QuerySet API

A QuerySet can be defined as a list containing all those objects we have created using the Django model.

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

query property – This property is used to get sql query of query set.

Syntax:- queryset.query

Methods that return new QuerySets

Retrieving all objects

all ( ) - This method is used to retrieve all objects. This returns a copy of current QuerySet.

Example:- Student.objects.all()

Retrieving specific objects

- filter (**kwargs) - It returns a new QuerySet containing objects that match the given *lookup parameters*. filter() will always give you a QuerySet, even if only a single object matches the query.

  Example:- Student.objects.filter(marks=70)

- exclude(**kwargs) - It returns a new QuerySet containing objects that do not match the given *lookup parameters*.

  Example:- Student.objects.exclude(marks=70)

Methods that return new QuerySets

order_by(*fields) – It orders the fields.

- ‘field’ – Asc order
- ‘-field’ – Desc Order
- ‘?’ – Randomly

reverse() – This works only when there is ordering in queryset.

values(*fields, **expressions) - It returns a QuerySet that returns dictionaries, rather than model instances, when used as an iterable. Each of those dictionaries represents an object, with the keys corresponding to the attribute names of model objects.

distinct(*fields) - This eliminates duplicate rows from the query results.

Methods that return new QuerySets

values_list(*fields, flat=False, named=False) - This is similar to values() except that instead of returning dictionaries, it returns tuples when iterated over.

- If you don’t pass any values to values_list(), it will return all the fields in the model, in the order they were declared.
- If you only pass in a single field, you can also pass in the flat parameter. If True, this will mean the returned results are single values, rather than one-tuples.
- You can pass named=True to get results as a namedtuple.

using(alias) - This method is for controlling which database the QuerySet will be evaluated against if you are using more than one database. The only argument this method takes is the alias of a database, as defined in DATABASES.

Example:- student_data = Student.objects.using('default')

Methods that return new QuerySets

dates(field, kind, order='ASC’) - It returns a QuerySet that evaluates to a list of datetime.date objects representing all available dates of a particular kind within the contents of the QuerySet.

Where,

field – It should be the name of a DateField of your model.

kind – It should be either "year", "month", "week", or "day".

"year" returns a list of all distinct year values for the field.

"month" returns a list of all distinct year/month values for the field.

"week" returns a list of all distinct year/week values for the field. All dates will be a Monday.

"day" returns a list of all distinct year/month/day values for the field.

order – It should be either 'ASC' or 'DESC'. This specifies how to order the results. defaults to 'ASC’.

Each datetime.date object in the result list is “truncated” to the given type.

Methods that return new QuerySets

datetimes(field_name, kind, order='ASC', tzinfo=None) – It returns a QuerySet that evaluates to a list of datetime.datetime objects representing all available dates of a particular kind within the contents of the QuerySet.

field_name – It should be the name of a DateTimeField of your model.

Kind - It should be either "year", "month", "week", or "day".

"year" returns a list of all distinct year values for the field.

"month" returns a list of all distinct year/month values for the field.

"week" returns a list of all distinct year/week values for the field. All dates will be a Monday.

"day" returns a list of all distinct year/month/day values for the field.

order – It should be either 'ASC' or 'DESC'. This specifies how to order the results. defaults to 'ASC'.

tzinfo – It defines the time zone to which datetimes are converted prior to truncation. This parameter must be a datetime.tzinfo object. If it's None, Django uses the current time zone. It has no effect when USE_TZ is False.

Each datetime.datetime object in the result list is "truncated" to the given type.

Methods that return new QuerySets

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.

Example:- student_data = Student.objects.none()

union(*other_qs, all=False) - Uses SQL's UNION operator to combine the results of two or more QuerySets. The UNION operator selects only distinct values by default. To allow duplicate values, use the all=True argument.

Example:- student_data = qs2.union(qs1, all=True)

intersection(*other_qs) – Uses SQL's INTERSECT operator to return the shared elements of two or more QuerySets.

Example:- student_data = qs1.intersection(qs2)

difference(*other_qs) - Uses SQL's EXCEPT operator to keep only elements present in the QuerySet but not in some other QuerySets.

