Many-to-one relationships

To define a many-to-one relationship, use `ForeignKey`.

In this example, a `Reporter` can be associated with many `Article` objects, but an `Article` can only have one `Reporter` object:

```
from django.db import models


class Reporter(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)
    email = models.EmailField()

    def __str__(self):
        return f"{self.first_name} {self.last_name}"


class Article(models.Model):
    headline = models.CharField(max_length=100)
    pub_date = models.DateField()
    reporter = models.ForeignKey(Reporter, on_delete=models.CASCADE)

    def __str__(self):
        return self.headline

    class Meta:
        ordering = ["headline"]
```

What follows are examples of operations that can be performed using the Python API facilities.

Create a few Reporters:

```
>>> r = Reporter(first_name="John", last_name="Smith", email="john@example.com")
>>> r.save()

>>> r2 = Reporter(first_name="Paul", last_name="Jones", email="paul@example.com")
>>> r2.save()
```

Create an Article:

```
>>> from datetime import date
>>> a = Article(id=None, headline="This is a test", pub_date=date(2005, 7, 27), reporter=r)
>>> a.save()

>>> a.reporter.id
1

>>> a.reporter
<Reporter: John Smith>
```

Note that you must save an object before it can be assigned to a foreign key relationship. For example, creating an `Article` with unsaved `Reporter` raises `ValueError`:

```
>>> r3 = Reporter(first_name="John", last_name="Smith", email="john@example.com")
>>> Article.objects.create(
...     headline="This is a test", pub_date=date(2005, 7, 27), reporter=r3
... )
Traceback (most recent call last):
...
ValueError: save() prohibited to prevent data loss due to unsaved related object 'reporter'.
```

Article objects have access to their related Reporter objects:

```
>>> r = a.reporter
```

Create an Article via the Reporter object:

```
>>> new_article = r.article_set.create(
...     headline="John's second story", pub_date=date(2005, 7, 29)
... )
>>> new_article
<Article: John's second story>
>>> new_article.reporter
<Reporter: John Smith>
>>> new_article.reporter.id
1
```

Create a new article:

```
>>> new_article2 = Article.objects.create(
...     headline="Paul's story", pub_date=date(2006, 1, 17), reporter=r
... )
>>> new_article2.reporter
<Reporter: John Smith>
>>> new_article2.reporter.id
1
>>> r.article_set.all()
<QuerySet [<Article: John's second story>, <Article: Paul's story>, <Article: This is a test>]>
```

Add the same article to a different article set - check that it moves:

```
>>> r2.article_set.add(new_article2)
>>> new_article2.reporter.id
2
>>> new_article2.reporter
<Reporter: Paul Jones>
```

Adding an object of the wrong type raises TypeError:

```
>>> r.article_set.add(r2)
Traceback (most recent call last):
...
TypeError: 'Article' instance expected, got <Reporter: Paul Jones>

>>> r.article_set.all()
<QuerySet [<Article: John's second story>, <Article: This is a test>]>
>>> r2.article_set.all()
<QuerySet [<Article: Paul's story>]>

>>> r.article_set.count()
2

>>> r2.article_set.count()
1
```

Note that in the last example the article has moved from John to Paul.

Related managers support field lookups as well. The API automatically follows relationships as far as you need. Use double underscores to separate relationships. This works as many levels deep as you want. There's no limit. For example:

```
>>> r.article_set.filter(headline__startswith="This")
<QuerySet [<Article: This is a test>]>

# Find all Articles for any Reporter whose first name is "John".
>>> Article.objects.filter(reporter__first_name="John")
<QuerySet [<Article: John's second story>, <Article: This is a test>]>
```

Exact match is implied here:

```
>>> Article.objects.filter(reporter__first_name="John")
<QuerySet [<Article: John's second story>, <Article: This is a test>]>
```

Query twice over the related field. This translates to an AND condition in the WHERE clause:

```
>>> Article.objects.filter(reporter__first_name="John", reporter__last_name="Smith")
<QuerySet [<Article: John's second story>, <Article: This is a test>]>
```

For the related lookup you can supply a primary key value or pass the related object explicitly:

```
>>> Article.objects.filter(reporter__pk=1)
<QuerySet [<Article: John's second story>, <Article: This is a test>]>
>>> Article.objects.filter(reporter=1)
<QuerySet [<Article: John's second story>, <Article: This is a test>]>
>>> Article.objects.filter(reporter=r)
<QuerySet [<Article: John's second story>, <Article: This is a test>]>
```

```
>>> Article.objects.filter(reporter__in=[1, 2]).distinct()
<QuerySet [<Article: John's second story>, <Article: Paul's story>, <Article: This is a test>]>
>>> Article.objects.filter(reporter__in=[r, r2]).distinct()
<QuerySet [<Article: John's second story>, <Article: Paul's story>, <Article: This is a test>]>
```

You can also use a queryset instead of a literal list of instances:

```
>>> Article.objects.filter(
...     reporter__in=Reporter.objects.filter(first_name="John")
... ).distinct()
<QuerySet [<Article: John's second story>, <Article: This is a test>]>
```

Querying in the opposite direction:

```
>>> Reporter.objects.filter(article__pk=1)
<QuerySet [<Reporter: John Smith>]>
>>> Reporter.objects.filter(article=1)
<QuerySet [<Reporter: John Smith>]>
>>> Reporter.objects.filter(article=a)
<QuerySet [<Reporter: John Smith>]>

>>> Reporter.objects.filter(article__headline__startswith="This")
<QuerySet [<Reporter: John Smith>, <Reporter: John Smith>, <Reporter: John Smith>]>
>>> Reporter.objects.filter(article__headline__startswith="This").distinct()
<QuerySet [<Reporter: John Smith>]>
```

Counting in the opposite direction works in conjunction with `distinct()`:

```
>>> Reporter.objects.filter(article__headline__startswith="This").count()
3
>>> Reporter.objects.filter(article__headline__startswith="This").distinct().count()
1
```

Queries can go round in circles:

```
>>> Reporter.objects.filter(article__reporter__first_name__startswith="John")
<QuerySet [<Reporter: John Smith>, <Reporter: John Smith>, <Reporter: John Smith>, <Reporter: John Smith>]>
>>> Reporter.objects.filter(article__reporter__first_name__startswith="John").distinct()
<QuerySet [<Reporter: John Smith>]>
>>> Reporter.objects.filter(article__reporter=r).distinct()
<QuerySet [<Reporter: John Smith>]>
```

If you delete a reporter, their articles will be deleted (assuming that the ForeignKey was defined with `django.db.models.ForeignKey.on_delete` set to `CASCADE`, which is the default):

```
>>> Article.objects.all()
<QuerySet [<Article: John's second story>, <Article: Paul's story>, <Article: This is a test>]>
>>> Reporter.objects.order_by("first_name")
<QuerySet [<Reporter: John Smith>, <Reporter: Paul Jones>]>
>>> r2.delete()
>>> Article.objects.all()
<QuerySet [<Article: John's second story>, <Article: This is a test>]>
>>> Reporter.objects.order_by("first_name")
<QuerySet [<Reporter: John Smith>]>
```

You can delete using a JOIN in the query:

```
>>> Reporter.objects.filter(article__headline__startswith="This").delete()
>>> Reporter.objects.all()
<QuerySet []>
>>> Article.objects.all()
<QuerySet []>
```