Models

January 7, 2019

1 Django ORM

Object Relational Mapping. `django.db` `models` provides class interface to a database. Model definition is converted into database queries and object mappings by django. *SqlAlchemy* is another popular tool. Django has its own implementation.

`models.py` is expected to contain your data description. Any class derived from `models.Model` will be a relational model.

```
from django.db import models

class Student(models.Model):
    sid = models.CharField(max_length=10, primary_key = True)
    name = models.CharField(max_length=30)
    surname = models.CharField(max_length=30)
    # if _str_ is implemented object details are displayed in query results
    def __str__(self):
        return ' '.join([str(self.sid),str(self.name),str(self.surname)])
```

This definition will generate all necessary queries to generate and access a row called `<app>_<class>`:

```
CREATE TABLE "student_student" (
"sid" varchar(10) NOT NULL PRIMARY KEY,
"name" varchar(30) NOT NULL,
"surname" varchar(30) NOT NULL);
```

Some important field types are: `AutoField`, `BinaryField`, `BooleanField`, `CharField`, `DecimalField`, `DateTimeField`, `DateField`, `TimeField`, `EmailField`, `FileField`, `FloatField`, `ImageField`, `IntegerField`, `GenericIPAddressField`, `URLField`

Fields get some options: * `null` (bool) if field can be null. * `primary_key` (bool) if this field is the primary key. If no primary key specified an autogenerated key field is added (called `id`) * `db_index` (bool) if index should be created in database. Secondary indexes are created by this flag. * `default` a default value if field is unspecified. * `unique` (bool) * `max_length` for string like fields, maximum size allowed * `auto_now` for date and time fields if current date/time is automatically set.

```
In [2]: import os
        import sys
```

```
os.environ.setdefault("DJANGO_SETTINGS_MODULE", "d2017.settings")
import django
django.setup()

from student.models import Department, Student, Course
```

1.1 Adding objects in database

simply constructing an object from class with parameters. All fields can be assigned in the object later. `save()` method creates the object (row in the table)

```
In [3]: st = Student(55727, 'Onur Tolga', 'ehitolu')
        st.save()
        Student(name='Ali', surname='Cin', sid=211112).save()
        st.sid = 44444
        # since primary key changed, save will generate another object
        st.save()
        # now query all objects
        Student.objects.all()

Out[3]: <QuerySet [<Student: 211112 Ali Cin>, <Student: 44444 Onur Tolga ehitolu>, <Student: 1
```

