# Creating forms from models

## ModelForm

```
class ModelForm
```
[source]

If you’re building a database-driven app, chances are you’ll have forms that map closely to Django models. For instance, you might have a `BlogComment` model, and you want to create a form that lets people submit comments. In this case, it would be redundant to define the field types in your form, because you’ve already defined the fields in your model.

For this reason, Django provides a helper class that lets you create a `Form` class from a Django model.

For example:

```
>>> from django.forms import ModelForm
>>> from myapp.models import Article

# Create the form class.
>>> class ArticleForm(ModelForm):
...     class Meta:
...         model = Article
...         fields = ["pub_date", "headline", "content", "reporter"]
...

# Creating a form to add an article.
>>> form = ArticleForm()

# Creating a form to change an existing article.
>>> article = Article.objects.get(pk=1)
>>> form = ArticleForm(instance=article)
```

### Field types

The generated `Form` class will have a form field for every model field specified, in the order specified in the `fields` attribute.

Each model field has a corresponding default form field. For example, a `CharField` on a model is represented as a `CharField` on a form. A model `ManyToManyField` is represented as a `MultipleChoiceField`. Here is the full list of conversions:

| Model field | Form field |
|---|---|
| `AutoField` | Not represented in the form |
| `BigAutoField` | Not represented in the form |
| `BigIntegerField` | `IntegerField` with `min_value` set to -9223372036854775808 and `max_value` set to 9223372036854775807. |
| `BinaryField` | `CharField`, if `editable` is set to `True` on the model field, otherwise not represented in the form. |
| `BooleanField` | `BooleanField`, or `NullBooleanField` if `null=True`. |
| `CharField` | `CharField` with `max_length` set to the model field's `max_length` and `empty_value` set to None if `null=True`. |
| `DateField` | `DateField` |
| `DateTimeField` | `DateTimeField` |
| `DecimalField` | `DecimalField` |
| `DurationField` | `DurationField` |
| `EmailField` | `EmailField` |
| `FileField` | `FileField` |
| `FilePathField` | `FilePathField` |
| `FloatField` | `FloatField` |
| `ForeignKey` | `ModelChoiceField` (see below) |
| `ImageField` | `ImageField` |
| `IntegerField` | `IntegerField` |
| `IPAddressField` | `IPAddressField` |
| `GenericIPAddressField` | `GenericIPAddressField` |
| `JSONField` | `JSONField` |
| `ManyToManyField` | `ModelMultipleChoiceField` (see below) |
| `PositiveBigIntegerField` | `IntegerField` |
| `PositiveIntegerField` | `IntegerField` |
| `PositiveSmallIntegerField` | `IntegerField` |
| `SlugField` | `SlugField` |
| `SmallAutoField` | Not represented in the form |
| `SmallIntegerField` | `IntegerField` |

| | |
|---|---|
| `TextField` | `CharField` with `widget=forms.Textarea` |
| `TimeField` | `TimeField` |
| `URLField` | `URLField` |
| `UUIDField` | `UUIDField` |

As you might expect, the `ForeignKey` and `ManyToManyField` model field types are special cases:

- `ForeignKey` is represented by `django.forms.ModelChoiceField`, which is a `ChoiceField` whose choices are a model `QuerySet`.
- `ManyToManyField` is represented by `django.forms.ModelMultipleChoiceField`, which is a `MultipleChoiceField` whose choices are a model `QuerySet`.

In addition, each generated form field has attributes set as follows:

- If the model field has `blank=True`, then `required` is set to `False` on the form field. Otherwise, `required=True`.
- The form field's `label` is set to the `verbose_name` of the model field, with the first character capitalized.
- The form field's `help_text` is set to the `help_text` of the model field.
- If the model field has `choices` set, then the form field's `widget` will be set to `Select`, with choices coming from the model field's `choices`. The choices will normally include the blank choice which is selected by default. If the field is required, this forces the user to make a selection. The blank choice will not be included if the model field has `blank=False` and an explicit `default` value (the `default` value will be initially selected instead).

Finally, note that you can override the form field used for a given model field. See Overriding the default fields below.

