

<div><label for="id\_form-1-ORDER">Order:</label><input type="number" name="form-1-ORDER" value="2"␣

,→id="id\_form-1-ORDER"></div>

<div><label for="id\_form-2-title">Title:</label><input type="text" name="form-2-title" id="id\_form-

,→2-title"></div>

<div><label for="id\_form-2-pub\_date">Pub date:</label><input type="text" name="form-2-pub\_date" id=

,→"id\_form-2-pub\_date"></div>

<div><label for="id\_form-2-ORDER">Order:</label><input type="number" name="form-2-ORDER" id="id\_

,→form-2-ORDER"></div>

This adds an additional field to each form. This new field is named ORDER and is an forms.IntegerField.

For the forms that came from the initial data it automatically assigned them a numeric value. Let’s look at

what will happen when the user changes these values:

>>> data = {

... "form-TOTAL\_FORMS": "3",

... "form-INITIAL\_FORMS": "2",

... "form-0-title": "Article #1",

... "form-0-pub\_date": "2008-05-10",

... "form-0-ORDER": "2",

... "form-1-title": "Article #2",

... "form-1-pub\_date": "2008-05-11",

... "form-1-ORDER": "1",

... "form-2-title": "Article #3",

... "form-2-pub\_date": "2008-05-01",

... "form-2-ORDER": "0",

... }

>>> formset = ArticleFormSet(

... data,

... initial=[

... {"title": "Article #1", "pub\_date": datetime.date(2008, 5, 10)},

... {"title": "Article #2", "pub\_date": datetime.date(2008, 5, 11)},

... ],

... )>>> formset.is\_valid()

True

>>> for form in formset.ordered\_forms:

... print(form.cleaned\_data)

...

{'pub\_date': datetime.date(2008, 5, 1), 'ORDER': 0, 'title': 'Article #3'}

{'pub\_date': datetime.date(2008, 5, 11), 'ORDER': 1, 'title': 'Article #2'}

{'pub\_date': datetime.date(2008, 5, 10), 'ORDER': 2, 'title': 'Article #1'}

**BaseFormSet** also provides an ordering\_widget attribute and get\_ordering\_widget() method that control

the widget used with can\_order.

**ordering\_widget**

BaseFormSet.ordering\_widget

Default: NumberInput

Set ordering\_widget to specify the widget class to be used with can\_order:

>>> from django.forms import BaseFormSet, formset\_factory

>>> from myapp.forms import ArticleForm

>>> class BaseArticleFormSet(BaseFormSet):

... ordering\_widget = HiddenInput

...>>> ArticleFormSet = formset\_factory(

... ArticleForm, formset=BaseArticleFormSet, can\_order=True

... )

**get\_ordering\_widget**

## BaseFormSet.get\_ordering\_widget()

Override get\_ordering\_widget() if you need to provide a widget instance for use with can\_order:

>>> from django.forms import BaseFormSet, formset\_factory

>>> from myapp.forms import ArticleForm

>>> class BaseArticleFormSet(BaseFormSet):

... def get\_ordering\_widget(self):

... return HiddenInput(attrs={"class": "ordering"})

...

>>> ArticleFormSet = formset\_factory(

... ArticleForm, formset=BaseArticleFormSet, can\_order=True

... )

## can\_delete

**BaseFormSet.can\_delete**

Default: False

Lets you create a formset with the ability to select forms for deletion:

>>> from django.forms import formset\_factory

>>> from myapp.forms import ArticleForm

>>> ArticleFormSet = formset\_factory(ArticleForm, can\_delete=True)

>>> formset = ArticleFormSet(

... initial=[

... {"title": "Article #1", "pub\_date": datetime.date(2008, 5, 10)},

... {"title": "Article #2", "pub\_date": datetime.date(2008, 5, 11)},

... ]

... )

>>> for form in formset:

... print(form)

...

