The Model Layer

Web Applications and Services

Spring Term

Naercio Magaia

UNIVERSITY
OF SUSSEX

Contents

- Models
- Fields
- Model class
- Making queries
- Migrations

UNIVERSITY OF SUSSEX

Models

- A Model is the single, definitive source of information about the data
- It contains the essential fields and behaviours of the data being stored
- Each model
  - maps to a single database table
  - Is a Python class that subclasses `django.db.models.Model`
  - Attribute represents a database field
- Django provides an automatically generated database-access API

UNIVERSITY OF SUSSEX

A model example

- Let us create a model for the `myapp` module that defines a `Person` having a `first_name` and a `last_name`

```
from django.db import models

class Person(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)
```

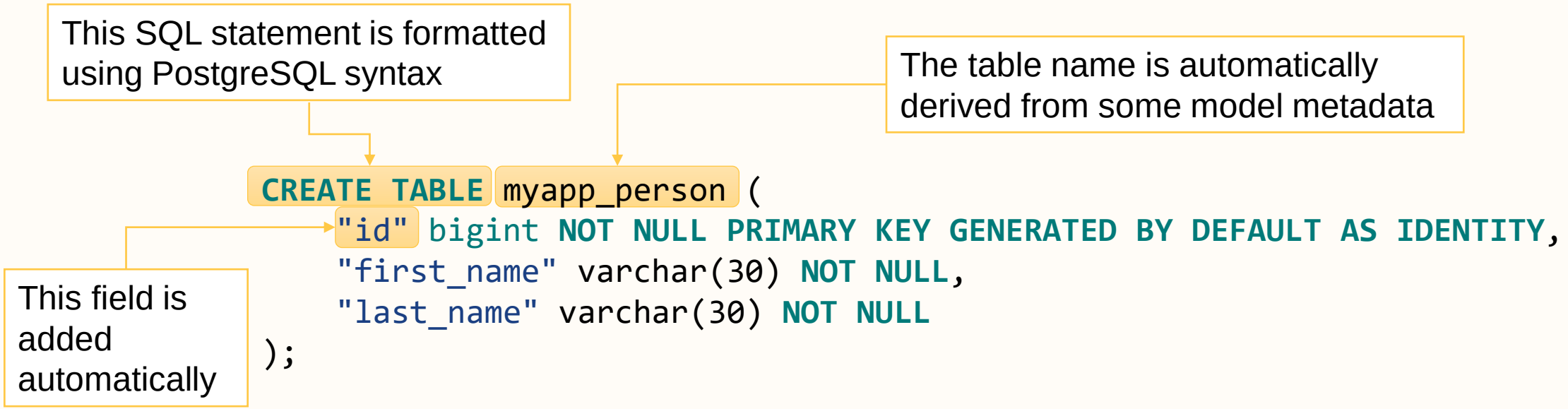
Field attributes

- Each field is specified as a class attribute, and each attribute maps to a database column.

UNIVERSITY OF SUSSEX

A model example

- The Person model would create the following database table



- Django uses SQL tailored to the database backend specified in your settings file

UNIVERSITY OF SUSSEX

Using models

- Once the model has been created, it is necessary to tell Django that we intend to use it
  - Edit the settings file and the module with the *models.py* file to the INSTALLED_APPS
- For the previous example
  - The models are in the module `myapp.models`
  - The package structure was created by the `managy.py startapp myapp` script

```
INSTALLED_APPS = [
    #...
    'myapp',
    #...
]
```

- When the new app has been added to the INSTALLED_APPS
  - *optionally* first make migrations with `manage.py makemigrations`
  - run `manage.py migrate`