Add more objects. You can use `objects.create()` method of class to create and save in a single call.

```
In [4]: for sid,name,sname in [(123415,'Bugs','Bunny'),(423122,'Daffy','Duck'),(314213,'Sylves
              (76341,'Marty','Martian'),(221412,'Tasmanian','Devil'),(652141,'Elmor','Fudd')]:
            Student.objects.create(sid=sid,name=name,surname=sname)


        ---------------------------------------------------------------------------

        IntegrityError                                Traceback (most recent call last)

        /usr/lib/python3/dist-packages/django/db/backends/utils.py in execute(self, sql, params
         63             else:
    ---> 64                 return self.cursor.execute(sql, params)
         65


        /usr/lib/python3/dist-packages/django/db/backends/sqlite3/base.py in execute(self, quer
        336         query = self.convert_query(query)
    --> 337         return Database.Cursor.execute(self, query, params)
        338


        IntegrityError: UNIQUE constraint failed: student_student.sid
```

```
The above exception was the direct cause of the following exception:


    IntegrityError                            Traceback (most recent call last)

    <ipython-input-4-6e71049dbb97> in <module>()
      1 for sid,name,sname in [(123415,'Bugs','Bunny'),(423122,'Daffy','Duck'),(314213,'Syl
      2       (76341,'Marty','Martian'),(221412,'Tasmanian','Devil'),(652141,'Elmor','Fudd
----> 3     Student.objects.create(sid=sid,name=name,surname=sname)


    /usr/lib/python3/dist-packages/django/db/models/manager.py in manager_method(self, *arg
     83         def create_method(name, method):
     84             def manager_method(self, *args, **kwargs):
---> 85                 return getattr(self.get_queryset(), name)(*args, **kwargs)
     86             manager_method.__name__ = method.__name__
     87             manager_method.__doc__ = method.__doc__


    /usr/lib/python3/dist-packages/django/db/models/query.py in create(self, **kwargs)
    397         obj = self.model(**kwargs)
    398         self._for_write = True
--> 399         obj.save(force_insert=True, using=self.db)
    400         return obj
    401


    /usr/lib/python3/dist-packages/django/db/models/base.py in save(self, force_insert, fo
    794
    795         self.save_base(using=using, force_insert=force_insert,
--> 796                        force_update=force_update, update_fields=update_fields)
    797     save.alters_data = True
    798


    /usr/lib/python3/dist-packages/django/db/models/base.py in save_base(self, raw, force_
    822             if not raw:
    823                 self._save_parents(cls, using, update_fields)
--> 824             updated = self._save_table(raw, cls, force_insert, force_update, using
    825         # Store the database on which the object was saved
    826         self._state.db = using


    /usr/lib/python3/dist-packages/django/db/models/base.py in _save_table(self, raw, cls,
    906
    907             update_pk = bool(meta.has_auto_field and not pk_set)
--> 908             result = self._do_insert(cls._base_manager, using, fields, update_pk, r
```

```
    909             if update_pk:
    910                 setattr(self, meta.pk.attname, result)


    /usr/lib/python3/dist-packages/django/db/models/base.py in _do_insert(self, manager, us
    945         """
    946         return manager._insert([self], fields=fields, return_id=update_pk,
--> 947                                using=using, raw=raw)
    948
    949     def delete(self, using=None, keep_parents=False):


    /usr/lib/python3/dist-packages/django/db/models/manager.py in manager_method(self, *arg
     83         def create_method(name, method):
     84             def manager_method(self, *args, **kwargs):
---> 85                 return getattr(self.get_queryset(), name)(*args, **kwargs)
     86             manager_method.__name__ = method.__name__
     87             manager_method.__doc__ = method.__doc__


    /usr/lib/python3/dist-packages/django/db/models/query.py in _insert(self, objs, fields
   1043         query = sql.InsertQuery(self.model)
   1044         query.insert_values(fields, objs, raw=raw)
-> 1045         return query.get_compiler(using=using).execute_sql(return_id)
   1046     _insert.alters_data = True
   1047     _insert.queryset_only = False


    /usr/lib/python3/dist-packages/django/db/models/sql/compiler.py in execute_sql(self, re
   1052         with self.connection.cursor() as cursor:
   1053             for sql, params in self.as_sql():
-> 1054                 cursor.execute(sql, params)
   1055             if not (return_id and cursor):
   1056                 return


    /usr/lib/python3/dist-packages/django/db/backends/utils.py in execute(self, sql, params
     77         start = time()
     78         try:
---> 79             return super(CursorDebugWrapper, self).execute(sql, params)
     80         finally:
     81             stop = time()


    /usr/lib/python3/dist-packages/django/db/backends/utils.py in execute(self, sql, params
     62                 return self.cursor.execute(sql)
     63             else:
---> 64                 return self.cursor.execute(sql, params)
```

```
     65
     66     def executemany(self, sql, param_list):


    /usr/lib/python3/dist-packages/django/db/utils.py in __exit__(self, exc_type, exc_value
     92                 if dj_exc_type not in (DataError, IntegrityError):
     93                     self.wrapper.errors_occurred = True
---> 94                 six.reraise(dj_exc_type, dj_exc_value, traceback)
     95
     96     def __call__(self, func):


    /usr/lib/python3/dist-packages/django/utils/six.py in reraise(tp, value, tb)
    683             value = tp()
    684         if value.__traceback__ is not tb:
--> 685             raise value.with_traceback(tb)
    686         raise value
    687


    /usr/lib/python3/dist-packages/django/db/backends/utils.py in execute(self, sql, params
     62                 return self.cursor.execute(sql)
     63             else:
---> 64                 return self.cursor.execute(sql, params)
     65
     66     def executemany(self, sql, param_list):


    /usr/lib/python3/dist-packages/django/db/backends/sqlite3/base.py in execute(self, quer
    335             return Database.Cursor.execute(self, query)
    336         query = self.convert_query(query)
--> 337         return Database.Cursor.execute(self, query, params)
    338
    339     def executemany(self, query, param_list):


    IntegrityError: UNIQUE constraint failed: student_student.sid
```

1.2 Queries

- `<classname>.objects` give a query interface with various methods. Some methods return a `QuerySet` object that can be iterated or accessed by an index (like a `list`)
- `all()` method returns all objects (`select * from <table>`)
- `get(<lookup>[,<lookup>]*)` method returns only one object. Query result should be unique
- `filter(<loookup>[,<lookup>]*)` method returns all objects matching lookup (`select ... where...`)

