<https://www.codementor.io/overiq/basics-of-django-orm-cwamhcerp>

# Basics of Django ORM.

Consider app name=blog and model name=Post,Author,Category,Tag

## how start/enter a model

if single model

>>> from blog.models import Post

if multiple model

>>> from blog.models import Post, Author, Category, Tag

At this point tables corresponding to these 4 models are empty.

## Create/Add attribute to this model

Let's start by creating an Author object by two type method:

Method1:

>>> from blog.models import Author

>>>Author.objects.create(name=’mahmud’,email='tom@email.com', active=True)

>>>a.save()

Or

Method2:

>>> from blog.models import Author

>>> a = Author(name='tom', email='tom@email.com', active=True)

>>>a.save()

Or

>>> a2 = Author.objects.create(name='jerry', email='jerry@mail.com' active=True)

>>> a2

<Author: jerry : jerry@mail.com>

>>> a2.pk

4

>>>a2.save()

## View/quarry all attribute

>>>Author.objects.all()

## View/query the individual model attribute

>>> a

<Author: Author object>

or

>>> print(a)

Author object

or

>>>a.name

## View//query fixed attribute

You can access the attributes of an object using the (.) dot operator.

>>> a.name

'tom'

>>>

>>> a.email

'tom@email.com'

>>>

>>> a.active

True

>>>

## Add date/time section

>>> import datetime

>>>

>>> r = Author(name="root", email="root@mail.com", active=True, created\_on=datet

ime.datetime.now(), last\_logged\_in=datetime.datetime.now())

>>> a.save()

>>>

## delete the object from the database.

>>>

>>> a.delete()

(1, {'blog.Author': 1})

>>>

This command removes author tom from the database. However, it is still exists inside the shell.

>>> a

<Author: Author object>

>>>

Sure, the object exists in shell.

## Defining \_\_str\_\_() method on model for multiple attribute

>>>

>>> a

<Author: Author object>

>>>

>>> print(a)

Author object

>>>

A \_\_str\_\_() is a special method which tells Python how to display an object in human readable form. Open models.py inside blog app and make the following changes to the Author model.

class Author(models.Model):

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

email = models.EmailField(unique=True)

active = models.BooleanField(default=False)

created\_on = models.DateTimeField(auto\_now\_add=True)

last\_logged\_in = models.DateTimeField(auto\_now=True)

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

return self.name + " : " + self.email

While we are at it lets add \_\_str\_\_() method to Category, Tag and Post model too.

class Category(models.Model):

...

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

return self.name

class Tag(models.Model):

...

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

return self.name

class Post(models.Model):

...

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

return self.title

Does this ring a bell ? You might say "We are changing our models so we should run makemigrations right ?".

Well No! Most of the time, we run makemigrations command only in the following two cases:

## When we migrations model

1. When we add/modify fields in the model.
2. When we adding/modify Meta classes.

See makemigrations returns "No changes detected".

After adding \_\_str\_\_() to models.py file, if you try to print Author object you would get the same output as before.

>>>

>>> a

<Author: Author object>

>>>

>>> print(a)

Author object

>>>

## How to ad pk/id

When you save an object the primary key is assigned automatically. Once object is saved in the database. You can refer to the primary key using id or pk attribute.

>>>

>>> a2 = Author.objects.create(name='jerry', email='jerry@mail.com' active=True)

>>> a2

<Author: jerry : jerry@mail.com>

>>> a2.pk

4

>>>a2.save()

or

>>>

>>> a.id

2

>>> a.pk

2

>>>

## How to update/replace model attribute

If you want to alter the value of attributes of an object just assign a new value and call the save()method again.

>>>

>>> a.name = 'Tom'

>>> a.email = 'tom@gmail.com'

>>>

>>> a

<Author: Tom : tom@gmail.com>

>>>

These changes are not yet saved to the database, you have to call save() to make the changes permanent.

>>>

>>> a.save()

>>>

## Database Access through Managers

To access objects manager type model class name followed by the (.) dot operator then the objects manager.

>>>

>>> Author.objects

<django.db.models.manager.Manager object at 0x00000000042CE978>

>>> type(Author.objects)

<class 'django.db.models.manager.Manager'>

>>>

Let's discuss some important methods of objects manager.

