MAKING QUERIES

Django mastery in Nepali

Making queries

Once you've created your data models, Django automatically gives you a database-abstraction API that lets you create, retrieve, update and delete objects. Throughout this guide (and in the reference), we'll refer to the following models, which comprise a blog application:

```
class Blog(models.Model):
    name = models.CharField(max_length=100)
    tagline = models.TextField()

    def __str__(self):
        return self.name


class Author(models.Model):
    name = models.CharField(max_length=200)
    email = models.EmailField()

    def __str__(self):
        return self.name


class Entry(models.Model):
    blog = models.ForeignKey(Blog, on_delete=models.CA
    headline = models.CharField(max_length=255)
    body_text = models.TextField()
    pub_date = models.DateField()
    mod_date = models.DateField(default=date.today)
    authors = models.ManyToManyField(Author)
    number_of_comments = models.IntegerField(default=0
    number_of_pingbacks = models.IntegerField(default=
    rating = models.IntegerField(default=5)

    def __str__(self):
        return self.headline
```

Creating objects

To represent database-table data in Python objects, Django uses an intuitive system: A model class represents a database table, and an instance of that class represents a particular record in the database table. To create an object, instantiate it using keyword arguments to the model class, then call save() to save it to the database. Assuming models live in a models.py file inside a blog Django app, here is an example:

```
>>> from blog.models import Blog
```

```
>>> b = Blog(name="Beatles Blog", tagline="All the latest Beatles news.")
>>> b.save()
```

This performs an INSERT SQL statement behind the scenes. Django doesn't hit the database until you explicitly call save(). The save() method has no return value.

Saving changes to objects

To save changes to an object that's already in the database, use save(). Given a Blog instance b5 that has already been saved to the database, this example changes its name and updates its record in the database:

```
>>> b5.name = "New name"
>>> b5.save()
```

This performs an UPDATE SQL statement behind the scenes. Django doesn't hit the database until you explicitly call save()

Saving ForeignKey and ManyToManyField fields

Updating a ForeignKey field works the same way as saving a normal field – assign an object of the right type to the field in question. This example updates the blog attribute of an Entry instance entry, assuming appropriate instances of Entry and Blog are already saved to the database (so we can retrieve them below):

```
>>> from blog.models import Blog, Entry
>>> entry = Entry.objects.get(pk=1)
>>> cheese_blog = Blog.objects.get(name="Cheddar Talk")
>>> entry.blog = cheese_blog
>>> entry.save()
```

Updating a ManyToManyField works a little differently – use the add() method on the field to add a record to the relation. This example adds the Author instance joe to the entry object:

```
>>> from blog.models import Author
>>> joe = Author.objects.create(name="Joe")
>>> entry.authors.add(joe)
```

To add multiple records to a ManyToManyField in one go, include multiple arguments in the call to add(), like this:

```
>>> john = Author.objects.create(name="John")
>>> paul = Author.objects.create(name="Paul")
>>> george = Author.objects.create(name="George")
>>> ringo = Author.objects.create(name="Ringo")
>>> entry.authors.add(john, paul, george, ringo)
```

Retrieving objects

To retrieve objects from your database, construct a QuerySet via a Manager on your model class. A QuerySet represents a collection of objects from your database. It can have zero, one or many filters. Filters narrow down the query results based on the given parameters. In SQL terms, a QuerySet equates to a SELECT statement, and a filter is a limiting clause such as WHERE or LIMIT. You get a QuerySet by using your model's Manager. Each model has at least one Manager, and it's called objects by default. Access it directly via the model class, like so:

Note: Managers are accessible only via model classes, rather than from model instances, to enforce a separation between "table-level" operations and "record-level" operations.

The Manager is the main source of QuerySets for a model. For example, Blog.objects.all() returns a QuerySet that contains all Blog objects in the database.

Retrieving all objects

The simplest way to retrieve objects from a table is to get all of them. To do this, use the all() method on a Manager:

```
>>> all_entries = Entry.objects.all()
```

Retrieving specific objects with filters

The QuerySet returned by all() describes all objects in the database table. Usually, though, you'll need to select only a subset of the complete set of objects. To create such a subset, you refine the initial QuerySet, adding filter conditions. The two most common ways to refine a QuerySet are:

filter(**kwargs)

Returns a new QuerySet containing objects that match the given lookup parameters.

exclude(**kwargs)

Returns a new QuerySet containing objects that do not match the given lookup parameters.

The lookup parameters (**kwargs in the above function definitions) should be in the format described in
Field lookups below.
For example, to get a QuerySet of blog entries from the year 2006, use filter() like so:

```
Entry.objects.filter(pub_date__year=2006)
```

Chaining filters

The result of refining a QuerySet is itself a QuerySet, so it's possible to chain refinements together. For example:

```
>>> Entry.objects.filter(headline__startswith="What").exclude(
...     pub_date__gte=datetime.date.today()
... ).filter(pub_date__gte=datetime.date(2005, 1, 30))
```

This takes the initial QuerySet of all entries in the database, adds a filter, then an exclusion, then another filter. The result is a QuerySet containing all entries with a headline that starts with "What", that were published between January 30, 2005, and the current day.

