MVT Development Pattern (P59-62)

- Consider the overall design of a database-driven Django Web application.
- Django is designed to encourage loose coupling and strict separation between pieces of an application - it's easy to make changes to one particular piece of the application without affecting the other pieces.
- In ***view functions***, it is importance to separate the ***business logic*** from the ***presentation logic*** by **using a template system**. With the database layer, apply that same philosophy to ***data access logic.***
- Those three pieces together — data access logic, business logic, and presentation logic — comprise a concept ***called the Model-View-Controller (MVC)*** pattern of software architecture. In this pattern,
  - "***Model***" refers to the ***data access layer***
  - "***View***" refers to the part of the system that ***selects what to display and how to display*** it, and
  - "***Controller***" refers to the part of the system that ***decides which view to use, depending on user input, accessing the model as needed.***

- Django follows this ***MVC pattern*** closely called an ***MVC framework***.
  - **M**, the data-access portion, is handled by Django's database layer
  - **V**, the portion that selects which data to display and how to display it, is ***handled by views and templates***.
  - The framework itself handles **C,** the portion that delegates **to a view depending on user input,** by ***following your URLconf*** and ***calling the appropriate Python function for the given URL***.
- Because the framework handles the "C" itself and most of the excitement in Django happens in models, templates, and views, Django has been referred to as an ***MTV framework***.
- In the MTV development pattern,
  - **M** stands for "**Model**," the **data access layer**. This layer contains anything and everything about the data: how to access it, how to validate it, which behaviors it has, and the relationships between the data.
  - **T** stands for "**Template**," the **presentation layer**. This layer contains presentation-related decisions: how something should be displayed on a Web page or other type of document.
  - **V** stands for "**View**," the **business logic layer**. This layer contains the logic that access the model and defers to the appropriate template(s). You can think of it as the bridge between models and templates.

Configuring Databases (P62-65)

- In Django's database layer, First, we need to take care of some initial configuration: we need to tell Django which database server to use and how to connect to it.
- We'll assume we've set up a database server - **SQLite**, activated it, and created a database within it (e.g., using a CREATE DATABASE statement).
- As with **TEMPLATE_DIRS**, database configuration lives in the Django settings file, called **settings.py** by default. Edit that file and look for the database settings:

```
DATABASE_ENGINE = ''
DATABASE_NAME = ''
```

```
DATABASE_USER = ''
DATABASE_PASSWORD = ''
DATABASE_HOST = ''
DATABASE_PORT = ''
```

- Here's a rundown of each setting. DATABASE_ENGINE tells Django which database engine to use. If you're using a database with Django, DATABASE_ENGINE must be set to one of the strings shown in Table.

Table. *Database Engine Settings*

Setting	Database	Required Adapter
postgresql	PostgreSQL	psycopg version 1.x, http://www.djangoproject.com/r/python-pgsql/1/.
postgresql_psycopg2	PostgreSQL	psycopg version 2.x, http://www.djangoproject.com/r/python-pgsql/.
mysql	MySQL	MySQLdb, http://www.djangoproject.com/r/python-mysql/.
sqlite3	SQLite	No adapter needed if using Python 2.5+. Otherwise, pysqlite, http://www.djangoproject.com/r/python-sqlite/.
oracle	Oracle	cx_Oracle, http://www.djangoproject.com/r/python-oracle/.

- ***DATABASE_NAME*** tells Django the **name of our database**. If we're using SQLite, specify the full filesystem path to the database file on your filesystem (e.g., '/home/django/mydata.db').
- ***DATABASE_USER*** tells Django **which username** to use when connecting to our database. If we're using SQLite, leave this blank.
- ***DATABASE_PASSWORD*** tells Django which password to use when connecting to your database. If we're using SQLite or have an empty password, leave this blank.
- ***DATABASE_HOST*** tells Django **which host to use when connecting to your database**. If our database is on the same computer as our Django installation (i.e., localhost), leave this blank. If we're using SQLite, leave this blank. MySQL is a special case here. If this value starts with a forward slash (/) and you're using MySQL, MySQL will connect via a Unix socket to the specified socket,
  for example: ***DATABASE_HOST = '/var/run/mysql'***. If you're using MySQL and this value doesn't start with a forward slash, then this value is assumed to be the host.
- ***DATABASE_PORT*** tells Django which port to use when connecting to your database. If we're using SQLite, leave this blank. Otherwise, if you leave this blank, the underlying database adapter will use whichever port is default for your given database server. In most cases, the default port is fine, so you can leave this blank.
- Once we've entered those settings, **test our configuration**. First, from within the mysite project directory we created, run the command ***python manage.py shell***.
- Once we've entered the shell, type these commands to test our database configuration:

```
>>> from django.db import connection
>>> cursor = connection.cursor()
```

