QuerySets allow you to read the data from the database, filter it and order it.

Django shell

Open up your local console

```
python manage.py shell
```

By executing the preceding command, you will get in an interactive Python shell configured for your Django project, where you can play around with the code, inspect classes, try out methods, or execute scripts on the fly. In this recipe, we will go through the most important functions that you need to know in order to work with the Django shell.

ORM

Django **Object-relational mapping** (**ORM**) comes with special abstraction constructs that can be used to build complex database queries. They are called **Query Expressions** and they allow you to filter data, order it, annotate new columns, and aggregate relations.

An object-relational mapper (ORM) is a code library that automates the transfer of data stored in relational databases tables into objects that are more commonly used in application code.

ORMs provide a high-level abstraction upon a relational database that allows a developer to write Python code instead of SQL to create, read, update and delete data and schemas in their database. Developers can use the programming language they are comfortable with to work with a database instead of writing SQL statements or stored procedures.

SQL

```
SELECT * FROM USERS WHERE zip_code=94107;
```

```
ORM
```

```
users = Users.objects.filter(zip_code=94107)
```

Modal.py (example)

```
class Post(models.Model):

    title   = models.CharField(max_length=200)
    slug    = models.SlugField(max_length=100)
    author  = models.ForeignKey(User, related_name="blog_posts"
,                  on_delete = models.DO_NOTHING,)
    body    = models.TextField()
    created = models.DateTimeField(auto_now_add=True)
    updated = models.DateTimeField(auto_now=True)
    status  = models.CharField(max_length=100 ,choices=STATUS_CHOICES,
                default='draft')
```

Now in shell or in view.py class you can write following code

In order to retrieve all of our posts form database you can use **all()** method

Sql

```
Select * from Post
```

Query set

```
from blog.models import Post

posts = Post.objects.all()

print(posts)
```

Iteration

```
posts = Post.objects.all()

for i in posts:
     print(i.title)
     print(i.slug)
     print(i.body)
```

order_By

Sql

```
select * from Post order by title;
```

Query set

```
Post.objects.order_by('title')
```

Sql

select * from Post order by title desc;

Query set

```
Post.objects.order_by('-title')
```

```
Or  you can do this in modal class by creating class meta(socialmedia)
```

```
class Post(models.Model):
     class Meta:
           ordering = ['-title']
```

filter()

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

Ex1

Sql

select * from Post where title="my post one";

Query set

```
Post.objects.filter(title='my post one')
```

Ex2

Sql

select * from Post where title="my post one" and status="published";

Query set

```
Post.objects.filter(title="shell Post" , status="published")
```

Ex3

Sql

select * from Post where title="my post one" or status="published";

Query set

```
from django.db.models import Q
```

```
Post.objects.filter(Q(title = "third post") | Q(status = "published"))
```

startswith

If you want to select posts that start with ’s’

```
Post.objects.filter(title__startswith=’s’)
```

endswith

If you want to select posts that ends with ’s’

```
Post.objects.filter(title__endswith=’s')
```

contains

If you want to select posts that contains ’my’

```
Post.objects.filter(title__contains=‘my')
```

lte (less than equal to <=)

If we need to get post that are post in last hour

```
# timezone.now() - timezone.timedelta(hours=1)
# above code will give us time exactly one hour ago
```

```
from django.utils import timezone
```

```
Post.objects.filter(created__lte=timezone.now() -
timezone.timedelta(hours=1))
```

exclude()

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

Ex1

Sql

```
select * from Post where title !='shell Post’;
```

Query set

```
Post.objects.exclude(title="shell Post")
```

Ex2

Sql

```
select * from Post where title ='shell Post’ and status!=‘draft’;
```

Query set

```
Post.objects.filter(title='shell Post').exclude(status='draft')
```

```
update table set field=field+1 where id=id
```

```
Model.objects.filter(id=id).update(field=F('field') +1))
```

ForeignKey

Model.py

```
class Songs(models.Model):
    title = models.CharField(max_length=255)

    def __str__(self):
        return '%s' % (self.title)


class Singer(models.Model):
    name = models.CharField(max_length=255)
    age = models.IntegerField()
    singer_songs = models.ForeignKey(Songs,
related_name="singer_gana",on_delete=models.CASCADE)

    def __str__(self):
        return '%s %s' % (self.name,self.singer_songs)
```

Query example 1:

```
q1=Singer.objects.all().values(‘name','singer_songs__title')
```

Query Example 2 with Filter:

```
q1 = Singer.objects.filter(name = 'neeraj', singer_songs__title = 'first
song’).values('name','singer_songs__title')
```

Query Example 3 ***Reverse Foreign*** key

```
q1 = Songs.objects.all().values('title','singer_gana__name')
```