Filtered QuerySets are unique

Each time you refine a QuerySet, you get a brand-new QuerySet that is in no way bound to the previous QuerySet. Each refinement creates a separate and distinct QuerySet that can be stored, used and reused.

```
>>> q1 = Entry.objects.filter(headline__startswith="What")
>>> q2 = q1.exclude(pub_date__gte=datetime.date.today())
>>> q3 = q1.filter(pub_date__gte=datetime.date.today())
```

QuerySets are lazy

QuerySets are lazy – the act of creating a QuerySet doesn't involve any database activity. You can stack filters together all day long, and Django won't run the query until the QuerySet is evaluated. Take a look at this example:

```
>>> q = Entry.objects.filter(headline__startswith="What")
>>> q = q.filter(pub_date__lte=datetime.date.today())
>>> q = q.exclude(body_text__icontains="food")
>>> print(q)
```

Though this looks like three database hits, in fact it hits the database only once, at the last line (print(q)). In general, the results of a QuerySet aren't fetched from the database until you "ask" for them. When you do, the QuerySet is evaluated by accessing the database.

Retrieving a single object with get()

filter() will always give you a QuerySet, even if only a single object matches the query - in this case, it will be a QuerySet containing a single element. If you know there is only one object that matches your query, you can use the get() method on a Manager which returns the object directly:

```
>>> one_entry = Entry.objects.get(pk=1)
```

Note that there is a difference between using get(), and using filter() with a slice of [0]. If there are no results that match the query, get() will raise a DoesNotExist exception. This exception is an attribute of the model class that the query is being performed on - so in the code above, if there is no Entry object with a primary key of 1, Django will raise Entry.DoesNotExist. Similarly, Django will complain if more than one item matches the get() query. In this case, it will raise MultipleObjectsReturned, which again is an attribute of the model class itself.

Other QuerySet methods

Most of the time you'll use all(), get(), filter() and exclude() when you need to look up objects from the database.

The field specified in a lookup must be the name of a model field. There's one exception though, in case of a ForeignKey you can specify the field name suffixed with _id. In this case, the value parameter is expected to contain the raw value of the foreign model's primary key. For example:

```
>>> Entry.objects.filter(blog_id=4)
```

If you pass an invalid keyword argument, a lookup function will raise TypeError. The database API supports about two dozen lookup types;

Limiting QuerySets

Use a subset of Python's array-slicing syntax to limit your QuerySet to a certain number of results. This is the equivalent of SQL's LIMIT and OFFSET clauses. For example, this returns the first 5 objects (LIMIT 5):

```
>>> Entry.objects.all()[:5]
```

This returns the sixth through tenth objects (`OFFSET 5 LIMIT 5`):

```
>>> Entry.objects.all()[5:10]
```

Negative indexing (i.e., Entry.objects.all()[-1]) is not supported.

Field lookups

Field lookups are how you specify the meat of an SQL WHERE clause. They're specified as keyword arguments to the QuerySet methods filter(), exclude() and get(). Basic lookups keyword arguments take the form field__lookuptype=value. (That's a double-underscore). For example:

```
>>> Entry.objects.filter(pub_date__lte="2006-01-01")
```

translates (roughly) into the following SQL:

```
SELECT * FROM blog_entry WHERE pub_date <= '2006-01-01';
```

The database API supports about two dozen lookup types; a complete reference can be found in the field lookup reference. To give you a taste of what's available, here's some of the more common lookups you'll probably use:

exact

An "exact" match. For example:

```
>>> Entry.objects.get(headline__exact="Cat bites dog")
```

If you don't provide a lookup type – that is, if your keyword argument doesn't contain a double underscore – the lookup type is assumed to be exact.

iexact

A case-insensitive match. So, the query:

```
>>> Blog.objects.get(name__iexact="beatles blog")
```

contains

Case-sensitive containment test. For example:

```
Entry.objects.get(headline__contains="Lennon")
```

startswith, endswith

Starts-with and ends-with search, respectively. There are also case-insensitive versions called istartswith and iendswith.

Lookups that span relationships

Django offers a powerful and intuitive way to "follow" relationships in lookups, taking care of the SQL JOINs for you automatically, behind the scenes. To span a relationship, use the field name of related fields across models, separated by double underscores, until you get to the field you want. This example retrieves all Entry objects with a Blog whose name is 'Beatles Blog':

```
>>> Entry.objects.filter(blog__name="Beatles Blog")
```

If you are filtering across multiple relationships and one of the intermediate models doesn't have a value that meets the filter condition, Django will treat it as if there is an empty (all values are NULL), but valid, object there. All this means is that no error will be raised.

```
Blog.objects.filter(entry__authors__name="Lennon")
```