## A full example

Consider this set of models:

```
from django.db import models
from django.forms import ModelForm

TITLE_CHOICES = {
    "MR": "Mr.",
    "MRS": "Mrs.",
    "MS": "Ms.",
}


class Author(models.Model):
    name = models.CharField(max_length=100)
    title = models.CharField(max_length=3, choices=TITLE_CHOICES)
    birth_date = models.DateField(blank=True, null=True)

    def __str__(self):
        return self.name
```

```
class Book(models.Model):
    name = models.CharField(max_length=100)
    authors = models.ManyToManyField(Author)


class AuthorForm(ModelForm):
    class Meta:
        model = Author
        fields = ["name", "title", "birth_date"]


class BookForm(ModelForm):
    class Meta:
        model = Book
        fields = ["name", "authors"]
```

With these models, the `ModelForm` subclasses above would be roughly equivalent to this (the only difference being the `save()` method, which we'll discuss in a moment.):

```
from django import forms


class AuthorForm(forms.Form):
    name = forms.CharField(max_length=100)
    title = forms.CharField(
        max_length=3,
        widget=forms.Select(choices=TITLE_CHOICES),
    )
    birth_date = forms.DateField(required=False)


class BookForm(forms.Form):
    name = forms.CharField(max_length=100)
    authors = forms.ModelMultipleChoiceField(queryset=Author.objects.all())
```

## Validation on a `ModelForm`

There are two main steps involved in validating a `ModelForm`:

1. Validating the form
2. Validating the model instance

Just like normal form validation, model form validation is triggered implicitly when calling `is_valid()` or accessing the `errors` attribute and explicitly when calling `full_clean()`, although you will typically not use the latter method in practice.

`Model` validation (`Model.full_clean()`) is triggered from within the form validation step, right after the form's `clean()` method is called.

**Warning:** The cleaning process modifies the model instance passed to the `ModelForm` constructor in various ways. For instance, any date fields on the model are converted into actual date objects. Failed validation may leave the underlying model instance in an inconsistent state and therefore it's not recommended to reuse it.

#### Overriding the `clean()` method

You can override the `clean()` method on a model form to provide additional validation in the same way you can on a normal form.

A model form instance attached to a model object will contain an `instance` attribute that gives its methods access to that specific model instance.

**Warning:** The `ModelForm.clean()` method sets a flag that makes the model validation step validate the uniqueness of model fields that are marked as `unique`, `unique_together` or `unique_for_date|month|year`.

If you would like to override the `clean()` method and maintain this validation, you must call the parent class's `clean()` method.

#### Interaction with model validation

As part of the validation process, `ModelForm` will call the `clean()` method of each field on your model that has a corresponding field on your form. If you have excluded any model fields, validation will not be run on those fields. See the form validation documentation for more on how field cleaning and validation work.

The model's `clean()` method will be called before any uniqueness checks are made. See Validating objects for more information on the model's `clean()` hook.

#### Considerations regarding model's `error_messages`

Error messages defined at the `form field` level or at the form Meta level always take precedence over the error messages defined at the `model field` level.

Error messages defined on `model fields` are only used when the `ValidationError` is raised during the model validation step and no corresponding error messages are defined at the form level.

You can override the error messages from `NON_FIELD_ERRORS` raised by model validation by adding the `NON_FIELD_ERRORS` key to the `error_messages` dictionary of the `ModelForm`'s inner `Meta` class:

```
from django.core.exceptions import NON_FIELD_ERRORS
from django.forms import ModelForm


class ArticleForm(ModelForm):
    class Meta:
        error_messages = {
            NON_FIELD_ERRORS: {
                "unique_together": "%(model_name)s's %(field_labels)s are not unique.",
            }
        }
```

## The `save()` method

Every `ModelForm` also has a `save()` method. This method creates and saves a database object from the data bound to the form. A subclass of `ModelForm` can accept an existing model instance as the keyword argument `instance`; if this is supplied, `save()` will update that instance. If it's not supplied, `save()` will create a new instance of the specified model:

```
>>> from myapp.models import Article
>>> from myapp.forms import ArticleForm

# Create a form instance from POST data.
>>> f = ArticleForm(request.POST)

# Save a new Article object from the form's data.
>>> new_article = f.save()

# Create a form to edit an existing Article, but use
# POST data to populate the form.
>>> a = Article.objects.get(pk=1)
>>> f = ArticleForm(request.POST, instance=a)
>>> f.save()
```

Note that if the form hasn't been validated, calling `save()` will do so by checking `form.errors`. A `ValueError` will be raised if the data in the form doesn't validate – i.e., if `form.errors` evaluates to `True`.

If an optional field doesn't appear in the form's data, the resulting model instance uses the model field `default`, if there is one, for that field. This behavior doesn't apply to fields that use `CheckboxInput`, `CheckboxSelectMultiple`, or `SelectMultiple` (or any custom widget whose `value_omitted_from_data()` method always returns `False`) since an unchecked checkbox and unselected `<select multiple>` don't appear in the data of an HTML form submission. Use a custom form field or widget if you're designing an API and want the default fallback behavior for a field that uses one of these widgets.

This `save()` method accepts an optional `commit` keyword argument, which accepts either `True` or `False`. If you call `save()` with `commit=False`, then it will return an object that hasn't yet been saved to the database. In this case, it's up to you to call `save()` on the resulting model instance. This is useful if you want to do custom processing on the object before saving it, or if you want to use one of the specialized model saving options. `commit` is `True` by default.