UNIVERSITY OF SUSSEX

Fields

- The *only* required part of a model is the list of **database fields** it defines
- Fields are specified by class attributes
  - Avoid choosing field name that conflict with the model API such as `clean`, `save`, or `delete`

```
from django.db import models

class Musician(models.Model):
    name = models.CharField(max_length=100)
    instrument = models.CharField(max_length=100)

class Album(models.Model):
    artist = models.ForeignKey(Musician, on_delete=models.CASCADE)
    title = models.CharField(max_length=100)
    release_date = models.DateField()
    num_stars = models.IntegerField()
```

Field types

- Each field in a model is an instance of the appropriate Field class.
- Django uses the field class types to determine
  - The column type, which tells the database what kind of data to store
    - For example: `INTEGER`, `VARCHAR`, `TEXT`
  - The default HTML widget to use when rendering a form field
    - For example: `<input type="text">`, `<select>`
  - The minimal validation requirements, used in Django's admin and in automatically-generated forms.
- Django comes with some built-in field types
  - It is possible to write new ones

UNIVERSITY OF SUSSEX

Field options

- Each field takes a certain set of field-specific arguments
  - The `CharField` (and its subclasses) require a `max_length` argument which specifies the size of the `VARCHAR` database field used to store the data.
- There are also some optional common arguments available to all field types
  - `null`: if True, Django will store empty values as `NULL` in the database.
  - `blank`: if True, the field is allowed to be blank.
  - `choices`: a sequence of 2-tuples to use as choices for this field.
    - A new migration is created each time the order of choices changes.
  - `default`: the default value for the field
  - `primary_key`: If True, this field is the primary key for the model.
    - Django will automatically add an `IntegerField` to hold the primary key, if not specified

```
BACHELOR_DEGREE = [
    ('BA', 'Bachelor of Arts'),
    ('BSc', 'Bachelor of Science'),
    ('BEd', 'Bachelor of Education'),
    ('BEng', 'Bachelor of Engineering'),
    ('LLB', Bachelor of Laws’),
    ('MB', Bachelor of Medicine’),
]
```

UNIVERSITY OF SUSSEX

Field options

```
from django.db import models

class Student(models.Model):
    BACHELOR_DEGREE = [
        ('BA', 'Bachelor of Arts'),
        ('BSc', 'Bachelor of Science'),
        ('BEd', 'Bachelor of Education'),
        ('BEng', 'Bachelor of Engineering'),
        ('LLB', 'Bachelor of Laws'),
        ('MB', Bachelor of Medicine'),
    ]
    name = models.CharField(max_length=60)
    bachelor_degree = models.CharField(max_length=4, choices=BACHELOR_DEGREE)
```

```
>>> s = Student(name="Fred Stone", bachelor_degree="BSc")
>>> s.save()
>>> s.bachelor_degree
'BSc'
```

Fields

- Django gives each model an auto-incrementing primary key
  - the type specified per app in `AppConfig.default_auto_field` or globally in the `DEFAULT_AUTO_FIELD` setting
  - Each model **requires** exactly one field to have `primary_key=True`
- Django offers ways to define the three most common types of database relationships:
  - many-to-one,
  - many-to-many, and
  - one-to-one.

Many-to-one relationships

- Use `django.db.models.ForeignKey` to define a many-to-one relationship
  - `ForeignKey` requires a positional argument, i.e., the class to which the model is related.

```
from django.db import models

class Manufacturer(models.Model):
    # ...
    pass

class Car(models.Model):
    manufacturer = models.ForeignKey(Manufacturer, on_delete=models.CASCADE)
    # ...
```

UNIVERSITY OF SUSSEX

Many-to-many relationships

- Use `ManyToManyField` to define a many-to-many relationship
  - `ManyToManyField` requires a positional argument, i.e., the class to which the model is related.

```
from django.db import models

class Topping(models.Model):
    # ...
    pass

class Pizza(models.Model):
    # ...
    toppings = models.ManyToManyField(Topping)
```

