**Model in django**

* A model is a class that represents table or collection in our DB, and where every attribute of the class is a field of the table or collection. Models are defined in the app/models.py
* Each model is a Python class that subclasses [django.db.models.Model](https://docs.djangoproject.com/en/3.1/ref/models/instances/" \l "django.db.models.Model" \o "django.db.models.Model).
* Each attribute of the model represents a database field

With all of this, Django gives you an automatically-generated database-access

Lets take a n exapme



The above **Person** model would create a database table like this:



**Model Meta options**

We will see some possible [metadata options](https://docs.djangoproject.com/en/dev/topics/db/models/#meta-options) that you can give your model in its internal **class Meta**.

Meta class is subclass in mode name/method of model./py file

1. **db\_table=<table\_name>**

The name of the database table to use for the model. By default django creates table by name <appsName\_modelname>

1. **app\_label=<myapp>**

If a model is defined outside of an application in [**INSTALLED\_APPS**](https://docs.djangoproject.com/en/dev/ref/settings/#std:setting-INSTALLED_APPS), it must declare which app it belongs to

1. **options.permissions/models.permissions=[permissions]**

Extra permissions to enter into the permissions table when creating this object. Add, change, delete, and view permissions are automatically created for each model

permissions = [('can\_deliver\_pizzas', 'Can deliver pizzas')]

This is a list or tuple of 2-tuples in the format **(permission\_code, human\_readable\_permission\_name)**.

1. **Options.default\_permissions<permissions>**
2. **Options.constraints=<list of constrainsts>**

A list of constraints that you want to define on the model

**Adding data to DB**



**QuerySets**

A django queryset is like its name says, basically a collection of (sql) queries, in your example above print(b.query) will show you the sql query generated from your django filter calls

Since querysets are lazy, the database operation isn't done immediately but operations happens only when needed or when the queryset is evaluated

**QuerySets are lazy**

QuerySets are lazy – the act of creating a QuerySet doesn’t involve any database activity. You can stack filters together all day long, and Django won’t actually run the query until the QuerySet is evaluated.

**>>>** q = Entry.objects.filter(headline\_\_startswith="What")

**>>>** q = q.filter(pub\_date\_\_lte=datetime.date.today())

**>>>** q = q.exclude(body\_text\_\_icontains="food")

**>>>** print(q)----At this line DB will be hitted,there are many easy to evalue querysets

**Evaluating QuerySets**

* 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.
* We can evaluate query in following ways

1. **Iteration**

A QuerySet is iterable, and it executes its database query the first time you iterate over it. For example, this will print the headline of all entries in the database



1. **repr()**

A QuerySet is evaluated when you call repr() on it. This is for convenience in the Python interactive interpreter, so you can immediately see your results when using the API interactively.

1. **len()**

A QuerySet is evaluated when you call len() on it.

*It returns the length of the result list ,*For this purpose django provides a **count()** method for precisely this reason.

1. **list()**

Force evaluation of a QuerySet by calling list() on it. For example:



1. **bool()**

Testing a QuerySet in a boolean context, such as using **bool(), or, and or an if** statement, will cause the query to be executed

If there is at least one result, the QuerySet is True, otherwise False.

1. **pickling/caching**
2. **slicing**

Note:

We can do chaining on queryset.

Not all methods from queryset API return a new query set when applied on a queryset

**Methods that return new QuerySets**

<https://docs.djangoproject.com/en/3.1/ref/models/querysets/#when-querysets-are-evaluated>

-------------- Methods which return a new query set

There are many methods for this but here we will see few of them

1. **filter(\*\*kwargs)/** **filter()**

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

If you need to execute more complex queries (for example, queries with OR statements), you can use Q objects.

1. **exclude()/exclude(\*\*kwargs)**

Returns a new QuerySet containing objects that do not match the given lookup parameters/conditions

1. **order\_by()/order\_by(\*field)**

By default, results returned by a QuerySet are ordered by the ordering tuple given by the ordering option in the model’s Meta.

You can override this on a per-QuerySet basis by using the order\_by method



1. **reverse()**

Use the reverse() method to reverse the order in which a queryset’s elements are returned.

Calling reverse() a second time restores the ordering back to the normal direction.



To retrieve last 5 items in querysets

**Operations which return a new queryset**

**AND(&)** and **OR** operation returns a new queryset

1. **AND**

Combines two QuerySets using the SQL AND operator.



Q(xyz)----- this is Q lass we will see it later in this notes

This queryset evaluates to



1. **OR**

Combines two QuerySets using the SQL OR operator.



Above queryset evaluates to



**Methods that do not return QuerySet**

<https://docs.djangoproject.com/en/3.1/ref/models/querysets/#methods-that-do-not-return-querysets>

------------ Methods which don’t return a new queryset

There are masny methods which don’t return queryset , we will see some of them

1. **get()/get(\*\*kwargs)**

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



**NOTE**:

1. If get() finds more than one object, it raises a Model.MultipleObjectsReturned exception.
2. If get() doesn’t find any object, it raises a Model.DoesNotExist except

We can handle those exceptions using "**django.core.exceptions**"

1. **2. create()/create(\*\*kwargs)**

A convenience method for creating an object and saving it all in one step.

1. **get\_or\_create()/get\_or\_create(\*kwargs)**

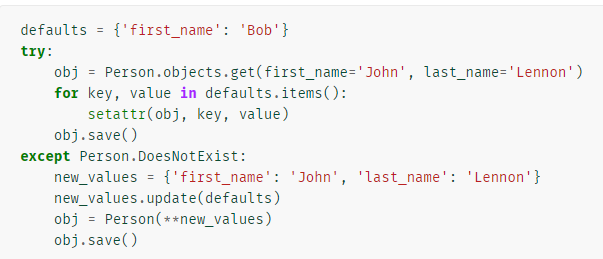
A convenience method for looking up an object with the given kwargs (may be empty if your model has defaults for all fields), creating one if necessary.

1. **update\_or\_create(defaults=None, \*\*kwargs)/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 callables.

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.

Let us take example of create or update manually.



Above can be written easily using create\_or\_update() as below



1. **bulk\_update()/bulk\_update(objs, fields, batch\_size=None)**
2. **count()**
3. **in\_bulk()/in\_bulk(id\_list=None, field\_name='pk')**
4. **latest()**
5. **latest(\*fields)**
6. **first()**
7. **bulk\_create()**
8. **bulk\_create(objs, batch\_size=None, ignore\_conflicts=False)**

**Field lookups**

**Caching and QuerySets**

**CURD operation in django**

**C - Create**

**U –** Update

**R –** Retrive

**D –** Delete