## The create() method

The create() method allows us to create and commit object to the database in one go, instead of separately calling the save() method. For example:

>>>

>>> a2 = Author.objects.create(name='jerry', email='jerry@mail.com')

>>> a2

<Author: jerry : jerry@mail.com>

>>> a2.pk

4

>>>a2.save()

## The bulk\_create() method

The bulk\_create() method allows us to create and commit multiple objects in one step. It accepts a list of objects.

>>>

>>> Author.objects.bulk\_create([

... Author(name='spike', email='spike@mail.com'),

... Author(name='tyke', email='tyke@mail.com'),

... Author(name='droopy', email='droopy@mail.com'),

... ])

[<Author: spike : spike@mail.com>, <Author: tyke : tyke@mail.com>, <Author: droo

py : droopy@mail.com>]

>>>

At this point blog\_author table should looks like this:

## The all() method

The all() method fetches all the records from the table. For example:

>>>

>>> Author.objects.all()

<QuerySet [<Author: tom : tom@email.com>, <Author: jerry : jerry@mail.com>, <Aut

hor: spike : spike@mail.com>, <Author: tyke : tyke@mail.com>, <Author: droopy :

droopy@mail.com>]>

>>>

The above command fetches all the records from the Author's table.

## access individual members in a QuerySet objects

The all() method returns a QuerySet object. A QuerySet object looks a lot like a list, but it is not an actual list, in some ways it behaves just like lists. For example, you can access individual members in a QuerySet objects using an index number.

>>>

>>> r = Author.objects.all()

>>> r

<QuerySet [<Author: tom : tom@email.com>, <Author: jerry : jerry@mail.com>, <Aut

hor: spike : spike@mail.com>, <Author: tyke : tyke@mail.com>, <Author: droopy :

droopy@mail.com>]>

>>>

>>> r[0]

<Author: tom : tom@email.com>

>>>

>>> r[1]

<Author: jerry : jerry@mail.com>

>>>

>>> r[2]

<Author: spike : spike@mail.com>

>>>

Although r points to an object of type QuerySet but r[0], r[1], r[2] and so on, points to an object of type Author.

>>> type(r[0])

<class 'blog.models.Author'>

>>>

>>> type(r[1])

<class 'blog.models.Author'>

>>>

>>> type(r[3])

<class 'blog.models.Author'>

>>>

It is important to note that some methods of objects manager returns QuerySet while some do not.

## Use for loop in query set /ORM

QuerySet is iterable just like a list. You can use a for loop to iterate through all of the objects in a QuerySet object.

>>>

>>> r = Author.objects.all()

>>> for a in r:

... print("Author: {0}".format(a.name))

...

Author: tom

Author: jerry

Author: spike

Author: tyke

Author: droopy

>>>

## The count() method

The count() method returns the total number of records in a database table.

>>>

>>> Author.objects.count()

5

>>>

Author.objects.all().count() also returns the same thing.

## Filtering records using the filter() method

Most of the time you would only want to work with a subset of data. Django provides a filter() method which returns a subset of data. It accepts field names as keyword arguments and returns a QuerySet object.

>>>

>>> Author.objects.filter(name='tom')

<QuerySet [<Author: tom : tom@email.com>]>

>>>

>>> Author.objects.filter(name='johnny')

<QuerySet []>

>>>

Author.objects.filter(name='tom') translates to SQL something like this:

SELECT \* from blog\_author

where name = 'tom'

As database has only one record where name is 'tom', the QuerySet object contains only a single record. If we had two records where name is 'tom' then filter() would have returned a QuerySet object containing two Author objects.

Similarly, Author.objects.filter(name='johnny') translates to SQL rougly as follows:

SELECT \* from blog\_author

where name = 'johnny'

As there are no records where name is 'johnny' an empty QuerySet is returned.

We can also directly print the raw SQL django uses to query the database using the query attribute of the QuerySet object.

>>>

>>> print(Author.objects.filter(name='tom').query)

SELECT "blog\_author"."id", "blog\_author"."name", "blog\_author"."email", "blog\_au

thor"."active", "blog\_author"."created\_on", "blog\_author"."last\_logged\_in" FROM

"blog\_author" WHERE "blog\_author"."name" = tom

>>>

Matching performed using keyword arguments are case-sensitive.