- ***If nothing happens, then your database is configured properly***. Otherwise, check the error message for clues about what's wrong. The following Table shows some common errors:

Table. *Database Configuration Error Messages*

Error Message	Solution
You haven't set the DATABASE_ENGINE setting yet.	Set the DATABASE_ENGINE setting to something other than an empty string.
Environment variable DJANGO_SETTINGS_MODULE is undefined.	Run the command `python manage.py shell` rather than `python`.
Error loading _____ module: No module named _____.	You haven't installed the appropriate database-specific adapter (e.g., psycopg or MySQLdb).
_____ isn't an available database backend.	Set your DATABASE_ENGINE setting to one of the valid engine settings described previously. Perhaps you made a typo?
database _____ does not exist	Change the DATABASE_NAME setting to point to a database that exists, or execute the appropriate CREATE DATABASE statement in order to create it.
role _____ does not exist	Change the DATABASE_USER setting to point to a user that exists, or create the user in your database.
could not connect to server	Make sure DATABASE_HOST and DATABASE_PORT are set correctly, and make sure the server is running.

Creating First App

- A **project** is an instance of a certain set of Django apps, plus the configuration for those apps. Technically, the only requirement of a project is that it supplies a settings file, which **defines the database connection information, the list of installed apps, the TEMPLATE_DIRS**, and so forth.
- An **app** is a portable set of Django functionality, usually including models and views that lives together in a single Python package. For example, Django comes with a number of apps, such as a commenting system and an automatic admin interface. A key thing to note about these apps is that they're portable and reusable across multiple projects.
- There are very few hard-and-fast rules about how we fit our Django code into this scheme; it's flexible.
    - If you're **building a simple Web site**, you may **use only a single app**.
    - If you're building a **complex Web site with several unrelated pieces** such as an e-commerce system and a message board, we'll probably want to **split those into separate apps** so that we'll be able to **reuse them individually in the future**.
- Don't necessarily need to create apps at all, as evidenced by the example view functions created a file called ***views.py***, filled it with view functions, and pointed our ***URLconf*** at those functions. No "apps" were needed.
- There's **one requirement regarding the app convention**: if we're **using Django's database layer (models), you must create a Django app**. Models must live within apps. Thus, in order to start writing our models, we'll need to create a new app.
- For example, within the ***mysite*** project directory discussed earlier, type this command to create a new app named books:

python manage.py startapp books

- This command does not produce any output, but it does create a books directory within the mysite directory. Let's look at the contents of that directory:

  books/
  __init__.py
  models.py
  views.py

- These files will contain the ***models*** and ***views*** for this app. Have a look at ***models.py*** and ***views.py*** in our **favorite text editor**. ***Both files are empty, except for an import in models.py.*** This is the blank slate for your Django app.

Defining and Implementing Models in Python (P65-82)

Topics Covered:

- ***Defining the Model***
- ***Installing the Model***
- ***Basic Data Access***
- ***Adding Model String Representations***
- ***Inserting and Updating Data***
- ***Selecting Objects*** (Filtering Data, Retrieving Single Objects, Ordering Data, Chaining Lookups, Slicing Data)
- ***Deleting Objects***
- ***Making Changes to a Database Schema (***Adding Fields, Removing Fields, Removing Many-to-Many Fields, Removing Models***)***

Defining the Model

- A **Django model is a description of the data in database, represented as Python code.**
- It's data layout— the equivalent of SQL CREATE TABLE statements—**except it's in Python instead of SQL**, and it includes more than just database column definitions.
- Django uses a model to execute SQL code behind the scenes and return convenient Python data structures representing the rows in your database tables.
- Django also uses models to represent higher-level concepts that SQL can't necessarily handle.

