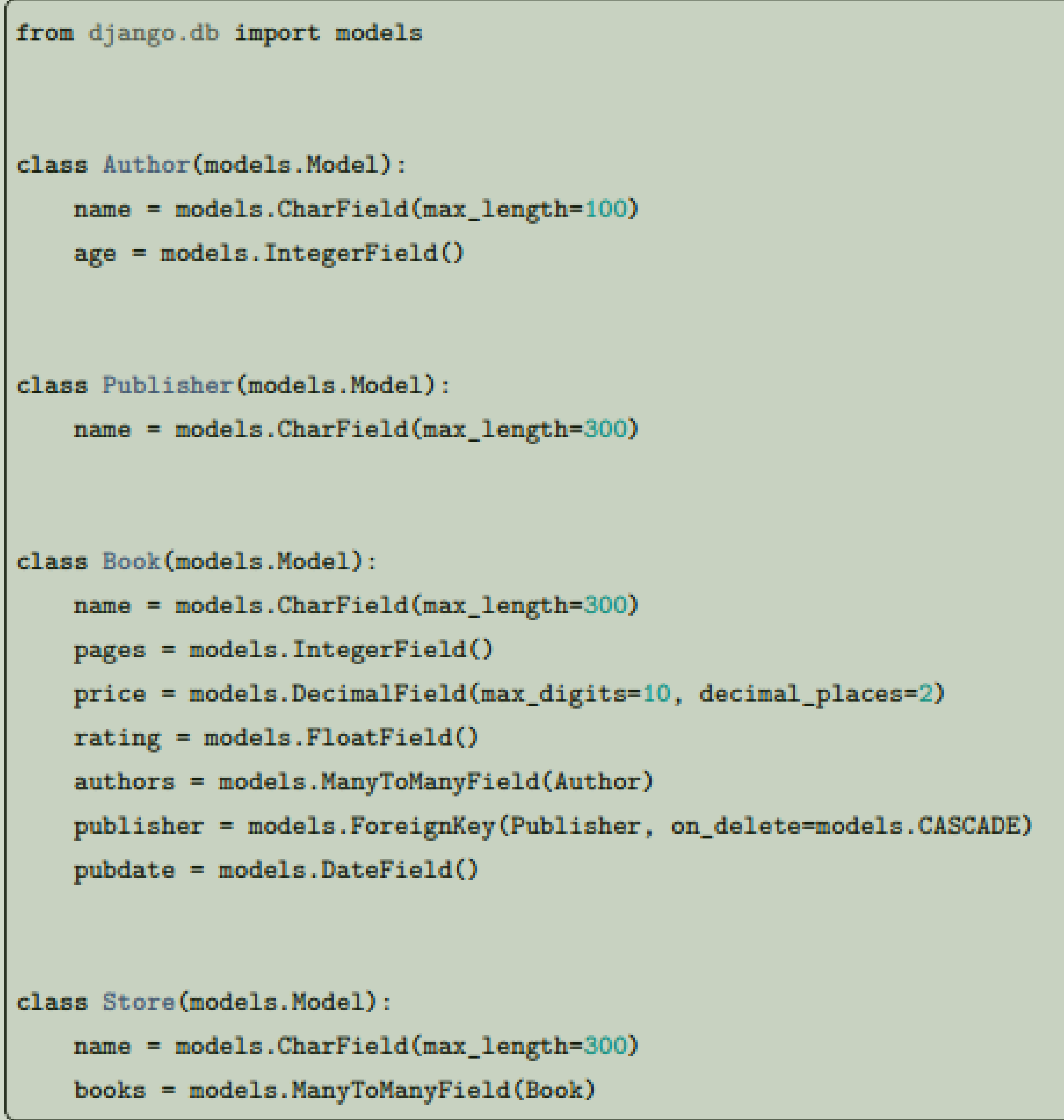
AGGREGATION

Django mastery in Nepali

Aggregation

The topic guide on Django’s database-abstraction API described the way that you can use Django queries
that create, retrieve, update and delete individual objects. However, sometimes you will need to retrieve
values that are derived by summarizing or aggregating a collection of objects. This topic guide describes the
ways that aggregate values can be generated and returned using Django queries.
Throughout this guide, we’ll refer to the following models. These models are used to track the inventory for
a series of online bookstores:



```
from django.db import models


class Author(models.Model):
    name = models.CharField(max_length=100)
    age = models.IntegerField()


class Publisher(models.Model):
    name = models.CharField(max_length=300)


class Book(models.Model):
    name = models.CharField(max_length=300)
    pages = models.IntegerField()
    price = models.DecimalField(max_digits=10, decimal_places=2)
    rating = models.FloatField()
    authors = models.ManyToManyField(Author)
    publisher = models.ForeignKey(Publisher, on_delete=models.CASCADE)
    pubdate = models.DateField()


class Store(models.Model):
    name = models.CharField(max_length=300)
    books = models.ManyToManyField(Book)
```

```
# Total number of books.
>>> Book.objects.count()
2452
# Total number of books with
publisher=BaloneyPress
>>>
Book.objects.filter(publisher__na
me="BaloneyPress").count()
73
# Average price across all books,
provide default to be returned
instead
# of None if no books exist.
>>> from django.db.models import
Avg
>>>
Book.objects.aggregate(Avg("price
", default=0))
{'price__avg': 34.35}
# Max price across all books,
provide default to be returned
instead of
# None if no books exist.
>>> from django.db.models import
Max
>>>
Book.objects.aggregate(Max("price
", default=0))
{'price__max': Decimal('81.20')}
```

```
# Difference between the highest
priced book and the average price
of all books.
>>> from django.db.models import
FloatField
>>> Book.objects.aggregate(
... price_diff=Max("price",
output_field=FloatField()) -
Avg("price")
... )
{'price_diff': 46.85}
# All the following queries
involve traversing the Book<-
>Publisher
# foreign key relationship
backwards.
# Each publisher, each with a
count of books as a "num_books"
attribute.
>>> from django.db.models import
Count
>>> pubs =
Publisher.objects.annotate(num_bo
oks=Count("book"))
>>> pubs
<QuerySet [<Publisher:
BaloneyPress>, <Publisher:
SalamiPress>, ...]>
>>> pubs[0].num_books
73
```

```