One-to-one relationships

- Use `OneToOneField` to define a one-to-one relationship
  - `OneToOneField` requires a positional argument, i.e., the class to which the model is related.

```
from django.db import models

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

class Restaurant(models.Model):
    place = models.OneToOneField(Place, on_delete=models.CASCADE, primary_key=True)
    serves_hot_dogs = models.BooleanField(default=False)
    serves_pizza = models.BooleanField(default=False)
    # ...
```

Model class

- The `Manager` is the most important model attribute
  - It is the interface that enables database query operations to be provided to Django models
  - It is used to retrieve the instances from the database
  - the default name is `objects`
  - It is only accessible via model classes, not the model instances
- Model methods define custom methods on a model to add custom “row-level” functionality to objects
  - Differently from Manager methods, model methods should act on a particular model instance
  - This allows keeping business logic in one place, i.e., the model

UNIVERSITY OF SUSSEX

Model class

- An example of a model has a few custom methods

```
from django.db import models

class Person(models.Model):
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)
    birth_date = models.DateField()

    def is_baby_boomer(self):
        import datetime
        if self.birth_date > datetime.date(1945, 8, 1) AND self.birth_date < datetime.date(1965, 1, 1):
            return "True"
        else:
            return "False"

    @property
    def full_name(self):
        return '%s %s' % (self.first_name, self.last_name)
```

Model class

- The following methods should most of the times be defined:
  - `__str__()`
    - It returns a string representation of any object
    - This is what Python and Django will use whenever a model instance needs to be coerced and displayed as a plain string
    - It is advisable to define as the default isn't very helpful at all.
  - `get_absolute_url()`
    - It tells Django how to calculate the URL for an object.
    - Django uses it in its admin interface, and any time it needs to figure out a URL for an object.
    - Any object that has a URL that uniquely identifies it should define this method.

Model class

- Model inheritance works similar to normal class inheritance works in Python
  - the base class should subclass `django.db.models.Model`
- Relevant question
  - Are the parent models to be models in their own right (with their own database tables)?, or
  - Are parents just holders of common information that will only be visible through the child models?
- The following styles of inheritance are possible in Django
  1. Often, one will use the parent class to hold information not to be added at each child model. This class isn't going to ever be used in isolation, so *Abstract base classes* are an option.

UNIVERSITY OF SUSSEX

Model class

- The following styles of inheritance are possible in Django
  2. If an existing model is being subclassed (possibly something from another application entirely) and it is desirable for each model to have its own database table, *Multi-table inheritance* is the way to go.
  3. Finally, if the goal is only to modify the Python-level behavior of a model, without changing the models' fields in any way, using *Proxy models* is a possibility.

Abstract base classes

- They are useful when you want to put some common information into a number of other models.
  - It is necessary to put abstract=True in the Meta class
    - This model will then not be used to create any database table.

```
from django.db import models

class CommonInfo(models.Model):
    name = models.CharField(max_length=100)
    age = models.PositiveIntegerField()

    class Meta:
        abstract = True

class Student(CommonInfo):
    home_group = models.CharField(max_length=5)
```

Multi-table inheritance

- Each model
  - in the hierarchy is a model all by itself
  - corresponds to its own database table and can be queried and created individually
- The inheritance relationship introduces links between the child model and each of its parents (via an automatically-created OneToOneField)

```
from django.db import models

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

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

Proxy models

- When using multi-table inheritance, a new database table is created for each subclass of a model.
- However, sometimes the goal is only to change the Python behavior of a model, e.g., to change the default manager, or add a new method
- Proxy model inheritance enables creating a proxy for the original model
- One can create, delete and update instances of the proxy model
  - All the data will be saved as if the original (non-proxied) model was being used

Model class

- Proxy models are declared like normal models.
  - One tells Django that it's a proxy model by setting the proxy attribute of the Meta class to True.
- For example, to add a method to the Person model

