Django fields

In this reading, you will learn about different field types in a model class.

Model

A model in Django is like any normal Python class. The ORM layer maps this model to a database table in the Django project. Each attribute of a model class represents a field in the table.
The Django ORM enables storing and retrieving data in tables, not by executing raw SQL queries, but by invoking the model methods.
A model class subclasses the **`django.models.Model`** class. A typical definition of a model class is done inside the app's **`models.py`** file.

```python
from django.db import models
class Person(models.Model):
    first_name = models.CharField(max_length=20)
    last_name = models.CharField(max_length=20)
```

When you migrate this model (you need to include your app in the **`INSTALLED_APPS`** setting), the **`myapp_person`** table will be created as would be done by the **`CREATE TABLE`** query:

```sql
CREATE TABLE Person (
    id          INTEGER      PRIMARY KEY,
    first_name VARCHAR (20),
    last_name  VARCHAR (20)
)
```

Note that the **`first_name`** and **`last_name`** are the class attributes corresponding to the fields in the table.

Django automatically names the table as **`appname_modelname`**, which you can override by assigning the desired name to **`db_table`** parameter of the Meta class, to be declared inside the model class.

```python
class Student(CommonInfo):
    # ...
    class Meta(CommonInfo.Meta):
        db_table = 'student_info'
```

You should choose the model field type appropriate for the data stored in the mapped field.

The types are defined in **`django.forms`** module. The choice of type determines the HTML widget to use when the form field is rendered. For example, for a **`CharField`**, HTML's text input type will be used. Django also auto-creates the form based on model definitions (it is called **`ModelForm`**), using field types.

Field properties

A Field object has common properties in addition to the field-specific properties.

`primary_key`

This parameter is False by default. You can set it to True if you want the mapped field in the table to be used as its primary key.
It's not mandatory for the model to have a field with a primary key. In its absence, Django, on its own, creates an **`IntegerField`** to hold a unique auto-incrementing number.

`default`

You can also specify any default in the form of a value or a function that will be called when a new object is created.

```python
class Person(models.Model):
    name = models.CharField(max_length=50)
    address = models.CharField(max_length=80, default='Mumbai')
```

`unique`

If this parameter in the field definition is set to True, it will ensure that each object will have a unique value for this field.

```
tax_code = models.CharField(
                    max_length = 20,
                    unique = True
                    )
```

`choices`

If you want to create a drop-down from which the user should select a value for this field, set this parameter to a list of two-item tuples.

```
SEMESTER_CHOICES = (
    ("1", "Civil"),
    ("2", "Electrical"),
    ("3", "Mechanical"),
    ("4", "CompSci"),
)

# declaring a Student Model

class Student(models.Model):
      semester = models.CharField(
        max_length = 20,
        choices = SEMESTER_CHOICES,
        default = '1'
        )
```

Field types

The **`django.models`** module has many field types to choose from.

CharField:

This is the most used field type. It can hold string data of length specified by **`max_lenth`** parameter. For a longer string, use **`TextField`**.

IntegerField:

The field can store an integer between -2147483648 to 2147483647. There are **`BigIntegerField`**, **`SmallIntegerField`** and **`AutoField`**types as well to store integers of varying lengths.

FloatField:

It can store a floating-point number. Its variant **`DecimalField`** stores a number with fixed digits in the fractional part.

```
class Student(Model):
    grade = models.DecimalField(
                        max_digits = 5,
                        decimal_places = 2)
```

DateTimeField

Stores the date and time as an object of Python's **`datetime.datetime`** class. The **`DateField`** stores **`datetime.date`** value.

EmailField

It’s actually a **`CharField`** with an in-built **`EmailValidator`**

FileField

This field is used to save the file uploaded by the user to a designated path specified by the **`upload_to`** parameter.

ImageField

This is a variant of **`FileField`**, having the ability to validate if the uploaded file is an image.

URLField

A **`CharField`** having in-built validation for URL.

Relationship fields

There can be three types of relationships between database models:

one-to-one

one-to-many

many-to-many

The **django.models** module has the following fields for establishing relationships between models.

