Digital Career Institute

Python Course - Databases - ORM

Digital Career Institute

Data Migration: Fixtures

To export the database data, **dumpdata** is used.

```
(env)$ python3 manage.py dumpdata > data.json
```

To import the data into a new instance, **loaddata** is used.

```
(env)$ python3 manage.py loaddata data.json
Installed 9 object(s) from 1 fixture(s)
```

The Model Manager & QuerySets

The Model Manager

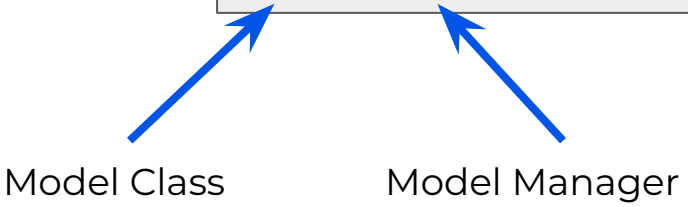
The model **Manager** is a property of the `Model` class.

The model manager is, on its turn, a class with **methods to create, delete and access** rows in the database table.

The Model Manager

The property holding the default manager is named **`objects`**.

```
Item.objects.<method>()
```



The Model Manager: Methods

The most basic methods of the model manager are:

CREATE

GET

FILTER

EXCLUDE

ALL

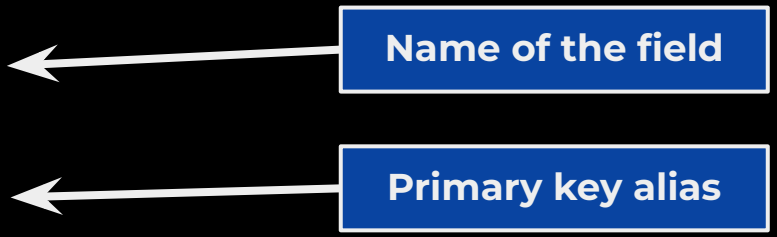
Create

```
(env)$ python3 manage.py shell
>>> from shop.models import Item
>>> tablet = Item.objects.create(name="Tablet",
...                              state="New")
...
>>> tablet
<Item: Item object (1)>
>>>
```

The model manager has a method named **`create`**. This method will store the new data in the database and will return the model object.

Create

```
>>> tablet = Item.objects.create(name="Tablet",
...                              state="New")
...
>>> tablet.id
1
>>> tablet.pk
1
```



A table created with the ORM will always have a primary key column. If one has not been defined in our code, the ORM will automatically add one named **`id`** (with an alias named **`pk`**).

If the model specifies a primary key with a name different than **`id`**, **`pk`** will point to that field.

Create

```
>>> tablet = Item(name="Tablet", state="Old")
>>> tablet.id
>>> tablet.save()
>>> tablet.id
2
```

An object can also be created with the standard instantiation. In this case, the ORM will create the object but will not save it in the database. Therefore, the object will not have an automated identifier.

Every model object has a method `save()` that will store the object in the database. Only then, the object is updated with the generated `id`.

Create

```
>>> tablet = Item(name="Tablet", state="Old")
>>> tablet.state = "Refurbished"
>>> tablet.save()
>>> tablet
Refurbished Tablet
```

A model object can also be modified like any other Python object. If the changes are made before calling the `save()` method, the changes will be stored in the database.

Get

```
>>> item = Item.objects.get(id=1)
>>> print(item)
<Item: Item object (1)>
```

The `get` method is used to retrieve a **single model object** from the database.

Get

```
>>> items = Item.objects.get(name="Tablet")
Traceback (most recent call last):
...
shop.models.Item.MultipleObjectsReturned: get() returned more
than one Item -- it returned 2!
```

The query arguments must return a single record.
If more than one is returned an error exception will be raised.

Get or Create?

An object can be created only when it does not yet exist.

```
>>> tablet, created = Item.objects.get_or_create(
...     name="Tablet", state="New"
... )
```

If the object **already exists** in the database,
`get_or_create` will execute like the `get` method.

If the object **does not exist** in the database,
`get_or_create` will execute like the `create` method.

The `get_or_create` method returns two values: the object and a Boolean indicating if it had to be created.

Filter & QuerySets

```
>>> tablets = Item.objects.filter(name="Tablet")
>>> tablets
<QuerySet [<Item: Item object (1)>, <Item: Item object (2)>]>
```

The `filter` method does not return a model object, but a `QuerySet` object.
A `QuerySet` is an iterable that provides access to the results of the query.

```
>>> for tablet in tablets:
...     print(tablet.state, tablet.name)
...
New Tablet
Refurbished Tablet
```

QuerySets: SQL Query

```
>>> tablets = Item.objects.filter(name="Tablet")
>>> print(tablets.query)
SELECT
    "shop_item"."id",
    "shop_item"."name",
    "shop_item"."state"
FROM "shop_item"
WHERE "shop_item"."name" = 'Tablet';
```

The `QuerySet` has a `query` property that returns the SQL that **will be executed**.

