Chapter 5: Models

In Chapter 3, we covered the fundamentals of building dynamic Web sites with Django: setting up views and URLconfs. As we explained, a view is responsible for doing *some arbitrary logic*, and then returning a response. In one of the examples, our arbitrary logic was to calculate the current date and time.

In modern Web applications, the arbitrary logic often involves interacting with a database. Behind the scenes, a *database-driven Web site* connects to a database server, retrieves some data out of it, and displays that data on a Web page. The site might also provide ways for site visitors to populate the database on their own.

Many complex Web sites provide some combination of the two. Amazon.com, for instance, is a great example of a database-driven site. Each product page is essentially a query into Amazon's product database formatted as HTML, and when you post a customer review, it gets inserted into the database of reviews.

Django is well suited for making database-driven Web sites, because it comes with easy yet powerful tools for performing database queries using Python. This chapter explains that functionality: Django's database layer.

(Note: While it's not strictly necessary to know basic relational database theory and SQL in order to use Django's database layer, it's highly recommended. An introduction to those concepts is beyond the scope of this book, but keep reading even if you're a database newbie. You'll probably be able to follow along and grasp concepts based on the context.)

The "Dumb" Way to Do Database Queries in Views

Just as Chapter 3 detailed a "dumb" way to produce output within a view (by hard-coding the text directly within the view), there's a "dumb" way to retrieve data from a database in a view. It's simple: just use any existing Python library to execute an SQL query and do something with the results.

In this example view, we use the MySQLdb library (available via http://www.djangoproject.com/r/python-mysql/) to connect to a MySQL database, retrieve some records, and feed them to a template for display as a Web page:

```
from django.shortcuts import render

import MySQLdb


def book_list(request):

    db = MySQLdb.connect(user='me', db='mydb',
passwd='secret', host='localhost')

    cursor = db.cursor()
```

```
    cursor.execute('SELECT name FROM books ORDER BY
name')

    names = [row[0] for row in cursor.fetchall()]

    db.close()

    return render(request, 'book_list.html', {'names':
names})
```

This approach works, but some problems should jump out at you immediately:

- We're hard-coding the database connection parameters. Ideally, these parameters would be stored in the Django configuration.
- We're having to write a fair bit of boilerplate code: creating a connection, creating a cursor, executing a statement, and closing the connection. Ideally, all we'd have to do is specify which results we wanted.
- It ties us to MySQL. If, down the road, we switch from MySQL to PostgreSQL, we'll have to use a different database adapter (e.g., psycopg rather than MySQLdb), alter the connection parameters, and – depending on the nature of the SQL statement – possibly rewrite the SQL. Ideally, the database server we're using would be abstracted, so that a database server change could be made in a single place. (This feature is particularly relevant if you're building an open-source Django application that you want to be used by as many people as possible.)

As you might expect, Django's database layer aims to solve these problems. Here's a sneak preview of how the previous view can be rewritten using Django's database API:

```
from django.shortcuts import render

from mysite.books.models import Book


def book_list(request):

    books = Book.objects.order_by('name')

    return render(request, 'book_list.html', {'books':
books})
```

We'll explain this code a little later in the chapter. For now, just get a feel for how it looks.

The MTV (or MVC) Development Pattern

Before we delve into any more code, let's take a moment to consider the overall design of a database-driven Django Web application.

As we mentioned in previous chapters, Django is designed to encourage loose coupling and strict separation between pieces of an application. If you follow this philosophy, it's easy to make

changes to one particular piece of the application without affecting the other pieces. In view functions, for instance, we discussed the importance of separating the business logic from the presentation logic by using a template system. With the database layer, we're applying that same philosophy to data access logic.

Those three pieces together – data access logic, business logic, and presentation logic – comprise a concept that's sometimes 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.

Why the Acronym?

The goal of explicitly defining patterns such as MVC is mostly to streamline communication among developers. Instead of having to tell your coworkers, "Let's make an abstraction of the data access, then let's have a separate layer that handles data display, and let's put a layer in the middle that regulates this," you can take advantage of a shared vocabulary and say, "Let's use the MVC pattern here."

Django follows this MVC pattern closely enough that it can be called an MVC framework. Here's roughly how the M, V, and C break down in Django:

- *M*, the data-access portion, is handled by Django's database layer, which is described in this chapter.
- *V*, the portion that selects which data to display and how to display it, is handled by views and templates.
- *C*, the portion that delegates to a view depending on user input, is handled by the framework itself by following your URLconf and calling the appropriate Python function for the given URL.

Because the "C" is handled by the framework 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.

If you're familiar with other MVC Web-development frameworks, such as Ruby on Rails, you may consider Django views to be the "controllers" and Django templates to be the "views." This is an unfortunate confusion brought about by differing interpretations of MVC. In Django's interpretation of MVC, the "view" describes the data that gets presented to the user; it's not necessarily just *how* the data looks, but *which*data is presented. In contrast, Ruby on Rails and similar frameworks suggest that the controller's job includes deciding which data gets presented to the user, whereas the view is strictly *how* the data looks, not *which* data is presented.