Example:- student_data = qs1.difference(qs2)

Methods that return new QuerySets

- select_related(*fields)
- defer(*fields)
- only(*fields)
- prefetch_related(*lookups)
- extra(select=None, where=None, params=None, tables=None, order_by=None, select_params=None)
- select_for_update(nowait=False, skip_locked=False, of=())
- raw(raw_query, params=None, translations=None)
- annotate(*args, **kwargs)

Operators that return new QuerySets

AND (&) - Combines two QuerySets using the SQL AND operator.

Example:-

```
student_data = Student.objects.filter(id=6) & Student.objects.filter(roll=106)
student_data = Student.objects.filter(id=6, roll=106)
student_data = Student.objects.filter(Q(id=6) & Q(roll=106))
```

OR (|) - Combines two QuerySets using the SQL OR operator.

Example:-

```
Student.objects.filter(id=11) | Student.objects.filter(roll=106)
Student.objects.filter(Q(id=11) | Q(roll=106))
```

Methods that do not return new QuerySets

Retrieving a single object

get ( ) - It returns one single object. If There is no result match it will raise DoesNotExist exception. If more than one item matches the get() query. It will raise MultipleObjectsReturned.

Example:- Student.objects.get(pk=1)

first() - It returns the first object matched by the queryset, or None if there is no matching object. If the QuerySet has no ordering defined, then the queryset is automatically ordered by the primary key.

Example:- student_data = Student.objects.first()

student_data = Student.objects.order_by('name').first()

last() - It returns the last object matched by the queryset, or None if there is no matching object. If the QuerySet has no ordering defined, then the queryset is automatically ordered by the primary key.

Methods that do not return new QuerySets

latest(*fields) - It returns the latest object in the table based on the given field(s).

Example:- student_data = Student.objects.latest('pass_date')

earliest(*fields) - It returns the earliest object in the table based on the given field(s).

Example:- student_data = Student.objects.earliest('pass_date’)

exists() - It 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.

Example:-

student_data = Student.objects.all()

print(student_data.exists())

Methods that do not return new QuerySets

create(**kwargs) - A convenience method for creating an object and saving it all in one step.

Example:-

s = Student(name='Sameer', roll=112, city='Bokaro', marks=60, pass_date='2020-5-4')

s.save(force_insert=True)

s = Student.objects.create(name='Sameer', roll=112, city='Bokaro', marks=60, pass_date='2020-5-4’)

get_or_create(defaults=None, **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.

It 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.

Example:-

student_data, created = Student.objects.get_or_create(name='Sameer', roll=112, city='Bokaro', marks=60, pass_date='2020-5-4')

print(student_data, created)

Methods that do not return new QuerySets

update(**kwargs) - Performs an SQL update query for the specified fields, and returns the number of rows matched (which may not be equal to the number of rows updated if some rows already have the new value).

Example:-

```
student_data = Student.objects.filter(id=12).update(name='Kabir', marks=80)


# Update student’s city Pass who has marks 60
student_data = Student.objects.filter(marks=60).update(city='Pass’)


student_data = Student.objects.get(id=12).update(name='Kabir', marks=80)
```

Methods that do not return new QuerySets

update_or_create(defaults=None, **kwargs) – 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.

It 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.

Example:-

student_data, created = Student.objects.update_or_create(id=14, name='Kohli’, defaults={'name':'Sameer'})

Methods that do not return new QuerySets

bulk_create(objs, batch_size=None, ignore_conflicts=False) – This method inserts the provided list of objects into the database in an efficient manner.

The model's save() method will not be called, and the pre_save and post_save signals will not be sent.

It does not work with child models in a multi-table inheritance scenario.

If the model's primary key is an AutoField it does not retrieve and set the primary key attribute, as save() does, unless the database backend supports it (currently PostgreSQL).

It does not work with many-to-many relationships.

It casts objs to a list, which fully evaluates objs if it's a generator. The cast allows inspecting all objects so that any objects with a manually set primary key can be inserted first.

The batch_size parameter controls how many objects are created in a single query. The default is to create all objects in one batch, except for SQLite where the default is such that at most 999 variables per query are used.

On databases that support it (all but Oracle), setting the ignore_conflicts parameter to True tells the database to ignore failure to insert any rows that fail constraints such as duplicate unique values. Enabling this parameter disables setting the primary key on each model instance.