If you're familiar with databases, your immediate thought might be, "Isn't it redundant to define data models in Python and in SQL?"

Django works the way it does for several reasons:

- Self-analysis requires overhead and is imperfect. In order to **provide convenient data access APIs, Django needs to know the database layout** somehow, and there are **two ways** of accomplishing this.
  - to **explicitly describe the data in Python**, and
  - to **self-analyse the database at runtime to determine the data models.**
- Django's developers aim **to trim as much framework overhead as possible**, and this approach has **succeeded in making Django faster than its high-level framework competitors** in benchmarks.)
- Second, **some databases, notably older versions of MySQL**, do not store sufficient metadata for accurate and complete introspection.

- Writing Python is fun, and **keeping everything in Python limits the number of times your brain has to do a "context switch."** It helps productivity if you keep yourself in a single programming environment/mentality for as long as possible.
- **Having data models stored as code rather than in your database makes it easier to keep your models under version control.** This way, you can easily keep track of changes to your data layouts.
- **SQL allows for only a certain level of metadata about a data layout.** Most database systems, for example, **do not provide a specialized data type for representing email addresses or URLs. Django models do. The advantage of higher-level data types is higher productivity and more reusable code.**
- **SQL is inconsistent across database platforms.** If you're distributing a Web application, for example, it's much **more realistic to distribute a Python module that describes your data layout than separate sets of CREATE TABLE statements for MySQL, PostgreSQL, and SQLite.**

First Model

- Now focus on a basic **book/author/ publisher** data layout - the **conceptual relationships between books, authors, and publishers.**
- The following concepts, fields, and relationships are:
  - An **author** has a salutation (e.g., Mr. or Mrs.), a first name, a last name, an email address, and a headshot photo.
  - A **publisher** has a name, a street address, a city, a state/province, a country, and a Web site.
  - A **book** has a title and a publication date. It also has one or more authors (a many-tomany relationship with authors) and a single publisher (a one-to-many relationship—aka foreign key—to publishers)
- The first step in using this database layout with Django is to express it as Python code. In the ***models.py*** file that was created by the ***startapp*** command, enter the following:

```
from django.db import models

class Publisher(models.Model):
    name = models.CharField(maxlength=30)
    address = models.CharField(maxlength=50)
    city = models.CharField(maxlength=60)
    state = models.CharField(maxlength=30)
    country = models.CharField(maxlength=50)
    website = models.URLField()

class Author(models.Model):
    salutation = models.CharField(maxlength=10)
    first_name = models.CharField(maxlength=30)
    last_name = models.CharField(maxlength=40)
    email = models.EmailField()
    headshot = models.ImageField(upload_to='/tmp')

class Book(models.Model):
    title = models.CharField(maxlength=100)
    authors = models.ManyToManyField(Author)
    publisher = models.ForeignKey(Publisher)
    publication_date = models.DateField()
```

- Let's quickly examine this code to cover the basics.
- The first thing to notice is that **each model is represented by a Python class** that is a ***subclass of django.db.models.Model***.
- The parent class, Model, contains all the machinery necessary to make these objects capable of interacting with a database—and that leaves our models responsible solely for defining their fields, in a nice and compact syntax, this is all the code we need to write to have basic data access with Django.
- The exception to the one-class-per-database-table rule is the case of many-to-many relationships. In our example models, Book has a ManyToManyField called authors. This designates that a book has one or many authors, but the Book database table doesn't get an authors column. Rather, Django creates an additional table—a many-to-many "join table"—that handles the mapping of books to authors.
- Finally, **unless we instruct it otherwise, Django automatically gives every model an integer primary key field called id**. Each Django model is **required to have a single-column primary key**.