>>>

>>> Author.objects.filter(email='jerry@mail.com')

<QuerySet [<Author: jerry : jerry@mail.com>]>

>>>

>>> Author.objects.filter(email='JERRY@mail.com')

<QuerySet []>

>>>

The last query returns an empty QuerySet because there are no records where email is "JERRY@mail.com", although there is a record where name is "jerry@mail.com".

You can also pass multiple keyword arguments to the filter() method.

>>>

>>> Author.objects.filter(name='spike', email='spike@mail.com')

<QuerySet [<Author: spike : spike@mail.com>]>

>>>

This traslates to SQL rougly as follows:

SELECT \* from blog\_author

where name = 'spike' and email ='spike@mail.com'

## Django Field Lookups

In addition to passing field names as keyword arguments. You can also use something called lookups.  
Managers and QuerySet objects comes with a feature called lookups. A lookup is composed of a model field followed by two underscores (\_\_) which is then followed by lookup name. Let's take some examples.

### \_\_contains lookup

>>>

>>> Author.objects.filter(name\_\_contains="ke")

<QuerySet [<Author: spike : spike@mail.com>, <Author: tyke : tyke@mail.com>]>

>>>

>>>

Here \_\_contains lookup finds all the records where name field contains the word "ke".

Author.objects.filter(name\_\_contains="ke") translates to SQL roughly as follows:

SELECT \* from blog\_author

where name like '%ke%'

Matching performed by \_\_contains lookup is case-sensitive. If you want to perform case-insensitive match use \_\_icontains. However, SQLite doesn't support case-sensitive LIKE statements. As a result \_\_contains and \_\_icontains returns the same result.

### \_\_startswith lookup

>>>

>>> Author.objects.filter(name\_\_startswith="t")

<QuerySet [<Author: tom : tom@email.com>, <Author: tyke : tyke@mail.com>]>

>>>

\_\_startswith lookup finds all the records whose name field start with "t". There also exists a complementary lookup called \_\_endswith.

>>>

>>> Author.objects.filter(email\_\_endswith="com")

<QuerySet [<Author: tom : tom@email.com>, <Author: jerry : jerry@mail.com>, <Aut

hor: spike : spike@mail.com>, <Author: tyke : tyke@mail.com>, <Author: droopy :

droopy@mail.com>]>

>>>

>>>

Here \_\_endswith lookup finds all the records whose email ends with "com". Both \_\_startswith and \_\_endswith are case-sensitive. Their case-insensitive equivalents are \_\_istartswith and \_\_iendswith.

## \_\_gt lookup

>>>

>>> Author.objects.filter(id\_\_gt=3)

<QuerySet [<Author: spike : spike@mail.com>, <Author: tyke : tyke@mail.com>, <Au

thor: droopy : droopy@mail.com>]>

>>>

here \_\_gt lookup finds all the records whose id or primary key (pk) is greater than 3. There also exists a complementary lookup called \_\_lt.

>>>

>>> Author.objects.filter(id\_\_lt=3)

<QuerySet [<Author: tom : tom@email.com>]>

>>>

>>>

Here \_\_lt lookups finds all the records whose primary key is less than 3. There are two more similar lookups called \_\_gte and \_\_lte which finds records which are greater than or equal to and less than or equal to respectively.

To view full list of field lookups check out the Django documentation on [lookups](https://docs.djangoproject.com/en/1.10/ref/models/querysets/#field-lookups).

## Retrieving a single record using the get() method

The filter() method described in the above section returns a QuerySet, sometimes we just want to fetch a single record from the table. To handle these situations objects manager provides a get()method. The get() method accepts same parameters as filter() method but it returns only a single object. If it finds multiple objects it raises a MultipleObjectsReturned exception. If it doesn't find any object it raises DoesNotExist exception.

>>>

>>> Author.objects.get(name="tom")

<Author: tom : tom@email.com>

>>>

>>> Author.objects.filter(name="tom")

<QuerySet [<Author: tom : tom@email.com>]>

>>>

Notice the difference between the output of get() and filter() method. For the same parameter they both two different results. The get() method returns a instance of Author while filter() methods returns a QuerySet object.

Lets see what happens, if get() method encounters multiple records.

