# Formsets

```
class BaseFormSet
```
[source]

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=[...])` .

**See also:** Creating formsets from models with model formsets.

## 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. See validate_max.

## 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()
```
[source]

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 template, you can include all the management data by rendering `{{ my_formset.management_form }}` (substituting 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 form-specific 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. See Limiting the maximum number of instantiated forms.

```
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()` [source]

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()
```
[source]

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` [source]

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()` [source]

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.