Installing the Model

- Now **create the tables in our database**. In order to do that, the **first step is to activate these models in our Django project**.
- We do that by adding the books app to the list of installed apps in the settings file.
- Edit the ***settings.py*** file again, and look for the ***INSTALLED_APPS*** setting. ***INSTALLED_APPS*** tells Django which apps are activated for a given project. By default, it looks something like this:

```
INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
)
```

- Modify the default ***MIDDLEWARE_CLASSES*** and ***TEMPLATE_CONTEXT_PROCESSORS*** settings. These depend on some of the apps we just commented out. Then, add **'mysite.books'** to the **INSTALLED_APPS** list, so the setting ends up looking like this:

```
MIDDLEWARE_CLASSES = []
TEMPLATE_CONTEXT_PROCESSORS = []
INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    'mysite.books',
)
```

- Now that the Django app has been activated in the settings file, we can create the database tables in our database. First, let's validate the models by running this command:

```
python manage.py validate
```

- The validate command checks whether your models' syntax and logic are correct. If all is well, you'll see the message 0 errors found.

- If your models are valid, run the following command for Django to generate **CREATE TABLE** statements for your models in the **books** app

  python manage.py sqlall books

- In this command, books is the name of the app. It's what you specified when you ran the command manage.py startapp. When you run the command, you should see something like this:

```
BEGIN;
CREATE TABLE "books_publisher" (
    "id" serial NOT NULL PRIMARY KEY,
    "name" varchar(30) NOT NULL,
    "address" varchar(50) NOT NULL,
    "city" varchar(60) NOT NULL,
    "state_province" varchar(30) NOT NULL,
    "country" varchar(50) NOT NULL,
    "website" varchar(200) NOT NULL
);
CREATE TABLE "books_book" (
    "id" serial NOT NULL PRIMARY KEY,
    "title" varchar(100) NOT NULL,
    "publisher_id" integer NOT NULL REFERENCES "books_publisher" ("id"),
    "publication_date" date NOT NULL
);
CREATE TABLE "books_author" (
    "id" serial NOT NULL PRIMARY KEY,
    "salutation" varchar(10) NOT NULL,
    "first_name" varchar(30) NOT NULL,
    "last_name" varchar(40) NOT NULL,
    "email" varchar(75) NOT NULL,
    "headshot" varchar(100) NOT NULL
);
CREATE TABLE "books_book_authors" (
    "id" serial NOT NULL PRIMARY KEY,
    "book_id" integer NOT NULL REFERENCES "books_book" ("id"),
    "author_id" integer NOT NULL REFERENCES "books_author" ("id"),
    UNIQUE ("book_id", "author_id")
);
CREATE INDEX books_book_publisher_id ON "books_book" ("publisher_id");
COMMIT;
```

- Note the following:
  - Table names are automatically generated by combining the name of the app (books) and the lowercased name of the model (publisher, book, and author).
  - As we mentioned earlier, Django adds a primary key for each table automatically—the id fields. You can override this, too.
  - By convention, Django appends "_id" to the foreign key field name. As you might have guessed, you can override this behavior as well.
  - The foreign key relationship is made explicit by a REFERENCES statement.

- These CREATE TABLE statements are tailored to the database you're using, so databasespecific field types such as auto_increment (MySQL), serial (PostgreSQL), or integer primary key (SQLite) are handled for you automatically. The same goes for quoting of column names (e.g., using double quotes or single quotes). This example output is in PostgreSQL syntax.

Note: The **sqlall** command **doesn't actually create the tables or otherwise touch your database—it just prints output to the screen** so you can see what SQL Django would execute if you asked it.

- Django provides an easier way of committing the SQL to the database. Run the syncdb command, like so: ***python manage.py syncdb.***
- You'll see something like this:

  Creating table books_publisher
  Creating table books_book
  Creating table books_author
  Installing index for books.Book model

- The ***syncdb*** command is a simple "sync" of your models to your database. ***It looks at all of the models in each app in your INSTALLED_APPS setting, checks the database to see whether the appropriate tables exist yet, and creates the tables if they don't yet exist***.
- Note that ***syncdb does not sync changes in models or deletions of models***; **if you make a change to a model or delete a model, and you want to update the database, *syncdb* will not handle that.**
- If you run python manage.py ***syncdb*** again, ***nothing happens***, because you haven't added any models to the books app or added any apps to ***INSTALLED_APPS***.

