One-to-one relationships[**¶**](https://docs.djangoproject.com/en/1.10/topics/db/examples/one_to_one/#one-to-one-relationships)

To define a one-to-one relationship, use **[OneToOneField](https://docs.djangoproject.com/en/1.10/ref/models/fields/" \l "django.db.models.OneToOneField" \o "django.db.models.OneToOneField)**.

In this example, a **Place** optionally can be a **Restaurant**:

**from** **django.db** **import** models

**class** **Place**(models.Model):

name = models.CharField(max\_length=50)

address = models.CharField(max\_length=80)

**def** \_\_str\_\_(self): *# \_\_unicode\_\_ on Python 2*

**return** “place: ”+self.name+” the place”

**class** **Restaurant**(models.Model):

place = models.OneToOneField(

Place,

on\_delete=models.CASCADE,

primary\_key=**True**,

)

serves\_hot\_dogs = models.BooleanField(default=**False**)

serves\_pizza = models.BooleanField(default=**False**)

**def** \_\_str\_\_(self):

**return** “Restaurant:”+self.place.name+”m the Restaurant”

**class** **Waiter**(models.Model):

restaurant = models.ForeignKey(Restaurant, on\_delete=models.CASCADE)

name = models.CharField(max\_length=50)

**def** \_\_str\_\_(self): *# \_\_unicode\_\_ on Python 2*

**return** "**%s** the waiter at **%s**" % (self.name, self.restaurant)

Create a couple of Places:

**>>>** p1 = Place(name='Demon Dogs', address='944 W. Fullerton')

**>>>** p1.save()

**>>>** p2 = Place(name='Ace Hardware', address='1013 N. Ashland')

**>>>** p2.save()

Create a Restaurant. Pass the ID of the “parent” object as this object’s ID:

**>>>** r = Restaurant(place=p1, serves\_hot\_dogs=True, serves\_pizza=False)

**>>>** r.save()

A Restaurant can access its place:

**>>>** r.place

<Place: Demon Dogs the place>

A Place can access its restaurant, if available:

**>>>** p1.restaurant

<Restaurant: Demon Dogs the restaurant>

p2 doesn’t have an associated restaurant:

**>>> from** **django.core.exceptions** **import** ObjectDoesNotExist

**>>> try**:

**>>>**  p2.restaurant

**>>> except** ObjectDoesNotExist:

**>>>**  **print**("There is no restaurant here.")

There is no restaurant here.

You can also use **hasattr** to avoid the need for exception catching:

**>>>** hasattr(p2, 'restaurant')

False

Set the place using assignment notation. Because place is the primary key on Restaurant, the save will create a new restaurant:

**>>>** r.place = p2

**>>>** r.save()

**>>>** p2.restaurant

<Restaurant: Ace Hardware the restaurant>

**>>>** r.place

<Place: Ace Hardware the place>

Set the place back again, using assignment in the reverse direction:

**>>>** p1.restaurant = r

**>>>** p1.restaurant

<Restaurant: Demon Dogs the restaurant>

Note that you must save an object before it can be assigned to a one-to-one relationship. For example, creating a **Restaurant** with unsaved **Place** raises **ValueError**:

**>>>** p3 = Place(name='Demon Dogs', address='944 W. Fullerton')

**>>>** Restaurant.objects.create(place=p3, serves\_hot\_dogs=True, serves\_pizza=False)

Traceback (most recent call last):

*...*

ValueError: save() prohibited to prevent data loss due to unsaved related object 'place'.

Restaurant.objects.all() just returns the Restaurants, not the Places. Note that there are two restaurants - Ace Hardware the Restaurant was created in the call to r.place = p2:

**>>>** Restaurant.objects.all()

<QuerySet [<Restaurant: Demon Dogs the restaurant>, <Restaurant: Ace Hardware the restaurant>]>

Place.objects.all() returns all Places, regardless of whether they have Restaurants:

**>>>** Place.objects.order\_by('name')

<QuerySet [<Place: Ace Hardware the place>, <Place: Demon Dogs the place>]>

You can query the models using [lookups across relationships](https://docs.djangoproject.com/en/1.10/topics/db/queries/#lookups-that-span-relationships):

**>>>** Restaurant.objects.get(place=p1)

<Restaurant: Demon Dogs the restaurant>

**>>>** Restaurant.objects.get(place\_\_pk=1)

<Restaurant: Demon Dogs the restaurant>

**>>>** Restaurant.objects.filter(place\_\_name\_\_startswith="Demon")

<QuerySet [<Restaurant: Demon Dogs the restaurant>]>

**>>>** Restaurant.objects.exclude(place\_\_address\_\_contains="Ashland")

<QuerySet [<Restaurant: Demon Dogs the restaurant>]>

This of course works in reverse:

**>>>** Place.objects.get(pk=1)

<Place: Demon Dogs the place>

**>>>** Place.objects.get(restaurant\_\_place=p1)

<Place: Demon Dogs the place>

**>>>** Place.objects.get(restaurant=r)

<Place: Demon Dogs the place>

**>>>** Place.objects.get(restaurant\_\_place\_\_name\_\_startswith="Demon")

<Place: Demon Dogs the place>

P1.delete()

Place.objects.all()

Restaurant.objects.all()

Add a Waiter to the Restaurant:

**>>>** w = r.waiter\_set.create(name='Joe')

**>>>** w

<Waiter: Joe the waiter at Demon Dogs the restaurant>

Query the waiters:

**>>>** Waiter.objects.filter(restaurant\_\_place=p1)

<QuerySet [<Waiter: Joe the waiter at Demon Dogs the restaurant>]>

**>>>** Waiter.objects.filter(restaurant\_\_place\_\_name\_\_startswith="Demon")

<QuerySet [<Waiter: Joe the waiter at Demon Dogs the restaurant>]>

# Many-to-one relationships[¶](https://docs.djangoproject.com/en/1.10/topics/db/examples/many_to_one/#many-to-one-relationships)

To define a many-to-one relationship, use **[ForeignKey](https://docs.djangoproject.com/en/1.10/ref/models/fields/" \l "django.db.models.ForeignKey" \o "django.db.models.ForeignKey)**:

**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): *# \_\_unicode\_\_ on Python 2*

**return** 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): *# \_\_unicode\_\_ on Python 2*

**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:

Manage.py task-🡪

Makemigrations onetomanyapp

Migrate onetomanyapp

>>>From onetomanyapp.models import Article,Reporter

>>> **from** **datetime** **import** date

**>>>** 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:

**>>>** 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:

**>>>** p = a.reporter

On Python 2, these are strings of type **str** instead of unicode strings because that’s what was used in the creation of this reporter (and we haven’t refreshed the data from the database, which always returns unicode strings):

**>>>** p.first\_name, p.last\_name

('John', 'Smith')

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, and add it to the article set:

**>>>** new\_article2 = Article(headline="Paul's story", pub\_date=date(2006, 1, 17))

**>>>new\_article2.save()**

**>>>** r.article\_set.add(new\_article2)

**>>>** 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

**>>>** 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:

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, his articles will be deleted (assuming that the ForeignKey was defined with **[django.db.models.ForeignKey.on\_delete](https://docs.djangoproject.com/en/1.10/ref/models/fields/" \l "django.db.models.ForeignKey.on_delete" \o "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 []>

# Many-to-many relationships[¶](https://docs.djangoproject.com/en/1.10/topics/db/examples/many_to_many/#many-to-many-relationships)

To define a many-to-many relationship, use **[ManyToManyField](https://docs.djangoproject.com/en/1.10/ref/models/fields/" \l "django.db.models.ManyToManyField" \o "django.db.models.ManyToManyField)**.

In this example, an **Article** can be published in multiple **Publication** objects, and a **Publication** has multiple **Article** objects:

**from** **django.db** **import** models

**class** **Publication**(models.Model):

title = models.CharField(max\_length=30)

**def** \_\_str\_\_(self): *# \_\_unicode\_\_ on Python 2*

**return** self.title

**class** **Meta**:

ordering = ('title',)

**class** **Article**(models.Model):

headline = models.CharField(max\_length=100)

publications = models.ManyToManyField(Publication)

**def** \_\_str\_\_(self): *# \_\_unicode\_\_ on Python 2*

**return** self.headline

**class** **Meta**:

ordering = ('headline',)

What follows are examples of operations that can be performed using the Python API facilities. Note that if you are using [an intermediate model](https://docs.djangoproject.com/en/1.10/topics/db/models/#intermediary-manytomany) for a many-to-many relationship, some of the related manager’s methods are disabled, so some of these examples won’t work with such models.

Create a couple of **Publications**:

**>>>** p1 = Publication(title='The Python Journal')

**>>>** p1.save()

**>>>** p2 = Publication(title='Science News')

**>>>** p2.save()

**>>>** p3 = Publication(title='Science Weekly')

**>>>** p3.save()

Create an **Article**:

**>>>** a1 = Article(headline='Django lets you build Web apps easily')

You can’t associate it with a **Publication** until it’s been saved:

**>>>** a1.publications.add(p1)

Traceback (most recent call last):

*...*

ValueError: 'Article' instance needs to have a primary key value before a many-to-many relationship can be used.

Save it!

