Theory: Field lookups

22 minutes 0 / 5 problems solved

§1. Introduction

You already know how to get and filter objects from a database. Yet sometimes the logic of an application is rather complex, so you should know how to make more particular queries.

The exact value of a field is a good starting point, but several problems remain. How to get all objects with a field greater or less than the given value, or filter objects which fields contain some substring or even a regular expression? Here **field lookups** will come in handy.

§2. Excluding data

The Quidditch league is gaining popularity with its new site we're making. Our users want to know the result of each game and find out some interesting statistics about the previous tournaments. We have models *Team* and *Game*. All the games from 1674 until now are stored in our database; we get them from the Hogwarts library archives. Don't even ask how much time it takes – the answer is a lot! This is why we prefer to keep the data in the database to get all the information we needed in a flash:

```
from django.db import models


class Team(models.Model):
    name = models.CharField(max_length=64)


class Game(models.Model):
    home_team = models.ForeignKey(Team, related_name='game_at_home', on_delete=models.CASCADE)
    home_team_points = models.IntegerField()

    rival_team = models.ForeignKey(Team, related_name='rival_game', on_delete=models.CASCADE)
    rival_team_points = models.IntegerField()

    date = models.DateField()
```

Since we haven't published all data to the website, we shall provide you with just a sample:

Current topic:

Field lookups

Topic depends on:

✕ Queries and filters

Topic is required for:

Aggregations and Ordering

Table of contents:

↑ Field lookups

§1. Introduction

§2. Excluding data

§3. Field Lookups

§4. Using Collections for Lookups

§5. Field Lookups for Foreign Keys

§6. Conclusion

Feedback & Comments

```
from datetime import date

falmouth_falcons = Team.objects.create(name="Falmouth Falcons")
montrose_magpies = Team.objects.create(name="Montrose Magpies")
tutshill_tornados = Team.objects.create(name="Tutshill Tornados")
appleby_arrows = Team.objects.create(name="Appleby Arrows")

Game.objects.create(home_team=falmouth_falcons, home_team_points=15,
                    rival_team=montrose_magpies, rival_team_points=12, date=date(1674, 5, 6))


Game.objects.create(home_team=falmouth_falcons, home_team_points=34,
                    rival_team=tutshill_tornados, rival_team_points=8, date=date(1774, 9, 30))


Game.objects.create(home_team=appleby_arrows, home_team_points=10,
                    rival_team=montrose_magpies, rival_team_points=19, date=date(1779, 7, 15))


Game.objects.create(home_team=tutshill_tornados, home_team_points=7,
                    rival_team=appleby_arrows, rival_team_points=27, date=date(2018, 6, 25))


Game.objects.create(home_team=montrose_magpies, home_team_points=24,
                    rival_team=tutshill_tornados, rival_team_points=16, date=date(1907, 5, 12))
```

Remember that you should migrate your models before using it!

If you're familiar with the `filter` method of Django Object Manager, you'll easily follow what the `exclude` method does. When filtering out data, we look for objects that satisfy some condition. Exclude, on the contrary, allows us to remove objects from QuerySet by a condition. Compare two queries:

```
# All home games for Falmouth Falcons
Game.objects.filter(home_team=falmouth_falcons)

# All games excluding games where Falmouth Falcons was a home team
Game.objects.exclude(home_team=falmouth_falcons)
```

We break all the games into two samples: one that satisfies the condition and another that doesn't. That is the point of the `exclude` method.

Note: the syntax rules for `exclude` are the same as for the `filter` method.

§3. Field Lookups

Filtering objects only by their exact value is not very convenient: you might want to get objects that satisfy a trickier condition. For example, it could be all Quidditch matches of the XX century or all games where the home team scored more than 12 points. We'll start with this query:

```
great_score_at_home_games = Game.objects.filter(home_team_points__gt=12)
```

The special syntax for the parameter is: field name, double underscores, field lookup. **Field lookup** is a special name for actions on the field value you want to make when filtering data. The "*gt*" lookup is a reduction of the word *greater,* so you can read this query as "filter games where the home team scored more than 12 points".

Let's look at another example:

```
from datetime import date

twentieth_century_games = Game.objects.filter(
    date__gte=date(1900, 1, 1), date__lte=date(1999, 12, 31)
)
```

The "*gte*" lookup looks similar to the "*gt*"; the difference is that "*gte*" means that the value of a field could be greater or equal than the given one. Accordingly, "*lte*" means *less than or equal*.

This way, we restrict the value of the date with upper and lower bounds. We could write our query with the use of another helper:

```
twentieth_century_games = Game.objects.filter(date__year__gte=1900, date__year__lte=1999)
```

Here we are combining two lookups. First, we extract a year from the date and then use condition on its value. The year lookup is only available for `DateField` and `DateTimeField`; we cannot use it for plain `IntegerField` or `CharField` because this wouldn't make any sense.

We could write our query more naturally:

```
twentieth_century_games = Game.objects.filter(date__year__range=(1900, 1999))
```

The syntax rule for any field lookup is simple: add double underscores before using it. When chaining them, add double underscores each time. The range means that we search values between two provided boundaries.

> It's strongly recommended not to use double underscores in field names. You should name all fields as you name all Python variables – join the parts with a single underscore mark.

§4. Using Collections for Lookups

We already know a lot about how to compare the field with a single value. Django allows us to use a collection in our queries. Of course, we can filter all home games for Falmouth Falcons and Montrose Magpies excluding all other options:

```
special_home_games = Game.objects.exclude(home_team=tutshill_tornados) \
                                 .exclude(home_team=appleby_arrows)
```

But what are we going to do when we have more conditions? Should we apply our profound copy-paste skills or make another smart move? We invite you to opt for the latter. Let's use the Python list to filter what we need:

```
special_home_games = Game.objects.filter(home_team__in=[falmouth_falcons, montrose_magpies])
```

If you're familiar with lists, you can use your knowledge for the "*in*" lookup. It works just like for any Python collection. We check that the collection contains the value, and if so, we keep the objects in our QuerySet.

§5. Field Lookups for Foreign Keys

By now you are able to filter out some data and use it for another query. First, we have `falmouth_falcons` as a variable. Then we can filter the games where Falcons was a home team. But do we need to store it in a variable to

make a query? It's not necessary because we can access fields of foreign keys directly through lookups.

```
falcons_home_games = Game.objects.filter(home_team__name="Falmouth Falcons")
```

You use double underscores again and it works. Now the construction is: foreign key field name, double underscores, foreign key model field.

To dive deeper, let's combine accessing the field of the foreign model with field lookups:

```
falcons_home_games = Game.objects.filter(home_team__name__contains="Falcons")
```

Double underscores are similar to access through the dot, fields are similar to class attributes and lookups resemble class methods. So you can combine them just like you can combine accesses through the dot.

> Be careful and don't make too complex queries with many double underscores: two pairs are enough. If you use more, the time can dramatically increase or your database will use a lot of memory for that query.

§6. Conclusion

Once you learn how to get particular objects, you quickly become dexterous in your queries. Using field lookups you can apply methods to fields of models or even those of foreign keys. It looks like we can get any objects we could think of! Surely, there's always room for perfection and more knowledge, so you may check out more lookups in the official documentation.

Report a typo

24 users liked this theory. **2** didn't like it. **What about you?**

Start practicing

Comments (5) Hints (0) Useful links (0) Show discussion