>>>

>>> Author.objects.filter(name\_\_contains="ke")

<QuerySet [<Author: spike : spike@mail.com>, <Author: tyke : tyke@mail.com>]>

>>>

>>>

>>> Author.objects.get(name\_\_contains="ke")

Traceback (most recent call last):

File "<console>", line 1, in <module>

File "C:\Users\K\TGDB\env\lib\site-packages\django\db\models\manager.py", line

85, in manager\_method

return getattr(self.get\_queryset(), name)(\*args, \*\*kwargs)

File "C:\Users\K\TGDB\env\lib\site-packages\django\db\models\query.py", line 3

89, in get

(self.model.\_meta.object\_name, num)

blog.models.MultipleObjectsReturned: get() returned more than one Author -- it r

eturned 2!

>>>

Here get() method raises a MultipleObjectsReturned because there are multiple objects in the database that matches the given parameter.

Similarly, if you try to access an object which do not exists then the get() method will raise an  
DoesNotExist exception.

>>>

>>> Author.objects.get(name\_\_contains="captain planet")

Traceback (most recent call last):

File "<console>", line 1, in <module>

File "C:\Users\K\TGDB\env\lib\site-packages\django\db\models\manager.py", line

85, in manager\_method

return getattr(self.get\_queryset(), name)(\*args, \*\*kwargs)

File "C:\Users\K\TGDB\env\lib\site-packages\django\db\models\query.py", line 3

85, in get

self.model.\_meta.object\_name

blog.models.DoesNotExist: Author matching query does not exist.

>>>

## Ordering Results

To order result we use order\_by() method, just like filter() it also returns a QuerySet object. It accepts field names that you want to sort by as positional arguments.

>>>

>>> Author.objects.order\_by("id")

<QuerySet [<Author: droopy : droopy@mail.com>, <Author: tyke : tyke@mail.com>, <

Author: spike : spike@mail.com>, <Author: jerry : jerry@mail.com>, <Author: tom

: tom@email.com>]>

>>>

This command retrieves all Author objects according to id field in ascending order. The above command translates to SQL roughly as follows:

SELECT \* from blog\_author

order by id

It turns out that we can chain methods which returns QuerySet objects. Doing so allows us to modify the database query further.

>>>

>>> Author.objects.filter(id\_\_gt=3).order\_by("name")

<QuerySet [<Author: droopy : droopy@mail.com>, <Author: spike : spike@mail.com>,

<Author: tyke : tyke@mail.com>]>

>>>

This command retrieves only those Author objects whose id is greater than 3 and orders those objects according to values in the name field in ascending order. The above command translates to SQL roughly as follows:

SELECT \* from blog\_author

where id > 3

order by name

To reverse the sorting ordering add - sign before the field name like this:

>>>

>>> Author.objects.filter(id\_\_gt=3).order\_by("-name")

<QuerySet [<Author: tyke : tyke@mail.com>, <Author: spike : spike@mail.com>, <Au

thor: droopy : droopy@mail.com>]>

>>>

The above command traslates to the SQL as follows:

SELECT \* from blog\_author

where id > 3

order by name DESC

You can also sort the result by multiple field like this.

>>>

>>> Author.objects.filter(id\_\_gt=3).order\_by("name", "-email")

<QuerySet [<Author: droopy : droopy@mail.com>, <Author: spike : spike@mail.com>,

<Author: tyke : tyke@mail.com>]>

>>>

This command will sort the result first by name in ascending and then by email in descending order

## Selecting the fields

When you run a query to database like this:

>>>

>>> Author.objects.filter(name\_\_contains='foo').order\_by("name")

>>>

It returns data from all the fields (columns). What if we want data only from one or two fields ? The objects manager provides a values\_list() method specially for this job. The values\_list() accepts optional one or more field names from which we want the data and returns a QuerySet. For example:

>>>

>>> Author.objects.values\_list("id", "name")

<QuerySet [(1, 'tom'), (2, 'jerry'), (3, 'spike'), (4, 'tyke'), (5, 'droopy')]>

>>>

Notice that the values\_list() method returns a QuerySet where each element is a tuple. And the tuple only contains data from the fields which we have specified in the values\_list().

>>>

>>> Author.objects.filter(id\_\_gt=3).values\_list("id", "name")

<QuerySet [(4, 'spike'), (5, 'tyke'), (6, 'droopy')]>

>>>

>>>

>>> r = Author.objects.filter(id\_\_gt=3).values\_list("id", "name")

>>> r

<QuerySet [(4, 'spike'), (5, 'tyke'), (6, 'droopy')]>

>>> r[0]

(4, 'spike')

>>> r[0][0]

4

>>> r[0][1]

'spike'

>>>

The objects manager also provides an identical method called values() which works exactly like  
values\_list() but it returns a QuerySet where each element is a dictionary instead of tuple.