```
from django.db import models

class Person(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)

class MyPerson(Person):
    class Meta:
        proxy = True

    def do_something(self):
        # ...
        pass
```

Organizing models in a package

- If there are many models, organizing them in separate files may be useful
- To do so, create a models package
  1. remove *models.py* file
  2. create a `myapp/models/` directory with an *__init__.py* file and the files to store your models
  3. import the models in the *__init__.py* file
- Consider that the `models` directory have *organic.py* and *synthetic.py*, the `myapp/models/__init__.py` should have

```
from .organic import Person
from .synthetic import Robot
```

To improve code readability, avoid using `from .models import *`

UNIVERSITY OF SUSSEX

Making queries

- Once the data models has been created, Django automatically gives a database-abstraction API that allowing to create, retrieve, update and delete objects
- Consider the following models: `Blog`, `Author`, and `Entry`.
  - A `Blog` has a name and a tag line.
  - An `Author` has a name and an email
  - An `Entry` has many fields such as: the headline, the body text, publication date, modification date, among others .
  - Each `Entry` is associated with a `Blog`.
- These concepts are represented by Python classes in the *myapp/models.py* file

Making queries

```
from datetime import date

from django.db import models

class Blog(models.Model):
    name = models.CharField(max_length=100)
    tagline = models.TextField()

class Author(models.Model):
    name = models.CharField(max_length=200)
    email = models.EmailField()

class Entry(models.Model):
    blog = models.ForeignKey(Blog, on_delete=models.CASCADE)
    headline = models.CharField(max_length=255)
    body_text = models.TextField()
    pub_date = models.DateField()
    mod_date = models.DateField(default=date.today)
    authors = models.ManyToManyField(Author)
    #...
```

Making queries

- Now, open the interactive Python shell to play with the free API provided by Django using the command

```
...\> py manage.py shell
```

- To create an object, instantiate it using keyword arguments to the model class, then call `save()` to save it to the database.

```
>>> from myapp.models import Blog
>>> b = Blog(name='Beatles Blog', tagline='All the latest Beatles news.')
>>> b.save()
```

Only here Django hits the database

This performs an INSERT SQL statement behind the scenes.

US
UNIVERSITY
OF SUSSEX

Making queries

- To save changes to an object that's already in the database, use `save()`.

```
>>> b.name = 'New name'
>>> b.save()
```

This performs an UPDATE SQL statement behind the scenes.

- Updating a `ForeignKey` field works exactly the same way as saving a normal field
  - assign an object of the right type to the field in question

```
>>> from myapp.models import Blog, Entry
>>> entry = Entry.objects.get(pk=1)
>>> cheese_blog = Blog.objects.get(name="Cheddar Talk")
>>> entry.blog = cheese_blog
>>> entry.save()
```

UNIVERSITY OF SUSSEX

Making queries

- To update a `ManyToManyField`, use the `add()` method on the field to add a record to the relation.

```
>>> from blog.models import Author
>>> joe = Author.objects.create(name="Joe")
>>> entry.authors.add(joe)
```

- To add multiple records to a ManyToManyField in one go, include multiple arguments in the call to add()

```
>>> john = Author.objects.create(name="John")
>>> paul = Author.objects.create(name="Paul")
>>> entry.authors.add(john, paul)
```

Django will complain if there is an attempt to assign or add an object of the wrong type.

UNIVERSITY OF SUSSEX

Making queries

- To retrieve objects from your database, construct a `QuerySet` via a `Manager` on your model class.

```
>>> Blog.objects
```

- The simplest way to retrieve objects from a table is to get all of them

```
>>> all_entries = Entry.objects.all()
```

- To get a QuerySet of blog entries from the year 2006
  - use `filter()`

```
>>> Entry.objects.filter(pub_date__year=2006)
```

  - With the default manager class

```
>>> Entry.objects.all().filter(pub_date__year=2006)
```

UNIVERSITY OF SUSSEX

Making queries

- Retrieving a single object with get()

```
>>> one_entry = Entry.objects.get(pk=1)
```

- Use a subset of Python's array-slicing syntax to limit your QuerySet to a certain number of results.