****

Basic Data Access

- Once we've ***created a model, Django automatically provides a high-level Python API for working with those models***. Try it out by running ***python manage.py shell*** and type the following:

```
>>> from books.models import Publisher
>>> p1 = Publisher(name='Addison-Wesley', address='75 Arlington St.',
... city='Boston', state='MA', country='U.S.A.',
... website='http://www.addison-wesley.com/')
>>> p.save()
>>> p2 = Publisher(name="O'Reilly", address='10 Fawcett St.',
... city='Cambridge', state='MA', country='U.S.A.',
... website='http://www.oreilly.com/')
>>> p2.save()
>>> publisher_list = Publisher.objects.all()
>>> publisher_list [, ]
[<Publisher: Publisher object>, <Publisher: Publisher object>]
```

- To ***create an object, just import the appropriate model class*** and instantiate it by passing in values for each field.
- To ***save the object to the database***, call the ***save()*** method on the object. ***Behind the scenes, Django executes an SQL INSERT statement here***.
- To **retrieve objects from the database**, use the attribute **Publisher.objects**. Fetch a list of all Publisher objects in the database with the statement **Publisher.objects.all().** Behind the scenes, Django executes an **SQL SELECT** statement here.

****

Adding Model String Representations

- When we printed out the list of publishers, all we got was this unhelpful display, which makes it difficult to tell the Publisher objects apart:
  [<Publisher: Publisher object>, <Publisher: Publisher object>]
- We can fix this easily by adding a method called ***__str__()*** to our Publisher object. A ***__str__()*** method tells Python how to display the "***string***" representation of an object. You can see this in action by adding a ***__str__()*** method to the ***three models***:

```
from django.db import models class
Publisher(models.Model):
    name = models.CharField(maxlength=30)
    address = models.CharField(maxlength=50)
    city = models.CharField(maxlength=60)
    state = models.CharField(maxlength=30) country
    = models.CharField(maxlength=50)
    website = models.URLField()

def __str__(self):
        return self.name
```

```
class Author(models.Model):
    salutation = models.CharField(maxlength=10)
    first_name = models.CharField(maxlength=30)
    last_name = models.CharField(maxlength=40)
    email = models.EmailField()
    headshot = models.ImageField(upload_to='/tmp')

def __str__(self):
        return '%s %s' % (self.first_name, self.last_name)
```

```
class Book(models.Model):
    title = models.CharField(maxlength=100)
    authors = models.ManyToManyField(Author)
    publisher = models.ForeignKey(Publisher)
    publication_date = models.DateField()
def __str__(self):
        return self.title
```

- For the changes to take effect, exit out of the Python shell and enter it again with python manage.py shell. (This is the simplest way to make code changes take effect.)
- Now the list of Publisher objects is much easier to understand:

```
>>> from books.models import Publisher
>>> publisher_list = Publisher.objects.all()
>>> publisher_list
[<Publisher: Publisher object>, <Publisher: Publisher object>]
```

- Make sure any model you define has a **__str__()** method—because Django uses the output of **__str__()** in several places when it needs to display objects.
- Finally, note that **__str__()** is a good example of adding behavior to models. A ***Django model describes more than the database table layout for an object.***

****

Inserting and Updating Data

- To ***insert a row into your database, first create an instance of your model*** using keyword arguments, like so:

```
>>> p = Publisher(name='Sarathy', address='28 Bannerghatta Road.', city='Bengaluru',
... state='Karnataka', country='INDIA', website='http://www.incerd.in/')
```

- To ***save the record*** into the database (i.e., to perform the SQL INSERT statement), call the object's save() method: ***>>> p.save().***
- In SQL, this can roughly be translated into the following:

```
INSERT INTO book_publisher
    (name, address, city, state, country, website)
VALUES
    ('Sarathy', '28 Bannerghatta Road.', 'Bengaluru', 'Karnataka', 'INDIA', 'http://www.incerd.in/');
```

