Django

DB's, Webapps, and API's - Oh My!

Dr. Scott Gorlin

Harvard University

Fall 2018

Agenda

- Mutability
- Living Data
- Django
- Django
- Django
- ORM
- Migrations
- Universal Django
- Django Code
- Readings

Mutability

Mutable Objects

Mutable

```
[]
{}
set()
d = Task()
d.a = 134  # The instance itself is mutable
```

Immutable

```
a = (1, 2)  # Tuple
>>> a[0] = 1
TypeError: 'tuple' object does not support item
assignment
>>> a.x = 1
AttributeError: 'tuple' object has no attribute 'x'
frozenset
namedtuple
# frozendict
```

Mutable Objects

Mutable

```
>>> a = [1, 2]
>>> o = a
>>> o is a
True
>>> a += [3]
>>> a
[1, 2, 3]
>>> o is a
True
```

Immutable

```
>>> a = (1, 2)
>>> o = a
>>> a += (3,)
>>> a
(1, 2, 3)
>>> o is a
False  # Same if a is an int
```

The += operator assigns a new tuple object to the label a, as opposed to modifying the list labelled a in place.

Mutable Objects

Some special treatment in python:

```
>>> s = set()
>>> (1,) in s
>>> [1] in s
TypeError: unhashable type: 'list'
```

This is a different use case for a hash. It's not a sha256. We may explore later.

Mutability vs State

Mutable

An object/system which ***can change*** (even if it never does)

Immutable objects can be more performant

Stateful

An object/system which ***has a state***, which may be mutable or not

State can impact execution, even if the state is immutable

> Stateless implies immutable!

Immutable Workflows

A Luigi workflow implies ***immutable artifacts***

Immutable data output is a Very Good Thing

It provides reproducibility and repeatability

Never ‘just edit the input’. You need a makefile, or ..

You can, of course, delete outputs

Salted Graphs provide a compromise: keep the immutable workflow, change version/target when output is expected to be different