Methods that do not return new QuerySets

Example:-

```
objs = [
 Student(name='Sonal', roll=120, city='Dhanbad', marks=40, pass_date='2020-5-4'),
 Student(name='Kunal', roll=121, city='Dumka', marks=50, pass_date='2020-5-7'),
 Student(name='Anisa', roll=122, city='Giridih', marks=70, pass_date='2020-5-9') ]
student_data = Student.objects.bulk_create(objs)
```

Methods that do not return new QuerySets

bulk_update(objs, fields, batch_size=None) - This method efficiently updates the given fields on the provided model instances, generally with one query. QuerySet.update() is used to save the changes, so this is more efficient than iterating through the list of models and calling save() on each of them.

You cannot update the model's primary key.

Each model's save() method isn't called, and the pre_save and post_save signals aren't sent.

If updating a large number of columns in a large number of rows, the SQL generated can be very large. Avoid this by specifying a suitable batch_size.

Updating fields defined on multi-table inheritance ancestors will incur an extra query per ancestor.

If objs contains duplicates, only the first one is updated.

The batch_size parameter controls how many objects are saved in a single query. The default is to update all objects in one batch, except for SQLite and Oracle which have restrictions on the number of variables used in a query.

Example:-

```
all_student_data = Student.objects.all()

for stu in all_student_data:
 stu.city = 'Bhel'

student_data = Student.objects.bulk_update(all_student_data, ['city'])
```

Methods that do not return new QuerySets

in_bulk(id_list=None, field_name='pk') – It takes a list of field values (id_list) and the field_name for those values, and returns a dictionary mapping each value to an instance of the object with the given field value. If id_list isn't provided, all objects in the queryset are returned. field_name must be a unique field, and it defaults to the primary key.

Example:-

```
student_data = Student.objects.in_bulk([1, 2])
print(student_data[1].name)
print()
student_data1 = Student.objects.in_bulk([])
print(student_data1)
print()
student_data2 = Student.objects.in_bulk()
print(student_data2)
print()
```

Methods that do not return new QuerySets

delete() - The delete method, conveniently, is named delete(). This method immediately deletes the object and returns the number of objects deleted and a dictionary with the number of deletions per object type.

Example:-

Delete One Record

student_data = Student.objects.get(pk=22)

deleted = student_data.delete()

Delete in Bulk

You can also delete objects in bulk. Every QuerySet has a delete() method, which deletes all members of that QuerySet.

Example:- student_data = Student.objects.filter(marks=50).delete()

Delete All Records

Example:- student_data = Student.objects.all().delete()

Methods that do not return new QuerySets

count() - It returns an integer representing the number of objects in the database matching the QuerySet. A count() call performs a SELECT COUNT(*) behind the scenes.

Example:-

```
student_data = Student.objects.all()

print(student_data.count())
```

explain(format=None, **options) – It returns a string of the QuerySet’s execution plan, which details how the database would execute the query, including any indexes or joins that would be used. Knowing these details may help you improve the performance of slow queries. explain() is supported by all built-in database backends except Oracle because an implementation there isn’t straightforward. The format parameter changes the output format from the databases’s default, usually text-based. PostgreSQL supports 'TEXT', 'JSON', 'YAML', and 'XML'. MySQL supports 'TEXT' (also called 'TRADITIONAL') and 'JSON'.

Example:- print(Student.objects.all().explain())

Methods that do not return new QuerySets

- aggregate(*args, **kwargs)
- as_manager()
- iterator(chunk_size=2000)

Field Lookups

Field lookups are how you specify the meet of an SQL WHERE clause.

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

If you pass an invalid keyword argument, a lookup function will raise TypeError.

Syntax:- field__lookuptype=value

Example:- Student.objects.filter(marks__lt=‘50’)

SELECT * FROM myapp_student WHERE marks < ’50’;

The field specified in a lookup has to be the name of a model field.

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.

Example:- Student.objects.filter(stu_id=10)

Field Lookups

exact - Exact match. If the value provided for comparison is None, it will be interpreted as an SQL NULL. This is case sensitive

Example:- Student.objects.get(name__exact='sonam')

iexact - Exact match. If the value provided for comparison is None, it will be interpreted as an SQL NULL. This is case insensitive

Example:- Student.objects.get(name__iexact='sonam')

contains - Case-sensitive containment test.

Example:- Student.objects.get(name__contains='kumar')

icontains - Case-insensitive containment test.

