Querying Django Models

class Language(models.Model):

    lang\_name = models.CharField(max\_length=30)

    def \_\_str\_\_(self):

        return self.lang\_name

class Framework(models.Model):

    frame\_name = models.CharField(max\_length=30)

    frame\_lang = models.ForeignKey(Language, on\_delete = models.CASCADE)

    def \_\_str\_\_(self):

        return self.frame\_name

One programming language can have multiple Frameworks

So one language have “One-To-Many” Relationship

**Creating Language objects using Django Shell**

python = Language.objects.create(lang\_name = 'Python')

java = Language(lang\_name = 'Java')

php = Language(lang\_name = 'PHP')

perl = Language(lang\_name = 'Perl')

net = Language(lang\_name = '.NET')

python.save()

java.save()

php.save()

perl.save()

net.save()

**Creating Framework objects using Django Shell**

flask = Framework.objects.create(frame\_name = 'Flask')

flask.frame\_lang = python

flask.save()

django = Framework(frame\_name = 'Django', frame\_lang = python)

bottle = Framework(frame\_name = 'Bottle', frame\_lang = python)

django.save()

bottle.save()

springmvc = Framework(frame\_name = 'SpringMVC', frame\_lang = java)

springboot = Framework(frame\_name = 'SpringBoot', frame\_lang = java)

jsp = Framework(frame\_name = 'JSP', frame\_lang = java)

springmvc.save()

springboot.save()

jsp.save()

larvel = Framework(frame\_name = 'Laravel', frame\_lang = php)

larvel.save()

ruby = Framework(frame\_name = 'Ruby on Rails')

ruby.frame\_lang = perl

ruby.save()

asp = Framework(frame\_name = 'ASP .NET')

asp.frame\_lang = net

asp.save()

========================Querying models data from database=====================

>>> Language.objects.all()

<QuerySet [<Language: Python>, <Language: Java>, <Language: PHP>, <Language: Perl>, <Language: .NET>]>

>>> Framework.objects.all()

<QuerySet [<Framework: Flask>, <Framework: Django>, <Framework: Bottle>, <Framework: SpringMVC>, <Framework: SpringBoot>, <Framework: JSP>, <Framework: Laravel>, <Framework: Ruby on Rails>, <Framework: ASP .NET>]>

>>> Language.objects.filter(lang\_name\_\_startswith = 'P')

<QuerySet [<Language: Python>, <Language: PHP>, <Language: Perl>]>

>>> Language.objects.all().filter(lang\_name = 'Python')

<QuerySet [<Language: Python>]>

**Chaining filters:**

>>> Language.objects.all().filter(lang\_name\_\_startswith = 'P').exclude(lang\_name = 'PHP')

<QuerySet [<Language: Python>, <Language: Perl>]>

> Language.objects.all().filter(lang\_name\_\_startswith = 'P').exclude(lang\_name = 'PHP').filter(lang\_name\_\_endswith = 'l')

<QuerySet [<Language: Perl>]>

#### Filtered QuerySets are unique:

#### Q1 = Language.objects.filter(lang\_name\_\_startswith = 'P')

#### Q1

#### <QuerySet [<Language: Python>, <Language: PHP>, <Language: Perl>]>

#### Q2 = Q1.exclude(lang\_name = 'Perl')

#### Q2

#### <QuerySet [<Language: Python>, <Language: PHP>]>

#### id(Q2)

#### 1895612174912

#### Q3 = Q1.filter(lang\_name\_\_contains = 'H')

#### Q3

#### <QuerySet [<Language: Python>, <Language: PHP>]>

#### id(Q3)

#### 1895611326024

#### Q2 and Q3 both are quried on the same Q1 but their result output objects are different, Hence

#### Filtered Querysets are Unique

#### QuerySets are lazy:

#### Q = Language.objects.filter(lang\_name\_\_startswith = 'P')

#### Q = Q.exclude(lang\_name = 'Perl')

#### Q = Q.filter(lang\_name\_\_contains = 'H')

#### Print(Q)

#### Its look like 3 database statements and its 3 times but not its it only one time when reached the line

#### “print(Q)”

### **Retrieving a single object with get():**

>>> Language.objects.get(lang\_name\_\_contains = 'y')

<Language: Python>

* If there are no results that match the query, [**get()**](https://docs.djangoproject.com/en/3.0/ref/models/querysets/#django.db.models.query.QuerySet.get) will raise a **DoesNotExist** exception

>>> Language.objects.get(lang\_name\_\_contains = 'z')

Practice.models.DoesNotExist: Language matching query does not exist.

* if more than one item matches the [**get()**](https://docs.djangoproject.com/en/3.0/ref/models/querysets/#django.db.models.query.QuerySet.get) query.

>>> Language.objects.get(lang\_name\_\_startswith = 'p')

Practice.models.MultipleObjectsReturned: get() returned more than one Language -- it returned 3!