**>>>** a1.save()

Associate the **Article** with a **Publication**:

**>>>** a1.publications.add(p1)

Create another **Article**, and set it to appear in both **Publications**:

**>>>** a2 = Article(headline='NASA uses Python')

**>>>** a2.save()

**>>>** a2.publications.add(p1, p2)

**>>>** a2.publications.add(p3)

Adding a second time is OK:

**>>>** a2.publications.add(p3)

Adding an object of the wrong type raises **[TypeError](https://docs.python.org/3/library/exceptions.html" \l "TypeError" \o "(in Python v3.6))**:

**>>>** a2.publications.add(a1)

Traceback (most recent call last):

*...*

TypeError: 'Publication' instance expected

Create and add a **Publication** to an **Article** in one step using [**create()**](https://docs.djangoproject.com/en/1.10/ref/models/relations/#django.db.models.fields.related.RelatedManager.create):

**>>>** new\_publication = a2.publications.create(title='Highlights for Children')

**Article** objects have access to their related **Publication** objects:

**>>>** a1.publications.all()

<QuerySet [<Publication: The Python Journal>]>

**>>>** a2.publications.all()

<QuerySet [<Publication: Highlights for Children>, <Publication: Science News>, <Publication: Science Weekly>, <Publication: The Python Journal>]>

**Publication** objects have access to their related **Article** objects:

**>>>** p2.article\_set.all()

<QuerySet [<Article: NASA uses Python>]>

**>>>** p1.article\_set.all()

<QuerySet [<Article: Django lets you build Web apps easily>, <Article: NASA uses Python>]>

**>>>** Publication.objects.get(id=4).article\_set.all()

<QuerySet [<Article: NASA uses Python>]>

Many-to-many relationships can be queried using [lookups across relationships](https://docs.djangoproject.com/en/1.10/topics/db/queries/#lookups-that-span-relationships):

**>>>** Article.objects.filter(publications\_\_id=1)

<QuerySet [<Article: Django lets you build Web apps easily>, <Article: NASA uses Python>]>

**>>>** Article.objects.filter(publications\_\_pk=1)

<QuerySet [<Article: Django lets you build Web apps easily>, <Article: NASA uses Python>]>

**>>>** Article.objects.filter(publications=1)

<QuerySet [<Article: Django lets you build Web apps easily>, <Article: NASA uses Python>]>

**>>>** Article.objects.filter(publications=p1)

<QuerySet [<Article: Django lets you build Web apps easily>, <Article: NASA uses Python>]>

**>>>** Article.objects.filter(publications\_\_title\_\_startswith="Science")

<QuerySet [<Article: NASA uses Python>, <Article: NASA uses Python>]>

**>>>** Article.objects.filter(publications\_\_title\_\_startswith="Science").distinct()

<QuerySet [<Article: NASA uses Python>]>

The [**count()**](https://docs.djangoproject.com/en/1.10/ref/models/querysets/#django.db.models.query.QuerySet.count) function respects [**distinct()**](https://docs.djangoproject.com/en/1.10/ref/models/querysets/#django.db.models.query.QuerySet.distinct) as well:

**>>>** Article.objects.filter(publications\_\_title\_\_startswith="Science").count()

2

**>>>** Article.objects.filter(publications\_\_title\_\_startswith="Science").distinct().count()

1

**>>>** Article.objects.filter(publications\_\_in=[1,2]).distinct()

<QuerySet [<Article: Django lets you build Web apps easily>, <Article: NASA uses Python>]>

**>>>** Article.objects.filter(publications\_\_in=[p1,p2]).distinct()

<QuerySet [<Article: Django lets you build Web apps easily>, <Article: NASA uses Python>]>

Reverse m2m queries are supported (i.e., starting at the table that doesn’t have a **[ManyToManyField](https://docs.djangoproject.com/en/1.10/ref/models/fields/" \l "django.db.models.ManyToManyField" \o "django.db.models.ManyToManyField)**):

**>>>** Publication.objects.filter(id=1)

<QuerySet [<Publication: The Python Journal>]>

**>>>** Publication.objects.filter(pk=1)

<QuerySet [<Publication: The Python Journal>]>

**>>>** Publication.objects.filter(article\_\_headline\_\_startswith="NASA")

<QuerySet [<Publication: Highlights for Children>, <Publication: Science News>, <Publication: Science Weekly>, <Publication: The Python Journal>]>

**>>>** Publication.objects.filter(article\_\_id=1)

<QuerySet [<Publication: The Python Journal>]>

**>>>** Publication.objects.filter(article\_\_pk=1)

<QuerySet [<Publication: The Python Journal>]>

**>>>** Publication.objects.filter(article=1)

<QuerySet [<Publication: The Python Journal>]>

**>>>** Publication.objects.filter(article=a1)

<QuerySet [<Publication: The Python Journal>]>

**>>>** Publication.objects.filter(article\_\_in=[1,2]).distinct()

<QuerySet [<Publication: Highlights for Children>, <Publication: Science News>, <Publication: Science Weekly>, <Publication: The Python Journal>]>