# Each publisher, with a separate
count of books with a rating
above and below 5
>>> from django.db.models import
Q
>>> above_5 = Count("book",
filter=Q(book__rating__gt=5))
>>> below_5 = Count("book",
filter=Q(book__rating__lte=5))
>>> pubs =
Publisher.objects.annotate(below_
5=below_5).annotate(above_5=above
_5)
>>> pubs[0].above_5
23
>>> pubs[0].below_5
12
# The top 5 publishers, in order
by number of books.
>>> pubs =
Publisher.objects.annotate(num_bo
oks=Count("book")).order_by("-
num_books")[:5]
>>> pubs[0].num_books
1323
```

Generating aggregates over a QuerySet

Django provides two ways to generate aggregates. The first way is to generate summary values over an entire QuerySet. For example, say you wanted to calculate the average price of all books available for sale.
Django's query syntax provides a means for describing the set of all books:

```
>>> Book.objects.all()
```

What we need is a way to calculate summary values over the objects that belong to this QuerySet. This is done by appending an aggregate() clause onto the QuerySet:

```
>>> from django.db.models
import Avg
>>>
Book.objects.all().aggregate(
Avg("price"))
{'price__avg': 34.35}
```

The all() is redundant in this example, so this could be simplified to:

```
>>>
Book.objects.aggregate(Avg("p
rice"))
{'price__avg': 34.35}
```

The argument to the aggregate() clause describes the aggregate value that we want to compute - in this case, the average of the price field on the Book model. A list of the aggregate functions that are available can be found in the QuerySet reference.
aggregate() 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. 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:

```
>>>
Book.objects.aggregate(averag
e_price=Avg("price"))
{'average_price': 34.35}
```

If you want to generate more than one aggregate, you add another argument to the aggregate() clause. So, if we also wanted to know the maximum and minimum price of all books, we would issue the query:

```
>>> from django.db.models
import Avg, Max, Min
>>>
Book.objects.aggregate(Avg("p
rice"), Max("price"),
Min("price"))
{'price__avg': 34.35,
'price__max':
Decimal('81.20'),
'price__min':
Decimal('12.99')}
```

Generating aggregates for each item in a QuerySet

The second way to generate summary values is to generate an independent summary for each object in a QuerySet. For example, if you are retrieving a list of books, you may want to know how many authors contributed to each book. Each Book has a many-to-many relationship with the Author; we want to summarize this relationship for each book in the QuerySet.