Another side effect of using `commit=False` is seen when your model has a many-to-many relation with another model. If your model has a many-to-many relation and you specify `commit=False` when you save a form, Django cannot immediately save the form data for the many-to-many relation. This is because it isn't possible to save many-to-many data for an instance until the instance exists in the database.

To work around this problem, every time you save a form using `commit=False`, Django adds a `save_m2m()` method to your `ModelForm` subclass. After you've manually saved the instance produced by the form, you can invoke `save_m2m()` to save the many-to-many form data. For example:

```
# Create a form instance with POST data.
>>> f = AuthorForm(request.POST)

# Create, but don't save the new author instance.
>>> new_author = f.save(commit=False)

# Modify the author in some way.
>>> new_author.some_field = "some_value"

# Save the new instance.
>>> new_author.save()

# Now, save the many-to-many data for the form.
>>> f.save_m2m()
```

Calling `save_m2m()` is only required if you use `save(commit=False)`. When you use a `save()` on a form, all data – including many-to-many data – is saved without the need for any additional method calls. For example:

```
# Create a form instance with POST data.
>>> a = Author()
>>> f = AuthorForm(request.POST, instance=a)

# Create and save the new author instance. There's no need to do anything else.
>>> new_author = f.save()
```

Other than the `save()` and `save_m2m()` methods, a `ModelForm` works exactly the same way as any other `forms` form. For example, the `is_valid()` method is used to check for validity, the `is_multipart()` method is used to determine whether a form requires multipart file upload (and hence whether `request.FILES` must be passed to the form), etc. See Binding uploaded files to a form for more information.

### Selecting the fields to use

It is strongly recommended that you explicitly set all fields that should be edited in the form using the `fields` attribute. Failure to do so can easily lead to security problems when a form unexpectedly allows a user to set certain fields, especially when new fields are added to a model. Depending on how the form is rendered, the problem may not even be visible on the web page.

The alternative approach would be to include all fields automatically, or remove only some. This fundamental approach is known to be much less secure and has led to serious exploits on major websites (e.g. GitHub).

There are, however, two shortcuts available for cases where you can guarantee these security concerns do not apply to you:

1. Set the `fields` attribute to the special value `'__all__'` to indicate that all fields in the model should be used. For

example:

```
from django.forms import ModelForm


class AuthorForm(ModelForm):
    class Meta:
        model = Author
        fields = "__all__"
```

2. Set the `exclude` attribute of the `ModelForm`'s inner `Meta` class to a list of fields to be excluded from the form.

   For example:

```
class PartialAuthorForm(ModelForm):
    class Meta:
        model = Author
        exclude = ["title"]
```

   Since the `Author` model has the 3 fields `name`, `title` and `birth_date`, this will result in the fields `name` and `birth_date` being present on the form.

If either of these are used, the order the fields appear in the form will be the order the fields are defined in the model, with `ManyToManyField` instances appearing last.

In addition, Django applies the following rule: if you set `editable=False` on the model field, *any* form created from the model via `ModelForm` will not include that field.

**Note:** Any fields not included in a form by the above logic will not be set by the form's `save()` method. Also, if you manually add the excluded fields back to the form, they will not be initialized from the model instance.

Django will prevent any attempt to save an incomplete model, so if the model does not allow the missing fields to be empty, and does not provide a default value for the missing fields, any attempt to `save()` a `ModelForm` with missing fields will fail. To avoid this failure, you must instantiate your model with initial values for the missing, but required fields:

```
author = Author(title="Mr")
form = PartialAuthorForm(request.POST, instance=author)
form.save()
```

Alternatively, you can use `save(commit=False)` and manually set any extra required fields:

```
form = PartialAuthorForm(request.POST)
author = form.save(commit=False)
author.title = "Mr"
author.save()
```

See the section on saving forms for more details on using `save(commit=False)`.

## Overriding the default fields

The default field types, as described in the Field types table above, are sensible defaults. If you have a `DateField` in your model, chances are you'd want that to be represented as a `DateField` in your form. But `ModelForm` gives you the flexibility of changing the form field for a given model.

To specify a custom widget for a field, use the `widgets` attribute of the inner `Meta` class. This should be a dictionary mapping field names to widget classes or instances.