Example:- Student.objects.get(name__icontains='kumar’)

in - In a given iterable; often a list, tuple, or queryset. It’s not a common use case, but strings (being iterables) are accepted.

Example:- Student.objects.filter(id__in=[1, 5, 7])

Field Lookups

gt - Greater than.

Example: - Student.objects.filter(marks__gt=50)

gte - Greater than or equal to.

Example: - Student.objects.filter(marks__gte=50)

lt - Less than.

Example: - Student.objects.filter(marks__lt=50)

lte - Less than or equal to.

Example: - Student.objects.filter(marks__lte=50)

Field Lookups

startswith - Case-sensitive starts-with.

Example: - Student.objects.filter(name__startswith='r')

istartswith - Case-insensitive starts-with.

Example: - Student.objects.filter(name__istartswith='r')

endswith - Case-sensitive ends-with.

Example:- Student.objects.filter(name__endswith='j')

iendswith - Case-insensitive ends-with.

Example:- Student.objects.filter(name__iendswith='j’)

Field Lookups

range - Range test (inclusive).

Example:- Student.objects.filter(passdate__range=('2020-04-01', '2020-05-05'))

SQL: - SELECT ... WHERE admission_date BETWEEN '2020-04-01' and '2020-05-05';

You can use range anywhere you can use BETWEEN in SQL — for dates, numbers and even characters.

date - For datetime fields, casts the value as date. Allows chaining additional field lookups. Takes a date value.

Example:-

Student.objects.filter(admdatetime__date=date(2020, 6, 5))

Student.objects.filter(admdatetime__date__gt=date(2020, 6, 5))

year - For date and datetime fields, an exact year match. Allows chaining additional field lookups. Takes an integer year.

Example:-

Student.objects.filter(passdate__year=2020)

Student.objects.filter(passdate__year__gt=2019)

Field Lookups

month - For date and datetime fields, an exact month match. Allows chaining additional field lookups. Takes an integer 1 (January) through 12 (December).

Example:-

Student.objects.filter(passdate__month=6)

Student.objects.filter(passdate__month__gt=5)

day - For date and datetime fields, an exact day match. Allows chaining additional field lookups. Takes an integer day.

Example:-

Student.objects.filter(passdate__day=5)

Student.objects.filter(passdate__day__gt=3)

This will match any record with a pub_date on the third day of the month, such as January 3, July 3, etc.

Field Lookups

week - For date and datetime fields, return the week number (1-52 or 53) according to ISO-8601, i.e., weeks start on a Monday and the first week contains the year's first Thursday.

Example:-

Student.objects.filter(passdate__week=23)

Student.objects.filter(passdate__week__gt=22)

week_day - For date and datetime fields, a 'day of the week' match. Allows chaining additional field lookups.

Takes an integer value representing the day of week from 1 (Sunday) to 7 (Saturday).

Example:-

Student.objects.filter(passdate__week_day=6)

Student.objects.filter(passdate__week_day__gt=5)

This will match any record with a admission_date that falls on a Monday (day 2 of the week), regardless of the month or year in which it occurs. Week days are indexed with day 1 being Sunday and day 7 being Saturday.

Field Lookups

quarter - For date and datetime fields, a ‘quarter of the year’ match. Allows chaining additional field lookups. Takes an integer value between 1 and 4 representing the quarter of the year.

Example to retrieve entries in the second quarter (April 1 to June 30):

Student.objects.filter(passdate__quarter=2)

time - For datetime fields, casts the value as time. Allows chaining additional field lookups. Takes a datetime.time value.

Example:- Student.objects.filter(admdatetime__time__gt=time(6,00))

hour - For datetime and time fields, an exact hour match. Allows chaining additional field lookups. Takes an integer between 0 and 23.

Example:- Student.objects.filter(admdatetime__hour__gt=5)

Field Lookups

minute - For datetime and time fields, an exact minute match. Allows chaining additional field lookups. Takes an integer between 0 and 59.

Example:- Student.objects.filter(admdatetime__minute__gt=50)

second - For datetime and time fields, an exact second match. Allows chaining additional field lookups. Takes an integer between 0 and 59.

Example:- Student.objects.filter(admdatetime__second__gt=30)

isnull - Takes either True or False, which correspond to SQL queries of IS NULL and IS NOT NULL, respectively.