The query will only be executed when the values are accessed.
This is called **lazy loading**.

Filter: Multiple AND Conditions

```
>>> tablets = Item.objects.filter(name="Tablet", state="New")
>>> print(tablets.query)
SELECT
    "shop_item"."id",
    "shop_item"."name",
    "shop_item"."state"
FROM "shop_item"
WHERE (     "shop_item"."name" = 'Tablet'
        AND "shop_item"."state" = 'New'
);
```

The `filter` method can be used with multiple keyword arguments.
It will execute them as `AND` logical expressions.

Filter: Multiple OR Conditions

```
>>> from django.db.models import Q
>>> phones_and_tablets = Item.objects.filter(
...     Q(name="Phone") | Q(name="Tablet")
... )
...
>>> print(phones_and_tablets.query)
… WHERE ("shop_item"."name" = 'Phone' OR "shop_item"."state"
= 'Tablet');
```

The `Q` function produces a query condition object.
The `|` operator is the `OR` logical operator.

Exclude

```
>>> items = Item.objects.exclude(state="Refurbished")
>>> print(items.query)
SELECT
    "shop_item"."id",
    "shop_item"."name",
    "shop_item"."state"
FROM "shop_item"
WHERE NOT ("shop_item"."state" = 'Refurbished');
```

The `exclude` method works very similarly to the filter method, but precedes the expression with a `NOT` logical operator.

All

```
>>> all_items = Item.objects.all()
>>> print(all_items.query)
SELECT
    "shop_item"."id",
    "shop_item"."name",
    "shop_item"."state"
FROM "shop_item";
```

The `all` method returns all rows in the table.

Limiting Fields Selected

```
>>> all_items = Item.objects.only("name")
>>> print(all_items.query)
SELECT
    "shop_item"."id",
    "shop_item"."name"
FROM "shop_item";
```

The `only` method returns only the fields indicated.
It automatically includes the primary key field.

Limiting Fields Selected

```
>>> all_items = Item.objects.values("name")
>>> print(all_items.query)
SELECT "shop_item"."name"
FROM "shop_item";
>>> print(all_items)
<QuerySet [{'name': 'Tablet'}, {'name': 'Tablet'}]>
>>> Item.objects.only("name")
<QuerySet [<Item: Item object (1)>, <Item: Item object (2)>]>
```

The **`values`** method strictly returns only the fields indicated.
Unlike **`only`**, the objects returned are dictionaries instead of model objects.

Field Lookups

A query argument can also be defined using the **`field_name`**, followed by a double score and a **`lookup`**.

```
>>> tablets = Item.objects.filter(
...     <field_name>__<lookup>=<value>
... )
```

Lookups are used often with the methods **`get`**, **`filter`** and **`exclude`**.

Field Lookups Example

```
>>> tablets = Item.objects.filter(
...     state__in=["New", "Refurbished"]
... )
>>> print(tablets.query)
SELECT
    "shop_item"."id",
    "shop_item"."name",
    "shop_item"."state"
FROM "shop_item"
WHERE "shop_item"."state" IN ('New', 'Refurbished');
```

Lookups may be used to specify different query operators and functions.

Filter Objects: Field Lookups

There are a variety of lookups available.

- **exact**
- iexact
- contains
- icontains
- in
- gt
- gte
- lt
- lte
- startswith
- istartswith
- endswith
- iendswith
- range
- date
- year
- iso_year
- month
- day
- week
- week_day
- iso_week_day
- quarter
- time
- hour
- minute
- second
- isnull
- regex
- iregex

If no lookup is defined, the `exact` lookup is used.

Reusing QuerySets

Django's ORM queries return **QuerySets** objects.

A QuerySet can be iterated, but it can also be manipulated again (filter, exclude,...).

```
>>> tablets = Item.objects.filter(name="Tablet")
>>> new_tablets = tablets.filter(state="New")
>>> tablets
<QuerySet [<Item: Item object (1)>, <Item: Item object (2)>]>
>>> new_tablets
<QuerySet [<Item: Item object (1)>]>
```

Chaining QuerySets

```
>>> from django.db.models import Q
>>> newish_tablets = Item.objects
...     .filter(name="Tablet")
...     .filter(Q(state="New") | Q(state="Like new"))
...
>>> print(newish_tablets.query)
SELECT … WHERE ("shop_item"."name" = Tablet AND
("shop_item"."state" = New OR "shop_item"."state" = Like
new))
```

QuerySets can also be simply chained one after the other.
Django's ORM will produce a single SQL query combining all conditions.

