Model field reference

This document contains all the API references of `Field` including the field options and field types Django offers.

> **See also**
>
> If the built-in fields don't do the trick, you can try django-localflavor (documentation), which contains assorted pieces of code that are useful for particular countries and cultures.
>
> Also, you can easily write your own custom model fields.

> **Note**
>
> Technically, these models are defined in `django.db.models.fields`, but for convenience they're imported into `django.db.models`; the standard convention is to use `from django.db import models` and refer to fields as `models.<Foo>Field`.

Field options

The following arguments are available to all field types. All are optional.

`null`

`Field.null`

If `True`, Django will store empty values as `NULL` in the database. Default is `False`.

Avoid using `null` on string-based fields such as `CharField` and `TextField` because empty string values will always be stored as empty strings, not as `NULL`. If a string-based field has `null=True`, that means it has two possible values for "no data": `NULL`, and the empty string. In most cases, it's redundant to have two possible values for "no data;" the Django convention is to use the empty string, not `NULL`.

For both string-based and non-string-based fields, you will also need to set `blank=True` if you wish to permit empty values in forms, as the `null` parameter only affects database storage (see `blank`).

> **Note**
>
> When using the Oracle database backend, the value `NULL` will be stored to denote the empty string regardless of this attribute.

If you want to accept `null` values with `BooleanField`, use `NullBooleanField` instead.

`blank`

`Field.blank`

If `True`, the field is allowed to be blank. Default is `False`.

Note that this is different than `null`. `null` is purely database-related, whereas `blank` is validation-related. If a field has `blank=True`, form validation will allow entry of an empty value. If a field has `blank=False`, the field will be required.

`choices`

`Field.choices`

An iterable (e.g., a list or tuple) consisting itself of iterables of exactly two items (e.g. `[(A, B), (A, B) ...]`) to use as choices for this field. If this is given, the default form widget will be a select box with these choices instead of the standard text field.

The first element in each tuple is the actual value to be set on the model, and the second element is the human-readable name. For example:

```
YEAR_IN_SCHOOL_CHOICES = (
    ('FR', 'Freshman'),
    ('SO', 'Sophomore'),
    ('JR', 'Junior'),
    ('SR', 'Senior'),
)
```

Generally, it's best to define choices inside a model class, and to define a suitably-named constant for each value:

```
from django.db import models

class Student(models.Model):
    FRESHMAN = 'FR'
    SOPHOMORE = 'SO'
    JUNIOR = 'JR'
    SENIOR = 'SR'
```

```
    YEAR_IN_SCHOOL_CHOICES = (
        (FRESHMAN, 'Freshman'),
        (SOPHOMORE, 'Sophomore'),
        (JUNIOR, 'Junior'),
        (SENIOR, 'Senior'),
    )
    year_in_school = models.CharField(
        max_length=2,
        choices=YEAR_IN_SCHOOL_CHOICES,
        default=FRESHMAN,
    )

    def is_upperclass(self):
        return self.year_in_school in (self.JUNIOR, self.SENIOR)
```

Though you can define a choices list outside of a model class and then refer to it, defining the choices and names for each choice inside the model class keeps all of that information with the class that uses it, and makes the choices easy to reference (e.g, `Student.SOPHOMORE` will work anywhere that the `Student` model has been imported).

You can also collect your available choices into named groups that can be used for organizational purposes:

```
MEDIA_CHOICES = (
    ('Audio', (
            ('vinyl', 'Vinyl'),
            ('cd', 'CD'),
        )
    ),
    ('Video', (
            ('vhs', 'VHS Tape'),
            ('dvd', 'DVD'),
        )
    ),
    ('unknown', 'Unknown'),
)
```

The first element in each tuple is the name to apply to the group. The second element is an iterable of 2-tuples, with each 2-tuple containing a value and a human-readable name for an option. Grouped options may be combined with ungrouped options within a single list (such as the unknown option in this example).

For each model field that has `choices` set, Django will add a method to retrieve the human-readable name for the field's current value. See `get_FOO_display()` in the database API documentation.

Note that choices can be any iterable object – not necessarily a list or tuple. This lets you construct choices dynamically. But if you find yourself hacking `choices` to be dynamic, you're probably better off using a proper database table with a `ForeignKey`. `choices` is meant for static data that doesn't change much, if ever.

Unless `blank=False` is set on the field along with a `default` then a label containing `"---------"` will be rendered with the select box. To override this behavior, add a tuple to `choices` containing `None`; e.g. `(None, 'Your String For Display')`. Alternatively, you can use an empty string instead of `None` where this makes sense - such as on a `CharField`.

db_column

`Field.db_column`

The name of the database column to use for this field. If this isn't given, Django will use the field's name.

If your database column name is an SQL reserved word, or contains characters that aren't allowed in Python variable names – notably, the hyphen – that's OK. Django quotes column and table names behind the scenes.

db_index

`Field.db_index`

If `True`, a database index will be created for this field.

db_tablespace

`Field.db_tablespace`

The name of the database tablespace to use for this field's index, if this field is indexed. The default is the project's `DEFAULT_INDEX_TABLESPACE` setting, if set, or the `db_tablespace` of the model, if any. If the backend doesn't support tablespaces for indexes, this option is ignored.

default

`Field.default`

The default value for the field. This can be a value or a callable object. If callable it will be called every time a new object is created.

The default can't be a mutable object (model instance, `list`, `set`, etc.), as a reference to the same instance of that object would be used as the default value in all new model instances. Instead, wrap the desired default in a callable. For example, if you want to specify a default `dict` for `JSONField`, use a function:

```
def contact_default():
    return {"email": "to1@example.com"}
```

```
contact_info = JSONField("ContactInfo", default=contact_default)
```

`lambda`s can't be used for field options like `default` because they can't be serialized by migrations. See that documentation for other caveats.

For fields like `ForeignKey` that map to model instances, defaults should be the value of the field they reference (`pk` unless `to_field` is set) instead of model instances.

The default value is used when new model instances are created and a value isn't provided for the field. When the field is a primary key, the default is also used when the field is set to `None`.

editable

`Field.editable`

If `False`, the field will not be displayed in the admin or any other `ModelForm`. They are also skipped during model validation. Default is `True`.

error_messages

`Field.error_messages`

The `error_messages` argument lets you override the default messages that the field will raise. Pass in a dictionary with keys matching the error messages you want to override.

Error message keys include `null`, `blank`, `invalid`, `invalid_choice`, `unique`, and `unique_for_date`. Additional error message keys are specified for each field in the Field types section below.

help_text

`Field.help_text`

Extra "help" text to be displayed with the form widget. It's useful for documentation even if your field isn't used on a form.