Example:- Student.objects.filter(roll__isnull=False)

regex

iregex

Aggregation

sometimes you will need to retrieve values that are derived by summarizing or aggregating a collection of objects.

aggregate() - It is a terminal clause for a QuerySet that, when invoked, returns a dictionary of name-value pairs. The name is an identifier for the aggregate value; the value is the computed aggregate. The name is automatically generated from the name of the field and the aggregate function.

Syntax:- aggregate(name=agg_function('field'), name=agg_function('field'),)

field - It describes the aggregate value that we want to compute.

name - If you want to manually specify a name for the aggregate value, you can do so by providing that name when you specify the aggregate clause.

annotate() - Per-object summaries can be generated using the annotate() clause. When an annotate() clause is specified, each object in the QuerySet will be annotated with the specified values. The output of the annotate() clause is a QuerySet; this QuerySet can be modified using any other QuerySet operation, including filter(), order_by(), or even additional calls to annotate().

Syntax:- annotate(name=agg_function('field'), name=agg_function('field'),)

Aggregation Functions

Django provides the following aggregation functions in the **django.db.models** module.

Avg(expression, output_field=None, distinct=False, filter=None, **extra) - It returns the mean value of the given expression, which must be numeric unless you specify a different output_field.

Default alias: <field>__avg

Return type: float if input is int, otherwise same as input field, or output_field if supplied

Has one optional argument:

distinct - If distinct=True, Avg returns the mean value of unique values. This is the SQL equivalent of AVG(DISTINCT <field>). The default value is False.

Count(expression, distinct=False, filter=None, **extra) - It returns the number of objects that are related through the provided expression.

Default alias: <field>__count

Return type: int

Has one optional argument:

distinct - If distinct=True, the count will only include unique instances. This is the SQL equivalent of COUNT(DISTINCT <field>). The default value is False.

Aggregation Functions

Max(expression, output_field=None, filter=None, **extra)- It returns the maximum value of the given expression.

Default alias: <field>__max

Return type: same as input field, or output_field if supplied

Min(expression, output_field=None, filter=None, **extra) - It returns the minimum value of the given expression.

Default alias: <field>__min

Return type: same as input field, or output_field if supplied

Sum(expression, output_field=None, distinct=False, filter=None, **extra) - It computes the sum of all values of the given expression.

Default alias: <field>__sum

Return type: same as input field, or output_field if supplied

Has one optional argument:

distinct - If distinct=True, Sum returns the sum of unique values. This is the SQL equivalent of SUM(DISTINCT <field>). The default value is False.

Aggregation Functions

StdDev(expression, output_field=None, sample=False, filter=None, **extra) - It returns the standard deviation of the data in the provided expression.

Default alias: <field>__stddev

Return type: float if input is int, otherwise same as input field, or output_field if supplied

Has one optional argument:

sample - By default, StdDev returns the population standard deviation. However, if sample=True, the return value will be the sample standard deviation.

Variance(expression, output_field=None, sample=False, filter=None, **extra) - It returns the variance of the data in the provided expression.

Default alias: <field>__variance

Return type: float if input is int, otherwise same as input field, or output_field if supplied

Has one optional argument:

sample - By default, Variance returns the population variance. However, if sample=True, the return value will be the sample variance.

Q Objects

Q object is an object used to encapsulate a collection of keyword arguments. These keyword arguments are specified as in “Field lookups” .

If you need to execute more complex queries, you can use Q objects.

Q objects can be combined using the & and | operators. When an operator is used on two Q objects, it yields a new Q object.

from django.db.models import Q

& (AND) Operator

Example:- Student.objects.filter(Q(id=6) & Q(roll=106))

| (OR) Operator

Example:- Student.objects.filter(Q(id=6) | Q(roll=108))

~ Negation Operator

Example:- Student.objects.filter(~Q(id=6))

Limiting QuerySets

Use a subset of Python's array-slicing syntax to limit your QuerySet to a certain number of results. This is the equivalent of SQL's LIMIT and OFFSET clauses.

Student.objects.all()[:5] - This returns First 5 objects

Student.objects.all()[5:10] - This returns sixth through tenth objects

Student.objects.all()[-1] - This is not valid.

Student.objects.all()[:10:2] - This returns a list of every second object of the first 10.