```
>>> Entry.objects.all()[5:10]
```

(OFFSET 5 LIMIT 5)

- Basic lookups keyword arguments take the form `field__lookuptype=value`

```
>>> Entry.objects.filter(pub_date__lte='2006-01-01')
```

translates (roughly) into the following SQL:

```
SELECT * FROM blog_entry WHERE pub_date <= '2006-01-01';
```

UNIVERSITY OF SUSSEX

Making queries

- An “exact” match

```
>>> Entry.objects.get(headline__exact="Cat bites dog")
```

- Case-sensitive containment test

```
>>> Entry.objects.get(headline__contains='Lennon')
```

- Lookups that span relationships
  - by retrieving all `Entry` objects with a `Blog` whose name is `'Beatles Blog’`

```
>>> Entry.objects.filter(blog__name='Beatles Blog’)
```

  - by retrieving all `Blog` objects which have at least one `Entry` whose headline contains `'Lennon'`

```
>>> Blog.objects.filter(entry__headline__contains='Lennon')
```

UNIVERSITY OF SUSSEX

Making queries

- Spanning multi-valued relationships
  - To select all blogs containing at least one entry from 2008 having “Lennon” in its headline (the same entry satisfying both conditions)

```
>>> Blog.objects.filter(entry__headline__contains='Lennon',
entry__pub_date__year=2008)
```

  - to perform a more permissive query selecting any blogs with merely some entry with “Lennon” in its headline and some entry from 2008

```
>>>Blog.objects.filter(entry__headline__contains='Lennon').filter(entry__
pub_date__year=2008)
```

UNIVERSITY OF SUSSEX

Migrations

- Are Django's way of propagating changes made to models into your database schema
  - For example, adding a field, deleting a model, etc.
- Are designed to be mostly automatic
  - Although it is necessary to know when to make and run them, and the common problems that may occur
- The migration files for each app
  - live in a `migrations` directory inside of that app, and
  - are designed to be committed to, and distributed as part of, its codebase.

UNIVERSITY OF SUSSEX

The Commands

- There are several commands necessary to interact with migrations and Django's handling of database schema
  - `migrate`, which is responsible for applying and unapplying migrations.
  - `makemigrations`, which is responsible for creating new migrations based on the changes made to the models.
  - `sqlmigrate`, which displays the SQL statements for a migration.
  - `showmigrations`, which lists a project's migrations and their status.
- They can be seen as a version control system for the database schema
  - `makemigrations` is responsible for packaging up the model changes into individual migration files - analogous to commits - and
  - `migrate` is responsible for applying those to the database.

UNIVERSITY OF SUSSEX

Migrations

- They should be made once on the development machine and then run the same migrations on the staging machines, and eventually the production machines
- They will run the same way on the same dataset and produce consistent results
  - It means that what is seen in development and staging is, under the same circumstances, exactly what will happen in production.
- Django will make migrations for any change to your models or fields, even options that don't affect the database
  - the only way it can reconstruct a field correctly is to have all the changes in the history

UNIVERSITY OF SUSSEX

Backend Support

- Migrations are supported on all backends that Django ships with, as well as any third-party backends if they have programmed in support for schema alteration
- Some databases are more capable than others when it comes to schema migrations
  - PostgreSQL is the most capable of all the databases in terms of schema support
  - MySQL lacks support for transactions around schema alteration operations
    - If a migration fails to apply you will have to manually unpick the changes in order to try again
  - SQLite has very little built-in schema alteration support, and so Django attempts to emulate it
    - Creating a new table with the new schema, copying the data across, dropping the old table, and renaming the new table to match the original name

UNIVERSITY OF SUSSEX

Migrations Workflow

1. If changes are made to models, e.g., add a field or remove a model, it is necessary to run `makemigrations`