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 syntax for these annotations is identical to that used for the aggregate() clause. Each argument to annotate() describes an aggregate that is to be calculated. For example, to annotate books with the number of authors:

```
# Build an annotated
queryset
>>> from django.db.models
import Count
>>> q =
Book.objects.annotate(Count(
"authors"))
# Interrogate the first
object in the queryset
>>> q[0]
<Book: The Definitive Guide
to Django>
>>> q[0].authors__count
2
# Interrogate the second
object in the queryset
>>> q[1]
<Book: Practical Django
Projects>
>>> q[1].authors__count
1
```

As with aggregate(), the name for the annotation is automatically derived from the name of the aggregate function and the name of the field being aggregated. You can override this default name by providing an alias when you specify the annotation:

```
>>> q =
Book.objects.annotate(num_au
thors=Count("authors"))
>>> q[0].num_authors
2
>>> q[1].num_authors
1
```

Unlike aggregate(), annotate() is not a terminal clause. 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().

Combining multiple aggregations

Combining multiple aggregations with annotate() will yield the wrong results because joins are used instead of subqueries:

```
>>> book = Book.objects.first()
>>> book.authors.count()
2
>>> book.store_set.count()
3
>>> q = Book.objects.annotate(Count("authors"),
Count("store"))
>>> q[0].authors__count
6
>>> q[0].store__count
6
```

For most aggregates, there is no way to avoid this problem, however, the Count aggregate has a distinct parameter that may help:

```
>>> q = Book.objects.annotate(
... Count("authors", distinct=True),
Count("store", distinct=True)
... )
>>> q[0].authors__count
2
>>> q[0].store__count
3
```

Joins and aggregates

So far, we have dealt with aggregates over fields that belong to the model being queried. However, sometimes the value you want to aggregate will belong to a model that is related to the model you are querying. When specifying the field to be aggregated in an aggregate function, Django will allow you to use the same double underscore notation that is used when referring to related fields in filters. Django will then handle any table joins that are required to retrieve and aggregate the related value. For example, to find the price range of books offered in each store, you could use the annotation:

```
>>> from django.db.models import Max, Min
>>>Store.objects.annotate(min_price=Min("books__pr
ice"), max_price=Max("books__price"))
```

This tells Django to retrieve the Store model, join (through the many-to-many relationship) with the Book model, and aggregate on the price field of the book model to produce a minimum and maximum value. The same rules apply to the aggregate() clause. If you wanted to know the lowest and highest price of any book that is available for sale in any of the stores, you could use the aggregate:

```
>>>Store.objects.aggregate(min_price=Min("books__p
rice"), max_price=Max("books__price"))
```

Join chains can be as deep as you require. For example, to extract the age of the youngest author of any book available for sale, you could issue the query:

```
>>>Store.objects.aggregate(youngest_age=Min("books
__authors__age"))
```

Following relationships backwards

In a way similar to Lookups that span relationships, aggregations and annotations on fields of models or models that are related to the one you are querying can include traversing “reverse” relationships. The lowercase name of related models and double-underscores are used here too.

For example, we can ask for all publishers, annotated with their respective total book stock counters (note how we use 'book' to specify the Publisher -> Book reverse foreign key hop):

```
>>> from django.db.models import Avg, Count, Min, Sum
>>> Publisher.objects.annotate(Count("book"))
```

(Every Publisher in the resulting QuerySet will have an extra attribute called book__count.)

We can also ask for the oldest book of any of those managed by every publisher:

```
>>> Publisher.objects.aggregate(oldest_pubdate=Min("book__pubdate"))
```

(The resulting dictionary will have a key called 'oldest_pubdate'. If no such alias were specified, it would be the rather long 'book__pubdate__min'.)

This doesn’t apply just to foreign keys. It also works with many-to-many relations. For example, we can ask for every author, annotated with the total number of pages considering all the books the author has (co-)authored (note how we use 'book' to specify the Author -> Book reverse many-to-many hop):

```
>>> Author.objects.annotate(total_pages=Sum("book__pages"))
```

(Every Author in the resulting QuerySet will have an extra attribute called total_pages. If no such alias were specified, it would be the rather long book__pages__sum.)

Or ask for the average rating of all the books written by author(s) we have on file:

```
>>>Author.objects.aggregate(average_rating=Avg("book__rating"))
```

(The resulting dictionary will have a key called 'average_rating'. If no such alias were specified, it would be the rather long 'book__rating__avg'.)

Aggregations and other QuerySet clauses

filter() and exclude()

Aggregates can also participate in filters. Any filter() (or exclude()) applied to normal model fields will have the effect of constraining the objects that are considered for aggregation.