- Because the Publisher model uses an autoincrementing primary key ***id***, the initial call to ***save()*** does one more thing: it calculates the primary key value for the record and sets it to the ***id*** attribute on the instance:

```
>>> p.id 52 # this will differ based on your own data
```

Subsequent calls to **save()** will save the record in place, without creating a new record (i.e., performing an **SQL UPDATE statement instead of an INSERT**):
 	
 `>>> p.name = 'Sarathy Publishing'`
 `>>> p.save()`
	The preceding save() statement will result in roughly the following SQL:
	 UPDATE book_publisher SET
	 name='Sarathy',
	 address='28 Bannerghatta Road.',
	 city='Bengaluru', **state**='Karnataka',
	 country='INDIA',
	 website='http://www.incerd.in/'
	 WHERE
	 id = 52;

****

Selecting Objects (Filtering Data, Retrieving Single Objects, Ordering Data, Chaining Lookups, Slicing Data)

- Look up all the data for a certain model:

```
>>> Publisher.objects.all()
[<Publisher: Addison-Wesley>, <Publisher: O'Reilly>, <Publisher: Sarathy Publishing>]
```

- This roughly translates to the following SQL:

```
SELECT id, name, address, city, state, country, website
FROM book_publisher;
```

- Let's take a close look at each part of this **Publisher.objects.all()** line:
  - First, we have the model we defined, Publisher.
  - Next, we have this objects business. Technically, this is a manager - take care of all "table-level" operations on data including, most important, data lookup.
  - Finally, we have **all()**. This is a method on the objects manager - that returns all the rows in the database.

Filtering Data

- Most of the time we're going to want to ***deal with a subset of the data with the filter() method***:

```
>>> Publisher.objects.filter(name="Sarathy Publishing")
[<Publisher: Sarathy Publishing>]
```

- ***filter()*** takes keyword arguments that get translated into the appropriate **SQL WHERE** clauses. The preceding example would get translated into **something like this**:

```
SELECT id, name, address, city, state, country, website
FROM book_publisher WHERE name = 'Sarathy Publishing';
```

- We can **pass multiple arguments** into ***filter()*** to narrow down things further:

```
>>> Publisher.objects.filter(country="INDIA", state="Karnataka")
[<Publisher: Sarathy Publishing>]
```

- Those multiple arguments get translated into SQL AND clauses. Thus, the example in the code snippet translates into the following:

```
SELECT id, name, address, city, state_province, country, website
FROM book_publisher WHERE country = 'INDIA' AND state = 'Karnataka';
```

- Other lookup types are available:

```
>>> Publisher.objects.filter(name__contains="press")
[<Publisher: Sarathy Publishing>]
```

- That's a double underscore there between name and contains. Django uses the double underscore to signal that the "__" contains part gets translated by Django into an **SQL LIKE** statement:

```
SELECT id, name, address, city, state_province, country, website FROM
book_publisher
WHERE name LIKE '%press%';
```

- Other types of lookups are
  1. ***icontains*** (case-insensitive LIKE)
  2. ***startswith*** and ***endswith***
  3. ***range*** (SQL BETWEEN queries)

****

Retrieving Single Objects

- Sometimes we want to fetch only a single object, use the ***get()*** method
  >>> **Publisher.objects.get**(name="Sarathy Publishing")
  [<Publisher: Sarathy Publishing>]
- Instead of a list (rather, QuerySet), only a single object is returned. Because of that, a query resulting in multiple objects will cause an exception:

```
>>> Publisher.objects.get(country="INDIA")
Traceback (most recent call last):
... AssertionError: get() returned more than one Publisher—it returned 2!
```

- ***A query that returns no objects also causes an exception***:

```
>>> Publisher.objects.get(name="Penguin")
Traceback (most recent call last):
... DoesNotExist: Publisher matching query does not exist
```

Ordering Data

- In the previous examples, the objects are being returned in a random order. The function ***order_by()*** is used to reorder our data into a useful list.

```
>>> Publisher.objects.order_by("name")
[<Publisher: Sarathy Publishing>, <Publisher: Addison-Wesley>, <Publisher: O'Reilly>]
```