(if there was a related Author model), if there was no author associated with an entry, it would be treated as if there was also no name attached, rather than raising an error because of the missing author. Usually this is exactly what you want to have happen. The only case where it might be confusing is if you are using isnull. Thus:

```
Blog.objects.filter(entry__authors__name__isnull=True)
```

will return Blog objects that have an empty name on the author and those which have an empty author on the entry. If you don't want those latter objects, you could write:

```
Blog.objects.filter(entry__authors__isnull=False,
entry__authors__name__isnull=True)
```

Spanning multi-valued relationships

When spanning a ManyToManyField or a reverse ForeignKey (such as from Blog to Entry), filtering on multiple attributes raises the question of whether to require each attribute to coincide in the same related object. We might seek blogs that have an entry from 2008 with "Lennon" in its headline, or we might seek blogs that merely have any entry from 2008 as well as some newer or older entry with "Lennon" in its headline. To select all blogs containing at least one entry from 2008 having "Lennon" in its headline (the same entry satisfying both conditions), we would write:

```
Blog.objects.filter(entry__headline__contains="Lennon",
entry__pub_date__year=2008)
```

Otherwise, to perform a more permissive query selecting any blogs with merely some entry with "Lennon" in its headline and some entry from 2008, we would write:

```
Blog.objects.filter(entry__headline__contains="Lennon").filter(
    entry__pub_date__year=2008
)
```

```
>>> from datetime import date
>>> beatles = Blog.objects.create(name="Beatles Blog")
>>> pop = Blog.objects.create(name="Pop Music Blog")
>>> Entry.objects.create(
...     blog=beatles,
...     headline="New Lennon Biography",
...     pub_date=date(2008, 6, 1),
... )
<Entry: New Lennon Biography>
>>> Entry.objects.create(
...     blog=beatles,
...     headline="New Lennon Biography in Paperback",
...     pub_date=date(2009, 6, 1),
... )
<Entry: New Lennon Biography in Paperback>
>>> Entry.objects.create(
...     blog=pop,
...     headline="Best Albums of 2008",
...     pub_date=date(2008, 12, 15),
... )
<Entry: Best Albums of 2008>
>>> Entry.objects.create(
...     blog=pop,
...     headline="Lennon Would Have Loved Hip Hop",
...     pub_date=date(2020, 4, 1),
... )
<Entry: Lennon Would Have Loved Hip Hop>
>>> Blog.objects.filter(
...     entry__headline__contains="Lennon",
...     entry__pub_date__year=2008,
... )
<QuerySet [<Blog: Beatles Blog>]>
>>> Blog.objects.filter(
...     entry__headline__contains="Lennon",
... ).filter(
...     entry__pub_date__year=2008,
... )
<QuerySet [<Blog: Beatles Blog>, <Blog: Beatles Blog>, <Blog: Pop Music Blog]>
```

Note: As the second (more permissive) query chains multiple filters, it performs multiple joins to the primary model, potentially yielding duplicates.

Note: The behavior of filter() for queries that span multi-value relationships, as described above, is not implemented equivalently for exclude(). Instead, the conditions in a single exclude() call will not necessarily refer to the same item . For example, the following query would exclude blogs that contain both entries with “Lennon” in the headline and entries published in 2008:

```
Blog.objects.exclude(
    entry__headline__contains="Lennon",
    entry__pub_date__year=2008,
)
```

```
Blog.objects.exclude(
    entry__in=Entry.objects.filter(
        headline__contains="Lennon",
        pub_date__year=2008,
    ),
)
```

However, unlike the behavior when using filter(), this will not limit blogs based on entries that satisfy both conditions. To do that, i.e., to select all blogs that do not contain entries published with “Lennon” that were published in 2008, you need to make two queries:

Break..... 1 ----

Filters can reference fields on the model

Django provides F expressions to allow such comparisons. Instances of F() act as a reference to a model field within a query. These references can then be used in query filters to compare the values of two different fields on the same model instance. For example, to find a list of all blog entries that have had more comments than pingbacks, we construct an F() object to reference the pingback count, and use that F() object in the query:

```
>>> from django.db.models import F
>>> Entry.objects.filter(number_of_comments__gt=F("number_of_pingbacks"))
```

Django supports the use of addition, subtraction, multiplication, division, modulo, and power arithmetic with F() objects, both with constants and with other F() objects. To find all the blog entries with more than twice as many comments as pingbacks, we modify the query:

```
>>> Entry.objects.filter(number_of_comments__gt=F("number_of_pingbacks") * 2)
```

You can also use the double underscore notation to span relationships in an F() object. An F() object with a double underscore will introduce any joins needed to access the related object. For example, to retrieve all the entries where the author's name is the same as the blog name, we could issue the query:

```
>>> Entry.objects.filter(authors__name=F("blog__name"))
```

For date and date/time fields, you can add or subtract a timedelta object. The following would return all entries that were modified more than 3 days after they were published:

```
>>> from datetime import timedelta
>>> Entry.objects.filter(mod_date__gt=F("pub_date") + timedelta(days=3))
```

The F() objects support bitwise operations by .bitand(), .bitor(), .bitxor(), .bitrightshift(), and .bitleftshift().
For example:

```
>>> F("somefield").bitand(16)
```

Expressions can reference transforms

Django supports using transforms in expressions.