Deterministic Workflows

Immutable data, version control, and data lineage (Salted) implies

100% deterministic output

If we have the code and the input data, we will get the same results every time

Unless, of course, you use a random number

Living Data

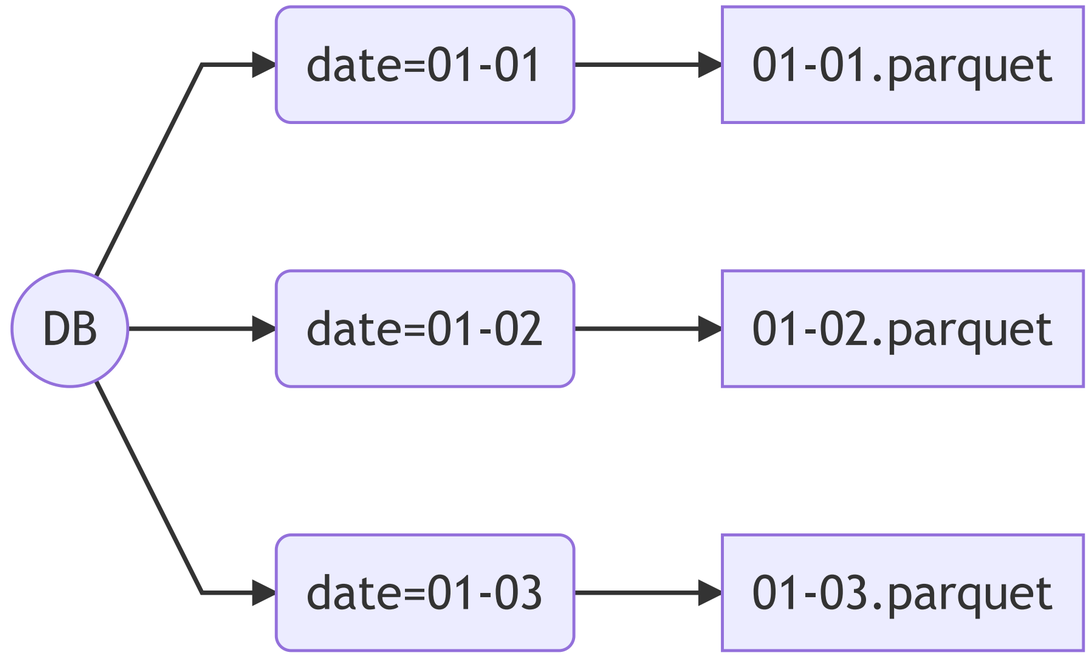
Not every workflow is deterministic

Weak Mutability

… eg streaming data

```
Oct 30 19:53:27 dnsmasq-dhcp[347]: DHCPREQUEST(br0)
192.168.1.224 a0:4e:a7:8a:91:72
Oct 30 19:53:27 dnsmasq-dhcp[347]: DHCPACK(br0)
192.168.1.224 a0:4e:a7:8a:91:72 iPad
Oct 30 19:53:27 dnsmasq-dhcp[347]: DHCPREQUEST(br0)
192.168.1.224 a0:4e:a7:8a:91:72
Oct 30 19:53:27 dnsmasq-dhcp[347]: DHCPACK(br0)
192.168.1.224 a0:4e:a7:8a:91:72 iPad
```

… can be solved with mini-batch



Run for a historical time interval (‘yesterday’), finished partitions, etc

Not every workflow is deterministic

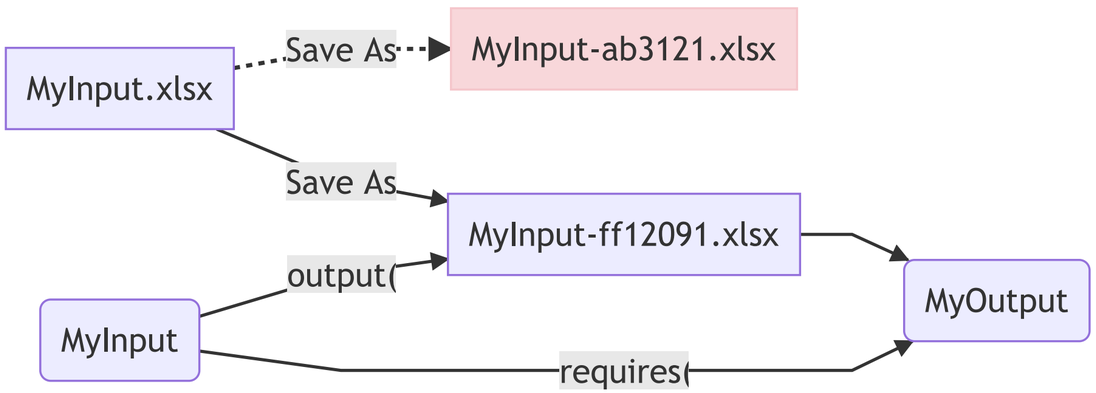
Weak Mutability

… eg a small input .xlsx

```
class MyInput(ExternalTask):
    __version__ = 'fixed_data'
    output = SaltedLocalOutput(ext='.xlsx')

class MyOutput(Task):
    my_input = Requirement(MyInput)
```

… can be solved by manually copying to a salted target!



Take snapshots of the input file, then rerun

Not every workflow is deterministic

Strong Mutability

… eg, most “up to date” databases

Snapshots

A normal DB:

name	grade
Scott	B

Append-only:

name	grade	tstamp
Scott	B	1:35
Scott	B-	2:05

Which can be queried:

```
select s.name, s.grade
from (
   select name,
   max(tstamp) as latest
   from students
   group by name
) as x
inner join students as s
on s.name = x.name
 and s.tstamp = x.latest;
```

Snapshots

A normal DB:

name	grade
Scott	B

Append-only:

name	grade	tstamp
Scott	B	1:35
Scott	B-	2:05

Deterministic query:

```
select s.name, s.grade
from (
   select name,
   max(tstamp) as latest
   from students
   WHERE tstamp < QTIME
   group by name
) as x
inner join students as s
on s.name = x.name
 and s.tstamp = x.latest;
```

Snapshots

Deterministic query time is not now()!

Consider an incremental backoff depending on when you expected data to have changed

```
from datetime import datetime, timedelta

def get_qtime(interval_end, asof=None):
    asof = asof or datetime.utcnow()
    if asof < interval_end:
        raise ValueError('Be patient!')
    delta = asof - interval_end
```

```
    if delta.days > 7:
        return (
            asof - timedelta(
                days=asof.isoweekday)
            ).date()
    elif delta.days:
        return asof.date()
    return interval_end + timedelta(
        hours=floor(delta.total_seconds/3600))
```