- some Field lookups are:
- `fieldname = value` or `fieldname__exact = value` . Equality (`select .... where fieldname=value`)
- `fieldname__contains = value` string contains the value (`select ... where fieldname like '%value%'`)
- `fieldname__gt = value` greater than (`select ... where fieldname > value`). Other comparison operators: `gte, lt, lte`
- `fieldname__startswith = value` string starts with the value (`select ... where fieldname like 'value%'`). Also`endswith' works same way.
- `fieldname__in = iteratable` if field name in the following set of values that are iterated. `fieldname_in = ['a','b','c']` (`select ... where fieldname IN ('a','b','c')`). Also other query result can be used as: `python cset = Department.objects.filter(did__ge='571') st = Student.objects.filter(department__in = cset)`
- `fieldname__range = (start, end)` range test, integer, string or date/time fields. (`SELECT ... WHERE fieldname BETWEEN (start,end)`. Range is inclusive.
- `fieldname__date=value`, matches date part of the date or datetime field. Similarly `year, month, day, week, time, hour, minute, second` can be used to match other fields.
- `fieldname__isnull` null test.
- `fieldname__regex = regexpattern` regular expression test.
- `iexact, istartswith, iendswith, icontains, iregex` are case insensitive versions of their corresponding lookups.
- Lookup parameters are combined with `AND` in SQL. as:

`filter(name='Ali',sname__endswith='olu', sid__gt = '51231')` is executed as: `SELECT ... WHERE name='Ali' AND sname LIKE '%olu' AND sid__gt=51231;` * `exclude(<lookup>)` Works like filter but negates the lookup. * QuerySets can be chained as:

`Student.filter(sid__ge = 10000).exclude(name__range = ('AA','IA')).filter(name__endswith = 'Z')` * For conjunction (SQL `OR`) and complex expression in parenthesis, you need `Q` objects (`from django.db.models import Q`) Not in scope of this course.

```
In [ ]: print("--all--\n",Student.objects.all())
        print("--get--\n",Student.objects.get(sid=76341))
        print("--get--\n",Student.objects.get(name='Bugs'))
        print("--filter--\n",Student.objects.filter(name__startswith='T'))
        print("--filter--\n",Student.objects.filter(name__contains='ty'))
        print("--filter-excl-\n",Student.objects.filter(name__contains='ty').exclude(sid__start
```

1.3 Updating Data

- Single objects can be updated by fetching, updating field and saving as: `python s=Student.objects.get(sid=55727) s.name = 'Onur' s.save()`
- Also `update()` method can be used in query sets: `python s.Student.objects.filter(sid=55727).update(name='Onur') s.Student.objects.filter(name__gt = 'TT').update(count=0)`
- `delete()` method on an object or query set deletes the object or objects in query set.

1.4 Model Class Relations

- **One to Many** relation maps an object field into a set of other objects. For example a Manifacturer has multiple Car models, a Bus has multiple Passengers, a Department has multiple Students. But inverse does not hold, a Car has only one Manifacturer as brand, a Passenger can be on a single Bus at a time. A Student can register one major Department.
- **Many to Many** relation maps many objects into more than one objects. For example a Book can be written by more than one Authors and each Author can write more than one Book. A Student can register to many Courses and each Course has many Students.
- **One to One** relation maps an object to another. For example Metadata of a file object is related to its Content, EmailBody is related to EmailHeader. This relation can be used as logical partitioning of the object.

Django provide this relations and creates required tables, fields, indexes automatically. Also provides SQL supported constraints so that when an object is deleted, the related object field is automatically set, object is deleted or error raised to enforce deletion of referring object first.

- `fieldname=ForeignKey('ClassName')` in a model creates a **One to Many** relation between the `ClassName` and the current class.

`Car.producedby=ForeignKey('Manifacturer')`, `Passenger.passat=ForeignKey('Bus')`, `Student.dept=ForeignKey('Department')`. * `fieldname=ManytoManyField('ClassName')` creates a **Many to Many** relation.

`Book.by=ManyToManyField('Author')`, `Student.registered=ManytoManyField('Course')`. * `fieldname=OnetoOneField('ClassName')` creates a **One to One** relation.

The relation among referring class to referred class is through the defined field (`producedby`, `dept`, `by`, `registered`. In the referred class the inverse relation is defined by `classname_set`. As in `Manifacturer`: `car_set`, in `Author`: `book_set`, in `Course`: `student_set` (class name is lowercased)

If there are more than one such relations or you need to give a more readable name, you can use `related_name` in the field definition.

