28 December 2019

Django

Django automatically gives you a database-abstraction API that lets you create, retrieve, update and delete objects.

Create an object

```
>>> from blog.models import Blog
>>> b = Blog(name='Beatles Blog', tagline='All the latest Beatles news.')

>>> b.save()
```

However no change is done in the database unless b.save() is called.

To create and save model instances, use the create method .

Blog.objects.create(……)

To save changes to an object that is already present int the database use the save() method.

To add to foreign key relationship, we can use the normal field updation syntax and then call save or we can use the add() method. For m2m we have to use the add() method.

Normal field updation syntax : b.name = "changed name"

QuerySet

A *QuerySet* represents a collection of objects from your database. It can have zero, one or many *filters*.

all() returns all the model instances.

However in most cases we want only a few instances. To do this we can use exclude or filter.

Entry.objects.filter(pub_date__year=2006)

With the default manager class, it is the same as:

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

The result of the above operations are query set itself and hence can be chained.

However remember query sets are lazy. Declaring them does not cause them to hit the database. Querysets are executed only when they are executed not initialised.

Filter always gives us a queryset. So even if we 0 matching or 1 matching instance, a queryset is returned.

If we know that only one object will be present we can use the get method. This method allows us to access a single object and returns the object instance and not the queryset.

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

We can use filter()[0]. This gives us the same result as get(). However filter can return 0 matched objects. While get will raise DoesNotExist exception. If more than one object is matched, get returns MultipleObjectsReturned error.

We can limit the queryset by using python’s slicing

Article.objects.filter(name=“Sid”)[:5]

# This will return 5 matching object instances

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

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

Field Lookups

Say we want instances of all user with age less than 10.

Person.objects.filter(age__lte=10)

The field specified in a lookup has to 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)
```

So blog_id allows us to access key the related models primary key.

This was a basic introduction. Following part will contain detailed information

<u>QuerySET API Reference</u>

Internally, a QuerySet can be constructed, filtered, sliced, and generally passed around without actually hitting the database. No database activity actually occurs until you do something to evaluate the queryset.

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

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

For both the above methods, if we have multiple lookup parameters they are joined by and.

This example excludes all entries whose pub_date is later than 2005-1-3 **AND** whose headline is “Hello”:

```
Entry.objects.exclude(pub_date__gt=datetime.date(2005, 1, 3), headline='Hello')
```

This example excludes all entries whose pub_date is later than 2005-1-3 **OR** whose headline is “Hello”:

```
Entry.objects.exclude(pub_date__gt=datetime.date(2005, 1, 3)).exclude(headline='Hello')
```

3. Annotate : Allows us to add new field in each object. Mostly used with **aggregation** function like count, max, min, avg etc.

```
>>> from django.db.models import Count
>>> q = Blog.objects.annotate(Count('entry')) # The name of the first blog
>>> q[0].name
'Blogasaurus'
# The number of entries on the first blog
>>> q[0].entry__count
42
```

4. order_by : By default, the queryset is ordered according to the parameter specified in the model’s Meta option. However this can be overrides by using the order_by method. However this method requires an queryset.

```
Entry.objects.filter(pub_date__year=2005).order_by('-pub_date', 'headline')
```

The result above will be ordered by pub_date descending, then by headline ascending. Negative sign indicates descending order. For random ordering use ?.

We can also use asc() and desc() for ordering.

5. Reverse : Used to reverse the queryset. (No change in the db). Use reverse only when we have some ordering defined.

6. Distinct : Used to return distinct elements in the queryset. Example in the Post model we want to return only 1 post per user.

```
Entry.objects.order_by('blog').distinct('blog')
```

7. Values : Returns a queryset that contains dictionary instead of model instances.

```
# This list contains a Blog object.

>>> Blog.objects.filter(name__startswith='Beatles') <QuerySet [<Blog: Beatles Blog>]>

# This list contains a dictionary.

>>> Blog.objects.filter(name__startswith='Beatles').values()
<QuerySet [{'id': 1, 'name': 'Beatles Blog', 'tagline': 'All the latest Beatles news.'}]>
```

We can even specify the fields we want

```
>>> Blog.objects.values('id', 'name')

<QuerySet [{'id': 1, 'name': 'Beatles Blog'}]>
```

This has some issue when using with foreign key fields.

8. value_list() : Similar to values() but instead of returning a dictionary it returns a tuple. This takes an extra parameter called **flat.** Flat can only be applied when value_list('something') is done. It can only work when we ask for one field.