Note that this value is *not* HTML-escaped in automatically-generated forms. This lets you include HTML in `help_text` if you so desire. For example:

```
help_text="Please use the following format: <em>YYYY-MM-DD</em>."
```

Alternatively you can use plain text and `django.utils.html.escape()` to escape any HTML special characters. Ensure that you escape any help text that may come from untrusted users to avoid a cross-site scripting attack.

primary_key

`Field.primary_key`

If `True`, this field is the primary key for the model.

If you don't specify `primary_key=True` for any field in your model, Django will automatically add an `AutoField` to hold the primary key, so you don't need to set `primary_key=True` on any of your fields unless you want to override the default primary-key behavior. For more, see Automatic primary key fields.

`primary_key=True` implies `null=False` and `unique=True`. Only one primary key is allowed on an object.

The primary key field is read-only. If you change the value of the primary key on an existing object and then save it, a new object will be created alongside the old one.

unique

`Field.unique`

If `True`, this field must be unique throughout the table.

This is enforced at the database level and by model validation. If you try to save a model with a duplicate value in a `unique` field, a `django.db.IntegrityError` will be raised by the model's `save()` method.

This option is valid on all field types except `ManyToManyField`, `OneToOneField`, and `FileField`.

Note that when `unique` is `True`, you don't need to specify `db_index`, because `unique` implies the creation of an index.

unique_for_date

`Field.unique_for_date`

Set this to the name of a `DateField` or `DateTimeField` to require that this field be unique for the value of the date field.

For example, if you have a field `title` that has `unique_for_date="pub_date"`, then Django wouldn't allow the entry of two records with the same `title` and `pub_date`.

Note that if you set this to point to a `DateTimeField`, only the date portion of the field will be considered. Besides, when `USE_TZ` is `True`, the check will be performed in the current time zone at the time the object gets saved.

This is enforced by `Model.validate_unique()` during model validation but not at the database level. If any `unique_for_date` constraint involves fields that are not part of a `ModelForm` (for example, if one of the fields is listed in `exclude` or has `editable=False`), `Model.validate_unique()` will skip validation for that particular constraint.

unique_for_month

`Field.unique_for_month`

Like `unique_for_date`, but requires the field to be unique with respect to the month.

unique_for_year

`Field.unique_for_year`

Like `unique_for_date` and `unique_for_month`.

verbose_name

`Field.verbose_name`

A human-readable name for the field. If the verbose name isn't given, Django will automatically create it using the field's attribute name, converting underscores to spaces. See Verbose field names.

validators

`Field.validators`

A list of validators to run for this field. See the validators documentation for more information.

Registering and fetching lookups

`Field` implements the lookup registration API. The API can be used to customize which lookups are available for a field class, and how lookups are fetched from a field.

Field types

AutoField

class `AutoField`(***options*)**[source]**

An `IntegerField` that automatically increments according to available IDs. You usually won't need to use this directly; a primary key field will automatically be added to your model if you don't specify otherwise. See Automatic primary key fields.

BigAutoField

class `BigAutoField`(***options*)**[source]**

New in Django 1.10.

A 64-bit integer, much like an `AutoField` except that it is guaranteed to fit numbers from `1` to `9223372036854775807`.

BigIntegerField

class `BigIntegerField`(***options*)**[source]**

A 64-bit integer, much like an `IntegerField` except that it is guaranteed to fit numbers from `-9223372036854775808` to `9223372036854775807`. The default form widget for this field is a `TextInput`.

BinaryField

class `BinaryField`(***options*)**[source]**

A field to store raw binary data. It only supports `bytes` assignment. Be aware that this field has limited functionality. For example, it is not possible to filter a queryset on a `BinaryField` value. It is also not possible to include a `BinaryField` in a `ModelForm`.

> **Abusing `BinaryField`**
>
> Although you might think about storing files in the database, consider that it is bad design in 99% of the cases. This field is *not* a replacement for proper static files handling.

BooleanField

class `BooleanField`(***options*)**[source]**

A true/false field.

The default form widget for this field is a `CheckboxInput`.

If you need to accept `null` values then use `NullBooleanField` instead.

The default value of `BooleanField` is `None` when `Field.default` isn't defined.

CharField

class `CharField`(*max_length=None*, ***options*)[source]

A string field, for small- to large-sized strings.

For large amounts of text, use `TextField`.

The default form widget for this field is a `TextInput`.

`CharField` has one extra required argument:

`CharField.max_length`

The maximum length (in characters) of the field. The max_length is enforced at the database level and in Django's validation.

> **Note**
>
> If you are writing an application that must be portable to multiple database backends, you should be aware that there are restrictions on `max_length` for some backends. Refer to the database backend notes for details.

> **MySQL users**
>
> If you are using this field with MySQLdb 1.2.2 and the `utf8_bin` collation (which is *not* the default), there are some issues to be aware of. Refer to the MySQL database notes for details.

CommaSeparatedIntegerField

class `CommaSeparatedIntegerField`(*max_length=None*, ***options*)[source]

Deprecated since version 1.9: This field is deprecated in favor of `CharField` with `validators=[validate_comma_separated_integer_list]`.

A field of integers separated by commas. As in `CharField`, the `max_length` argument is required and the note about database portability mentioned there should be heeded.

DateField

class `DateField`(*auto_now=False*, *auto_now_add=False*, ***options*)[source]

A date, represented in Python by a `datetime.date` instance. Has a few extra, optional arguments:

`DateField.auto_now`

Automatically set the field to now every time the object is saved. Useful for "last-modified" timestamps. Note that the current date is *always* used; it's not just a default value that you can override.

The field is only automatically updated when calling `Model.save()`. The field isn't updated when making updates to other fields in other ways such as `QuerySet.update()`, though you can specify a custom value for the field in an update like that.

`DateField.auto_now_add`

Automatically set the field to now when the object is first created. Useful for creation of timestamps. Note that the current date is *always* used; it's not just a default value that you can override. So even if you set a value for this field when creating the object, it will be ignored. If you want to be able to modify this field, set the following instead of `auto_now_add=True`:

- For `DateField`: `default=date.today` - from `datetime.date.today()`
- For `DateTimeField`: `default=timezone.now` - from `django.utils.timezone.now()`

The default form widget for this field is a `TextInput`. The admin adds a JavaScript calendar, and a shortcut for "Today". Includes an additional `invalid_date` error message key.

The options `auto_now_add`, `auto_now`, and `default` are mutually exclusive. Any combination of these options will result in an error.

> **Note**
>
> As currently implemented, setting `auto_now` or `auto_now_add` to `True` will cause the field to have `editable=False` and `blank=True` set.

> **Note**
>
> The `auto_now` and `auto_now_add` options will always use the date in the default timezone at the moment of creation or update. If you need something different, you may want to consider simply using your own callable default or overriding `save()` instead of using `auto_now` or `auto_now_add`; or using

a `DateTimeField` instead of a `DateField` and deciding how to handle the conversion from datetime to date at display time.