Let us improve our model:

```
class Department(models.Model):
    did = models.CharField(max_length=10, primary_key = True)
    name = models.CharField(max_length=30)

class Course(models.Model):
    cid = models.CharField(max_length=10, primary_key = True)
    name = models.CharField(max_length=100)
    # this is tricky, it creates  many to many relation to between Course and Course
    # symmetrical=True automatically makes relation bidirectional as in friend-friend,
    # if A is friend of B, B is friend of A. prerequisite relations is not like that
    prereq = models.ManyToManyField('self', symmetrical=False, blank=True)
    def __str__(self):
        return ':'.join([str(self.cid),str(self.name)])


class Student(models.Model):
```

```
    sid = models.CharField(max_length=10, primary_key = True)
    name = models.CharField(max_length=30)
    surname = models.CharField(max_length=30)
    # many to one from Department
    department = models.ForeignKey(Department,blank=True, null=True)
    # many to many with courses
    took = models.ManyToManyField(Course,related_name = 'taken')
    # many to many with courses
    registered = models.ManyToManyField(Course,related_name = 'enrolled')

    def __str__(self):
        return ' '.join([str(self.sid),str(self.name),str(self.surname)])
```

You need to call `manage.py makemigrations` afterwards.
This will create the following SQL tables (without indexes):

```
CREATE TABLE "student_course" (
    "cid" varchar(10) NOT NULL PRIMARY KEY,
    "name" varchar(100) NOT NULL);

CREATE TABLE "student_course_prereq" (
    "id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,
    "from_course_id" varchar(10) NOT NULL REFERENCES "student_course" ("cid"),
    "to_course_id" varchar(10) NOT NULL REFERENCES "student_course" ("cid"));

CREATE TABLE "student_department" (
    "did" varchar(10) NOT NULL PRIMARY KEY,
    "name" varchar(30) NOT NULL);

CREATE TABLE "student_student" (
    "sid" varchar(10) NOT NULL PRIMARY KEY,
    "name" varchar(30) NOT NULL,
    "surname" varchar(30) NOT NULL,
    "department_id" varchar(10) NULL REFERENCES "student_department" ("did"));

CREATE TABLE "student_student_registered" (
    "id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,
    "student_id" varchar(10) NOT NULL REFERENCES "student_student" ("sid"),
    "course_id" varchar(10) NOT NULL REFERENCES "student_course" ("cid"));

CREATE TABLE "student_student_took" (
    "id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,
    "student_id" varchar(10) NOT NULL REFERENCES "student_student" ("sid"),
    "course_id" varchar(10) NOT NULL REFERENCES "student_course" ("cid"));
```

```
In [ ]: for cid,cname in [('ceng100','Computer Engineering Orientation.'), ('ceng111','Introduc
        ('ceng140','C Programming.'), ('ceng213','Data Structures.'), ('ceng223','Discrete Comp
```

```
        ('ceng232','Logic Design.'), ('ceng242','Programming Language Concepts.'), ('ceng280',
        ('ceng300','Summer Practice - I.'), ('ceng315','Algorithms.'), ('ceng331','Computer Org
        ('ceng334','Introduction to Operating Systems.'), ('ceng336','Int. to Embedded Systems
        ('ceng351','Data Management And File Structures.'), ('ceng378','Computer Graphics - I.
        ('ceng400','Summer Practice - II.'), ('ceng435','Data Communications and Networking.')
        ('ceng477','Introduction to Computer Graphics.'), ('ceng491','Computer Engineering Desi
            Course.objects.create(cid=cid, name=cname)
        for did,dname in [('572','Aerospace Engineering'), ('563','Chemical Engineering'), ('56
        ('571','Computer Engineering'), ('567','Electrical and Electronics Engineering'), ('560
        ('573','Food Engineering'), ('564','Geological Engineering'), ('568','Industrial Engine
        ('569','Mechanical Engineering'), ('570','Metallurgical and Materials Engineering'), (
            Department.objects.create(did=did,name=dname)
```

```