### Limiting QuerySets:

>>> Framework.objects.all()

<QuerySet [<Framework: Flask>, <Framework: Django>, <Framework: Bottle>, <Framework: SpringMVC>, <Framework: SpringBoot>, <Framework: JSP>, <Framework: Laravel>, <Framework: Ruby on Rails>, <Framework: ASP .NET>]>

>>> Framework.objects.all()[:3] This is Equal to “LIMIT = 3” IN SQL

<QuerySet [<Framework: Flask>, <Framework: Django>, <Framework: Bottle>]>

>>> Framework.objects.all()[3:7] This is Equal to OFFSET = 5 LIMIT = 7 IN SQL

<QuerySet [<Framework: SpringMVC>, <Framework: SpringBoot>, <Framework: JSP>, <Framework: Laravel>]>

>>> Framework.objects.all()[::2] 2 is the step value in list slicing

[<Framework: Flask>, <Framework: Bottle>, <Framework: SpringBoot>, <Framework: Laravel>, <Framework: ASP .NET>]

* Negative indexing is not supported.

>>> Framework.objects.all()[-1]

AssertionError: Negative indexing is not supported.

### Field lookups

>>> Framework.objects.filter(frame\_name = 'Bottle')

<QuerySet [<Framework: Bottle>]>

An “exact” match.

>>> Framework.objects.filter(frame\_name\_\_exact = 'flask')

<QuerySet []>

#### A case-insensitive match.

#### >>> Framework.objects.filter(frame\_name\_\_iexact = 'flask')

#### <QuerySet [<Framework: Flask>]>

#### >>> Framework.objects.filter(frame\_name\_\_contains = 'a') This is Equal to LIKE in SQL

#### <QuerySet [<Framework: Flask>, <Framework: Django>, <Framework: Laravel>, <Framework: Ruby on Rails>, <Framework: ASP .NET>]>

#### Contains -------- icontains i -----🡪 means = ignore case sensitve

#### Startswith -------- istartswith

#### Endswith -------- iendswith

### Lookups that span relationships:

>>> Framework.objects.filter(frame\_lang\_\_lang\_name = 'Python')

<QuerySet [<Framework: Flask>, <Framework: Django>, <Framework: Bottle>]>

frame\_lang = > Forigenkey field name of Framework model

\_\_lang­\_name => field name of Lnguage model

from Practice.models import Blog, Author, Entry

>>>Blog.objects.filter(entry\_\_authors\_\_name = 'Vissu')

<QuerySet [<Blog: Django Tutorial>, <Blog: Exploratory data analysis>]>

>>> Blog.objects.filter(entry\_\_authors\_\_name = 'Mano')

<QuerySet []>

>>> Author.objects.filter(entry\_\_blog\_\_name\_\_istartswith = 'd')

<QuerySet [<Author: Vissu>, <Author: mani>]>

**Syntax to Query Multi relation models:**

Resultmodel.objects.filter(relations implemented model\_\_field name of relation to other model\_\_field name\_\_look up= some value)

**More than one filter:**

>>> Blog.objects.filter(entry\_\_headline\_\_icontains = 'Pandas', entry\_\_pub\_date\_\_lte = datetime.date.today())

<QuerySet [<Blog: pandas hack for machine learning>]>

>>>

>>> Blog.objects.filter(entry\_\_headline\_\_icontains = 'Pandas').filter(entry\_\_pub\_date\_\_lte = datetime.date.today())

<QuerySet [<Blog: pandas hack for machine learning>]>

**Note:**

so far, we have constructed filters that compare the value of a model field with a constant. But what if you want to compare the value of a model field with another field on the same model?

Django provides [**F expressions**](https://docs.djangoproject.com/en/3.0/ref/models/expressions/#django.db.models.F) to allow such comparisons.

**>>> Entry.objects.filter(pub\_date = F('mod\_date'))**

**<QuerySet [<Entry: Preprocessing data before feeding it to ML model>, <Entry: Visualizing data using python libraries>]>**

**>>> Entry.objects.filter(pub\_date\_\_lt = F('mod\_date'))**

**<QuerySet [<Entry: Pandas Hack for ML>, <Entry: Django2.0.07>]>**

**>>> Entry.objects.filter(pub\_date\_\_lt = mod\_date)**

**Traceback (most recent call last):**

**File "<console>", line 1, in <module>**

**NameError: name 'mod\_date' is not defined**

**pub\_date and mod\_date are two field names in the same model named as Entry, it’s not allowed to compare directely two fields of a same model in the filter Query. We should use “F” expression to valued field (i.e. Right-side field).**

**Note:**

Django supports the use of **addition, subtraction, multiplication, division, modulo, and power arithmetic** with **F()** objects, both with constants and with other **F()** objects.

**>>> from** **datetime** **import** timedelta

**>>>** Entry.objects.filter(mod\_date\_\_gt=F('pub\_date') + timedelta(days=3))

<QuerySet [<Entry: Pandas Hack for ML>]>

**Note:**

You can also use the double underscore notation to span relationships in an **F()** object. An **F()** object with a double underscore will introduce any joins needed to access the related object.