DateTimeField

class `DateTimeField`**(***auto_now=False*, *auto_now_add=False*, ***options*)**[source]**

A date and time, represented in Python by a `datetime.datetime` instance. Takes the same extra arguments as `DateField`.

The default form widget for this field is a single `TextInput`. The admin uses two separate `TextInput` widgets with JavaScript shortcuts.

DecimalField

class `DecimalField`**(***max_digits=None*, *decimal_places=None*, ***options*)**[source]**

A fixed-precision decimal number, represented in Python by a `Decimal` instance. Has two **required** arguments:

DecimalField.max_digits

The maximum number of digits allowed in the number. Note that this number must be greater than or equal to `decimal_places`.

DecimalField.decimal_places

The number of decimal places to store with the number.

For example, to store numbers up to `999` with a resolution of 2 decimal places, you'd use:

```
models.DecimalField(..., max_digits=5, decimal_places=2)
```

And to store numbers up to approximately one billion with a resolution of 10 decimal places:

```
models.DecimalField(..., max_digits=19, decimal_places=10)
```

The default form widget for this field is a `NumberInput` when `localize` is `False` or `TextInput` otherwise.

Note

For more information about the differences between the `FloatField` and `DecimalField` classes, please see FloatField vs. DecimalField.

DurationField

class `DurationField`**(*****options*)**[source]**

A field for storing periods of time - modeled in Python by `timedelta`. When used on PostgreSQL, the data type used is an `interval` and on Oracle the data type is `INTERVAL DAY(9) TO SECOND(6)`. Otherwise a `bigint` of microseconds is used.

Note

Arithmetic with `DurationField` works in most cases. However on all databases other than PostgreSQL, comparing the value of a `DurationField` to arithmetic on `DateTimeField` instances will not work as expected.

EmailField

class `EmailField`**(***max_length=254*, ***options*)**[source]**

A `CharField` that checks that the value is a valid email address. It uses `EmailValidator` to validate the input.

FileField

class `FileField`**(***upload_to=None*, *max_length=100*, ***options*)**[source]**

A file-upload field.

Note

The `primary_key` and `unique` arguments are not supported, and will raise a `TypeError` if used.

Has two optional arguments:

FileField.upload_to

This attribute provides a way of setting the upload directory and file name, and can be set in two ways. In both cases, the value is passed to the `Storage.save()` method.

If you specify a string value, it may contain `strftime()` formatting, which will be replaced by the date/time of the file upload (so that uploaded files don't fill up the given directory). For example:

```
class MyModel(models.Model):
    # file will be uploaded to MEDIA_ROOT/uploads
    upload = models.FileField(upload_to='uploads/')
    # or...
    # file will be saved to MEDIA_ROOT/uploads/2015/01/30
    upload = models.FileField(upload_to='uploads/%Y/%m/%d/')
```

If you are using the default `FileSystemStorage`, the string value will be appended to your `MEDIA_ROOT` path to form the location on the local filesystem where uploaded files will be stored. If you are using a different storage, check that storage's documentation to see how it handles `upload_to`.

`upload_to` may also be a callable, such as a function. This will be called to obtain the upload path, including the filename. This callable must accept two arguments and return a Unix-style path (with forward slashes) to be passed along to the storage system. The two arguments are:

Argument	Description
`instance`	An instance of the model where the `FileField` is defined. More specifically, this is the particular instance where the current file is being attached. In most cases, this object will not have been saved to the database yet, so if it uses the default `AutoField`, *it might not yet have a value for its primary key field*.
`filename`	The filename that was originally given to the file. This may or may not be taken into account when determining the final destination path.

For example:

```
def user_directory_path(instance, filename):
    # file will be uploaded to MEDIA_ROOT/user_<id>/<filename>
    return 'user_{0}/{1}'.format(instance.user.id, filename)

class MyModel(models.Model):
    upload = models.FileField(upload_to=user_directory_path)
```

`FileField.storage`

A storage object, which handles the storage and retrieval of your files. See Managing files for details on how to provide this object.

The default form widget for this field is a `ClearableFileInput`.

Using a `FileField` or an `ImageField` (see below) in a model takes a few steps:

1. In your settings file, you'll need to define `MEDIA_ROOT` as the full path to a directory where you'd like Django to store uploaded files. (For performance, these files are not stored in the database.) Define `MEDIA_URL` as the base public URL of that directory. Make sure that this directory is writable by the Web server's user account.
2. Add the `FileField` or `ImageField` to your model, defining the `upload_to` option to specify a subdirectory of `MEDIA_ROOT` to use for uploaded files.
3. All that will be stored in your database is a path to the file (relative to `MEDIA_ROOT`). You'll most likely want to use the convenience `url` attribute provided by Django. For example, if your `ImageField` is called `mug_shot`, you can get the absolute path to your image in a template with `{{ object.mug_shot.url }}`.

For example, say your `MEDIA_ROOT` is set to `'/home/media'`, and `upload_to` is set to `'photos/%Y/%m/%d'`. The `'%Y/%m/%d'` part of `upload_to` is `strftime()` formatting; `'%Y'` is the four-digit year, `'%m'` is the two-digit month and `'%d'` is the two-digit day. If you upload a file on Jan. 15, 2007, it will be saved in the directory `/home/media/photos/2007/01/15`.

If you wanted to retrieve the uploaded file's on-disk filename, or the file's size, you could use the `name` and `size` attributes respectively; for more information on the available attributes and methods, see the `File` class reference and the Managing files topic guide.

> **Note**
>
> The file is saved as part of saving the model in the database, so the actual file name used on disk cannot be relied on until after the model has been saved.

The uploaded file's relative URL can be obtained using the `url` attribute. Internally, this calls the `url()` method of the underlying `Storage` class.

Note that whenever you deal with uploaded files, you should pay close attention to where you're uploading them and what type of files they are, to avoid security holes. *Validate all uploaded files* so that you're sure the files are what you think they are. For example, if you blindly let somebody upload files, without validation, to a directory that's within your Web server's document root, then somebody could upload a CGI or PHP script and execute that script by visiting its URL on your site. Don't allow that.

Also note that even an uploaded HTML file, since it can be executed by the browser (though not by the server), can pose security threats that are equivalent to XSS or CSRF attacks.

`FileField` instances are created in your database as `varchar` columns with a default max length of 100 characters. As with other fields, you can change the maximum length using the `max_length` argument.

`FileField` and `FieldFile`

class `FieldFile`**[source]**

When you access a `FileField` on a model, you are given an instance of `FieldFile` as a proxy for accessing the underlying file.