Annotating Objects

```
>>> from django.db.models import Value
>>> tablets = Item.objects.filter(name="Tablet")
...      .annotate(description=Value("Cool tablets!"))
...
>>> print(tablets.query)
SELECT "shop_item"."id",
       "shop_item"."name",
       "shop_item"."state",
       'Cool tablets!' AS "description"
FROM …
```

The returned rows can be annotated by adding a virtual field and using a literal value. The `Value` function must be used to assign a literal value to the annotation field.

Annotating Objects

```
>>> from django.db.models import Q, ExpressionWrapper, BooleanField
>>> tablets = Item.objects.annotate(is_new=ExpressionWrapper(
...     Q(state="New") | Q(state="Like new"),
...     output_field=BooleanField()
... ))
...
>>> print(tablets.values())
<QuerySet [
    {'id': 1, 'name': 'Tablet', 'state': 'New', 'is_new': True},
    {'id': 2, 'name': 'Tablet', 'state': 'Refurbished', 'is_new': False}
]>
```

The returned rows can be annotated using an expression.

Aggregation

```
>>> from django.db.models import Count
>>> tablets = Item.objects.values("name").annotate(amount=Count("name"))
>>> print(tablets)
<QuerySet [{'name': 'Tablet', 'total': 2}]>
>>> # Inverting the methods will produce a different result
>>> tablets = Item.objects.annotate(amount=Count("name")).values("name")
>>> print(tablets)
<QuerySet [{'name': 'Tablet'}, {'name': 'Tablet'}]>
```

Aggregation functions, like `Count`, can be used if they are preceded by the `values` method indicating the fields to be used as key for the aggregation.

This will use the SQL `GROUP BY` clause.

Count or Limit Returning Rows

```
>>> # The count method returns the number of rows
>>> tablets = Item.objects.values("name").annotate(amount=Count("name"))
>>> print(tablets.count())
1
>>> # Limiting the number of rows to return can be done with Python
>>> tablets = Item.objects.filter(name="Tablet")[:1]
>>> print(tablets.count())
1
>>> print(tablets.query)
SELECT … WHERE "shop_item"."name" = 'Tablet' LIMIT 1
```

The **`count`** method returns the number of rows returned by the query.
Using standard Python syntax, the number of rows can be limited.

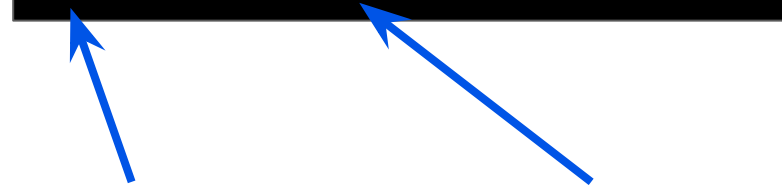
Ordering Rows

```
>>> Item.objects.order_by("state")
<QuerySet [<Item: Item object (1)>, <Item: Item object (2)>]>
>>> # A descending order can be defined using the minus sign
>>> Item.objects.order_by("-state")
<QuerySet [<Item: Item object (2)>, <Item: Item object (1)>]>
```

Deleting All Rows

Deleting all rows can be done by using the `delete` method after the `all` method.

```
>>> Item.objects.all().delete()
(2, {'shop.Item': 2})
```



Total objects deleted.

Total objects deleted by Model.

Deleting Some Rows

Deleting a set of rows can be done by filtering and then using the `delete` method of the *queryset* object.

```
>>> Item.objects.filter(state="New").delete()
(1, {'shop.Item': 1})
```

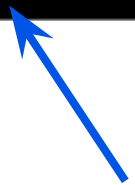
Deleting a single row can be done by using `get` and then using the `delete` method of the *model* object.

```
>>> Item.objects.get(id=1).delete()
(1, {'shop.Item': 1})
```

Updating All Rows

Updating all rows can be done by using the `update` method.

```
>>> Item.objects.update(state="Like New")
2
```



Rows affected.

Updating Some Rows

Updating a set of rows can be done by filtering and then using the `update` method of the queryset object.

```
>>> Item.objects.filter(state="New").update(state="Like New")
1
```

Updating a single row can **NOT** be done by using `get` and then using the `update` method. The model object does not have such method.

```
>>> Item.objects.get(id=1).update(state="Like new")
Traceback (most recent call last):
  File "<console>", line 1, in <module>
AttributeError: 'Item' object has no attribute 'update'
```

Updating a Single Row

A single row can be updated using `get` first, applying the changes and then saving the object.

```
>>> item = Item.objects.get(id=1)
>>> item.state = "Old"
>>> item.save()
```

Update or Create a Single Row

The method `update_or_create` will first execute a `get` query using the keyword arguments.

```
>>> item, created = Item.objects.update_or_create(
...     name="Tablet", state="New",
...     defaults={"state": "Old"}
... )
>>> print(item.state, item.value)
Old Tablet
```

If there is an object with those keyword arguments, it will update it with the `defaults`. If not, it will create the object anew.

This method can only be used to update (or create) a single record.

Contact Details
DCI Digital Career Institute gGmbH

Digital Career Institute