Custom Salts

```python
from abc import ABC, abstractmethod

class SaltableTask(ABC, Task):

    @abstractmethod
    def get_salted_signature(self):
        """Provide a custom salt"""
        ...
```

```python
    @classmethod
    def __subclasshook__(cls, C):
        # Don't force inheritance
        # just check for method
        return hasattr(C,
            'get_salted_signature')
```

We can - and should - allow tasks to create their own salts

```python
def get_salted_version(task):
    msg = ... # Get reqs/salts

    # Allow override
    if isinstance(task, SaltableTask):
        msg += task.get_salted_signature()
    else:
        msg += task.__class__.__name__ ...
```

Custom Salts

Simple example - add new class param w/o changing salt

```
class MyTask(SaltableTask):
    # This used to be hard coded
    # but now is a parameter
    param = Parameter(default=3)

    def get_salted_signature(self):
        all_params = {...} # Significant params
        if self.param == MyTask.param.default:
            # Remove from hashable config
            del all_params['param']
        return ','.join([
            '{}={}'.format(k, all_params[k])
            for k in sorted(all_params)
        ])
```

Data-Driven Salts

Base signature on query time… data will automatically refresh!

```
class MyTask(SaltableTask):
    date = DateParameter()

    def get_qtime(self):
        return get_qtime(self.date+timedelta(days=1))

    def get_salted_signature(self):
        qtime = self.get_qtime()  # and use in signature!
        ...

    def run(self):
        qtime = self.get_qtime()
        target = self.output()  # Don't wait :)
        ...
```

Data-Driven Salts

Use the actual data to sign the output!!

Many big data stores (HBase) store data versions like this automatically.

Use this paradigm to kick off Big Data pipelines only when the data has changed!

```
class MyTask(SaltableTask):

    def query_max_time(self):
        # Cache or persist this result if it's slow
        'select max(tstamp) from students'
        ' where tstamp < qtime'
        ...

    def get_salted_signature(self):
        # Now, the signature only changes
        # IFF there is new data!!
        last_update = self.query_max_time()
        ...
```

```
    def run(self):
        # Still need to filter to guarantee
        # deterministic output
        qtime = self.query_max_time()
        ...
```

Beware!

It feels like we may be working too hard! If we have live data, maybe we want live (and eventually consistent) outputs

We don’t necessarily need a deterministic workflow or skeleton. Time to explore another paradigm!