```
...\> py manage.py makemigrations
Migrations for 'books':
  books/migrations/0003_auto.py:
    - Alter field author on book
```

2. Django scans and compares the models to the versions currently contained in the migration files
3. Then, a new set of migrations will be written out

UNIVERSITY OF SUSSEX

Migrations Workflow

4. The developer needs to read the output to see what `makemigrations` thinks has been changed
   - It's not perfect, and for complex changes it might not be detecting what is expected
5. Once the new migration files are ready, they should be applied to the database to ensure they work as expected

```
...\> py manage.py migrate
Operations to perform:
  Apply all migrations: books
Running migrations:
  Rendering model states... DONE
  Applying books.0003_auto... OK
```

Migrations Workflow

6. Once the migration is applied, commit the migration and the models change to the version control system as a single commit
   - When other developers (or the production servers) check out the code, they'll get both the changes to the models and the accompanying migration at the same time
- Migration(s) can be given a meaningful name instead of a generated one by using the `makemigrations --name` option

```
...\> py manage.py makemigrations --name changed_my_model your_app_label
```

UNIVERSITY OF SUSSEX

Version control

- Because migrations are stored in version control, it may occasionally happen that two developers have both committed a migration to the same app at the same time, resulting in two migrations with the same number.
  - The numbers are just there for developers' reference, Django just cares that each migration has a different name
  - Migrations specify which other migrations they depend on, including earlier migrations in the same app, in the file
  - Therefore, it's possible to detect when there're two new migrations for the same app that aren't ordered.

Version control

- When this happens, Django will prompt and give some options to the developer
  - If it thinks it's safe enough, it will offer to automatically linearise the two migrations for him
  - If not, the developer will have to go in and modify the migrations himself

Dependencies

- While migrations are per-app, the tables and relationships implied by the models are too complex to be created for one app at a time
- When a migration is made that requires something else to run, the resulting migration will contain a dependency on another migration
- This dependency behavior affects most migration operations where one restrict to a single app
  - Restricting to a single app (either in `makemigrations` or `migrate`) is a best-efforts promise, and not a guarantee
  - any other apps that need to be used to get dependencies correct will be.

Apps without migrations must not have relations to apps with migrations.

UNIVERSITY OF SUSSEX

Migration files

- Migrations are stored as an on-disk format, known as *migration files*.
  - These files are normal Python files with an agreed-upon object layout, written in a declarative style.
- A basic migration file

```
from django.db import migrations, models

class Migration(migrations.Migration):

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

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

a list of migrations this one depends on

a list of Operation classes that define what this migration does

Migration files

- The operations are the key
  1. They are a set of declarative instructions which tell Django what schema changes need to be made
  2. Django scans them and builds an in-memory representation of all of the schema changes to all apps, and
  3. Uses this to generate the SQL which makes the schema changes
- One should rarely, if ever, need to edit migration files by hand
  - It's entirely possible to write them manually, if necessary
- Some of the more complex operations are not autodetectable and are only available via a hand-written migration

UNIVERSITY OF SUSSEX

Migrations

- New apps come preconfigured to accept migrations
  - One can add migrations by running `makemigrations` once some changes have been made
- If the app already has models and database tables, and doesn't have migrations yet, it is possible to convert it to use migrations by running `makemigrations` and `migrate --fake-initial`
- Migrations can be reversed with `migrate` by passing the number of the previous migration

Migrations

- to reverse migration books.0003

```
...\> py manage.py migrate books 0002
Operations to perform:
  Target specific migration: 0002_auto, from books
Running migrations:
  Rendering model states... DONE
  Unapplying books.0003_auto... OK
```

- to reverse all migrations applied for an app

```
...\> py manage.py migrate books zero
Operations to perform:
  Unapply all migrations: books
Running migrations:
  Rendering model states... DONE
  Unapplying books.0002_auto... OK
  Unapplying books.0001_initial... OK
```