For example, if you want the `CharField` for the `name` attribute of `Author` to be represented by a `<textarea>` instead of its default `<input type="text">`, you can override the field's widget:

```
from django.forms import ModelForm, Textarea
from myapp.models import Author


class AuthorForm(ModelForm):
    class Meta:
        model = Author
        fields = ["name", "title", "birth_date"]
        widgets = {
            "name": Textarea(attrs={"cols": 80, "rows": 20}),
        }
```

The `widgets` dictionary accepts either widget instances (e.g., `Textarea(...)`) or classes (e.g., `Textarea`). Note that the `widgets` dictionary is ignored for a model field with a non-empty `choices` attribute. In this case, you must override the form field to use a different widget.

Similarly, you can specify the `labels`, `help_texts` and `error_messages` attributes of the inner `Meta` class if you want to further customize a field.

For example if you wanted to customize the wording of all user facing strings for the `name` field:

```
from django.utils.translation import gettext_lazy as _


class AuthorForm(ModelForm):
    class Meta:
        model = Author
        fields = ["name", "title", "birth_date"]
        labels = {
            "name": _("Writer"),
        }
        help_texts = {
            "name": _("Some useful help text."),
        }
        error_messages = {
            "name": {
                "max_length": _("This writer's name is too long."),
            },
        }
```

You can also specify `field_classes` or `formfield_callback` to customize the type of fields instantiated by the form.

For example, if you wanted to use `MySlugFormField` for the `slug` field, you could do the following:

```
from django.forms import ModelForm
from myapp.models import Article


class ArticleForm(ModelForm):
    class Meta:
        model = Article
        fields = ["pub_date", "headline", "content", "reporter", "slug"]
        field_classes = {
            "slug": MySlugFormField,
        }
```

or:

```
from django.forms import ModelForm
from myapp.models import Article


def formfield_for_dbfield(db_field, **kwargs):
    if db_field.name == "slug":
        return MySlugFormField()
    return db_field.formfield(**kwargs)


class ArticleForm(ModelForm):
    class Meta:
        model = Article
        fields = ["pub_date", "headline", "content", "reporter", "slug"]
        formfield_callback = formfield_for_dbfield
```

Finally, if you want complete control over of a field – including its type, validators, required, etc. – you can do this by declaratively specifying fields like you would in a regular `Form`.

If you want to specify a field's validators, you can do so by defining the field declaratively and setting its `validators` parameter:

```
from django.forms import CharField, ModelForm
from myapp.models import Article


class ArticleForm(ModelForm):
    slug = CharField(validators=[validate_slug])

    class Meta:
        model = Article
        fields = ["pub_date", "headline", "content", "reporter", "slug"]
```

**Note:** When you explicitly instantiate a form field like this, it is important to understand how `ModelForm` and regular `Form` are related.

`ModelForm` is a regular `Form` which can automatically generate certain fields. The fields that are automatically generated depend on the content of the `Meta` class and on which fields have already been defined declaratively. Basically, `ModelForm` will **only** generate fields that are **missing** from the form, or in other words, fields that weren't defined declaratively.

Fields defined declaratively are left as-is, therefore any customizations made to `Meta` attributes such as `widgets`, `labels`, `help_texts`, or `error_messages` are ignored; these only apply to fields that are generated automatically.

Similarly, fields defined declaratively do not draw their attributes like `max_length` or `required` from the corresponding model. If you want to maintain the behavior specified in the model, you must set the relevant arguments explicitly when declaring the form field.

For example, if the `Article` model looks like this:

```
class Article(models.Model):
    headline = models.CharField(
        max_length=200,
        null=True,
        blank=True,
        help_text="Use puns liberally",
    )
    content = models.TextField()
```

and you want to do some custom validation for `headline`, while keeping the `blank` and `help_text` values as specified, you might define `ArticleForm` like this:

```
class ArticleForm(ModelForm):
    headline = MyFormField(
        max_length=200,
        required=False,
        help_text="Use puns liberally",
    )

    class Meta:
        model = Article
        fields = ["headline", "content"]
```

You must ensure that the type of the form field can be used to set the contents of the corresponding model field. When they are not compatible, you will get a `ValueError` as no implicit conversion takes place.

See the form field documentation for more information on fields and their arguments.

## Enabling localization of fields

By default, the fields in a `ModelForm` will not localize their data. To enable localization for fields, you can use the `localized_fields` attribute on the `Meta` class.

```
>>> from django.forms import ModelForm
>>> from myapp.models import Author
>>> class AuthorForm(ModelForm):
...     class Meta:
...         model = Author
...         localized_fields = ['birth_date']
```

If `localized_fields` is set to the special value `'__all__'`, all fields will be localized.

## Form inheritance

As with basic forms, you can extend and reuse `ModelForms` by inheriting them. This is useful if you need to declare extra fields or extra methods on a parent class for use in a number of forms derived from models. For example, using the previous `ArticleForm` class:

```
>>> class EnhancedArticleForm(ArticleForm):
...     def clean_pub_date(self): ...
...
```

This creates a form that behaves identically to `ArticleForm`, except there's some extra validation and cleaning for the `pub_date` field.