<div><label for="id\_form-0-title">Title:</label><input type="text" name="form-0-title" value=

,→"Article #1" id="id\_form-0-title"></div>

<div><label for="id\_form-0-pub\_date">Pub date:</label><input type="text" name="form-0-pub\_date"␣

,→value="2008-05-10" id="id\_form-0-pub\_date"></div>

<div><label for="id\_form-0-DELETE">Delete:</label><input type="checkbox" name="form-0-DELETE" id=

,→"id\_form-0-DELETE"></div>

<div><label for="id\_form-1-title">Title:</label><input type="text" name="form-1-title" value=

,→"Article #2" id="id\_form-1-title"></div>

<div><label for="id\_form-1-pub\_date">Pub date:</label><input type="text" name="form-1-pub\_date"␣

,→value="2008-05-11" id="id\_form-1-pub\_date"></div>

<div><label for="id\_form-1-DELETE">Delete:</label><input type="checkbox" name="form-1-DELETE" id=

,→"id\_form-1-DELETE"></div>

<div><label for="id\_form-2-title">Title:</label><input type="text" name="form-2-title" id="id\_form-

,→2-title"></div>

<div><label for="id\_form-2-pub\_date">Pub date:</label><input type="text" name="form-2-pub\_date" id=

,→"id\_form-2-pub\_date"></div>

<div><label for="id\_form-2-DELETE">Delete:</label><input type="checkbox" name="form-2-DELETE" id=

,→"id\_form-2-DELETE"></div>

Similar to can\_order this adds a new field to each form named DELETE and is a forms.BooleanField. When

data comes through marking any of the delete fields you can access them with deleted\_forms:

>>> data = {

... "form-TOTAL\_FORMS": "3",

... "form-INITIAL\_FORMS": "2",

... "form-0-title": "Article #1",

... "form-0-pub\_date": "2008-05-10",

... "form-0-DELETE": "on",

... "form-1-title": "Article #2",

... "form-1-pub\_date": "2008-05-11",

... "form-1-DELETE": "",

... "form-2-title": "",

... "form-2-pub\_date": "",

... "form-2-DELETE": "",

... }

>>> formset = ArticleFormSet(

... data,

... initial=[

... {"title": "Article #1", "pub\_date": datetime.date(2008, 5, 10)},

... {"title": "Article #2", "pub\_date": datetime.date(2008, 5, 11)},

... ],

... )

>>> [form.cleaned\_data for form in formset.deleted\_forms][{'DELETE': True, 'pub\_date': datetime.date(2008, 5, 10), 'title': 'Article #1'}]

If you are using a **ModelFormSet**, model instances for deleted forms will be deleted when you call formset.

save().

If you call **formset.save(commit=False),** objects will not be deleted automatically. You’ll need to call

delete() on each of the **formset.deleted\_objects** to actually delete them:

>>> instances = formset.save(commit=False)

>>> for obj in formset.deleted\_objects:

... obj.delete()

...

On the other hand, if you are using a plain **FormSet**, it’s up to you to handle formset.deleted\_forms, perhaps

in your formset’s save() method, as there’s no general notion of what it means to delete a form.

**BaseFormSet** also provides a **deletion\_widget** attribute and **get\_deletion\_widget()** method that control

the widget used with **can\_delete.**

## deletion\_widget

**BaseFormSet.deletion\_widget**

Default: CheckboxInput

Set deletion\_widget to specify the widget class to be used with can\_delete:

>>> from django.forms import BaseFormSet, formset\_factory

>>> from myapp.forms import ArticleForm

>>> class BaseArticleFormSet(BaseFormSet):

... deletion\_widget = HiddenInput

...

>>> ArticleFormSet = formset\_factory(

... ArticleForm, formset=BaseArticleFormSet, can\_delete=True

... )

## get\_deletion\_widget

**BaseFormSet.get\_deletion\_widget()**

Override get\_deletion\_widget() if you need to provide a widget instance for use with can\_delete:

>>> from django.forms import BaseFormSet, formset\_factory

>>> from myapp.forms import ArticleForm

>>> class BaseArticleFormSet(BaseFormSet):

... def get\_deletion\_widget(self):

... return HiddenInput(attrs={"class": "deletion"})

...

>>> ArticleFormSet = formset\_factory(

... ArticleForm, formset=BaseArticleFormSet, can\_delete=True

... )

### can\_delete\_extra

**BaseFormSet.can\_delete\_extra**

Default: **True**

While setting **can\_delete=True**, specifying **can\_delete\_extra=False** will remove the option to delete extra

forms.