- SQL now includes a specific ordering: **SELECT *id, name, address, city, state, country, website* FROM *book_publisher* ORDER BY *name;***
- We can **order by any field** we like:
  >>> Publisher.objects.order_by("address")
  [<Publisher: O'Reilly>, <Publisher: Sarathy Publishing>, <Publisher: Addison-Wesley>]
  >>> Publisher.objects.order_by("state")
  [<Publisher: Sarathy Publishing>, <Publisher: Addison-Wesley>, <Publisher: O'Reilly>]
- Order by **multiple fields**:
  >>> Publisher.objects.order_by("country", "address")

[<Publisher: Sarathy Publishing>, <Publisher: O'Reilly>, <Publisher: Addison-Wesley>]

- We can also specify reverse ordering by prefixing the field name with a – (a minus sign character):

  >>> Publisher.objects.order_by("-name")

  [<Publisher: O'Reilly>, <Publisher: Sarathy Publishing>, <Publisher: Addison-Wesley>]
- While this flexibility is useful, using **order_by()** all the time can be quite repetitive, **most of the time we'll have a particular field we usually want to order by**. In these cases, **Django lets you attach a default ordering to the model**:

```
class Publisher(models.Model):
    name = models.CharField(maxlength=30)
    address = models.CharField(maxlength=50)
    city = models.CharField(maxlength=60)
    state_province = models.CharField(maxlength=30)
    country = models.CharField(maxlength=50)
    website = models.URLField()

    def __str__(self):
        return self.name

    class Meta:
        ordering = ["name"]
```

This ordering = ["name"] bit tells Django that unless an ordering is given explicitly with order_by(), all publishers should be ordered by name.

Chaining Lookups

- We've seen how **can filter data**, and we've seen **how can order it**. At times, of course, we're going to want to do both, in these cases, simply **"chain" the lookups together**:

  >>> Publisher.objects.filter(country="U.S.A.").order_by("-name")

  [<Publisher: O'Reilly>, <Publisher: Sarathy Publishing>, <Publisher: Addison-Wesley>]
- As you might expect, this translates to an **SQL query with both a WHERE and an ORDER BY**:

  SELECT id, name, address, city, state_province, country, website FROM book_publisher WHERE country = 'U.S.A' ORDER BY name DESC;
- **We can keep chaining queries as long as we like. There is no limit.**

Slicing Data

- **Another common need is to look up only a fixed number of rows**.
- Imagine we have **thousands of publishers in database**, but we want to display only the first one. You can do this using Python's standard list slicing syntax:

  >>> Publisher.objects.all()[0]

  <Publisher: Addison-Wesley>
- This translates roughly to the following: **SELECT** id, name, address, city, state_province, country, website **FROM** book_publisher **ORDER BY** name **LIMIT** 1;

****

Deleting Objects **(P-78)**

- To delete objects, simply call the ***delete()*** method on your object:

```
>>> apress = Publisher.objects.get(name="Addison-Wesley")
>>> apress.delete()
>>> Publisher.objects.all()
[<Publisher: Apress Publishing>, <Publisher: O'Reilly>]
```

- You can also delete objects in ***bulk by calling delete()*** on the result of some lookup:

```
>>> publishers = Publisher.objects.all()
>>> publishers.delete()
>>> Publisher.objects.all()
[]
```

- **Deletions are permanent**, it's usually a good idea to avoid deleting objects unless absolutely have to—relational databases don't do "undo" so well, and restoring from backups is painful.
- It's **often a good idea to add "active" flags to your data models**. We can look up only "active" objects, and **simply set the active field to False instead of deleting the object**. Then, if we **realize we've made a mistake, we can simply flip the flag back.**

Making Changes to a Database Schema

Topics Covered:

- ***Adding Fields***
- ***Removing Fields***
- ***Removing Many-to-Many Fields***
- ***Removing Models)***

- **"syncdb" command creates tables** that don't yet exist in our database—**it does not sync changes in models or perform deletions of models**. If add or change a model's field, or if delete a model, we need to make the change in database manually.
- **When dealing with schema changes, it's important to keep a few things in mind about how Django's database layer works:**
  - **Django will complain loudly if a model contains a field that has not yet been created in the database table**. This will cause an error the first time you use the Django database API to query the given table (i.e., **it will happen at code execution time, not at compilation time**).
  - **Django does not care if a database table contains columns that are not defined in the model.**
  - **Django does not care if a database contains a table that is not represented by a model.**
- Making schema changes is a matter of changing the various pieces—the Python code and the database itself—in the right order.