You can also subclass the parent's `Meta` inner class if you want to change the `Meta.fields` or `Meta.exclude` lists:

```
>>> class RestrictedArticleForm(EnhancedArticleForm):
...     class Meta(ArticleForm.Meta):
...         exclude = ["body"]
...
```

This adds the extra method from the `EnhancedArticleForm` and modifies the original `ArticleForm.Meta` to remove one field.

There are a couple of things to note, however.

- Normal Python name resolution rules apply. If you have multiple base classes that declare a `Meta` inner class, only the first one will be used. This means the child's `Meta`, if it exists, otherwise the `Meta` of the first parent, etc.
- It's possible to inherit from both `Form` and `ModelForm` simultaneously, however, you must ensure that `ModelForm` appears first in the MRO. This is because these classes rely on different metaclasses and a class can only have one metaclass.
- It's possible to declaratively remove a `Field` inherited from a parent class by setting the name to be `None` on the subclass.

You can only use this technique to opt out from a field defined declaratively by a parent class; it won't prevent the `ModelForm` metaclass from generating a default field. To opt-out from default fields, see Selecting the fields to use.

### Providing initial values

As with regular forms, it's possible to specify initial data for forms by specifying an `initial` parameter when instantiating the form. Initial values provided this way will override both initial values from the form field and values from an attached model instance. For example:

```
>>> article = Article.objects.get(pk=1)
>>> article.headline
'My headline'
>>> form = ArticleForm(initial={"headline": "Initial headline"}, instance=article)
>>> form["headline"].value()
'Initial headline'
```

### ModelForm factory function

You can create forms from a given model using the standalone function `modelform_factory()`, instead of using a class definition. This may be more convenient if you do not have many customizations to make:

```
>>> from django.forms import modelform_factory
>>> from myapp.models import Book
>>> BookForm = modelform_factory(Book, fields=["author", "title"])
```

This can also be used to make modifications to existing forms, for example by specifying the widgets to be used for a given field:

```
>>> from django.forms import Textarea
>>> Form = modelform_factory(Book, form=BookForm, widgets={"title": Textarea()})
```

The fields to include can be specified using the `fields` and `exclude` keyword arguments, or the corresponding attributes on the `ModelForm` inner `Meta` class. Please see the `ModelForm` Selecting the fields to use documentation.

... or enable localization for specific fields:

```
>>> Form = modelform_factory(Author, form=AuthorForm, localized_fields=["birth_date"])
```

## Model formsets

`class models.BaseModelFormSet`

Like regular formsets, Django provides a couple of enhanced formset classes to make working with Django models more convenient. Let's reuse the `Author` model from above:

```
>>> from django.forms import modelformset_factory
>>> from myapp.models import Author
>>> AuthorFormSet = modelformset_factory(Author, fields=["name", "title"])
```

Using `fields` restricts the formset to use only the given fields. Alternatively, you can take an "opt-out" approach, specifying which fields to exclude:

```
>>> AuthorFormSet = modelformset_factory(Author, exclude=["birth_date"])
```

This will create a formset that is capable of working with the data associated with the `Author` model. It works just like a regular formset:

```
>>> formset = AuthorFormSet()
>>> print(formset)
<input type="hidden" name="form-TOTAL_FORMS" value="1" id="id_form-TOTAL_FORMS"><input type="hidden"
name="form-INITIAL_FORMS" value="0" id="id_form-INITIAL_FORMS"><input type="hidden" name="form-
MIN_NUM_FORMS" value="0" id="id_form-MIN_NUM_FORMS"><input type="hidden" name="form-MAX_NUM_FORMS"
value="1000" id="id_form-MAX_NUM_FORMS">
<div><label for="id_form-0-name">Name:</label><input id="id_form-0-name" type="text" name="form-0-name"
maxlength="100"></div>
<div><label for="id_form-0-title">Title:</label><select name="form-0-title" id="id_form-0-title">
<option value="" selected>---------</option>
<option value="MR">Mr.</option>
<option value="MRS">Mrs.</option>
<option value="MS">Ms.</option>
</select><input type="hidden" name="form-0-id" id="id_form-0-id"></div>
```

**Note:** `modelformset_factory()` uses `formset_factory()` to generate formsets. This means that a model formset is an extension of a basic formset that knows how to interact with a particular model.

**Note:** When using multi-table inheritance, forms generated by a formset factory will contain a parent link field (by default `<parent_model_name>_ptr`) instead of an `id` field.