`ForeignKey`

It is used to establisha **one-to-many** relationship between two models. It requires two positional arguments - the model with which it is related, and the **`on_delete`** option to define the behavior of the delete operation. Suppose you have a **Customer** and **Vehicle** model with a one-to-many relationship. A customer can have more than one vehicle.

```python
class Customer(models.Model):
    name = models.CharField(max_length=255)

class Vehicle(models.Model):
    name = models.CharField(max_length=255)
    customer = models.ForeignKey(
        Customer,
        on_delete=models.CASCADE,
        related_name='Vehicle'
    )
```

The **`on_delete`** option specifies the behavior in case the associated object in the primary model is deleted. The values are:

CASCADE: deletes the object containing the **`ForeignKey`**

PROTECT: Prevent deletion of the referenced object

RESTRICT: Prevent deletion of the referenced object by raising **`RestrictedError`**

Now let's expand upon these values in more detail:

CASCADE

If the **`on_delete`** parameter is set to **`CASCADE`**, deleting the reference object will also delete the referred object. Suppose a vehicle belongs to a customer. When the Customer is deleted, all the vehicles that reference the customer will be automatically deleted.

PROTECT

The effect of the **`PROTECT`** option is the opposite of **`CASCADE`**. It prevents the deletion of a referenced objectif it has an object referencing it in the database. Suppose a vehicle belongs to a customer.
If a customer has vehicles, it cannot be deleted. It's important to know that if you forcefully delete the customer, Django raises the **`ProtectedError`**.

RESTRICT

The difference between **`PROTECT`** and **`RESTRICT`** is that when you delete the referenced object, the **`on_delete`** option raises the **`RestrictedError`**.

The deletion of the referenced object is allowed if it also references a different object that is being deleted in the same operation, but via a **`CASCADE`** relationship.
Let's use the example of an Artist model – a **`CASCADE`** relationship with an Album and Song model. The Artist model, in turn, has a **`RESTRICT`** relationship with the song model.

```
class Artist(models.Model):
    name = models.CharField(max_length=10)

class Album(models.Model):
    artist = models.ForeignKey(Artist, on_delete=models.CASCADE)

class Song(models.Model):
    artist = models.ForeignKey(Artist, on_delete=models.CASCADE)
    album = models.ForeignKey(Album, on_delete=models.RESTRICT)
```

Next, you can create a few instances of these models:

```
>>> artist1 = Artist.objects.create(name='Danny')
>>> artist2 = Artist.objects.create(name='John')
>>> album1 = Album.objects.create(artist=artist1)
>>> album2 = Album.objects.create(artist=artist2)
>>> song1 = Song.objects.create(artist=artist1, album=album1)
>>> song_two = Song.objects.create(artist=artis1, album=album2)
```

You can safely delete the **`artist1`** instance. If you try to delete artist2, the **`RestrictedError`** is raised.

OneToOneField:
This field in one of the models establishes a one-to-one relationship between the two models.

Although a **`ForeignKey`** field with **`unique=True`** setting, behaves similarly, the reverse side of the relationship will always return a single object.
The following model definition demonstrates a one-to-one relationship between the **college** model and a **principal** model.
A college can have only one principal and one person can be a principal of only one college.

```
class college(Model):
    CollegeID = models.IntegerField(primary_key = True)
    name = models.CharField(max_length=50)
    strength = models.IntegerField()
    website=models.URLField()

class Principal(models.Model):
    CollegeID = models.OneToOneField(
                College,
                on_delete=models.CASCADE
                )
    Qualification = models.CharField(max_length=50)
    email = models.EmailField(max_length=50)
```

ManyToManyField:

This field helps in setting a many-to-many relationship between two models.

Here, multiple objects of one model can be associated with multiple objects of another model.

For example, in the case of **Subject** and **Teacher** models, a subject is taught by more than one teacher.

Similarly, a teacher can teach more than one subject. This is represented in the following model definitions:

```
class Teacher(models.Model):
    TeacherID = models.ItegerField(primary_key=True)
    Qualification = models.CharField(max_length=50)
    email = models.EmailField(max_length=50)

class Subject(models.Model):
    Subjectcode = models.IntegerField(primary_key = True)
    name = models.CharField(max_length=30)
    credits = model.IntegerField()
    teacher = model.ManyToManyField(Teacher)
```

In this reading you learned about the different fields to be used as the type of attributes in a model class.