**Complex lookups with Q objects**

Keyword argument queries – in [filter()](https://docs.djangoproject.com/en/2.2/ref/models/querysets/#django.db.models.query.QuerySet.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**](https://docs.djangoproject.com/en/2.2/ref/models/querysets/#django.db.models.Q).

A [**Q object**](https://docs.djangoproject.com/en/2.2/ref/models/querysets/#django.db.models.Q)**(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')

This is equivalent to the following SQL WHERE clause:

WHERE question LIKE 'Who%' OR question LIKE '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()**](https://docs.djangoproject.com/en/2.2/ref/models/querysets/#django.db.models.query.QuerySet.filter), [**exclude()**](https://docs.djangoproject.com/en/2.2/ref/models/querysets/#django.db.models.query.QuerySet.exclude), [**get()**](https://docs.djangoproject.com/en/2.2/ref/models/querysets/#django.db.models.query.QuerySet.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. For example:

Poll.objects.get(

Q(pub\_date=date(2005, 5, 2)) | Q(pub\_date=date(2005, 5, 6)),

question\_\_startswith='Who',

)

… would be a valid query, equivalent to the previous example; but:

Poll.objects.get(

question\_\_startswith='Who',

Q(pub\_date=date(2005, 5, 2)) | Q(pub\_date=date(2005, 5, 6))

)

… would not be valid.

\_\_\_\_\_\_\_[SO Question](https://stackoverflow.com/questions/44017772/how-to-use-q-objects-in-django)\_\_\_\_\_\_\_

Why a simple result.filter(Q(somedbfield\_icontains=q)) returns an error

The simplest variant would be result.filter(somedbfield\_\_icontains=q) Q isn't needed there, Q is used to extend your filtering with logic operators (and, or, not). Also, notice the double underscore before icontains.

\_\_\_\_\_\_\_ Example \_\_\_\_\_\_\_

>>> from blogApp.models import Post

>>> from django.db.models import Q

>>> ps = Post.objects.all()

>>> for p in ps:

... print(p.title)

...

Progressively streamline viral deliverables

Compellingly formulate [Updated#2]

Continually pesontificate distindctive updated

Proactively simplify world-class action items via premier ROI

Synergistically seize

>>> res = Post.objects.filter(Q(title\_\_icontains='continually') | Q(title\_\_startswith='Synergistically'))

>>> for i in res:

... print(i)

...

Continually pesontificate distindctive updated

Synergistically seize

\_\_\_\_\_\_\_ Example \_\_\_\_\_\_\_

>>> for p in posts:

... print(p.title)

...

Progressively engineer collaborative

Progressively streamline viral deliverables

Competently implement excellent

Compellingly formulate [Updated#2]

Continually pesontificate distindctive updated

Assertively negotiate client-based

Proactively simplify world-class action items via premier ROI

Synergistically seize

Compellingly restore ethical ROIs

>>> res = Post.objects.filter(Q(title\_\_icontains='continually') |~Q(title\_\_startswith='Compellingly')) **# using ~Q**

>>> for i in res:

... print(i)

...

Progressively engineer collaborative

Progressively streamline viral deliverables

Competently implement excellent

Continually pesontificate distindctive updated

Assertively negotiate client-based

Proactively simplify world-class action items via premier ROI

Synergistically seize