The API of `FieldFile` mirrors that of `File`, with one key difference: *The object wrapped by the class is not necessarily a wrapper around Python's built-in file object.* Instead, it is a wrapper around the result of the `Storage.open()` method, which may be a `File` object, or it may be a custom storage's implementation of the `File` API.

In addition to the API inherited from `File` such as `read()` and `write()`, `FieldFile` includes several methods that can be used to interact with the underlying file:

> **Warning**
>
> Two methods of this class, `save()` and `delete()`, default to saving the model object of the associated `FieldFile` in the database.

`FieldFile.name`

The name of the file including the relative path from the root of the `Storage` of the associated `FileField`.

`FieldFile.size`

The result of the underlying `Storage.size()` method.

`FieldFile.url`

A read-only property to access the file's relative URL by calling the `url()` method of the underlying `Storage` class.

`FieldFile.open`**(***mode='rb'***)**[source]

Opens or reopens the file associated with this instance in the specified `mode`. Unlike the standard Python `open()` method, it doesn't return a file descriptor.

Since the underlying file is opened implicitly when accessing it, it may be unnecessary to call this method except to reset the pointer to the underlying file or to change the `mode`.

`FieldFile.close`**()**[source]

Behaves like the standard Python `file.close()` method and closes the file associated with this instance.

`FieldFile.save`**(***name***,** *content***,** *save=True***)**[source]

This method takes a filename and file contents and passes them to the storage class for the field, then associates the stored file with the model field. If you want to manually associate file data with `FileField` instances on your model, the `save()` method is used to persist that file data.

Takes two required arguments: `name` which is the name of the file, and `content` which is an object containing the file's contents. The optional `save` argument controls whether or not the model instance is saved after the file associated with this field has been altered. Defaults to `True`.

Note that the `content` argument should be an instance of `django.core.files.File`, not Python's built-in file object. You can construct a `File` from an existing Python file object like this:

```
from django.core.files import File
# Open an existing file using Python's built-in open()
f = open('/path/to/hello.world')
myfile = File(f)
```

Or you can construct one from a Python string like this:

```
from django.core.files.base import ContentFile
myfile = ContentFile("hello world")
```

For more information, see Managing files.

`FieldFile.delete`**(***save=True***)**[source]

Deletes the file associated with this instance and clears all attributes on the field. Note: This method will close the file if it happens to be open when `delete()` is called.

The optional `save` argument controls whether or not the model instance is saved after the file associated with this field has been deleted. Defaults to `True`.

Note that when a model is deleted, related files are not deleted. If you need to cleanup orphaned files, you'll need to handle it yourself (for instance, with a custom management command that can be run manually or scheduled to run periodically via e.g. cron).

`FilePathField`

class `FilePathField`**(***path=None***,** *match=None***,** *recursive=False***,** *max_length=100***,** ***options***)**[source]

A `CharField` whose choices are limited to the filenames in a certain directory on the filesystem. Has three special arguments, of which the first is **required**:

`FilePathField.path`

Required. The absolute filesystem path to a directory from which this `FilePathField` should get its choices. Example: `"/home/images"`.

`FilePathField.match`

Optional. A regular expression, as a string, that `FilePathField` will use to filter filenames. Note that the regex will be applied to the base filename, not the full path. Example: `"foo.*\.txt$"`, which will match a file called `foo23.txt` but not `bar.txt` or `foo23.png`.

`FilePathField.recursive`

Optional. Either `True` or `False`. Default is `False`. Specifies whether all subdirectories of `path` should be included

`FilePathField.allow_files`

Optional. Either `True` or `False`. Default is `True`. Specifies whether files in the specified location should be included. Either this or `allow_folders` must be `True`.

`FilePathField.allow_folders`

Optional. Either `True` or `False`. Default is `False`. Specifies whether folders in the specified location should be included. Either this or `allow_files` must be `True`.

Of course, these arguments can be used together.

The one potential gotcha is that `match` applies to the base filename, not the full path. So, this example:

```
FilePathField(path="/home/images", match="foo.*", recursive=True)
```

...will match `/home/images/foo.png` but not `/home/images/foo/bar.png` because the `match` applies to the base filename (`foo.png` and `bar.png`).

`FilePathField` instances are created in your database as `varchar` columns with a default max length of 100 characters. As with other fields, you can change the maximum length using the `max_length` argument.

FloatField

class `FloatField`**(*****options***)[source]**

A floating-point number represented in Python by a `float` instance.

The default form widget for this field is a `NumberInput` when `localize` is `False` or `TextInput` otherwise.

> `FloatField` **vs.** `DecimalField`
>
> The `FloatField` class is sometimes mixed up with the `DecimalField` class. Although they both represent real numbers, they represent those numbers differently. `FloatField` uses Python's `float` type internally, while `DecimalField` uses Python's `Decimal` type. For information on the difference between the two, see Python's documentation for the `decimal` module.

ImageField

class `ImageField`**(***upload_to=None***,** *height_field=None***,** *width_field=None***,** *max_length=100***,** ***options***) [source]**

Inherits all attributes and methods from `FileField`, but also validates that the uploaded object is a valid image.

In addition to the special attributes that are available for `FileField`, an `ImageField` also has `height` and `width` attributes.

To facilitate querying on those attributes, `ImageField` has two extra optional arguments:

`ImageField.height_field`

Name of a model field which will be auto-populated with the height of the image each time the model instance is saved.

`ImageField.width_field`

Name of a model field which will be auto-populated with the width of the image each time the model instance is saved.

Requires the Pillow library.

`ImageField` instances are created in your database as `varchar` columns with a default max length of 100 characters. As with other fields, you can change the maximum length using the `max_length` argument.

The default form widget for this field is a `ClearableFileInput`.

IntegerField

class `IntegerField`(***options*)[source]

An integer. Values from `-2147483648` to `2147483647` are safe in all databases supported by Django. The default form widget for this field is a `NumberInput` when `localize` is `False` or `TextInput` otherwise.

GenericIPAddressField

class `GenericIPAddressField`(*protocol='both'*, *unpack_ipv4=False*, ***options*)[source]

An IPv4 or IPv6 address, in string format (e.g. `192.0.2.30` or `2a02:42fe::4`). The default form widget for this field is a `TextInput`.

The IPv6 address normalization follows **RFC 4291#section-2.2** section 2.2, including using the IPv4 format suggested in paragraph 3 of that section, like `::ffff:192.0.2.0`. For example, `2001:0::0:01` would be normalized to `2001::1`, and `::ffff:0a0a:0a0a` to `::ffff:10.10.10.10`. All characters are converted to lowercase.

`GenericIPAddressField.protocol`

Limits valid inputs to the specified protocol. Accepted values are `'both'` (default), `'IPv4'` or `'IPv6'`. Matching is case insensitive.