For example, to find all Entry objects published in the same year as they were last modified:

```
>>> from django.db.models import F
>>>Entry.objects.filter(pub_date__year=F("mod_date__year"))
```

To find the earliest year an entry was published, we can issue the query:

```
>>> from django.db.models import Min
>>>Entry.objects.aggregate(first_published_year=Min("pub_date__year"))
```

The pk lookup shortcut

For convenience, Django provides a pk lookup shortcut, which stands for "primary key". In the example Blog model, the primary key is the id field, so these three statements are equivalent:

```
>>> Blog.objects.get(id__exact=14) # Explicit form
>>> Blog.objects.get(id=14) # __exact is implied
>>> Blog.objects.get(pk=14) # pk implies id__exact
```

The use of pk isn't limited to __exact queries – any query term can be combined with pk to perform a query on the primary key of a model:

Caching and QuerySets

Each QuerySet contains a cache to minimize database access. Understanding how it works will allow you to write the most efficient code.

In a newly created QuerySet, the cache is empty. The first time a QuerySet is evaluated – and, hence, a database query happens – Django saves the query results in the QuerySet's cache and returns the results that have been explicitly requested (e.g., the next element, if the QuerySet is being iterated over). Subsequent evaluations of the QuerySet reuse the cached results.

Keep this caching behavior in mind, because it may bite you if you don't use your QuerySets correctly. For example, the following will create two QuerySets, evaluate them, and throw them away:

```
>>> print([e.headline for e in Entry.objects.all()])
>>> print([e.pub_date for e in Entry.objects.all()])
```

That means the same database query will be executed twice, effectively doubling your database load. Also, there's a possibility the two lists may not include the same database records, because an Entry may have been added or deleted in the split second between the two requests. To avoid this problem, save the QuerySet and reuse it:

```
>>> queryset = Entry.objects.all()
>>> print([p.headline for p in queryset]) # Evaluate the query set.
>>> print([p.pub_date for p in queryset]) # Reuse the cache from the evaluation
```

When QuerySets are not cached

Querysets do not always cache their results. When evaluating only part of the queryset, the cache is checked, but if it is not populated then the items returned by the subsequent query are not cached. Specifically, this means that limiting the queryset using an array slice or an index will not populate the cache. For example, repeatedly getting a certain index in a queryset object will query the database each time:

```
>>> queryset = Entry.objects.all()
>>> print(queryset[5]) # Queries the database
>>> print(queryset[5]) # Queries the database again
```

However, if the entire queryset has already been evaluated, the cache will be checked instead:

```
# Queries the database
>>>
print(queryset[5]) #
Uses cache
>>>
print(queryset[5]) #
```

Uses cache

Here are some examples of other actions that will result in the entire queryset being evaluated and therefore populate the cache:

```
>>> [entry for entry
in queryset]
>>> bool(queryset)
>>> entry in queryset
>>> list(queryset)
```

Asynchronous queries

If you are writing asynchronous views or code, you cannot use the ORM for queries in quite the way we have described above, as you cannot call blocking synchronous code from asynchronous code - it will block up the event loop (or, more likely, Django will notice and raise a SynchronousOnlyOperation to stop that from happening). Fortunately, you can do many queries using Django's asynchronous query APIs. Every method that might block - such as get() or delete() - has an asynchronous variant (aget() or adelete()), and when you iterate over results, you can use asynchronous iteration (async for) instead.

Transactions

Transactions are not currently supported with asynchronous queries and updates. You will find that trying to use one raises SynchronousOnlyOperation.

Querying JSONField

Lookups implementation is different in JSONField, mainly due to the existence of key transformations. To demonstrate, we will use the following example model:

```
from django.db import models

class Dog(models.Model):
    name = models.CharField(max_length=200)
    data = models.JSONField(null=True)

    def __str__(self):
        return self.name
```

Storing and querying for None

As with other fields, storing None as the field's value will store it as SQL NULL. While not recommended, it is possible to store JSON scalar null instead of SQL NULL by using Value(None, JSONField()). Whichever of the values is stored, when retrieved from the database, the Python representation of the JSON scalar null is the same as SQL NULL, i.e. None. Therefore, it can be hard to distinguish between them. This only applies to None as the top-level value of the field. If None is inside a list or dict, it will always be interpreted as JSON null. When querying, None value will always be interpreted as JSON null. To query for SQL NULL, use isnull:

```
>>> Dog.objects.create(name="Max", data=None) # SQL NULL.
<Dog: Max>
>>> Dog.objects.create(name="Archie", data=Value(None, JSONField())) # JSON null.
<Dog: Archie>
>>> Dog.objects.filter(data=None)
<QuerySet [<Dog: Archie>]>
>>> Dog.objects.filter(data=Value(None, JSONField()))
<QuerySet [<Dog: Archie>]>
>>> Dog.objects.filter(data__isnull=True)
<QuerySet [<Dog: Max>]>
>>> Dog.objects.filter(data__isnull=False)
<QuerySet [<Dog: Archie>]>
```

Unless you are sure you wish to work with SQL NULL values, consider setting null=False and providing a suitable default for empty values, such as default=dict.

