Python | Relational fields in Django models

Django models represent real-world entities, and it is rarely the case that real-world entities are entirely independent of each other. Hence Django supports relational databases and allows us to establish relations between different models. There are three types of relational fields which Django supports: many-to-one, many-to-many and one-to-one.

**Many-to-one fields:**

This is used when one record of a model A is related to multiple records of another model B. For example – a model **Song** has many-to-one relationship with a model **Album**, i.e. an album can have many songs, but one song cannot be part of multiple albums. Many-to-one relations are defined using **ForeignKey** field of **django.db.models**.

Below is an example to demonstrate the same.

|  |
| --- |
| from django.db import models    class Album(models.Model):      title = models.CharField(max\_length = 100)      artist = models.CharField(max\_length = 100)    class Song(models.Model):      title = models.CharField(max\_length = 100)      album = models.ForeignKey(Album, on\_delete = models.CASCADE) |

It is a good practice to name the many-to-one field with the same name as the related model, lowercase.

**Many-to-many fields:**

This is used when one record of a model A is related to multiple records of another model B and vice versa. For example – a model **Book** has many-to-many relationship with a model **Author**, i.e. an book can be written by multiple authors and an author can write multiple books. Many-to-many relations are defined using **ManyToManyField** field of **django.db.models**.

Below is an example to demonstrate the same.

|  |
| --- |
| from django.db import models    class Author(models.Model):      name = models.CharField(max\_length = 100)      desc = models.TextField(max\_length = 300)    class Book(models.Model):      title = models.CharField(max\_length = 100)      desc = models.TextField(max\_length = 300)      authors = models.ManyToManyField(Author) |

It is a good practice to name the many-to-many field with the plural version of the related model, lowercase. It doesn’t matter which of the two models contain the many-to-many field, but it shouldn’t be put in both the models.

**One-to-one fields:**

This is used when one record of a model A is related to exactly one record of another model B. This field can be useful as a primary key of an object if that object extends another object in some way. For example – a model **Car** has one-to-one relationship with a model **Vehicle**, i.e. a car is a vehicle. One-to-one relations are defined using **OneToOneField** field of **django.db.models**.

Below is an example to demonstrate the same.

|  |
| --- |
| from django.db import models    class Vehicle(models.Model):      reg\_no = models.IntegerField()      owner = models.CharField(max\_length = 100)    class Car(models.Model):      vehicle = models.OneToOneField(Vehicle,            on\_delete = models.CASCADE, primary\_key = True)      car\_model = models.CharField(max\_length = 100) |

It is a good practice to name the one-to-one field with the same name as that of the related model, lowercase.

**Data integrity options:**

Since we are creating models which depend on other models, we need to define the behavior of a record in one model when the corresponding record in the other is deleted. This is achieved by adding an optional **on\_delete** parameter in the relational field, which can take the following values:

* **on\_delete = models.CASCADE** – This is the default value. It automatically deletes all the related records when a record is deleted.(e.g. when an Album record is deleted all the Song records related to it will be deleted)
* **on\_delete = models.PROTECT** – It blocks the deletion of a record having relation with other records.(e.g. any attempt to delete an Album record will be blocked)
* **on\_delete = models.SET\_NULL** – It assigns NULL to the relational field when a record is deleted, provided **null = True** is set.
* **on\_delete = models.SET\_DEFAULT** – It assigns default values to the relational field when a record is deleted, a default value has to be provided.
* **on\_delete = models.SET()** – It can either take a default value as parameter, or a callable, the return value of which will be assigned to the field.
* **on\_delete = models.DO\_NOTHING** – Takes no action. Its a bad practice to use this value.