`GenericIPAddressField.unpack_ipv4`

Unpacks IPv4 mapped addresses like `::ffff:192.0.2.1`. If this option is enabled that address would be unpacked to `192.0.2.1`. Default is disabled. Can only be used when `protocol` is set to `'both'`.

If you allow for blank values, you have to allow for null values since blank values are stored as null.

NullBooleanField

class `NullBooleanField`(***options*)[source]

Like a `BooleanField`, but allows `NULL` as one of the options. Use this instead of a `BooleanField` with `null=True`. The default form widget for this field is a `NullBooleanSelect`.

PositiveIntegerField

class `PositiveIntegerField`(***options*)[source]

Like an `IntegerField`, but must be either positive or zero (`0`). Values from `0` to `2147483647` are safe in all databases supported by Django. The value `0` is accepted for backward compatibility reasons.

PositiveSmallIntegerField

class `PositiveSmallIntegerField`(***options*)[source]

Like a `PositiveIntegerField`, but only allows values under a certain (database-dependent) point. Values from `0` to `32767` are safe in all databases supported by Django.

SlugField

class `SlugField`(*max_length=50*, ***options*)[source]

Slug is a newspaper term. A slug is a short label for something, containing only letters, numbers, underscores or hyphens. They're generally used in URLs.

Like a CharField, you can specify `max_length` (read the note about database portability and `max_length` in that section, too). If `max_length` is not specified, Django will use a default length of 50.

Implies setting `Field.db_index` to `True`.

It is often useful to automatically prepopulate a SlugField based on the value of some other value. You can do this automatically in the admin using `prepopulated_fields`.

`SlugField.allow_unicode`

New in Django 1.9.

If `True`, the field accepts Unicode letters in addition to ASCII letters. Defaults to `False`.

SmallIntegerField

class `SmallIntegerField`(***options*)[source]

Like an `IntegerField`, but only allows values under a certain (database-dependent) point. Values from `-32768` to `32767` are safe in all databases supported by Django.

TextField

class `TextField`(***options*)[source]

A large text field. The default form widget for this field is a `Textarea`.

If you specify a `max_length` attribute, it will be reflected in the `Textarea` widget of the auto-generated form field. However it is not enforced at the model or database level. Use a `CharField` for that.

> **MySQL users**
>
> If you are using this field with MySQLdb 1.2.1p2 and the `utf8_bin` collation (which is *not* the default), there are some issues to be aware of. Refer to the MySQL database notes for details.

TimeField

class `TimeField`**(***auto_now=False*, *auto_now_add=False*, ***options*)**[source]**

A time, represented in Python by a `datetime.time` instance. Accepts the same auto-population options as `DateField`.

The default form widget for this field is a `TextInput`. The admin adds some JavaScript shortcuts.

URLField

class `URLField`**(***max_length=200*, ***options*)**[source]**

A `CharField` for a URL.

The default form widget for this field is a `TextInput`.

Like all `CharField` subclasses, `URLField` takes the optional `max_length` argument. If you don't specify `max_length`, a default of 200 is used.

UUIDField

class `UUIDField`**(*****options*)**[source]**

A field for storing universally unique identifiers. Uses Python's `UUID` class. When used on PostgreSQL, this stores in a `uuid` datatype, otherwise in a `char(32)`.

Universally unique identifiers are a good alternative to `AutoField` for `primary_key`. The database will not generate the UUID for you, so it is recommended to use `default`:

```
import uuid
from django.db import models

class MyUUIDModel(models.Model):
    id = models.UUIDField(primary_key=True, default=uuid.uuid4, editable=False)
    # other fields
```

Note that a callable (with the parentheses omitted) is passed to `default`, not an instance of `UUID`.

Relationship fields

Django also defines a set of fields that represent relations.

ForeignKey

class `ForeignKey`**(***othermodel*, *on_delete*, ***options*)**[source]**

A many-to-one relationship. Requires a positional argument: the class to which the model is related.

Changed in Django 1.9:

`on_delete` can now be used as the second positional argument (previously it was typically only passed as a keyword argument). It will be a required argument in Django 2.0.

To create a recursive relationship – an object that has a many-to-one relationship with itself – use `models.ForeignKey('self', on_delete=models.CASCADE)`.

If you need to create a relationship on a model that has not yet been defined, you can use the name of the model, rather than the model object itself:

```
from django.db import models

class Car(models.Model):
    manufacturer = models.ForeignKey(
        'Manufacturer',
        on_delete=models.CASCADE,
    )
    # ...

class Manufacturer(models.Model):
    # ...
    pass
```

Relationships defined this way on abstract models are resolved when the model is subclassed as a concrete model and are not relative to the abstract model's `app_label`:

products/models.py

```
from django.db import models

class AbstractCar(models.Model):
    manufacturer = models.ForeignKey('Manufacturer', on_delete=models.CASCADE)

    class Meta:
        abstract = True
```

production/models.py

```
from django.db import models
from products.models import AbstractCar

class Manufacturer(models.Model):
    pass

class Car(AbstractCar):
    pass

# Car.manufacturer will point to `production.Manufacturer` here.
```

To refer to models defined in another application, you can explicitly specify a model with the full application label. For example, if the `Manufacturer` model above is defined in another application called `production`, you'd need to use:

```
class Car(models.Model):
    manufacturer = models.ForeignKey(
        'production.Manufacturer',
        on_delete=models.CASCADE,
    )
```

This sort of reference can be useful when resolving circular import dependencies between two applications.

A database index is automatically created on the `ForeignKey`. You can disable this by setting `db_index` to `False`. You may want to avoid the overhead of an index if you are creating a foreign key for consistency rather than joins, or if you will be creating an alternative index like a partial or multiple column index.

Database Representation

Behind the scenes, Django appends `"_id"` to the field name to create its database column name. In the above example, the database table for the `Car` model will have a `manufacturer_id` column. (You can change this explicitly by specifying `db_column`) However, your code should never have to deal with the database column name, unless you write custom SQL. You'll always deal with the field names of your model object.

Arguments

`ForeignKey` accepts other arguments that define the details of how the relation works.

`ForeignKey.on_delete`

When an object referenced by a `ForeignKey` is deleted, Django will emulate the behavior of the SQL constraint specified by the `on_delete` argument. For example, if you have a nullable `ForeignKey` and you want it to be set null when the referenced object is deleted:

```
user = models.ForeignKey(
    User,
    models.SET_NULL,
    blank=True,
    null=True,
)
```

Deprecated since version 1.9: `on_delete` will become a required argument in Django 2.0. In older versions it defaults to `CASCADE`.