Key, index, and path transforms

To query based on a given dictionary key, use that key as the lookup name:

```
>>> Dog.objects.create(
...     name="Rufus",
...     data={
...         "breed": "labrador",
...         "owner": {
...             "name": "Bob",
...             "other_pets": [
...                 {
...                 "name": "Fishy",
...                 }
...             ],
...         },
...     },
... )
<Dog: Rufus>
>>> Dog.objects.create(name="Meg", data={"breed": "collie", "owner": None})
<Dog: Meg>
>>> Dog.objects.filter(data__breed="collie")
<QuerySet [<Dog: Meg>]>
```

Multiple keys can be chained together to form a path lookup:

```
>>> Dog.objects.filter(data__owner__name="Bob")
<QuerySet [<Dog: Rufus>]>
```

If the key is an integer, it will be interpreted as an index transform in an array:

```
>>>
Dog.objects.filter(data__owner__other_pets__0__name="Fishy"
)
<QuerySet [<Dog: Rufus>]>
```

Note: The lookup examples given above implicitly use the exact lookup. Key, index, and path transforms can also be chained with: icontains, endswith, iendswith, iexact, regex, iregex, startswith, istartswith, lt, lte, gt, and gte, as well as with Containment and key lookups.

KT() expressions

class KT(lookup)

Represents the text value of a key, index, or path transform of JSONField. You can use the double underscore notation in lookup to chain dictionary key and index transforms.

For example:

```
>>> from django.db.models.fields.json import KT
>>> Dog.objects.create(
... name="Shep",
... data={
...         "owner": {"name": "Bob"},
...         "breed": ["collie", "lhasa apso"],
...     },
... )
<Dog: Shep>
>>> Dogs.objects.annotate(
... first_breed=KT("data__breed__1"), owner_name=KT("data__owner__name")
... ).filter(first_breed__startswith="lhasa", owner_name="Bob")
<QuerySet [<Dog: Shep>]>
```

Containment and key lookups

contains

The contains lookup is overridden on JSONField. The returned objects are those where the given dict of key-value pairs are all contained in the top-level of the field. For example:

```
>>> Dog.objects.create(name="Rufus", data={"breed": "labrador", "owner": "Bob"})
<Dog: Rufus>
>>> Dog.objects.create(name="Meg", data={"breed": "collie", "owner": "Bob"})
<Dog: Meg>
>>> Dog.objects.create(name="Fred", data={})
<Dog: Fred>
>>> Dog.objects.filter(data__contains={"owner": "Bob"})
<QuerySet [<Dog: Rufus>, <Dog: Meg>]>
>>> Dog.objects.filter(data__contains={"breed": "collie"})
<QuerySet [<Dog: Meg>]>
```

contained_by

This is the inverse of the contains lookup - the objects returned will be those where the key-value pairs on the object are a subset of those in the value passed. For example:

```
>>> Dog.objects.create(name="Rufus", data={"breed": "labrador", "owner": "Bob"})
<Dog: Rufus>
>>> Dog.objects.create(name="Meg", data={"breed": "collie", "owner": "Bob"})
<Dog: Meg>
>>> Dog.objects.create(name="Fred", data={})
<Dog: Fred>
>>> Dog.objects.filter(data__contained_by={"breed": "collie", "owner": "Bob"})
<QuerySet [<Dog: Meg>, <Dog: Fred>]>
>>> Dog.objects.filter(data__contained_by={"breed": "collie"})
<QuerySet [<Dog: Fred>]>
```

has_key

Returns objects where the given key is in the top-level of the data. For example:

```
>>> Dog.objects.create(name="Rufus", data={"breed": "labrador"})
<Dog: Rufus>
>>> Dog.objects.create(name="Meg", data={"breed": "collie", "owner": "Bob"})
<Dog: Meg>
>>> Dog.objects.filter(data__has_key="owner")
<QuerySet [<Dog: Meg>]>
```

has_keys

Returns objects where all of the given keys are in the top-level of the data. For example:

```
>>> Dog.objects.create(name="Rufus", data={"breed": "labrador"})
<Dog: Rufus>
>>> Dog.objects.create(name="Meg", data={"breed": "collie", "owner": "Bob"})
<Dog: Meg>
>>> Dog.objects.filter(data__has_keys=["breed", "owner"])
<QuerySet [<Dog: Meg>]>
```

has_any_keys

Returns objects where any of the given keys are in the top-level of the data. For example:

```
>>> Dog.objects.create(name="Rufus", data={"breed": "labrador"})
<Dog: Rufus>
>>> Dog.objects.create(name="Meg", data={"owner": "Bob"})
<Dog: Meg>
>>> Dog.objects.filter(data__has_any_keys=["owner", "breed"])
<QuerySet [<Dog: Rufus>, <Dog: Meg>]>
```