```
>>> Entry.objects.values_list('id').order_by('id')

<QuerySet[(1,), (2,), (3,), ...]>

>>> Entry.objects.values_list('id', flat=True).order_by('id')

<QuerySet [1, 2, 3, ...]>
```

Flat cannot be used when we have more than one field in the params.

Both value and value_lists() are used to access a subset of the entire data instead of the entire model instance.

9. None : Calling none() will create a queryset that never returns any objects and no query will be executed when accessing the results. A qs.none() queryset is an instance of EmptyQuerySet.

```
>>> from django.db.models.query import EmptyQuerySet

>>> isinstance(Entry.objects.none(), EmptyQuerySet)

                True
```

10. All : Returns all the model instances in the form of queryset.

11. We can also do **intersection , difference, union .**

12. select_related : This is mostly used to reduce database hits.Using get and then accessing a field from the model instance is 2 hits.

```
# Hits the database.

e = Entry.objects.select_related('blog').get(id=5)

# Doesn't hit the database, because e.blog has been pre-populated # in the previous query.
b = e.blog
```

select_related can be even applied on an queryset and it will return a queryset with those fields only.

Problem can arise when m2m relations are included.

13. Prefetch_related : *Returns a QuerySet that will automatically retrieve, in a single batch, related objects for each of the specified lookups. Similar to selected_range however, select_related is limited to single-valued relationships - foreign key and one-to-one.*

```
>>> Pizza.objects.all().prefetch_related('toppings')
```

14. Defer : In real life situations we do not want to provide all the model instance fields. We may want to fetch those only on some user action. To do this we use the defer field. Defer field is responsible for fetching the field only when that field is accessed.

```
Entry.objects.defer("headline", "body")
```

15. Only : Similar to defer but somewhat opposite. The field passed as params is the only one that is not deferred. Others will be deferred.

```
Person.objects.only("name")
```

16. & : Used to combine two querysets using and operator

```
Model.objects.filter(x=1) & Model.objects.filter(y=2) Model.objects.filter(x=1, y=2)
```

SQL Equivalent :

SELECT ... **WHERE** x=1 **AND** y=2

17. | : Used to combine two querysets using or operator

Model.objects.filter(x=1) | Model.objects.filter(y=2)

ALL THE ABOVE METHODS RETURNED NEW QUERYSETS

THE METHODS SPECIFIED BELOW DO NOT RETURN A QUERYSET.

1. **Get** : Returns the object matching the given lookup parameters, which should be in the format described in *Field lookups*.

get() raises *MultipleObjectsReturned* if more than one object was found. The *MultipleObjectsReturned* exception is an attribute of the model class.

get() raises a *DoesNotExist* exception if an object wasn't found for the given parameters. This exception is an attribute of the model class. Example:

```
from django.core.exceptions import ObjectDoesNotExist

try:

        e = Entry.objects.get(id=3)

        b = Blog.objects.get(id=1)

except ObjectDoesNotExist:

        print("Either the entry or blog doesn't exist.")
```

2. **Create** : This saves us the worry of calling save(). Create uses force_insert=True.

The *force_insert* parameter is documented elsewhere, but all it means is that a new object will always be created. Normally you won't need to worry about this. However, if your model contains a manual primary key value that you set and if that value already exists in the database, a call to create() will fail with an *IntegrityError* since primary keys must be unique. Be prepared to handle the exception if you are using manual primary keys.

3. **get_or_create** : A convenience method for looking up an object with the given kwargs (may be empty if your model has defaults for all fields), creating one if necessary.

Returns a tuple of (object, created), where object is the retrieved or created object and created is a boolean specifying whether a new object was created.

If multiple objects are found, get_or_create() raises *MultipleObjectsReturned*. If an object is *not* found, get_or_create() will instantiate and save a new object, returning a tuple of the new object and True.

Any keyword arguments passed to get_or_create() — *except* an optional one called defaults — will be used in a *get()* call. If an object is found, get_or_create() returns a tuple of that object and False.