**>>>** Publication.objects.filter(article\_\_in=[a1,a2]).distinct()

<QuerySet [<Publication: Highlights for Children>, <Publication: Science News>, <Publication: Science Weekly>, <Publication: The Python Journal>]>

Excluding a related item works as you would expect, too (although the SQL involved is a little complex):

**>>>** Article.objects.exclude(publications=p2)

<QuerySet [<Article: Django lets you build Web apps easily>]>

If we delete a **Publication**, its **Articles** won’t be able to access it:

**>>>** p1.delete()

**>>>** Publication.objects.all()

<QuerySet [<Publication: Highlights for Children>, <Publication: Science News>, <Publication: Science Weekly>]>

**>>>** a1 = Article.objects.get(pk=1)

**>>>** a1.publications.all()

<QuerySet []>

If we delete an **Article**, its **Publications** won’t be able to access it:

**>>>** a2.delete()

**>>>** Article.objects.all()

<QuerySet [<Article: Django lets you build Web apps easily>]>

**>>>** p2.article\_set.all()

<QuerySet []>

Adding via the ‘other’ end of an m2m:

**>>>** a4 = Article(headline='NASA finds intelligent life on Earth')

**>>>** a4.save()

**>>>** p2.article\_set.add(a4)

**>>>** p2.article\_set.all()

<QuerySet [<Article: NASA finds intelligent life on Earth>]>

**>>>** a4.publications.all()

<QuerySet [<Publication: Science News>]>

Adding via the other end using keywords:

**>>>** new\_article = p2.article\_set.create(headline='Oxygen-free diet works wonders')

**>>>** p2.article\_set.all()

<QuerySet [<Article: NASA finds intelligent life on Earth>, <Article: Oxygen-free diet works wonders>]>

**>>>** a5 = p2.article\_set.all()[1]

**>>>** a5.publications.all()

<QuerySet [<Publication: Science News>]>

Removing **Publication** from an **Article**:

**>>>** a4.publications.remove(p2)

**>>>** p2.article\_set.all()

<QuerySet [<Article: Oxygen-free diet works wonders>]>

**>>>** a4.publications.all()

<QuerySet []>

And from the other end:

**>>>** p2.article\_set.remove(a5)

**>>>** p2.article\_set.all()

<QuerySet []>

**>>>** a5.publications.all()

<QuerySet []>

Relation sets can be set:

**>>>** a4.publications.all()

<QuerySet []>

**>>>** a4.publications.set([p3])

**>>>** a4.publications.all()

<QuerySet [<Publication: Science Weekly>]>

Relation sets can be cleared:

**>>>** p2.article\_set.clear()

**>>>** p2.article\_set.all()

<QuerySet []>

And you can clear from the other end:

**>>>** p2.article\_set.add(a4, a5)

**>>>** p2.article\_set.all()

<QuerySet [<Article: NASA finds intelligent life on Earth>, <Article: Oxygen-free diet works wonders>]>

**>>>** a4.publications.all()

<QuerySet [<Publication: Science News>, <Publication: Science Weekly>]>

**>>>** a4.publications.clear()

**>>>** a4.publications.all()

<QuerySet []>

**>>>** p2.article\_set.all()

<QuerySet [<Article: Oxygen-free diet works wonders>]>

Recreate the **Article** and **Publication** we have deleted:

**>>>** p1 = Publication(title='The Python Journal')

**>>>** p1.save()

**>>>** a2 = Article(headline='NASA uses Python')

**>>>** a2.save()

**>>>** a2.publications.add(p1, p2, p3)

Bulk delete some **Publications** - references to deleted publications should go:

**>>>** Publication.objects.filter(title\_\_startswith='Science').delete()

**>>>** Publication.objects.all()

<QuerySet [<Publication: Highlights for Children>, <Publication: The Python Journal>]>

**>>>** Article.objects.all()

<QuerySet [<Article: Django lets you build Web apps easily>, <Article: NASA finds intelligent life on Earth>, <Article: NASA uses Python>, <Article: Oxygen-free diet works wonders>]>

**>>>** a2.publications.all()

<QuerySet [<Publication: The Python Journal>]>

Bulk delete some articles - references to deleted objects should go:

**>>>** q = Article.objects.filter(headline\_\_startswith='Django')

**>>> print**(q)

<QuerySet [<Article: Django lets you build Web apps easily>]>

**>>>** q.delete()

After the [**delete()**](https://docs.djangoproject.com/en/1.10/ref/models/querysets/#django.db.models.query.QuerySet.delete), the **[QuerySet](https://docs.djangoproject.com/en/1.10/ref/models/querysets/" \l "django.db.models.query.QuerySet" \o "django.db.models.query.QuerySet)** cache needs to be cleared, and the referenced objects should be gone:

**>>> print**(q)

<QuerySet []>

**>>>** p1.article\_set.all()

<QuerySet [<Article: NASA uses Python>]>