>>>

>>> r = Author.objects.filter(id\_\_gt=3).values("id", "name")

>>>

>>> r

<QuerySet [{'name': 'spike', 'id': 4}, {'name': 'tyke', 'id': 5}, {'name': 'droo

py', 'id': 6}]>

>>>

>>> type(r[0])

<class 'dict'>

>>>

>>> r[0]

{'name': 'spike', 'id': 4}

>>>

>>> r[0]['name']

'spike'

>>> r[0]['id']

4

>>>

## Slicing Results

You can use Python list slicing syntax i.e [start:end] to limit your QuerySet object to certain number of results.

**Example 1:**

>>>

>>> # returns the second record after sorting the result

>>>

>>> Author.objects.order\_by("-id")[1]

<Author: tyke : tyke@mail.com>

>>>

This command roughly translates to SQL as follows:

SELECT \* from blog\_author

order by -id

limit 1, 1

**Example 2:**

>>>

>>> # returns the first three objects after sorting the result

>>>

>>> Author.objects.order\_by("-id")[:3]

<QuerySet [<Author: droopy : droopy@mail.com>, <Author: tyke : tyke@mail.com>, <

Author: spike : spike@mail.com>]>

>>>

>>>

This command roughly translates to SQL as follows:

SELECT \* from blog\_author

order by -id

limit 0, 3

**Example 3:**

>>>

>>> # returns objects from 3rd index to 5th index after sorting the result

>>>

>>> Author.objects.order\_by("-id")[2:5]

<QuerySet [<Author: spike : spike@mail.com>, <Author: jerry : jerry@mail.com>, <

Author: tom : tom@email.com>]>

>>>

>>>

This command roughly translates to SQL as follows:

SELECT \* from blog\_author

order by -id

limit 2, 3

Negative slicing is not supported.

>>>

>>> Author.objects.order\_by("-id")[-1]

Traceback (most recent call last):

File "<console>", line 1, in <module>

File "C:\Users\K\TGDB\env\lib\site-packages\django\db\models\query.py", line 2

75, in \_\_getitem\_\_

"Negative indexing is not supported."

AssertionError: Negative indexing is not supported.

>>>

>>>

## Updating Multiple Objects

Recall that one way to update an object to call save() method after updating it's attributes. For example:

>>>

>>>

>>> a = Author.objects.get(pk=2)

>>> a

<Author: tom : tom@email.com>

>>>

>>> a.name = 'tommy'

>>> a.email = 'tommy@mail.com'

>>>

>>> a.save()

>>>

>>> a = Author.objects.get(pk=2)

>>> a

<Author: tommy : tommy@mail.com>

>>>

>>>

The objects manager provides a method called update() to update one or multiple records in one step. Just like filter() method it accepts one or more keyword arguments. If update is successful it returns number of rows updated.

>>>

>>> Author.objects.filter(pk=2).update(email='tom@yahoo.com')

1

>>>

This command will update the email of author whose pk is 2 to tom@yahoo.com.

This statement is equivalent to:

UPDATE blog\_author SET

email='tom@mail.com'

WHERE id = 2;

### Updating all objects

>>>

>>>

>>> Author.objects.all().update(active=True)

5

>>>

>>>

The above command updates the value of active field to True for all the records in the Author's table. The above command is equivalent to the following command:

Author.objects.update(active=True)

The SQL equivalent of the above command is:

UPDATE blog\_author SET

active=1

## Deleting records

The delete() method is used to delete one or more objects. For example:

### Deleting a single object.

>>>

>>> a = Author.objects.get(pk=2)

>>>

>>> a

<Author: tom : tom@mail.com>

>>>

>>> a.delete()

(1, {'blog.Author': 1})

>>>

>>>

### Deleting multiple records.

>>>

>>> r = Author.objects.all().delete()

>>> r

(4, {'blog.Author': 4})

>>>

>>>

You should now have a solid understanding of Django ORM. In the next lesson, we will discuss how to access data from multiple tables using Django ORM.