In [6]: print(Course.objects.all())
        print(Department.objects.all())
        print(Student.objects.all())
```

```
<QuerySet [<Course: ceng100:Computer Engineering Orientation.>, <Course: ceng111:Introduction t
<QuerySet [<Department: Department object>, <Department: Department object>, <Department: Depar
<QuerySet [<Student: 211112 Ali Cin>, <Student: 44444 Onur Tolga ehitolu>, <Student: 123415 Bug
```

```
In [7]: s = Student.objects.get(sid='55727')
        print(s)
```

```
55727 Onur Tolga ehitolu
```

```
In [11]: # setting a foreign key field
         s.department = Department.objects.get(did='571')
         s.save()
```

```
In [ ]: # student_set is defined in Department to get all Students
        d=Department(did='571')
        # it is a set object that can be queried
        d.student_set.all()
```

```
In [ ]: # setting in inverse direction is also possible
        d.student_set.add(Student.objects.get(sid='211112'))
        d.student_set.all()
```

```
In [14]: # ManytoMany fields return a set when inspected
         print(repr(s.took))
         # you canget queries on the sets
         print(s.took.all())
         s.took.add(Course.objects.get(cid='ceng350'))
         s.took.add(Course.objects.get(cid='ceng315'))
```

```
<django.db.models.fields.related_descriptors.create_forward_many_to_many_manager.<locals>.Manyl
<QuerySet [<Course: ceng315:Algorithms.>, <Course: ceng350:Software Engineering.>, <Course: cen
```

```
In [15]: s.took.all()

Out[15]: <QuerySet [<Course: ceng315:Algorithms.>, <Course: ceng350:Software Engineering.>, <C

In [16]: c=Course.objects.get(cid='ceng350')
         c.taken.all()

Out[16]: <QuerySet [<Student: 123415 Bugs Bunny>, <Student: 55727 Onur Tolga ehitolu>, <Student

In [18]: c.taken.add(Student.objects.get(name='Tweety'))

In [19]: Student.objects.get(name='Tweety').took.all()

Out[19]: <QuerySet [<Course: ceng350:Software Engineering.>]>

In [20]: qs = Student.objects.all()

In [21]: for st in qs.order_by('surname'):
             print('\n===\n',st.name, st.surname)
             print('took:')
             for c in st.took.all():
                 print('  ->', c.cid, c.name)
             print('registered:')
             for c in st.registered.all():
                 print('  ->', c.cid, c.name)

         for c in Course.objects.all().order_by('cid'):
             print('\n===\n', c.cid, c.name)
             for s in c.taken.all():
                 print('  ->', s.sid, s.name, s.surname)
             for s in c.enrolled.all():
                 print('  ->', s.sid, s.name, s.surname)

         Student.objects.exclude(name__range=('AA','MM')).filter(sid__lt = '4')


===
 Bugs Bunny
took:
  -> ceng350 Software Engineering.
registered:
  -> ceng223 Discrete Computational Structures.
  -> ceng351 Data Management And File Structures.

===
 Sylvester Cat
took:
  -> ceng232 Logic Design.
registered:
```

```
  -> ceng100 Computer Engineering Orientation.
  -> ceng111 Introduction to Computer Engineering Concepts.
  -> ceng140 C Programming.
  -> ceng223 Discrete Computational Structures.
  -> ceng378 Computer Graphics - I.
  -> ceng384 Signals and Systems for Computer Engineers.
  -> ceng436 Data Communications and Computer Networking.

===
 Ali Cin
took:
  -> ceng140 C Programming.
registered:
  -> ceng111 Introduction to Computer Engineering Concepts.
  -> ceng232 Logic Design.

===
 Tazmanian Devil
took:
registered:
  -> ceng223 Discrete Computational Structures.

===
 Daffy Duck
took:
registered:

===
 Hara Gürele
took:
registered:

===
 Tweety Tweets
took:
  -> ceng350 Software Engineering.
registered:
  -> ceng100 Computer Engineering Orientation.

===
 Hello World
took:
registered:

===
 asdasd  lczxc zxzxc
took:
registered:
```

```
===
 weqwe qweqwe
took:
registered:

===
 Onur Tolga ehitolu
took:
registered:
  -> ceng350 Software Engineering.
  -> ceng351 Data Management And File Structures.

===
 Onur Tolga ehitolu
took:
  -> ceng315 Algorithms.
  -> ceng350 Software Engineering.
  -> ceng491 Computer Engineering Design I.
registered:
  -> ceng140 C Programming.
  -> ceng223 Discrete Computational Structures.
  -> ceng378 Computer Graphics - I.
  -> ceng435 Data Communications and Networking.
  -> ceng491 Computer Engineering Design I.

===
 ceng100 Computer Engineering Orientation.
  -> 314213 Sylvester Cat
  -> 423145 Tweety Tweets

===
 ceng111 Introduction to Computer Engineering Concepts.
  -> 314213 Sylvester Cat
  -> 211112 Ali Cin

===
 ceng140 C Programming.
  -> 211112 Ali Cin
  -> 314213 Sylvester Cat
  -> 55727 Onur Tolga ehitolu

===
 ceng213 Data Structures.

===
 ceng223 Discrete Computational Structures.
  -> 221412 Tazmanian Devil
```

```
  -> 55727 Onur Tolga ehitolu
  -> 314213 Sylvester Cat
  -> 123415 Bugs Bunny

===
 ceng232 Logic Design.
  -> 314213 Sylvester Cat
  -> 211112 Ali Cin

===
 ceng242 Programming Language Concepts.

===
 ceng280 Formal Languages And Abstract Machines.

===
 ceng300 Summer Practice - I.

===
 ceng315 Algorithms.
  -> 55727 Onur Tolga ehitolu

===
 ceng331 Computer Organization.

===
 ceng334 Introduction to Operating Systems.

===
 ceng336 Int. to Embedded Systems Development.

===
 ceng350 Software Engineering.
  -> 123415 Bugs Bunny
  -> 55727 Onur Tolga ehitolu
  -> 423145 Tweety Tweets
  -> 44444 Onur Tolga ehitolu

===
 ceng351 Data Management And File Structures.
  -> 123415 Bugs Bunny
  -> 44444 Onur Tolga ehitolu

===
 ceng378 Computer Graphics - I.
  -> 55727 Onur Tolga ehitolu
  -> 314213 Sylvester Cat
```

```
===
 ceng384 Signals and Systems for Computer Engineers.
  -> 314213 Sylvester Cat

===
 ceng400 Summer Practice - II.

===
 ceng435 Data Communications and Networking.
  -> 55727 Onur Tolga ehitolu

===
 ceng436 Data Communications and Computer Networking.
  -> 314213 Sylvester Cat

===
 ceng477 Introduction to Computer Graphics.

===
 ceng491 Computer Engineering Design I.
  -> 55727 Onur Tolga ehitolu
  -> 55727 Onur Tolga ehitolu

===
 ceng492 Computer Engineering Design II.
```

```
Out[21]: <QuerySet [<Student: 221412 Tazmanian Devil>, <Student: 23523523 weqwe qweqwe>, <Stude
```

```
In [22]: Student.objects.all().values('sid','name')
```

```
Out[22]: <QuerySet [{'name': 'Ali', 'sid': '211112'}, {'name': 'Onur Tolga', 'sid': '44444'},
```

```
In [23]: Student.objects.all()
```

```
Out[23]: <QuerySet [<Student: 211112 Ali Cin>, <Student: 44444 Onur Tolga ehitolu>, <Student:
```

```
In [24]: Student.objects.filter(sid__gt = '5').delete()
```

```
Out[24]: (11,
          {'student.Student': 3,
           'student.Student_registered': 5,
           'student.Student_took': 3})
```

```
In [25]: d = Department.objects.get(did='571')
         d.student_set.all()
```

```
Out[25]: <QuerySet [<Student: 123415 Bugs Bunny>]>
```

```
In [26]: d.student_set.add(Student.objects.get(name='Bugs'))
```

```
In [27]: d.student_set.all()

Out[27]: <QuerySet [<Student: 123415 Bugs Bunny>]>
```

What is left?: * advanced queries by using django.db.Q * advanced update by django.db.F * aggrate functions django.db. Count, Avg, Sum, ... * group by functionality through annote() and aggragate functions * write your custom field lookups * write your custom aggrate functions

```
In [28]: from django.db.models import Q

In [29]: Student.objects.filter(Q(name__gt='T') | Q(sid__lt='4000'))

Out[29]: <QuerySet [<Student: 211112 Ali Cin>, <Student: 123415 Bugs Bunny>, <Student: 314213 S

In [ ]: from django.db.models import Avg,Max,Count
        allstd=Student.objects.all()
        print(allstd.count())
        print(allstd.aggregate(Avg('sid')))
        print(allstd.aggregate(Max('sid')), allstd.aggregate(Max('sid')))
        for v in Student.objects.values('sid').annotate(Count('took'),Count('registered')):
            print(v)
        print('====')
        for c in Course.objects.values('cid','name').annotate(Count('enrolled')):
            print(c)
```