## Changing the queryset

By default, when you create a formset from a model, the formset will use a queryset that includes all objects in the model (e.g., `Author.objects.all()`). You can override this behavior by using the `queryset` argument:

```
>>> formset = AuthorFormSet(queryset=Author.objects.filter(name__startswith="O"))
```

Alternatively, you can create a subclass that sets `self.queryset` in `__init__` :

```
from django.forms import BaseModelFormSet
from myapp.models import Author


class BaseAuthorFormSet(BaseModelFormSet):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.queryset = Author.objects.filter(name__startswith="O")
```

Then, pass your `BaseAuthorFormSet` class to the factory function:

```
>>> AuthorFormSet = modelformset_factory(
...     Author, fields=["name", "title"], formset=BaseAuthorFormSet
... )
```

If you want to return a formset that doesn't include *any* preexisting instances of the model, you can specify an empty QuerySet:

```
>>> AuthorFormSet(queryset=Author.objects.none())
```

### Changing the form

By default, when you use `modelformset_factory` , a model form will be created using `modelform_factory()`. Often, it can be useful to specify a custom model form. For example, you can create a custom model form that has custom validation:

```
class AuthorForm(forms.ModelForm):
    class Meta:
        model = Author
        fields = ["name", "title"]

    def clean_name(self):
        # custom validation for the name field
        ...
```

Then, pass your model form to the factory function:

```
AuthorFormSet = modelformset_factory(Author, form=AuthorForm)
```

It is not always necessary to define a custom model form. The `modelformset_factory` function has several arguments which are passed through to `modelform_factory` , which are described below.

### Specifying widgets to use in the form with `widgets`

Using the `widgets` parameter, you can specify a dictionary of values to customize the `ModelForm`'s widget class for a particular field. This works the same way as the `widgets` dictionary on the inner `Meta` class of a `ModelForm` works:

```
>>> AuthorFormSet = modelformset_factory(
...     Author,
...     fields=["name", "title"],
...     widgets={"name": Textarea(attrs={"cols": 80, "rows": 20})},
... )
```

### Enabling localization for fields with `localized_fields`

Using the `localized_fields` parameter, you can enable localization for fields in the form.

```
>>> AuthorFormSet = modelformset_factory(
...     Author, fields=['name', 'title', 'birth_date'],
...     localized_fields=['birth_date'])
```

If `localized_fields` is set to the special value `'__all__'`, all fields will be localized.

### Providing initial values

As with regular formsets, it's possible to specify initial data for forms in the formset by specifying an `initial` parameter when instantiating the model formset class returned by `modelformset_factory()`. However, with model formsets, the initial values only apply to extra forms, those that aren't attached to an existing model instance. If the length of `initial` exceeds the number of extra forms, the excess initial data is ignored. If the extra forms with initial data aren't changed by the user, they won't be validated or saved.

### Saving objects in the formset

As with a `ModelForm`, you can save the data as a model object. This is done with the formset's `save()` method:

```
# Create a formset instance with POST data.
>>> formset = AuthorFormSet(request.POST)

# Assuming all is valid, save the data.
>>> instances = formset.save()
```

The `save()` method returns the instances that have been saved to the database. If a given instance's data didn't change in the bound data, the instance won't be saved to the database and won't be included in the return value (`instances`, in the above example).

When fields are missing from the form (for example because they have been excluded), these fields will not be set by the `save()` method. You can find more information about this restriction, which also holds for regular `ModelForms`, in Selecting the fields to use.

Pass `commit=False` to return the unsaved model instances:

```
# don't save to the database
>>> instances = formset.save(commit=False)
>>> for instance in instances:
...     # do something with instance
...     instance.save()
...
```

This gives you the ability to attach data to the instances before saving them to the database. If your formset contains a `ManyToManyField`, you'll also need to call `formset.save_m2m()` to ensure the many-to-many relationships are saved properly.

After calling `save()`, your model formset will have three new attributes containing the formset's changes:

`models.BaseModelFormSet.changed_objects`

`models.BaseModelFormSet.deleted_objects`

`models.BaseModelFormSet.new_objects`

### Limiting the number of editable objects

As with regular formsets, you can use the `max_num` and `extra` parameters to `modelformset_factory()` to limit the number of extra forms displayed.

`max_num` does not prevent existing objects from being displayed:

```
>>> Author.objects.order_by("name")
<QuerySet [<Author: Charles Baudelaire>, <Author: Paul Verlaine>, <Author: Walt Whitman>]>

>>> AuthorFormSet = modelformset_factory(Author, fields=["name"], max_num=1)
>>> formset = AuthorFormSet(queryset=Author.objects.order_by("name"))
>>> [x.name for x in formset.get_queryset()]
['Charles Baudelaire', 'Paul Verlaine', 'Walt Whitman']
```

Also, `extra=0` doesn't prevent creation of new model instances as you can add additional forms with JavaScript or send additional POST data. See Preventing new objects creation on how to do this.