When used with an annotate() clause, a filter has the effect of constraining the objects for which an annotation is calculated. For example, you can generate an annotated list of all books that have a title starting with "Django" using the query:

```
>>> from django.db.models import Avg, Count
>>>Book.objects.filter(name__startswith="Django").annotate(num_authors=Count("author
s"))
```

When used with an aggregate() clause, a filter has the effect of constraining the objects over which the aggregate is calculated. For example, you can generate the average price of all books with a title that starts with "Django" using the query:

```
>>>Book.objects.filter(name__startswith="Django").aggregate(Avg("price"))
```

Filtering on annotations

Annotated values can also be filtered. The alias for the annotation can be used in filter() and exclude() clauses in the same way as any other model field.

For example, to generate a list of books that have more than one author, you can issue the query:

```
>>> Book.objects.annotate(num_authors=Count("authors")).filter(num_authors__gt=1)
```

This query generates an annotated result set, and then generates a filter based upon that annotation.

If you need two annotations with two separate filters you can use the filter argument with any aggregate.

For example, to generate a list of authors with a count of highly rated books:

```
>>> highly_rated = Count("book", filter=Q(book__rating__gte=7))
>>> Author.objects.annotate(num_books=Count("book"),
highly_rated_books=highly_rated)
```

Each Author in the result set will have the num_books and highly_rated_books attributes.

Order of annotate() and filter() clauses

When developing a complex query that involves both annotate() and filter() clauses, pay particular attention to the order in which the clauses are applied to the QuerySet.
When an annotate() clause is applied to a query, the annotation is computed over the state of the query up to the point where the annotation is requested. The practical implication of this is that filter() and annotate() are not commutative operations.
Given:

- Publisher A has two books with ratings 4 and 5.
- Publisher B has two books with ratings 1 and 4.
- Publisher C has one book with rating 1.

Here's an example with the Count aggregate:

```
>>> a, b =
Publisher.objects.annotate(num_books=Count("
book", distinct=True)).filter(
... book__rating__gt=3.0
... )
>>> a, a.num_books
(<Publisher: A>, 2)
>>> b, b.num_books
(<Publisher: B>, 2)
>>> a, b =
Publisher.objects.filter(book__rating__gt=3.0).a
nnotate(num_books=Count("book"))
>>> a, a.num_books
(<Publisher: A>, 2)
>>> b, b.num_books
(<Publisher: B>, 1)
```

Both queries return a list of publishers that have at least one book with a rating exceeding 3.0, hence publisher C is excluded.
In the first query, the annotation precedes the filter, so the filter has no effect on the annotation.
distinct=True is required to avoid a query bug.
The second query counts the number of books that have a rating exceeding 3.0 for each publisher. The filter precedes the annotation, so the filter constrains the objects considered when calculating the annotation.
Here's another example with the Avg aggregate:

```
>>> a, b =
Publisher.objects.annotate(avg_rating=Avg("bo
ok__rating")).filter(
... book__rating__gt=3.0
... )
>>> a, a.avg_rating
(<Publisher: A>, 4.5) # (5+4)/2
>>> b, b.avg_rating
(<Publisher: B>, 2.5) # (1+4)/2
>>> a, b =
Publisher.objects.filter(book__rating__gt=3.0).a
nnotate(
... avg_rating=Avg("book__rating")
... )
>>> a, a.avg_rating
(<Publisher: A>, 4.5) # (5+4)/2
>>> b, b.avg_rating
(<Publisher: B>, 4.0) # 4/1 (book with rating 1
excluded)
```

The first query asks for the average rating of all a publisher's books for publisher's that have at least one book with a rating exceeding 3.0. The second query asks for the average of a publisher's book's ratings for only those ratings exceeding 3.0.
It's difficult to intuit how the ORM will translate complex querysets into SQL queries so when in doubt, inspect the SQL with str(queryset.query) and write plenty of tests.

order_by()

Annotations can be used as a basis for ordering. When you define an order_by() clause, the aggregates you provide can reference any alias defined as part of an annotate() clause in the query. For example, to order a QuerySet of books by the number of authors that have contributed to the book, you could use the following query:

```
>>> Book.objects.annotate(num_authors=Count("authors")).order_by("num_authors")
```

values()