Adding additional fields to a formset

If you need to add additional fields to the formset this can be easily accomplished. The formset base class

provides an add\_fields method. You can override this method to add your own fields or even redefine the

default fields/attributes of the order and deletion fields:

>>> from django.forms import BaseFormSet

>>> from django.forms import formset\_factory

>>> from myapp.forms import ArticleForm

>>> class BaseArticleFormSet(BaseFormSet):

... def add\_fields(self, form, index):

... super().add\_fields(form, index)

... form.fields["my\_field"] = forms.CharField()

...

>>> ArticleFormSet = formset\_factory(ArticleForm, formset=BaseArticleFormSet)

>>> formset = ArticleFormSet()

>>> for form in formset:... print(form)

...

<div><label for="id\_form-0-title">Title:</label><input type="text" name="form-0-title" id="id\_form-

,→0-title"></div>

<div><label for="id\_form-0-pub\_date">Pub date:</label><input type="text" name="form-0-pub\_date" id=

,→"id\_form-0-pub\_date"></div>

<div><label for="id\_form-0-my\_field">My field:</label><input type="text" name="form-0-my\_field" id=

,→"id\_form-0-my\_field"></div>

## Passing custom parameters to formset forms

Sometimes your form class takes custom parameters, like MyArticleForm. You can pass this parameter when

instantiating the formset:

>>> from django.forms import BaseFormSet

>>> from django.forms import formset\_factory

>>> from myapp.forms import ArticleForm

>>> class MyArticleForm(ArticleForm):

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

... self.user = user

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

...

>>> ArticleFormSet = formset\_factory(MyArticleForm)

>>> formset = ArticleFormSet(form\_kwargs={"user": request.user})

The **form\_kwargs** may also depend on the specific form instance. The formset base class provides a

**get\_form\_kwargs** method. The method takes a single argument - the index of the form in the formset.

The index is None for the empty\_form:

>>> from django.forms import BaseFormSet

>>> from django.forms import formset\_factory

>>> class BaseArticleFormSet(BaseFormSet):

... def get\_form\_kwargs(self, index):

... kwargs = super().get\_form\_kwargs(index)

... kwargs["custom\_kwarg"] = index

... return kwargs

...

>>> ArticleFormSet = formset\_factory(MyArticleForm, formset=BaseArticleFormSet)

>>> formset = ArticleFormSet()

### Customizing a formset’s prefix

In the rendered HTML, formsets include a prefix on each field’s name. By default, the prefix is 'form', but

it can be customized using the formset’s prefix argument.

For example, in the default case, you might see:

<label for="id\_form-0-title">Title:</label>

<input type="text" name="form-0-title" id="id\_form-0-title">

But with ArticleFormset(prefix='article') that becomes:

<label for="id\_article-0-title">Title:</label>

<input type="text" name="article-0-title" id="id\_article-0-title">

This is useful if you want to use more than one formset in a view.

# Using a formset in views and templates

Formsets have the following attributes and methods associated with rendering:

**BaseFormSet.renderer**

Specifies the renderer to use for the formset. Defaults to the renderer specified by the FORM\_RENDERER

setting.

**BaseFormSet.template\_name**

The name of the template rendered if the formset is cast into a string, e.g. via **print(formset)** or in a

template via **{{ formset }}.**

By default, a property returning the value of the renderer’s **formset\_template\_name**. You may set it

as a string template name in order to override that for a particular formset class.

This template will be used to render the formset’s management form, and then each form in the formset

as per the template defined by the form’s **template\_name**.

**BaseFormSet.template\_name\_div**

The name of the template used when calling **as\_div().** By default this is **"django/forms/formsets/**

**div.html"**. This template renders the formset’s management form and then each form in the formset

as per the form’s **as\_div()** method.

**BaseFormSet.template\_name\_p**

The name of the template used when calling **as\_p().** By default this is **"django/forms/formsets/p.**

**html**". This template renders the formset’s management form and then each form in the formset as

per the form’s **as\_p()** method.

**BaseFormSet.template\_name\_table**

The name of the template used when calling as\_table(). By default this is "django/forms/formsets/

table.html". This template renders the formset’s management form and then each form in the formset

as per the form’s as\_table() method.