The possible values for `on_delete` are found in `django.db.models`:

- `CASCADE`**[source]**

  Cascade deletes. Django emulates the behavior of the SQL constraint ON DELETE CASCADE and also deletes the object containing the ForeignKey.

- `PROTECT`**[source]**

  Prevent deletion of the referenced object by raising `ProtectedError`, a subclass of `django.db.IntegrityError`.

- `SET_NULL`**[source]**

  Set the `ForeignKey` null; this is only possible if `null` is `True`.

- `SET_DEFAULT`**[source]**

  Set the `ForeignKey` to its default value; a default for the `ForeignKey` must be set.

- `SET()`**[source]**

  Set the `ForeignKey` to the value passed to `SET()`, or if a callable is passed in, the result of calling it. In most cases, passing a callable will be necessary to avoid executing queries at the time your models.py is imported:

```
from django.conf import settings
from django.contrib.auth import get_user_model
from django.db import models

def get_sentinel_user():
    return get_user_model().objects.get_or_create(username='deleted')[0]

class MyModel(models.Model):
    user = models.ForeignKey(
        settings.AUTH_USER_MODEL,
        on_delete=models.SET(get_sentinel_user),
    )
```

- DO_NOTHING[source]

  Take no action. If your database backend enforces referential integrity, this will cause an `IntegrityError` unless you manually add an SQL `ON DELETE` constraint to the database field.

ForeignKey.limit_choices_to

Sets a limit to the available choices for this field when this field is rendered using a `ModelForm` or the admin (by default, all objects in the queryset are available to choose). Either a dictionary, a `Q` object, or a callable returning a dictionary or `Q` object can be used.

For example:

```
staff_member = models.ForeignKey(
    User,
    on_delete=models.CASCADE,
    limit_choices_to={'is_staff': True},
)
```

causes the corresponding field on the `ModelForm` to list only `Users` that have `is_staff=True`. This may be helpful in the Django admin.

The callable form can be helpful, for instance, when used in conjunction with the Python `datetime` module to limit selections by date range. For example:

```
def limit_pub_date_choices():
    return {'pub_date__lte': datetime.date.utcnow()}

limit_choices_to = limit_pub_date_choices
```

If `limit_choices_to` is or returns a `Q object`, which is useful for complex queries, then it will only have an effect on the choices available in the admin when the field is not listed in `raw_id_fields` in the `ModelAdmin` for the model.

> **Note**
>
> If a callable is used for `limit_choices_to`, it will be invoked every time a new form is instantiated. It may also be invoked when a model is validated, for example by management commands or the admin. The admin constructs querysets to validate its form inputs in various edge cases multiple times, so there is a possibility your callable may be invoked several times.

ForeignKey.related_name

The name to use for the relation from the related object back to this one. It's also the default value for `related_query_name` (the name to use for the reverse filter name from the target model). See the related objects documentation for a full explanation and example. Note that you must set this value when defining relations on abstract models; and when you do so some special syntax is available.

If you'd prefer Django not to create a backwards relation, set `related_name` to `'+'` or end it with `'+'`. For example, this will ensure that the `User` model won't have a backwards relation to this model:

```
user = models.ForeignKey(
    User,
    on_delete=models.CASCADE,
    related_name='+',
)
```

ForeignKey.related_query_name

The name to use for the reverse filter name from the target model. It defaults to the value of `related_name` or `default_related_name` if set, otherwise it defaults to the name of the model:

```
# Declare the ForeignKey with related_query_name
class Tag(models.Model):
    article = models.ForeignKey(
        Article,
        on_delete=models.CASCADE,
        related_name="tags",
        related_query_name="tag",
    )
    name = models.CharField(max_length=255)

# That's now the name of the reverse filter
Article.objects.filter(tag__name="important")
```

Like `related_name`, `related_query_name` supports app label and class interpolation via some special syntax.

`ForeignKey.to_field`

The field on the related object that the relation is to. By default, Django uses the primary key of the related object. If you reference a different field, that field must have `unique=True`.

`ForeignKey.db_constraint`

Controls whether or not a constraint should be created in the database for this foreign key. The default is `True`, and that's almost certainly what you want; setting this to `False` can be very bad for data integrity. That said, here are some scenarios where you might want to do this:

- You have legacy data that is not valid.
- You're sharding your database.

If this is set to `False`, accessing a related object that doesn't exist will raise its `DoesNotExist` exception.

`ForeignKey.swappable`

Controls the migration framework's reaction if this `ForeignKey` is pointing at a swappable model. If it is `True` - the default - then if the `ForeignKey` is pointing at a model which matches the current value of `settings.AUTH_USER_MODEL` (or another swappable model setting) the relationship will be stored in the migration using a reference to the setting, not to the model directly.

You only want to override this to be `False` if you are sure your model should always point towards the swapped-in model - for example, if it is a profile model designed specifically for your custom user model.

Setting it to `False` does not mean you can reference a swappable model even if it is swapped out - `False` just means that the migrations made with this ForeignKey will always reference the exact model you specify (so it will fail hard if the user tries to run with a User model you don't support, for example).

If in doubt, leave it to its default of `True`.

ManyToManyField

class `ManyToManyField`**(***othermodel***,** ***options***)[source]**

A many-to-many relationship. Requires a positional argument: the class to which the model is related, which works exactly the same as it does for `ForeignKey`, including recursive and lazy relationships.

Related objects can be added, removed, or created with the field's `RelatedManager`.

Database Representation

Behind the scenes, Django creates an intermediary join table to represent the many-to-many relationship. By default, this table name is generated using the name of the many-to-many field and the name of the table for the model that contains it. Since some databases don't support table names above a certain length, these table names will be automatically truncated to 64 characters and a uniqueness hash will be used. This means you might see table names like `author_books_9cdf4`; this is perfectly normal. You can manually provide the name of the join table using the `db_table` option.

Arguments

`ManyToManyField` accepts an extra set of arguments – all optional – that control how the relationship functions.

`ManyToManyField.related_name`

Same as `ForeignKey.related_name`.

`ManyToManyField.related_query_name`

Same as `ForeignKey.related_query_name`.

`ManyToManyField.limit_choices_to`

Same as `ForeignKey.limit_choices_to`.

`limit_choices_to` has no effect when used on a `ManyToManyField` with a custom intermediate table specified using the `through` parameter.

`ManyToManyField.symmetrical`

Only used in the definition of ManyToManyFields on self. Consider the following model:

```
from django.db import models

class Person(models.Model):
    friends = models.ManyToManyField("self")
```

When Django processes this model, it identifies that it has a `ManyToManyField` on itself, and as a result, it doesn't add a `person_set` attribute to the `Person` class. Instead, the `ManyToManyField` is assumed to be symmetrical – that is, if I am your friend, then you are my friend.