If the value of `max_num` is greater than the number of existing related objects, 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`:

```
>>> AuthorFormSet = modelformset_factory(Author, fields=["name"], max_num=4, extra=2)
>>> formset = AuthorFormSet(queryset=Author.objects.order_by("name"))
>>> for form in formset:
...     print(form)
...
```

```
<div><label for="id_form-0-name">Name:</label><input id="id_form-0-name" type="text" name="form-0-name" value="Charles Baudelaire" maxlength="100"><input type="hidden" name="form-0-id" value="1" id="id_form-0-id"></div>
<div><label for="id_form-1-name">Name:</label><input id="id_form-1-name" type="text" name="form-1-name" value="Paul Verlaine" maxlength="100"><input type="hidden" name="form-1-id" value="3" id="id_form-1-id"></div>
<div><label for="id_form-2-name">Name:</label><input id="id_form-2-name" type="text" name="form-2-name" value="Walt Whitman" maxlength="100"><input type="hidden" name="form-2-id" value="2" id="id_form-2-id"></div>
<div><label for="id_form-3-name">Name:</label><input id="id_form-3-name" type="text" name="form-3-name" maxlength="100"><input type="hidden" name="form-3-id" id="id_form-3-id"></div>
```

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.

### Preventing new objects creation

Using the `edit_only` parameter, you can prevent creation of any new objects:

```
>>> AuthorFormSet = modelformset_factory(
...     Author,
...     fields=["name", "title"],
...     edit_only=True,
... )
```

Here, the formset will only edit existing `Author` instances. No other objects will be created or edited.

### Using a model formset in a view

Model formsets are very similar to formsets. Let's say we want to present a formset to edit `Author` model instances:

```python
from django.forms import modelformset_factory
from django.shortcuts import render
from myapp.models import Author


def manage_authors(request):
    AuthorFormSet = modelformset_factory(Author, fields=["name", "title"])
    if request.method == "POST":
        formset = AuthorFormSet(request.POST, request.FILES)
        if formset.is_valid():
            formset.save()
            # do something.
    else:
        formset = AuthorFormSet()
    return render(request, "manage_authors.html", {"formset": formset})
```

As you can see, the view logic of a model formset isn't drastically different than that of a "normal" formset. The only difference is that we call `formset.save()` to save the data into the database. (This was described above, in Saving objects in the formset.)

### Overriding `clean()` on a `ModelFormSet`

Just like with `ModelForms`, by default the `clean()` method of a `ModelFormSet` will validate that none of the items in the formset violate the unique constraints on your model (either `unique`, `unique_together` or `unique_for_date|month|year`). If you want to override the `clean()` method on a `ModelFormSet` and maintain this validation, you must call the parent class's `clean` method:

```
from django.forms import BaseModelFormSet


class MyModelFormSet(BaseModelFormSet):
    def clean(self):
        super().clean()
        # example custom validation across forms in the formset
        for form in self.forms:
            # your custom formset validation
            ...
```

Also note that by the time you reach this step, individual model instances have already been created for each `Form`. Modifying a value in `form.cleaned_data` is not sufficient to affect the saved value. If you wish to modify a value in `ModelFormSet.clean()` you must modify `form.instance`:

```
from django.forms import BaseModelFormSet


class MyModelFormSet(BaseModelFormSet):
    def clean(self):
        super().clean()

        for form in self.forms:
            name = form.cleaned_data["name"].upper()
            form.cleaned_data["name"] = name
            # update the instance value.
            form.instance.name = name
```

### Using a custom queryset

As stated earlier, you can override the default queryset used by the model formset:

```
from django.forms import modelformset_factory
from django.shortcuts import render
from myapp.models import Author


def manage_authors(request):
    AuthorFormSet = modelformset_factory(Author, fields=["name", "title"])
    queryset = Author.objects.filter(name__startswith="O")
    if request.method == "POST":
        formset = AuthorFormSet(
            request.POST,
            request.FILES,
            queryset=queryset,
        )
        if formset.is_valid():
            formset.save()
            # Do something.
    else:
        formset = AuthorFormSet(queryset=queryset)
    return render(request, "manage_authors.html", {"formset": formset})
```

Note that we pass the `queryset` argument in both the `POST` and `GET` cases in this example.

## Using the formset in the template

There are three ways to render a formset in a Django template.

First, you can let the formset do most of the work:

```
<form method="post">
    {{ formset }}
</form>
```

Second, you can manually render the formset, but let the form deal with itself:

```
<form method="post">
    {{ formset.management_form }}
    {% for form in formset %}
        {{ form }}
    {% endfor %}
</form>
```

When you manually render the forms yourself, be sure to render the management form as shown above. See the management form documentation.