Django

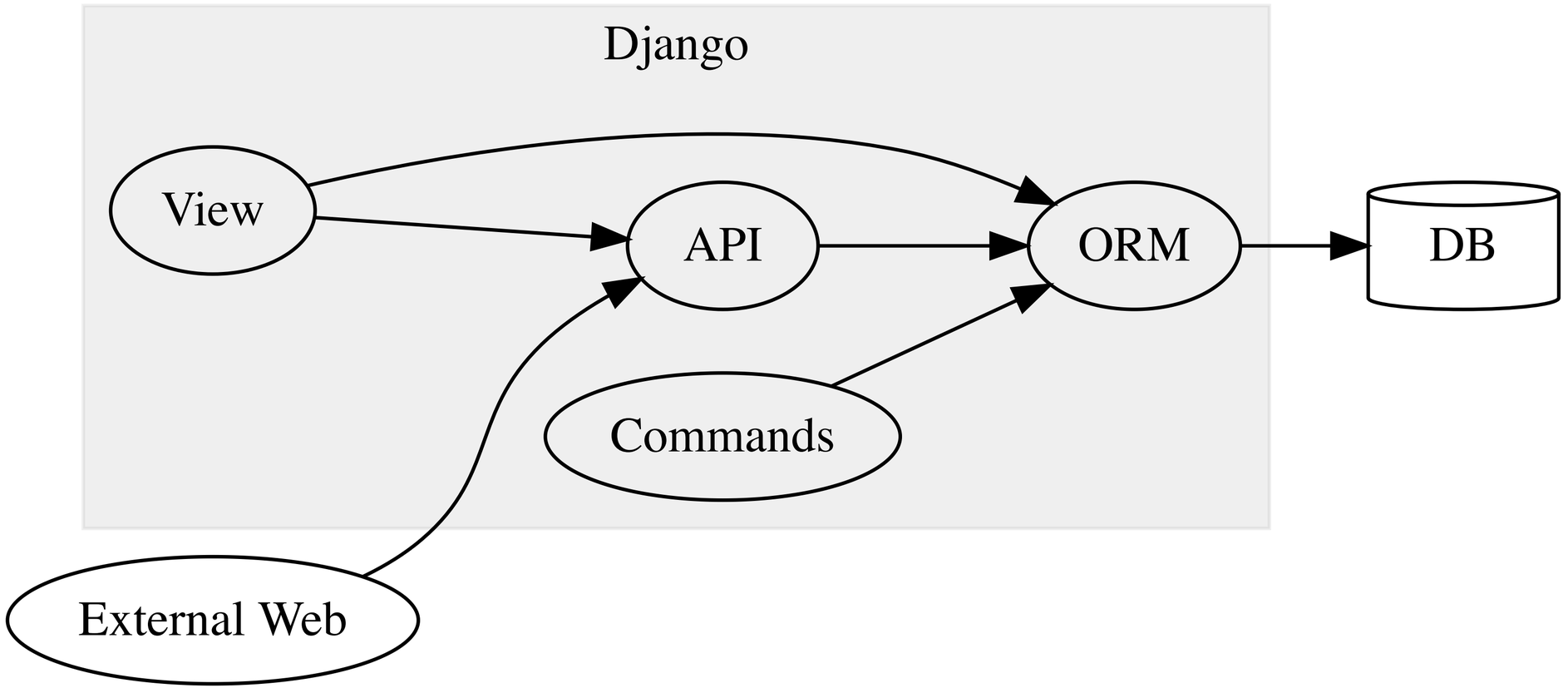
DJANGO

DJANGO

A Framework

> *The web framework for perfectionists with deadlines.*
>
> — *Django Project*

A Framework



Django is ***an entire application framework*** for webapps and more

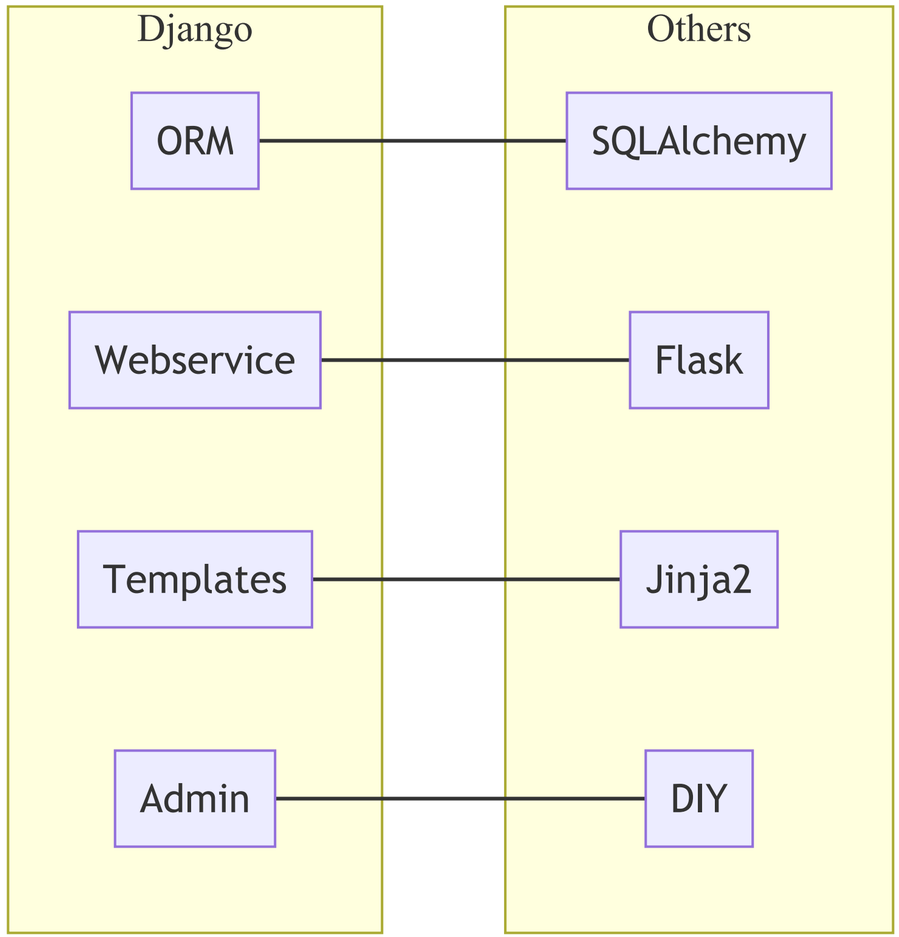
Full Stack Django

Django is ***not*** best-of-breed.
It is, however, a fantastic all-in-one.