Break……2--

Complex lookups with Q objects

Keyword argument queries – in filter(), etc. – are "AND "ed together. If you need to execute more complex queries (for example, queries with OR statements), you can use Q objects. A Q object (django.db.models.Q) is an object used to encapsulate a collection of keyword arguments. These keyword arguments are specified as in "Field lookups" above. For example, this Q object encapsulates a single LIKE query:

```
from django.db.models import Q
Q(question__startswith="What")
```

Q objects can be combined using the &, |, and ^ operators. When an operator is used on two Q objects, it yields a new Q object. For example, this statement yields a single Q object that represents the "OR" of two *"question__startswith"* queries*:*

```
Q(question__startswith="Who") | Q(question__startswith="What")
```

You can compose statements of arbitrary complexity by combining Q objects with the &, |, and ^ operators and use parenthetical grouping. Also, Q objects can be negated using the ~ operator, allowing for combined lookups that combine both a normal query and a negated (NOT) query:

```
Q(question__startswith="Who") | ~Q(pub_date__year=2005)
```

Each lookup function that takes keyword-arguments (e.g. filter(), exclude(), get()) can also be passed one or more Q objects as positional (not-named) arguments. If you provide multiple Q object arguments to a lookup function, the arguments will be "AND"ed together. For example:

```
Poll.objects.get(
    Q(question__startswith="Who"),
    Q(pub_date=date(2005, 5, 2)) | Q(pub_date=date(2005, 5, 6)),
)
```

... roughly translates into the SQL:

```
SELECT * from polls WHERE question LIKE 'Who%'
    AND (pub_date = '2005-05-02' OR pub_date = '2005-05-06')
```

Lookup functions can mix the use of Q objects and keyword arguments. All arguments provided to a lookup function (be they keyword arguments or Q objects) are "AND "ed together. However, if a Q object is provided, it must precede the definition of any keyword arguments.

Comparing objects

To compare two model instances, use the standard Python comparison operator, the double equals sign: ==.

Behind the scenes, that compares the primary key values of two models. Using the Entry example above, the following two statements are equivalent:

```
>>> some_entry == other_entry
>>> some_entry.id == other_entry.id
```

If a model's primary key isn't called id, no problem. Comparisons will always use the primary key, whatever it's called.

Deleting objects

The delete method, conveniently, is named delete(). This method immediately deletes the object and returns the number of objects deleted and a dictionary with the number of deletions per object type. Example:

```
>>> e.delete()
(1, {'blog.Entry': 1})
```

You can also delete objects in bulk. Every QuerySet has a delete() method, which deletes all members of that QuerySet. For example, this deletes all Entry objects with a pub_date year of 2005:

```
>>>
Entry.objects.filter(pub_date__year=2005)
.delete()
(5, {'webapp.Entry': 5})
```

Keep in mind that this will, whenever possible, be executed purely in SQL, and so the delete() methods of individual object instances will not necessarily be called during the process. If you've provided a custom delete() method on a model class and want to ensure that it is called, you will need to "manually" delete instances of that model (e.g., by iterating over a QuerySet and calling delete() on each object individually) rather than using the bulk delete() method of a QuerySet.
When Django deletes an object, by default it emulates the behavior of the SQL constraint ON DELETE CASCADE
– in other words, any objects which had foreign keys pointing at the object to be deleted will be deleted along with it. For example:

```
b = Blog.objects.get(pk=1)
# This will delete the Blog and all of its Entry objects.
b.delete()
```

This cascade behavior is customizable via the on_delete argument to the ForeignKey.
Note that delete() is the only QuerySet method that is not exposed on a Manager itself. This is a safety mechanism to prevent you from accidentally requesting Entry.objects.delete(), and deleting all the entries. If you do want to delete all the objects, then you have to explicitly request a complete query set:

```
Entry.objects.all().delete()
```

Copying model instances

Although there is no built-in method for copying model instances, it is possible to easily create new instance with all fields' values copied. In the simplest case, you can set pk to None and _state.adding to True. Using our blog example:

```
blog = Blog(name="My blog", tagline="Blogging is easy")
blog.save() # blog.pk == 1
blog.pk = None
blog._state.adding = True
blog.save() # blog.pk == 2
```

Things get more complicated if you use inheritance. Consider a subclass of Blog:

```
class ThemeBlog(Blog):
    theme = models.CharField(max_length=200)
django_blog = ThemeBlog(name="Django",
tagline="Django is easy", theme="python")
django_blog.save() # django_blog.pk == 3
```

Due to how inheritance works, you have to set both pk and id to None, and _state.adding to True:

```
django_blog.pk = None
django_blog.id = None
django_blog._state.adding = True
django_blog.save() # django_blog.pk == 4
```