UNIVERSITY OF SUSSEX

Migrations

- A migration is irreversible if it contains any irreversible operations
  - Attempting to reverse such migrations will raise `IrreversibleError`

```
...\> py manage.py migrate books 0002
Operations to perform:
  Target specific migration: 0002_auto, from books
Running migrations:
  Rendering model states... DONE
  Unapplying books.0003_auto...Traceback (most recent call last):
django.db.migrations.exceptions.IrreversibleError: Operation <RunSQL
sql='DROP TABLE demo_books'> in books.0003_auto is not reversible
```

UNIVERSITY OF SUSSEX

Migrations

- Removing custom model fields from the project or third-party app will cause a problem if they are referenced in old migrations
  - Django provides some model field attributes to assist with model field deprecation using the system checks framework
- Add the `system_check_deprecated_details` attribute to the model field

```
class IPAddressField(Field):
    system_check_deprecated_details = {
        'msg': (
            'IPAddressField has been deprecated. Support for it (except '
            'in historical migrations) will be removed in Django 1.9.'
        ),
        'hint': 'Use GenericIPAddressField instead.',  # optional
        'id': 'fields.W900',  # pick a unique ID for your field.
    }
```

Migrations

- After a deprecation period of your choosing, change the `system_check_deprecated_details` attribute to `system_check_removed_details` and update the dictionary

```
class IPAddressField(Field):
    system_check_removed_details = {
        'msg': (
            'IPAddressField has been removed except for support in '
            'historical migrations.'
        ),
        'hint': 'Use GenericIPAddressField instead.',
        'id': 'fields. E900',  # pick a unique ID for your field.
    }
```

UNIVERSITY OF SUSSEX

Data migrations

- Besides changing the database schema, migrations can be used to change the data in the database itself, in conjunction with the schema, if necessary
- Data migrations are those that alter data
  - They're best written as separate migrations, sitting alongside schema migrations
- Django can't automatically generate data migrations, as it does with schema migrations

1. Start by making an empty migration file and Django will put it in the right place, suggest a name, and add dependencies

```
...\> py manage.py makemigrations --empty yourappname
```

UNIVERSITY OF SUSSEX

Data migrations

2. Then, open the file and it should look like

```
# Generated by Django A.B on YYYY-MM-DD HH:MM
from django.db import migrations

class Migration(migrations.Migration):

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

    operations = [
    ]
```

UNIVERSITY OF SUSSEX

Data migrations

3. Now, create a new function and have `RunPython` use it
   - `RunPython` expects a callable as its argument which takes two arguments
     - an *app registry*, which has the historical versions of all the models loaded into it to match where in the history the migration sits, and
     - a *SchemaEditor*, which can be used to manually effect database schema changes
- For example, let's write a migration that populates our new `name` field with the combined values of `first_name` and `last_name`
  - All we need to do is use the historical model and iterate over the rows

Data migrations

```
from django.db import migrations

def combine_names(apps, schema_editor):
    # We can't import the Person model directly as it may be a newer
    # version than this migration expects. We use the historical version.
    Person = apps.get_model('yourappname', 'Person')
    for person in Person.objects.all():
        person.name = '%s %s' % (person.first_name, person.last_name)
        person.save()

class Migration(migrations.Migration):

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

    operations = [
        migrations.RunPython(combine_names),
    ]
```

Data migrations

- Once that's done, run `python manage.py migrate` as normal
  - the data migration will run in place alongside other migrations
- It is possible to pass a second callable to `RunPython` to run whatever logic is necessary to be executed when migrating backwards
  - If this callable is omitted, migrating backwards will raise an exception.

Next Lecture ...

- ✓ Introduction
- ✓ HTTP, Caching, and CDNs
- ✓ Views
- ✓ Templates
- ✓ Forms
- ✓ Models
- ➢ **Security**

- Transactions
- RPC/RMI/Thrift
- Web Services
- RESTful Services
- Time
- Elections/Group Communication
- Zookeeper

UNIVERSITY OF SUSSEX