Query Example 4 ***Reverse Foreign*** key Filter

```
q1 = Songs.objects.filter(title='first song',
singer_gana__name='neeraj').values('title','singer_gana__name')
```

Foreign key In Foreign key

Model.py

```
class Home(models.Model):
    name = models.CharField(max_length=255)

    def __str__(self):
        return '%s' % (self.name)


class Songs(models.Model):
    title = models.CharField(max_length=255)

    def __str__(self):
        return '%s' % (self.title)


class Singer(models.Model):
    name = models.CharField(max_length=255)
    age = models.IntegerField()
    singer_songs = models.ForeignKey(Songs,
related_name="singer_gana",on_delete=models.CASCADE)
    singer_home = models.ForeignKey(Home, related_name='singer_garh',
on_delete=models.DO_NOTHING)

    def __str__(self):
        return '%s %s' % (self.name,self.singer_songs)
```

Query 1

```
q1 = Songs.objects.all().values('title', 'singer_gana__name',
'singer_gana__singer_home__name')
```

Query 2 with filter

```
q1 = Songs.objects.filter(singer_gana__name = 'neeraj',
singer_gana__singer_home__name= '#777').values('title',
'singer_gana__name', 'singer_gana__singer_home__name')
```

OneToOne Relation

OneToOne relationship is similar to a `ForeignKey` with `unique=True.` Like in example bellow Singer has OneToOneRelation with home, what that mean is one singer can have only one home and a home can assign to a singe singer.

Example

If you have one Home H1 and two Singers S1 and S2. And Home H1 is assign to S1 singer, then you can not assign H1 home to S2 singer.

Models.py file

```
class Home(models.Model):
    name = models.CharField(max_length=255)

    def __str__(self):
        return '%s' % (self.name)


class Songs(models.Model):
    title = models.CharField(max_length=255)

    def __str__(self):
        return '%s' % (self.title)

class Singer(models.Model):
    name = models.CharField(max_length=255)
    age = models.IntegerField()
    singer_songs = models.ForeignKey(Songs,
related_name="singer_gana",on_delete=models.CASCADE)
    singer_home = models.OneToOneField(Home,
related_name='singer_garh',on_delete=models.CASCADE)

    def __str__(self):
        return '%s %s' % (self.name,self.singer_songs)
```

In One TO One Field you can **retrieve data** same as you retrieve in foreign key.

Example

```
q1 = Singer.objects.all().values(‘name','singer_home__name')
```

Example Reverse

```
q=Home.objects.all().values(‘name','singer_garh__name','singer_garh__sin
ger_songs__title')
```

And in order to **SAVE** OneToOne relationship you don’t need to do any thing extra.

```
song = Songs.objects.get(id=1)
```

```
Home.objects.get(id=2)
```

```
q1 = Singer.objects.create(name = 'happy', age=21, singer_songs = song,
singer_home = h1)
```

ManyToMany Relation

In many to many relation ship you don’t need to declare *on_delete* in model.

A many to many relationship implies that many records can have many other records associated amongst one another.

That mean, a singer can assign multiple home and a home can assign to multiple singers.

Example

- If you have one home H1 and two singers S1 and S2, in many to many relations S1 and S2 can share H1.
- If you have two home H1 and H2 and one singer S1 then both H1 and H2 can assign to S1.
- If you have two home H1 and H2 and two singers S1 and S2 then both S1 and S2 can assign to H1 and H2

```
class Home(models.Model):
    name = models.CharField(max_length=255)

    def __str__(self):
        return '%s' % (self.name)


class Songs(models.Model):
    title = models.CharField(max_length=255)

    def __str__(self):
        return '%s' % (self.title)

class Singer(models.Model):
    name = models.CharField(max_length=255)
    age = models.IntegerField()
    singer_songs = models.ForeignKey(Songs, 
related_name="singer_gana",on_delete=models.CASCADE)
    singer_home = models.ManyToManyField(Home, 
related_name='singer_garh')

    def __str__(self):
        return '%s %s' % (self.name,self.singer_songs)
```

Create or **Enter** data with many to many relation ship

```
song = Songs.objects.get(id=1)
home = Home.objects.get(id=1)
home2 = Home.objects.get(id=2)

q1 = Singer.objects.create(name='views2',age=22, singer_songs=song)
q1.save()
q1.singer_home.add(home,home2)
```

retrieve data same as you retrieve in foreign key.

Example

```
q1 = Singer.objects.all().values(‘name','singer_home__name')
```

Example Reverse

```
q=Home.objects.all().values(‘name','singer_garh__name','singer_garh__sin
ger_songs__title')
```