It optimizes ***practicality***.

It is ***opinionated***.

It is ***likely*** good enough for you.



When you need Django

Web App

Easy, awesome, interactivity and presentability

Hosted/shareable

Deploy your analytics for non-techies to run

Your DB

You want to store and track live, mutable data

Microservice

You want to provide value to another team via exposing your work via an API

ORM

Two Types of Data Use

Analytics

I care about summary stats:

```
select is_grad, mean(grade)
from students
```

and in general care about specific columns, aggregations, etc

Objects

I care about the rows:

```
select *
from students
where grade > 75
```

and in general want all of the data per row, and often save new rows

Two Types of Data Use

Analytics - Data Mapper

```
class Student(Table):
    # table/column metadata
    name = StringColumn()
    is_grad = BoolColumn()
    grade = FloatColumn()

select([
    Student.is_grad,
    mean(Student.grade)
]).group_by(Student.is_grad)
```

Query results in arbitrary output w/ `is_grad` and `grade` columns

Objects - Active Record

```
select([
    Student
]).where(Student.grade > 75)

student = Student.fetch(name='Scott')
student.grade += 5
student.save()
```

Query results are instances of `Student`

NB: this is pseudocode

The Python <-> DB Layer

SQL

```
SELECT users.id, users.name
FROM users
```

Most flexibility, but is DB dependent, and not scriptable

Expression Language

```
from sqlalchemy import *

users = Table(
    'users', MetaData(),
    Column('id', Integer),
    Column('name', String))

>>> select([users]).first()
(1, u'jack')
```

DB agnostic, flexible, debuggable, scriptable

ORM

```
from sqlalchemy import *

class User(DeclarativeBase):
    __tablename__ = 'users'
    id = Column(Integer)
    name = Column(String)

our_user = session.query(
    User
).filter_by(name='ed').first()
>>> our_user
<User(name='ed')>
```

Opinionated, but valuable

Django ORM

ORM: Object Relational Mapping

Table -> Class, Row -> Instance

Django ***only*** provides an ORM; it allows raw SQL and some pythonic expressions, but not as first class citizens

```
from django.db.models import *

class Question(Model):
    question_text = CharField(max_length=200)
    pub_date = DateTimeField('date published')

class Choice(Model):
    question = ForeignKey(Question)
    choice_text = CharField(max_length=200)
    votes = IntegerField(default=0)
```

Django ORM

Basic stats are easy!

Though the syntax is bizarre for noobs, it is internally consistent and easy to master

```
# The manager/query 'objects'
Choice.objects.order_by('-votes')[0]
Choice.objects.values(
    'question'
    ).annotate(
        total_votes=Sum('votes')
    )  # Note this will not be a Choice
```

`Table.values(x).annotate(y=f(...))` is Django idiom for `select x, f(...) as y from table group by x`

Django ORM

Creating the database is ***handled*** ***magically*** through migrations

```
# Inspects the models.py
# and create a **migration plan**
# No change to DB!
$ python manage.py makemigrations polls
```

```
# Run the migration!
$ python manage.py migrate polls
```

Migrations

Migrations

Because your DB is live, you need to upgrade it as the schema changes

You may want to add or delete columns, change data types, create new foreign keys, …

The ORM Model is what would be created from scratch given your ***current*** code

A ***migration*** freezes the current model definition and commits it to the codebase. It then generates the steps to upgrade an ***existing*** DB from the previous migration snapshot.

Migrations

Schema Migration

- Add table
- Change column
  - Float -> Int
  - Non-null
  - New default value
- Add column

Data Migration

- Change existing column values
- Initial table data ???

Migrations

Usually, Django can auto generate the migration for you. It's awesome.

Note the DAG and declarative syntax!

```
from django.db import migrations
from django.db.models import IntegerField

class Migration(migrations.Migration):

    dependencies = [
        ('migrations', '0001_initial')
    ]

    operations = [
        migrations.DeleteModel('Tribble'),
        migrations.AddField(
            'Author', 'rating', IntegerField(default=0)),
    ]
```

Live Migrations

Consider live migrations vs 'downtime'

Can all code safely read/write at any point?

Shut it down

1. Stop all code
2. Add a new column
3. Populate the column
4. Deploy new code expecting new column

Live, rolling

1. Add new nullable column
2. Write to column for new rows
3. Backfill/finish deploy
4. Make column non-nullable, and use the value

Universal Django

ORM as Translation

Django effectively translates across many DB dialects

Django:

```
Student.objects.filter(name='Scott')
```

MySQL:

```
select * from students where name = "Scott"
```

Postgres:

```
select * from students where name = 'Scott'
```

SQL has an ANSI compliance standard, so code should work everywhere. However, it doesn't!