If you do not want symmetry in many-to-many relationships with `self`, set `symmetrical` to `False`. This will force Django to add the descriptor for the reverse relationship, allowing `ManyToManyField` relationships to be non-symmetrical.

`ManyToManyField.through`

Django will automatically generate a table to manage many-to-many relationships. However, if you want to manually specify the intermediary table, you can use the `through` option to specify the Django model that represents the intermediate table that you want to use.

The most common use for this option is when you want to associate extra data with a many-to-many relationship.

If you don't specify an explicit `through` model, there is still an implicit `through` model class you can use to directly access the table created to hold the association. It has three fields to link the models.

If the source and target models differ, the following fields are generated:

- `id`: the primary key of the relation.
- `<containing_model>_id`: the `id` of the model that declares the `ManyToManyField`.
- `<other_model>_id`: the `id` of the model that the `ManyToManyField` points to.

If the `ManyToManyField` points from and to the same model, the following fields are generated:

- `id`: the primary key of the relation.
- `from_<model>_id`: the `id` of the instance which points at the model (i.e. the source instance).
- `to_<model>_id`: the `id` of the instance to which the relationship points (i.e. the target model instance).

This class can be used to query associated records for a given model instance like a normal model.

`ManyToManyField.through_fields`

Only used when a custom intermediary model is specified. Django will normally determine which fields of the intermediary model to use in order to establish a many-to-many relationship automatically. However, consider the following models:

```
from django.db import models

class Person(models.Model):
    name = models.CharField(max_length=50)

class Group(models.Model):
    name = models.CharField(max_length=128)
    members = models.ManyToManyField(
        Person,
        through='Membership',
        through_fields=('group', 'person'),
    )

class Membership(models.Model):
    group = models.ForeignKey(Group, on_delete=models.CASCADE)
    person = models.ForeignKey(Person, on_delete=models.CASCADE)
    inviter = models.ForeignKey(
        Person,
        on_delete=models.CASCADE,
        related_name="membership_invites",
    )
    invite_reason = models.CharField(max_length=64)
```

`Membership` has *two* foreign keys to `Person` (`person` and `inviter`), which makes the relationship ambiguous and Django can't know which one to use. In this case, you must explicitly specify which foreign keys Django should use using `through_fields`, as in the example above.

`through_fields` accepts a 2-tuple `('field1', 'field2')`, where `field1` is the name of the foreign key to the model the `ManyToManyField` is defined on (`group` in this case), and `field2` the name of the foreign key to the target model (`person` in this case).

When you have more than one foreign key on an intermediary model to any (or even both) of the models participating in a many-to-many relationship, you *must* specify `through_fields`. This also applies to recursive relationships when an intermediary model is used and there are more than two foreign keys to the model, or you want to explicitly specify which two Django should use.

Recursive relationships using an intermediary model are always defined as non-symmetrical – that is, with `symmetrical=False` – therefore, there is the concept of a "source" and a "target". In that case `'field1'` will be treated as the "source" of the relationship and `'field2'` as the "target".

`ManyToManyField.db_table`

The name of the table to create for storing the many-to-many data. If this is not provided, Django will assume a default name based upon the names of: the table for the model defining the relationship and the name of the field itself.

`ManyToManyField.db_constraint`

Controls whether or not constraints should be created in the database for the foreign keys in the intermediary table. The default is `True`, and that's almost certainly what you want; setting this to `False` can be very bad for data integrity. That said, here are some scenarios where you might want to do this:

- You have legacy data that is not valid.
- You're sharding your database.

It is an error to pass both `db_constraint` and `through`.

`ManyToManyField.swappable`

Controls the migration framework's reaction if this `ManyToManyField` is pointing at a swappable model. If it is `True` - the default - then if the `ManyToManyField` is pointing at a model which matches the current value of `settings.AUTH_USER_MODEL` (or another swappable model setting) the relationship will be stored in the migration using a reference to the setting, not to the model directly.

You only want to override this to be `False` if you are sure your model should always point towards the swapped-in model - for example, if it is a profile model designed specifically for your custom user model.

If in doubt, leave it to its default of `True`.

`ManyToManyField` does not support `validators`.

`null` has no effect since there is no way to require a relationship at the database level.

OneToOneField

class `OneToOneField`**(***othermodel*, *on_delete*, *parent_link=False*, ***options*)**[source]**

A one-to-one relationship. Conceptually, this is similar to a `ForeignKey` with `unique=True`, but the "reverse" side of the relation will directly return a single object.

Changed in Django 1.9:

`on_delete` can now be used as the second positional argument (previously it was typically only passed as a keyword argument). It will be a required argument in Django 2.0.

This is most useful as the primary key of a model which "extends" another model in some way; Multi-table inheritance is implemented by adding an implicit one-to-one relation from the child model to the parent model, for example.

One positional argument is required: the class to which the model will be related. This works exactly the same as it does for `ForeignKey`, including all the options regarding recursive and lazy relationships.

If you do not specify the `related_name` argument for the `OneToOneField`, Django will use the lower-case name of the current model as default value.

With the following example:

```
from django.conf import settings
from django.db import models

class MySpecialUser(models.Model):
    user = models.OneToOneField(
        settings.AUTH_USER_MODEL,
        on_delete=models.CASCADE,
    )
    supervisor = models.OneToOneField(
        settings.AUTH_USER_MODEL,
        on_delete=models.CASCADE,
        related_name='supervisor_of',
    )
```

your resulting `User` model will have the following attributes:

```
>>> user = User.objects.get(pk=1)
>>> hasattr(user, 'myspecialuser')
True
>>> hasattr(user, 'supervisor_of')
True
```

A `DoesNotExist` exception is raised when accessing the reverse relationship if an entry in the related table doesn't exist. For example, if a user doesn't have a supervisor designated by `MySpecialUser`:

```
>>> user.supervisor_of
Traceback (most recent call last):
    ...
DoesNotExist: User matching query does not exist.
```

Additionally, `OneToOneField` accepts all of the extra arguments accepted by `ForeignKey`, plus one extra argument:

`OneToOneField.parent_link`

When `True` and used in a model which inherits from another concrete model, indicates that this field should be used as the link back to the parent class, rather than the extra `OneToOneField` which would normally be implicitly created by subclassing.

See One-to-one relationships for usage examples of `OneToOneField`.

Field API reference

class `Field`**[source]**

`Field` is an abstract class that represents a database table column. Django uses fields to create the database table (`db_type()`), to map Python types to database (`get_prep_value()`) and vice-versa (`from_db_value()`).

A field is thus a fundamental piece in different Django APIs, notably, `models` and `querysets`.

In models, a field is instantiated as a class attribute and represents a particular table column, see Models. It has attributes such as `null` and `unique`, and methods that Django uses to map the field value to database-specific values.