This process doesn't copy relations that aren't part of the model's database table. For example, Entry has aManyToManyField to Author. After duplicating an entry, you must set the many-to-many relations for the new entry:

```
entry = Entry.objects.all()[0] # some previous entry
old_authors = entry.authors.all()
entry.pk = None
entry._state.adding = True
entry.save()
entry.authors.set(old_authors)
```

For a OneToOneField, you must duplicate the related object and assign it to the new object's field to avoid violating the one-to-one unique constraint. For example, assuming entry is already duplicated as above:

```
detail = EntryDetail.objects.all()[0]
detail.pk = None
detail._state.adding = True
detail.entry = entry
detail.save()
```

Updating multiple objects at once

Sometimes you want to set a field to a particular value for all the objects in a QuerySet. You can do this with the update() method. For example:

```
# Update all the headlines with pub_date in
2007.
Entry.objects.filter(pub_date__year=2007).upda
te(headline="Everything is the same")
```

You can only set non-relation fields and ForeignKey fields using this method. To update a non-relation field, provide the new value as a constant. To update ForeignKey fields, set the new value to be the new model instance you want to point to. For example:

```
>>> b = Blog.objects.get(pk=1)
# Change every Entry so that it belongs to
this Blog.
>>> Entry.objects.update(blog=b)
```

The update() method is applied instantly and returns the number of rows matched by the query (which may not be equal to the number of rows updated if some rows already have the new value). The only restriction on the QuerySet being updated is that it can only access one database table: the model's main table. You can filter based on related fields, but you can only update columns in the model's main table. Example:

```
>>> b = Blog.objects.get(pk=1)
# Update all the headlines belonging to this
Blog.
>>>
Entry.objects.filter(blog=b).update(headline="
Everything is the same")
```

Be aware that the update() method is converted directly to an SQL statement. It is a bulk operation for direct updates. It doesn't run any save() methods on your models, or emit the pre_save or post_save signals (which are a consequence of calling save()), or honor the auto_now field option. If you want to save every item in a QuerySet and make sure that the save() method is called on each instance, you don't need any special function to handle that. Loop over them and call save():

```
for item in my_queryset:
item.save()
```

Calls to update can also use F expressions to update one field based on the value of another field in the model. This is especially useful for incrementing counters based upon their current value. For example, to increment the pingback count for every entry in the blog:

Break…… 3 ---

Related objects

When you define a relationship in a model (i.e., `a ForeignKey, OneToOneField, or ManyToManyField)`, instances of that model will have a convenient API to access the related object(s). Using the models at the top of this page, for example, an Entry object e can get its associated Blog object by accessing the blog attribute: `e.blog.` (Behind the scenes, this functionality is implemented by Python descriptors. This shouldn't really matter to you, but we point it out here for the curious.) Django also creates API accessors for the "other" side of the relationship – the link from the related model to
the model that defines the relationship. For example, a Blog object b has access to a list of all related Entry
objects via the `entry_set` attribute: `b.entry_set.all().` All examples in this section use the sample Blog, Author and Entry models defined at the top of this page.

One-to-many relationships

Forward

If a model has a ForeignKey, instances of that model will have access to the related (foreign) object via an attribute of the model.
Example:

```
>>> e = Entry.objects.get(id=2)
>>> e.blog # Returns the related Blog object.
```

You can get and set via a foreign-key attribute. As you may expect, changes to the foreign key aren't saved to the database until you call save(). Example:

```
>>> e = Entry.objects.get(id=2)
>>> e.blog = some_blog
>>> e.save()
```

If a ForeignKey field has null=True set (i.e., it allows NULL values), you can assign None to remove the relation. Example:

```
>>> e = Entry.objects.get(id=2)
>>> e.blog = None
>>> e.save() # "UPDATE blog_entry SET blog_id = NULL ...;"
```

Forward access to one-to-many relationships is cached the first time the related object is accessed. Subsequent
accesses to the foreign key on the same object instance are cached. Example:

```
>>> e = Entry.objects.get(id=2)
>>> print(e.blog) # Hits the database to retrieve the associated Blog.
>>> print(e.blog) # Doesn't hit the database; uses cached version.
```

Note that the select_related() QuerySet method recursively prepopulates the cache of all one-to-many relationships ahead of time. Example:

```
>>> e = Entry.objects.select_related().get(id=2)
>>> print(e.blog) # Doesn't hit the database; uses cached version.
>>> print(e.blog) # Doesn't hit the database; uses cached version.
```

Following relationships "backward"

If a model has a ForeignKey, instances of the foreign-key model will have access to a Manager that returns all instances of the first model. By default, this Manager is named FOO_set, where FOO is the source model name, lowercased. This Manager returns QuerySets, which can be filtered and manipulated as described in the "Retrieving objects" section above.