ORM as Translation

But I only use one database. Why do I need translation?

False! Unittests!

Django will create in-memory SQLite DB's for your tests!

You can, of course, run the same tests on a 'real' DB with configuration

```python
from django.test import TestCase as DJTest
from myapp.models import Animal

class AnimalTestCase(DJTestCase):
    def setUp(self):
        Animal.objects.create(name="lion", sound="roar")
        Animal.objects.create(name="cat", sound="meow")

    def test_animals_can_speak(self):
        """Animals that can speak are correctly identified"""
        lion = Animal.objects.get(name="lion")
        cat = Animal.objects.get(name="cat")
        self.assertEqual(lion.speak(), 'The lion says "roar"')
        self.assertEqual(cat.speak(), 'The cat says "meow"')
```

ORM AS LCD

As a ***universal*** ORM, Django targets the Lowest Common Denominator of major databases

It will avoid officially supporting DB's that don't implement most of its standard features

Django ***does*** expose some DB-specific features, eg via `django.contrib.postgres` and GeoDjango. Using them implies more work on the testing and adoption side, since things are no longer DB agnostic.

ORM as Translation

Unittests and migrations are a great way to test your app on new DB's, eg when moving from MySQL to Postgres.

Common DB's

SQLite

- local DB
- In memory or file
- Fine for basics

MySQL

- good overall

Postgres

- Good default
- Extra features
- Atomic migrations

Oracle, MSSQL

- yuck
- Someone tells you to use

Redshift

- columnar store!
- Big data
- Unofficial support

NoSQL

- Um… kind of?
- Possible, but mismatched

Django Code

Inheritable Models

Models must inherit from Django classes to ensure the tables are created, migrateable, etc

Inheritance usually means a new table. Make sure that's what you want.

Model Inheritance

```
from django.db.models import *

class Place(Model):
    name = CharField(max_length=50)
    address = CharField(max_length=80)

class Restaurant(Place):
    serves_hot_dogs = BooleanField(default=False)
    serves_pizza = BooleanField(default=False)
```

The `Restaurant` table FK's to `Place`!

Composable Models

Django models make great use of composability

```
class Place(Model):
    name = CharField(max_length=50)

p = Place(name='Harvard')
Place.objects.filter(name='Harvard')
```

The `Field` classes handle:

- Schema creation
- Instantiation
- Migration
- Serialization
- Querying…

Composable Models

`Manager` objects: composable ‘views’

```python
# First, define the Manager subclass.
class DahlBookManager(Manager):
    def get_queryset(self):
        return super().get_queryset().filter(author='Roald Dahl')

# Then hook it into the Book model explicitly.
class Book(Model):
    title = CharField(max_length=100)
    author = CharField(max_length=50)

    objects = Manager() # The default manager.
    dahl_objects = DahlBookManager() # The Dahl-specific manager.
```

The Point

If you know little to nothing about proper DB creation and design, Django is your new best friend. It will do everything Mostly Right.

You can still do cool columnar analytics with persisted data in Dask and Luigi. Use Django for your webapps and small row-oriented data.

Even if you don't need a DB, Django `Model` classes are an excellent case study in modern programming design

… even for their warts

ORM Breakdowns

```python
class Timespan(Model):
    start = DateTimeField()
    end = DateTimeField()

    @property
    def duration(self):
        # Python side implementation
        return self.end - self.start
```

```python
# SQL-side implementation
# NB: F('field') is a field lookup
Timespan.objects.annotate(
    duration=F('end') - F('start'))
```

Putting logic in the ORM layer risks violating DRY

Other ORM implementations may handle this better

The Competition

Django

ORM designed to make 80% of your work 100% better

Universal (LCD)

Difficult or impossible for complex work without using raw SQL

Largely considered Good Enough

SQLAlchemy

ORM layer explicitly wraps SQL

Directly targets DB features, while translating across dialects

Expressions: can avoid the ORM entirely

Largely considered superior.
Only sane choice for analytics.

Eg, joining two arbitrary tables!

Readings

- Why I Hate the Django ORM
- SQLAlchemy and You
- The Vietnam of Computer Science
- Django vs SQLAlchemy
- Django Tutorials