A `Field` is a subclass of `RegisterLookupMixin` and thus both `Transform` and `Lookup` can be registered on it to be used in `QuerySet`s (e.g. `field_name__exact="foo"`). All built-in lookups are registered by default.

All of Django's built-in fields, such as `CharField`, are particular implementations of `Field`. If you need a custom field, you can either subclass any of the built-in fields or write a `Field` from scratch. In either case, see Writing custom model fields.

`description`

A verbose description of the field, e.g. for the `django.contrib.admindocs` application.

The description can be of the form:

```
description = _("String (up to %(max_length)s)")
```

where the arguments are interpolated from the field's `__dict__`.

To map a `Field` to a database-specific type, Django exposes several methods:

`get_internal_type()`**[source]**

Returns a string naming this field for backend specific purposes. By default, it returns the class name.

See Emulating built-in field types for usage in custom fields.

`db_type(`*connection*`)`**[source]**

Returns the database column data type for the `Field`, taking into account the `connection`.

See Custom database types for usage in custom fields.

`rel_db_type(`*connection*`)`**[source]**

New in Django 1.10.

Returns the database column data type for fields such as `ForeignKey` and `OneToOneField` that point to the `Field`, taking into account the `connection`.

See Custom database types for usage in custom fields.

There are three main situations where Django needs to interact with the database backend and fields:

- when it queries the database (Python value -> database backend value)
- when it loads data from the database (database backend value -> Python value)
- when it saves to the database (Python value -> database backend value)

When querying, `get_db_prep_value()` and `get_prep_value()` are used:

`get_prep_value(`*value*`)`**[source]**

`value` is the current value of the model's attribute, and the method should return data in a format that has been prepared for use as a parameter in a query.

See Converting Python objects to query values for usage.

`get_db_prep_value(`*value*`,` *connection*`,` *prepared=False*`)`**[source]**

Converts `value` to a backend-specific value. By default it returns `value` if `prepared=True` and `get_prep_value()` if is `False`.

See Converting query values to database values for usage.

When loading data, `from_db_value()` is used:

`from_db_value(`*value*`,` *expression*`,` *connection*`,` *context*`)`

Converts a value as returned by the database to a Python object. It is the reverse of `get_prep_value()`.

This method is not used for most built-in fields as the database backend already returns the correct Python type, or the backend itself does the conversion.

See Converting values to Python objects for usage.

> **Note**

> For performance reasons, `from_db_value` is not implemented as a no-op on fields which do not require it (all Django fields). Consequently you may not call `super` in your definition.

When saving, `pre_save()` and `get_db_prep_save()` are used:

`get_db_prep_save`(*value*, *connection*)[source]

Same as the `get_db_prep_value()`, but called when the field value must be *saved* to the database. By default returns `get_db_prep_value()`.

`pre_save`(*model_instance*, *add*)[source]

Method called prior to `get_db_prep_save()` to prepare the value before being saved (e.g. for `DateField.auto_now`).

`model_instance` is the instance this field belongs to and `add` is whether the instance is being saved to the database for the first time.

It should return the value of the appropriate attribute from `model_instance` for this field. The attribute name is in `self.attname` (this is set up by `Field`).

See Preprocessing values before saving for usage.

Fields often receive their values as a different type, either from serialization or from forms.

`to_python`(*value*)[source]

Converts the value into the correct Python object. It acts as the reverse of `value_to_string()`, and is also called in `clean()`.

See Converting values to Python objects for usage.

Besides saving to the database, the field also needs to know how to serialize its value:

`value_to_string`(*obj*)[source]

Converts `obj` to a string. Used to serialize the value of the field.

See Converting field data for serialization for usage.

When using `model forms`, the `Field` needs to know which form field it should be represented by:

`formfield`(*form_class=None*, *choices_form_class=None*, ***kwargs*)[source]

Returns the default `django.forms.Field` of this field for `ModelForm`.

By default, if both `form_class` and `choices_form_class` are `None`, it uses `CharField`. If the field has `choices` and `choices_form_class` isn't specified, it uses `TypedChoiceField`.

See Specifying the form field for a model field for usage.

`deconstruct`()[source]

Returns a 4-tuple with enough information to recreate the field:

1. The name of the field on the model.
2. The import path of the field (e.g. `"django.db.models.IntegerField"`). This should be the most portable version, so less specific may be better.
3. A list of positional arguments.
4. A dict of keyword arguments.

This method must be added to fields prior to 1.7 to migrate its data using Migrations.

Field attribute reference

Every `Field` instance contains several attributes that allow introspecting its behavior. Use these attributes instead of `isinstance` checks when you need to write code that depends on a field's functionality. These attributes can be used together with the Model._meta API to narrow down a search for specific field types. Custom model fields should implement these flags.

Attributes for fields

`Field.auto_created`

Boolean flag that indicates if the field was automatically created, such as the `OneToOneField` used by model inheritance.

`Field.concrete`

Boolean flag that indicates if the field has a database column associated with it.

`Field.hidden`

Boolean flag that indicates if a field is used to back another non-hidden field's functionality (e.g. the `content_type` and `object_id` fields that make up a `GenericForeignKey`). The `hidden` flag is used to distinguish what constitutes the public subset of fields on the model from all the fields on the model.

> **Note**
>
> `Options.get_fields()` excludes hidden fields by default. Pass in `include_hidden=True` to return hidden fields in the results.

`Field.is_relation`

Boolean flag that indicates if a field contains references to one or more other models for its functionality (e.g. `ForeignKey`, `ManyToManyField`, `OneToOneField`, etc.).

`Field.model`

Returns the model on which the field is defined. If a field is defined on a superclass of a model, `model` will refer to the superclass, not the class of the instance.

Attributes for fields with relations

These attributes are used to query for the cardinality and other details of a relation. These attribute are present on all fields; however, they will only have boolean values (rather than `None`) if the field is a relation type (`Field.is_relation=True`).

`Field.many_to_many`

Boolean flag that is `True` if the field has a many-to-many relation; `False` otherwise. The only field included with Django where this is `True` is `ManyToManyField`.

`Field.many_to_one`

Boolean flag that is `True` if the field has a many-to-one relation, such as a `ForeignKey`; `False` otherwise.

`Field.one_to_many`

Boolean flag that is `True` if the field has a one-to-many relation, such as a `GenericRelation` or the reverse of a `ForeignKey`; `False` otherwise.

`Field.one_to_one`

Boolean flag that is `True` if the field has a one-to-one relation, such as a `OneToOneField`; `False` otherwise.

`Field.related_model`

Points to the model the field relates to. For example, `Author` in `ForeignKey(Author, on_delete=models.CASCADE)`. The `related_model` for a `GenericForeignKey` is always `None`.

« previous | up | next »