****

Adding Fields

- When adding a field to a table/model in a production setting, the trick is to take advantage of the fact that **Django doesn't care if a table contains columns that aren't defined in the model**. The **strategy is to add the column in the database and then update the Django model** to **include the new field**.

- In order to know how the new database column should be expressed in SQL, need to look at the output of ***Django's manage.py sqlall*** command, which requires that the **field exist in the model.**
- **First, take these steps in the development environment** (i.e., not on the production server):
  1. Add the field to model.
  2. Run ***manage.py sqlall [yourapp]*** to see the new **CREATE TABLE** statement for the model.
  3. **Start database's interactive shell** (e.g., psql or mysql, or you can use manage.py dbshell). **Execute an ALTER TABLE** statement that adds your new column.
  4. (Optional.) Launch the Python interactive shell with manage.py shell and verify that the new field was added properly by importing the model and selecting from the table (e.g., ***MyModel.objects.all()[:5]***).
- Then on the production server perform these steps:
  1. Start database's interactive shell.
  2. Execute the ALTER TABLE statement used in step 3 of the development environment steps.
  3. Add the field to model. If we're using source-code revision control and you checked in your change in development environment step 1, now is the time to update the code (e.g., svn update, with Subversion) on the production server.
  4. Restart the Web server for the code changes to take effect.
- For example, if we add a ***num_pages*** field to the **Book model** described. First, alter the model in our development environment to look like this:

```
class Book(models.Model):
        title = models.CharField(maxlength=100)
        authors = models.ManyToManyField(Author)
        publisher = models.ForeignKey(Publisher)
        publication_date = models.DateField()
        num_pages = models.IntegerField(blank=True, null=True)

        def __str__(self):
            return self.title
```

- Then run the command ***manage.py sqlall books*** to see the **CREATE TABLE** statement. It would look something like this:

```
CREATE TABLE "books_book" (
    "id" serial NOT NULL PRIMARY KEY,
    "title" varchar(100) NOT NULL,
    "publisher_id" integer NOT NULL REFERENCES "books_publisher" ("id"),
    "publication_date" date NOT NULL,
    "num_pages" integer NULL
);
```

- Next, start the database's interactive shell for our development database by typing ***psql (for PostgreSQL)***, and execute the following statements:

```
ALTER TABLE books_book ADD COLUMN num_pages integer;
```

- After the **ALTER TABLE** statement, verify that the change worked properly by starting the Python shell and running this code:

```
>>> from mysite.books.models import Book
>>> Book.objects.all()[:5]
```

- If **that code didn't cause errors**, we'd **switch to our production server and execute the ALTER TABLE statement on the production database**. Then, **update the model in the production environment and restart the Web server**.

Removing Fields

- Removing a field from a model is a lot easier than adding one. To remove a field, just follow these steps:
  1. Remove the field from your model and restart the Web server.
  2. Remove the column from your database, using a command like this: ***ALTER TABLE books_book DROP COLUMN num_pages;***

Removing Many-to-Many Fields

- Because many-to-many fields are different from normal fields, the removal process is different:
  1. Remove the ManyToManyField from your model and restart the Web server.
  2. Remove the many-to-many table from your database, using a command like this: ***DROP TABLE books_books_publishers;***

Removing Models

- Removing a model entirely is as easy as removing a field. To remove a model, just follow these steps:
  1. Remove the model from your models.py file and restart the Web server.
  2. Remove the table from your database, using a command like this: ***DROP TABLE books_book;***