```
obj, created = Person.objects.get_or_create( first_name='John',
last_name='Lennon',
defaults={'birthday': date(1940, 10, 9)},

)
```

If the fields used in the keyword arguments do not have a uniqueness constraint, concurrent calls to this method may result in multiple rows with the same parameters being inserted.

4. **update_or_create** : A convenience method for updating an object with the given kwargs, creating a new one if necessary. The defaults is a dictionary of (field, value) pairs used to update the object.

The values in defaults can be callable. Returns a tuple of (object, created), where object is the created or updated object and created is a boolean specifying whether a new object was created.

The update_or_create method tries to fetch an object from database based on the given kwargs. If a match is found, it updates the fields passed in the defaults dictionary.

5. **Count :** Returns an integer representing the number of objects in the database matching the QuerySet.

```
# Returns the total number of entries in the database.

Entry.objects.count()

# Returns the number of entries whose headline contains 'Lennon'
Entry.objects.filter(headline__contains='Lennon').count()
```

6. **Latest** : Returns the latest object in the table based on the given field(s).

```
Entry.objects.latest('pub_date')

Entry.objects.latest('pub_date', '-expire_date')
```

The negative sign in '-expire_date' means to sort expire_date in *descending* order. Since latest() gets the last result, the Entry with the earliest expire_date is selected. Like *get()*, earliest() and latest() raise *DoesNotExist* if there is no object with the given parameters.

If your model's *Meta* specifies *get_latest_by*, you can omit any arguments to earliest() or latest(). The fields specified in *get_latest_by* will be used by default.

If null values can exist then ordering may not be correct

```
Entry.objects.filter(pub_date__isnull=False).latest('pub_date')
```

7. **Earliset**

8. **first, last** :

first : Returns the first object matched by the queryset, or None if there is no matching object.

9. **Exists** : Returns True if the *QuerySet* contains any results, and False if not. This tries to perform the query in the simplest and fastest way possible, but it *does* execute nearly the same query as a normal *QuerySet* query.

The most efficient method of finding whether a model with a unique field (e.g. primary_key) is a member of a QuerySet is:

```
entry = Entry.objects.get(pk=123)
if some_queryset.filter(pk=entry.pk).exists():

   print("Entry contained in queryset”)
```

And to find whether a queryset contains any items:

```
if some_queryset.exists():
print("There is at least one object in some_queryset")
```

10. **Update** : Performs a SQL Update operation. Can be done over a model instance or the queryset. Returns the number of rows affected. However it cannot update on model’s related table.

```
Entry.objects.filter(id=10).update(comments_on=False)
```

This is preferred than getting the model instance, updating the value and then calling save(). Reduces database hits.

11. **Delete** : Performs SQL Delete operation. Used to delete model instances.

```
b = Blog.objects.get(pk=1)

# Delete all the entries belonging to this Blog.

>>> Entry.objects.filter(blog=b).delete()
(4, {'weblog.Entry': 2, 'weblog.Entry_authors': 2})
```

Field Lookups

Used in the SQL Where clause.

They're specified as keyword arguments to the QuerySet methods *filter()*, *exclude()* and *get()*.

WHEN NO LOOKUP TYPE IS PROVIDED, THE LOOKUP TYPE IS ASSUMED TO BE EXACT.

All field lookups come after double underscores __

1. Exact : Case sensitive match
2. iExact : Case insensitive match
3. Contains : Case sensitive containment test
4. iContains : Case insensitive containment test
5. In : In a given iterable; often a list, tuple, or queryset. It's not a common use case, but strings (being iterables) are accepted.
6. Gt : greater than
7. Gte : greater than or equal to
8. Lt : Lesser than
9. Lte : Lesser than or equal to
10. Startswith, istartswith
11. endswith, iendswith
12. Range : Range test (inclusive).

Entry.objects.filter(pub_date__range=(start_date, end_date))

13. isnull : Checks if the given field is null or not

14. Regex : Case sensitive regex match

15. iRegex : Case insensitive regex match