**BaseFormSet.template\_name\_ul**

The name of the template used when calling **as\_ul().** By default this is "django/forms/formsets/

ul.html". This template renders the formset’s management form and then each form in the formset

as per the form’s as\_**ul()** method.

**BaseFormSet.get\_context()**

Returns the context for rendering a formset in a template.

The available context is:

• **formset** : The instance of the formset.

**BaseFormSet.render(template\_name=None, context=None, renderer=None)**

The render method is called by **\_\_str\_\_** as well as the **as\_div(), as\_p(), as\_ul(),** and **as\_table()**

methods. All arguments are optional and will default to:

• template\_name: template\_name

• context: Value returned by **get\_context()**

• renderer: Value returned by renderer

**BaseFormSet.as\_div()**

Renders the formset with the **template\_name\_div** template.

**BaseFormSet.as\_p()**

Renders the formset with the **template\_name\_p** template.

**BaseFormSet.as\_table()**

Renders the formset with the **template\_name\_table** template.

**BaseFormSet.as\_ul()**

Renders the formset with the template\_name\_ul template.

Using a formset inside a view is not very different from using a regular Form class. The only thing you will

want to be aware of is making sure to use the management form inside the template. Let’s look at a sample view:

from django.forms import formset\_factory

from django.shortcuts import render

from myapp.forms import ArticleForm

def manage\_articles(request):

ArticleFormSet = formset\_factory(ArticleForm)

if request.method == "POST":

formset = ArticleFormSet(request.POST, request.FILES)

if formset.is\_valid():

# do something with the formset.cleaned\_data

pass

else:

formset = ArticleFormSet()

return render(request, "manage\_articles.html", {"formset": formset})

The manage\_articles.html template might look like this:

<form method="post">

{{ formset.management\_form }}

<table>

{% for form in formset %}

{{ form }}

{% endfor %}

</table>

</form>

However, there’s a slight shortcut for the above by letting the formset itself deal with the management form:

<form method="post">

<table>

{{ formset }}

</table>

</form>

The above ends up calling the **BaseFormSet.render()** method on the formset class. This renders the formset

using the template specified by the **template\_name** attribute. Similar to forms, by default the formset will

be rendered **as\_table**, with other helper methods of **as\_p** and **as\_ul** being available. The rendering of the

formset can be customized by specifying the **template\_name** attribute, or more generally by overriding the

default template.

## Manually rendered can\_delete and can\_order

If you manually render fields in the template, you can render **can\_delete** parameter with **{{ form.DELETE**

**}}**:

<form method="post">

{{ formset.management\_form }}

{% for form in formset %}

<ul>

<li>{{ form.title }}</li>

<li>{{ form.pub\_date }}</li>

{% if formset.can\_delete %}

<li>{{ form.DELETE }}</li>

{% endif %}

</ul>

{% endfor %}

</form>

Similarly, if the formset has the ability to order (can\_order=True), it is possible to render it with {{ form.

ORDER }}

## Using more than one formset in a view

You are able to use more than one formset in a view if you like. Formsets borrow much of its behavior from

forms. With that said you are able to use prefix to prefix formset form field names with a given value to

allow more than one formset to be sent to a view without name clashing. Let’s take a look at how this might

be accomplished:

from django.forms import formset\_factory

from django.shortcuts import render

from myapp.forms import ArticleForm, BookForm

def manage\_articles(request):

ArticleFormSet = formset\_factory(ArticleForm)

BookFormSet = formset\_factory(BookForm)

if request.method == "POST":

article\_formset = ArticleFormSet(request.POST, request.FILES, prefix="articles")

book\_formset = BookFormSet(request.POST, request.FILES, prefix="books")

if article\_formset.is\_valid() and book\_formset.is\_valid():

# do something with the cleaned\_data on the formsets.

pass

else:

article\_formset = ArticleFormSet(prefix="articles") book\_formset = BookFormSet(prefix="books")

return render(

request,

"manage\_articles.html",

{

"article\_formset": article\_formset,

"book\_formset": book\_formset,

},

)

You would then render the **formsets** as normal. It is important to point out that you need to pass prefix on

both the **POST** and **non-POST** cases so that it is rendered and processed correctly.

Each formset’s prefix replaces the default form prefix that’s added to each field’s name and id HTML attributes.

--- --- END --- ---