Ordinarily, annotations are generated on a per-object basis - an annotated QuerySet will return one result for each object in the original QuerySet. However, when a values() clause is used to constrain the columns that are returned in the result set, the method for evaluating annotations is slightly different. Instead of returning an annotated result for each result in the original QuerySet, the original results are grouped according to the unique combinations of the fields specified in the values() clause. An annotation is then provided for each unique group; the annotation is computed over all members of the group. For example, consider an author query that attempts to find out the average rating of books written by each author:

```
>>> Author.objects.annotate(average_rating=Avg("book__rating"))
```

This will return one result for each author in the database, annotated with their average book rating.
However, the result will be slightly different if you use a values() clause:

```
>>> Author.objects.values("name").annotate(average_rating=Avg("book__rating"))
```

In this example, the authors will be grouped by name, so you will only get an annotated result for each unique author name. This means if you have two authors with the same name, their results will be merged into a single result in the output of the query; the average will be computed as the average over the books written by both authors.

Order of annotate() and values() clauses

As with the filter() clause, the order in which annotate() and values() clauses are applied to a query is significant. If the values() clause precedes the annotate(), the annotation will be computed using the grouping described by the values() clause. However, if the annotate() clause precedes the values() clause, the annotations will be generated over the entire query set. In this case, the values() clause only constrains the fields that are generated on output. For example, if we reverse the order of the values() and annotate() clause from our previous example:

```
>>> Author.objects.annotate(average_rating=Avg("book__rating")).values(
...  "name", "average_rating"
... )
```

This will now yield one unique result for each author; however, only the author's name and the average_rating annotation will be returned in the output data. You should also note that average_rating has been explicitly included in the list of values to be returned. This is required because of the ordering of the values() and annotate() clause. If the values() clause precedes the annotate() clause, any annotations will be automatically added to the result set. However, if the values() clause is applied after the annotate() clause, you need to explicitly include the aggregate column.

Interaction with order_by()

Fields that are mentioned in the order_by() part of a queryset are used when selecting the output data, even if they are not otherwise specified in the values() call. These extra fields are used to group “like” results together and they can make otherwise identical result rows appear to be separate. This shows up, particularly, when counting things.

By way of example, suppose you have a model like this:

```
from django.db import
models
class Item(models.Model):
name =
models.CharField(max_leng
th=10)
data =
models.IntegerField()
```

If you want to count how many times each distinct data value appears in an ordered queryset, you might try this:

```
items =
Item.objects.order_by("na
me")
# Warning: not quite
correct!
items.values("data").anno
tate(Count("id"))
```

. . .which will group the Item objects by their common data values and then count the number of id values in each group. Except that it won’t quite work. The ordering by name will also play a part in the grouping, so this query will group by distinct (data, name) pairs, which isn’t what you want. Instead, you should construct this queryset:

```
items.values("data").anno
tate(Count("id")).order_b
y()
```

. . .clearing any ordering in the query. You could also order by, say, data without any harmful effects, since that is already playing a role in the query.

This behavior is the same as that noted in the queryset documentation for distinct() and the general rule is the same: normally you won’t want extra columns playing a part in the result, so clear out the ordering, or at least make sure it’s restricted only to those fields you also select in a values() call

Aggregating annotations

You can also generate an aggregate on the result of an annotation. When you define an aggregate() clause, the aggregates you provide can reference any alias defined as part of an annotate() clause in the query.
For example, if you wanted to calculate the average number of authors per book you first annotate the set of books with the author count, then aggregate that author count, referencing the annotation field:

```
>>> from django.db.models import Avg, Count
>>>Book.objects.annotate(num_authors=Count("authors")).aggregate(Avg("num_authors")
) #output {'num_authors__avg': 1.66}
```

Aggregating on empty querysets or groups

When an aggregation is applied to an empty queryset or grouping, the result defaults to its default parameter, typically None. This behavior occurs because aggregate functions return NULL when the executed query returns no rows. You can specify a return value by providing the default argument for most aggregations. However, since Count does not support the default argument, it will always return 0 for empty querysets or groups. For example, assuming that no book contains web in its name, calculating the total price for this book set would return None since there are no matching rows to compute the Sum aggregation on:

```
>>> from django.db.models import Sum
>>> Book.objects.filter(name__contains="web").aggregate(Sum("price"))
{"price__sum": None}
However, the default argument can be set when calling Sum to return a different default value
if no books
can be found:
>>> Book.objects.filter(name__contains="web").aggregate(Sum("price", default=0))
{"price__sum": Decimal("0")}
```

Under the hood, the default argument is implemented by wrapping the aggregate function with Coalesce.