Example:

```
>>> b = Blog.objects.get(id=1)
>>> b.entry_set.all() # Returns all Entry objects related to Blog.
    # b.entry_set is a Manager that returns QuerySets.
>>> b.entry_set.filter(headline__contains="Lennon")
>>> b.entry_set.count()
```

You can override the FOO_set name by setting the related_name parameter in the ForeignKey definition. For example, if the Entry model was altered to blog = ForeignKey(Blog, on_delete=models.CASCADE, related_name='entries'), the above example code would look like this:

```
>>> b = Blog.objects.get(id=1)
>>> b.entries.all() # Returns all Entry objects related to Blog.
   # b.entries is a Manager that returns QuerySets.
>>> b.entries.filter(headline__contains="Lennon")
>>> b.entries.count()
```

Using a custom reverse manager

By default the RelatedManager used for reverse relations is a subclass of the default manager for that model.If you would like to specify a different manager for a given query you can use the following syntax:

```
from django.db import models
class Entry(models.Model):
    # ...
    objects = models.Manager() # Default Manager
    entries = EntryManager() # Custom Manager
b = Blog.objects.get(id=1)
b.entry_set(manager="entries").all()
```

If EntryManager performed default filtering in its get_queryset() method, that filtering would apply to theall() call.

Specifying a custom reverse manager also enables you to call its custom methods:

```
b.entry_set(manager="entries").is_published()
```

Interaction with prefetching

When calling prefetch_related() with a reverse relation, the default manager will be used. If you want to prefetch related objects using a custom reverse manager, use Prefetch(). For example:

```
from django.db.models import Prefetch
prefetch_manager = Prefetch("entry_set", queryset=Entry.entries.all())
Blog.objects.prefetch_related(prefetch_manager)
```

Additional methods to handle related objects

In addition to the QuerySet methods defined in "Retrieving objects" above, the ForeignKey Manager has additional methods used to handle the set of related objects. A synopsis of each is below, and complete details can be found in the related objects reference.

add(obj1, obj2, ...)

Adds the specified model objects to the related object set.

create(kwargs)**

Creates a new object, saves it and puts it in the related object set. Returns the newly created object.

remove(obj1, obj2, ...)

Removes the specified model objects from the related object set.

clear()

Removes all objects from the related object set.

set(objs)

Replace the set of related objects.

To assign the members of a related set, use the set() method with an iterable of object instances. For example, if e1 and e2 are Entry instances:

```
b = Blog.objects.get(id=1)
b.entry_set.set([e1, e2])
```

If the clear() method is available, any preexisting objects will be removed from the entry_set before all objects in the iterable (in this case, a list) are added to the set. If the clear() method is not available, all objects in the iterable will be added without removing any existing elements. Each "reverse" operation described in this section has an immediate effect on the database. Every addition, creation and deletion is immediately and automatically saved to the database.

Break..... 4 ----

Many-to-many relationships

Both ends of a many-to-many relationship get automatic API access to the other end. The API works similar to a "backward" one-to-many relationship, above. One difference is in the attribute naming: The model that defines the ManyToManyField uses the attribute name of that field itself, whereas the "reverse" model uses the lowercased model name of the original model, plus '_set' (just like reverse one-to-many relationships). An example makes this easier to understand:

```
e = Entry.objects.get(id=3)
e.authors.all() # Returns all Author objects for this Entry.

e.authors.count()
e.authors.filter(name__contains="John")
a = Author.objects.get(id=5)
a.entry_set.all() # Returns all Entry objects for this Author.
```

Like ForeignKey, ManyToManyField can specify related_name. In the above example, if the ManyToManyField in Entry had specified related_name='entries', then each Author instance would have an entries attribute instead of entry_set. Another difference from one-to-many relationships is that in addition to model instances, the add(), set(), and remove() methods on many-to-many relationships accept primary key values. For example, if e1 and e2 are Entry instances, then these set() calls work identically:

```
a = Author.objects.get(id=5)
a.entry_set.set([e1, e2])
a.entry_set.set([e1.pk, e2.pk])
```

One-to-one relationships

One-to-one relationships are very similar to many-to-one relationships. If you define a OneToOneField on your model, instances of that model will have access to the related object via an attribute of the model.
For example:

```
class EntryDetail(models.Model):
    entry = models.OneToOneField(Entry, on_delete=models.CASCADE)
    details = models.TextField()

ed = EntryDetail.objects.get(id=2)
ed.entry # Returns the related Entry object.
```

The difference comes in "reverse" queries. The related model in a one-to-one relationship also has access to a Manager object, but that Manager represents a single object, rather than a collection of objects:

```
e = Entry.objects.get(id=2)
e.entrydetail # returns the related EntryDetail object
```

If no object has been assigned to this relationship, Django will raise a DoesNotExist exception. Instances can be assigned to the reverse relationship in the same way as you would assign the forward relationship:

```
e.entrydetail = ed
```

Falling back to raw SQL

If you find yourself needing to write an SQL query that is too complex for Django's database-mapper to handle, you can fall back on writing SQL by hand. Django has a couple of options for writing raw SQL queries; Finally, it's important to note that the Django database layer is merely an interface to your database. You can access your database via other tools, programming languages or database frameworks; there's nothing Django-specific about your database.