>>> Entry.objects.filter(authors\_\_name = F(blog\_\_name))

Traceback (most recent call last):

File "<console>", line 1, in <module>

NameError: name 'blog\_\_name' is not defined

>>> Entry.objects.filter(authors\_\_name = F('blog\_\_name'))

<QuerySet []>

There are no entries with blog name same as author name.

**For date and date/time fields, you can add or subtract a [timedelta](https://docs.python.org/3/library/datetime.html" \l "datetime.timedelta" \o "(in Python v3.8)) object.**

**>>> from** **datetime** **import** timedelta

**>>>** Entry.objects.filter(mod\_date\_\_gt=F('pub\_date') + timedelta(days=3))

### **The pk lookup shortcut:**

>>> Blog.objects.get(pk\_\_exact = 1)

<Blog: Exploratory data analysis>

>>>

>>> Blog.objects.get(pk = 1)

<Blog: Exploratory data analysis>

>>>

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

<Blog: Exploratory data analysis>

>>>

>>> Blog.objects.get(id\_\_exact = 1)

<Blog: Exploratory data analysis>

>>>

>>> Author.objects.get(id\_\_exact = 3)

<Author: dinesh>

>>> Author.objects.get(pk=3)

<Author: dinesh>

The use of **pk** isn’t limited to **\_\_exact** queries – any query term can be combined with **pk** to perform a query on the primary key of a model:

**>>> Blog.objects.filter(pk\_\_in = [1,5,3])**

**<QuerySet [<Blog: Data visualization>, <Blog: Exploratory data analysis>]>**

**>>> Blog.objects.filter(id\_\_gt = 2)**

**<QuerySet [<Blog: Data visualization>, <Blog: Django Tutorial>]>**

**pk** **lookups also work across joins.**

**>>> Entry.objects.filter(blog\_\_pk\_\_in = [1,2,6])**

**<QuerySet [<Entry: Pandas Hack for ML>, <Entry: Preprocessing data before feeding it to ML model>]>**

**>>> Entry.objects.filter(authors\_\_id\_\_exact = 3)**

**<QuerySet [<Entry: Pandas Hack for ML>]>**

**>>> Entry.objects.filter(authors\_\_id\_\_lt = 3)**

**<QuerySet [<Entry: Django2.0.07>, <Entry: Preprocessing data before feeding it to ML model>, <Entry: Visualizing data using python libraries>]>**

### **Caching and QuerySets:**

Each **[QuerySet](https://docs.djangoproject.com/en/3.0/ref/models/querysets/" \l "django.db.models.query.QuerySet" \o "django.db.models.query.QuerySet)** contains a cache to minimize database access. Understanding how it works will allow you to write the most efficient code.

In a newly created **[QuerySet](https://docs.djangoproject.com/en/3.0/ref/models/querysets/" \l "django.db.models.query.QuerySet" \o "django.db.models.query.QuerySet)**, the cache is empty. The first time a **[QuerySet](https://docs.djangoproject.com/en/3.0/ref/models/querysets/" \l "django.db.models.query.QuerySet" \o "django.db.models.query.QuerySet)** is evaluated – and, hence, a database query happens – Django saves the query results in the **[QuerySet](https://docs.djangoproject.com/en/3.0/ref/models/querysets/" \l "django.db.models.query.QuerySet" \o "django.db.models.query.QuerySet)**’s cache and returns the results that have been explicitly requested Subsequent evaluations of the **[QuerySet](https://docs.djangoproject.com/en/3.0/ref/models/querysets/" \l "django.db.models.query.QuerySet" \o "django.db.models.query.QuerySet)** reuse the cached results

**>>>** print([e.headline **for** e **in** Entry.objects.all()])

**>>>** print([e.pub\_date **for** e **in** Entry.objects.all()])

That means the same database query will be executed twice, effectively doubling your database load. Also, there’s a possibility the two lists may not include the same database records, because an **Entry** may have been added or deleted in the split second between the two requests.

To avoid this problem, save the [**QuerySet**](https://docs.djangoproject.com/en/3.0/ref/models/querysets/#django.db.models.query.QuerySet) and reuse it:

**>>>** queryset = Entry.objects.all()

**>>>** print([p.headline **for** p **in** queryset]) *# Evaluate the query set.*

**>>>** print([p.pub\_date **for** p **in** queryset]) *# Re-use the cache from the evaluation.*

#### When QuerySets are not cached:

Querysets do not always cache their results. When evaluating only part of the queryset, the cache is checked, but if it is not populated then the items returned by the subsequent query are not cached.

**>>>** queryset = Entry.objects.all()

**>>>** print(queryset[5]) *# Queries the database*

**>>>** print(queryset[5]) *# Queries the database again*

**However, if the entire queryset has already been evaluated, the cache will be checked instead:**

**>>>** queryset = Entry.objects.all()

**>>>** [entry **for** entry **in** queryset] *# Queries the database*

**>>>** print(queryset[5]) *# Uses cache*

**>>>** print(queryset[5]) *# Uses cache*