Third, you can manually render each field:

```
<form method="post">
    {{ formset.management_form }}
    {% for form in formset %}
        {% for field in form %}
            {{ field.label_tag }} {{ field }}
        {% endfor %}
    {% endfor %}
</form>
```

If you opt to use this third method and you don't iterate over the fields with a `{% for %}` loop, you'll need to render the primary key field. For example, if you were rendering the `name` and `age` fields of a model:

```
<form method="post">
    {{ formset.management_form }}
    {% for form in formset %}
        {{ form.id }}
        <ul>
            <li>{{ form.name }}</li>
            <li>{{ form.age }}</li>
        </ul>
    {% endfor %}
</form>
```

Notice how we need to explicitly render `{{ form.id }}`. This ensures that the model formset, in the `POST` case, will work correctly. (This example assumes a primary key named `id`. If you've explicitly defined your own primary key that isn't called `id`, make sure it gets rendered.)

## Inline formsets

```
class models.BaseInlineFormSet
```

Inline formsets is a small abstraction layer on top of model formsets. These simplify the case of working with related objects via a foreign key. Suppose you have these two models:

```
from django.db import models


class Author(models.Model):
    name = models.CharField(max_length=100)


class Book(models.Model):
    author = models.ForeignKey(Author, on_delete=models.CASCADE)
    title = models.CharField(max_length=100)
```

If you want to create a formset that allows you to edit books belonging to a particular author, you could do this:

```
>>> from django.forms import inlineformset_factory
>>> BookFormSet = inlineformset_factory(Author, Book, fields=["title"])
>>> author = Author.objects.get(name="Mike Royko")
>>> formset = BookFormSet(instance=author)
```

`BookFormSet`'s prefix is `'book_set'` (`<model name>_set`). If `Book`'s `ForeignKey` to `Author` has a `related_name`, that's used instead.

**Note:** `inlineformset_factory()` uses `modelformset_factory()` and marks `can_delete=True` .

**See also:** Manually rendered can_delete and can_order.

**Overriding methods on an `InlineFormSet`**

When overriding methods on `InlineFormSet` , you should subclass `BaseInlineFormSet` rather than `BaseModelFormSet`.

For example, if you want to override `clean()` :

```
from django.forms import BaseInlineFormSet


class CustomInlineFormSet(BaseInlineFormSet):
    def clean(self):
        super().clean()
        # example custom validation across forms in the formset
        for form in self.forms:
            # your custom formset validation
            ...
```

See also Overriding clean() on a ModelFormSet.

Then when you create your inline formset, pass in the optional argument `formset` :

```
>>> from django.forms import inlineformset_factory
>>> BookFormSet = inlineformset_factory(
...     Author, Book, fields=["title"], formset=CustomInlineFormSet
... )
>>> author = Author.objects.get(name="Mike Royko")
>>> formset = BookFormSet(instance=author)
```

**More than one foreign key to the same model**

If your model contains more than one foreign key to the same model, you'll need to resolve the ambiguity manually using `fk_name` . For example, consider the following model:

```
class Friendship(models.Model):
    from_friend = models.ForeignKey(
        Friend,
        on_delete=models.CASCADE,
        related_name="from_friends",
    )
    to_friend = models.ForeignKey(
        Friend,
        on_delete=models.CASCADE,
        related_name="friends",
    )
    length_in_months = models.IntegerField()
```

To resolve this, you can use `fk_name` to `inlineformset_factory()`:

```
>>> FriendshipFormSet = inlineformset_factory(
...     Friend, Friendship, fk_name="from_friend", fields=["to_friend", "length_in_months"]
... )
```

### Using an inline formset in a view

You may want to provide a view that allows a user to edit the related objects of a model. Here's how you can do that:

```
def manage_books(request, author_id):
    author = Author.objects.get(pk=author_id)
    BookInlineFormSet = inlineformset_factory(Author, Book, fields=["title"])
    if request.method == "POST":
        formset = BookInlineFormSet(request.POST, request.FILES, instance=author)
        if formset.is_valid():
            formset.save()
            # Do something. Should generally end with a redirect. For example:
            return HttpResponseRedirect(author.get_absolute_url())
    else:
        formset = BookInlineFormSet(instance=author)
    return render(request, "manage_books.html", {"formset": formset})
```

Notice how we pass `instance` in both the `POST` and `GET` cases.

### Specifying widgets to use in the inline form

`inlineformset_factory` uses `modelformset_factory` and passes most of its arguments to `modelformset_factory`. This means you can use the `widgets` parameter in much the same way as passing it to `modelformset_factory`. See Specifying widgets to use in the form with widgets above.