Neither interpretation is more "correct" than the other. The important thing is to understand the underlying concepts.

Configuring the Database

With all of that philosophy in mind, let's start exploring 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 you've set up a database server, activated it, and created a database within it (e.g., using a`CREATE DATABASE` statement). If you're using SQLite, no such setup is required, because SQLite uses standalone files on the filesystem to store its data.

As with `TEMPLATE_DIRS` in the previous chapter, database configuration lives in the Django settings file, called `settings.py` by default. Edit that file and look for the database settings:

```
DATABASES = {

    'default': {

        'ENGINE': 'django.db.backends.', # Add
'postgresql_psycopg2', 'mysql', 'sqlite3' or 'oracle'.

        'NAME': '',                      # Or path to
database file if using sqlite3.

        'USER': '',                      # Not used
with sqlite3.

        'PASSWORD': '',                  # Not used
with sqlite3.

        'HOST': '',                      # Set to empty
string for localhost. Not used with sqlite3.

        'PORT': '',                      # Set to empty
string for default. Not used with sqlite3.

    }

}
```

Here's a rundown of each setting.

- `ENGINE` tells Django which database engine to use. If you're using a database with Django, `ENGINE`must be set to one of the strings shown in Table 5-1.

Table 5-1. Database Engine Settings

Setting	Database	Required Adapter
django.db.backends.postgresql_psycopg2	PostgreSQL	psycopg version 2.x,http://www.djangoproject.com/r/python-pgsql/.
django.db.backends.mysql	MySQL	MySQLdb,http://www.djangoproject.com/r/python-mysql/.
django.db.backends.sqlite3	SQLite	No adapter needed.
django.db.backends.oracle	Oracle	cx_Oracle,http://www.djangoproject.com/r/python-oracle/.

- Note that for whichever database back-end you use, you'll need to download and install the appropriate database adapter. Each one is available for free on the Web; just follow the links in the "Required Adapter" column in Table 5-1. If you're on Linux, your distribution's package-management system might offer convenient packages. (Look for packages called python-postgresql orpython-psycopg, for example.)

- Example:

- `'ENGINE': 'django.db.backends.postgresql_psycopg2',`

- NAME tells Django the name of your database. For example:

```
'NAME': 'mydb',
```

If you're using SQLite, specify the full filesystem path to the database file on your filesystem. For example:

```
'NAME': '/home/django/mydata.db',
```

As for where you put that SQLite database, we're using the /home/django directory in this example, but you should pick a directory that works best for you.

- USER tells Django which username to use when connecting to your database. For example: If you're using SQLite, leave this blank.

- PASSWORD tells Django which password to use when connecting to your database. If you're using SQLite or have an empty password, leave this blank.

- HOST tells Django which host to use when connecting to your database. If your database is on the same computer as your Django installation (i.e., localhost), leave this blank. If you'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:

```
'HOST': '/var/run/mysql',
```

- PORT tells Django which port to use when connecting to your database. If you'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 you've entered those settings and saved settings.py, it's a good idea to test your configuration. To do this, run python manage.py shell as in the last chapter, from within the mysite project directory. (As we pointed out last chapter manage.py shell is a way to run the Python interpreter with the correct Django settings activated. This is necessary in our case, because Django needs to know which settings file to use in order to get your database connection information.)

In the shell, type these commands to test your 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. Table 5-2 shows some common errors.

Table 5-2. Database Configuration Error Messages

Error Message	Solution
You haven't set the ENGINE setting yet.	Set the ENGINE setting to something other than an empty string. Valid values are in Table 5-1.
Environment variable DJANGO_SETTINGS_MODULE is undefined.	Run the command python manage.py shellrather than python.
Error loading _____ module: No module named _____.	You haven't installed the appropriate database-specific adapter (e.g., psycopg orMySQLdb). Adapters are *not* bundled with Django, so it's your responsibility to download and install them on your own.
_____ isn't an available database backend.	Set your ENGINE setting to one of the valid engine settings described previously. Perhaps you made a typo?
database _____ does not exist	Change the NAME setting to point to a database that exists, or execute the appropriateCREATE DATABASE statement in order to create it.
role _____ does not exist	Change the USER setting to point to a user that exists, or create the user in your database.
could not connect to server	Make sure HOST and PORT are set correctly, and make sure the database server is running.

Your First App

Now that you've verified the connection is working, it's time to create a *Django app* – a bundle of Django code, including models and views, that lives together in a single Python package and represents a full Django application.

It's worth explaining the terminology here, because this tends to trip up beginners. We'd already created a*project*, in Chapter 2, so what's the difference between a *project* and an *app*? The difference is that of configuration vs. code:

- 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 you fit your Django code into this scheme. 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, you'll probably want to split those into separate apps so that you'll be able to reuse them individually in the future.

  Indeed, you don't necessarily need to create apps at all, as evidenced by the example view functions we've created so far in this book. In those cases, we simply created a file called views.py, filled it with view functions, and pointed our URLconf at those functions. No "apps" were needed.

  However, there's one requirement regarding the app convention: if you'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.

  Within the mysite project directory, type this command to create a books app:

```
python manage.py startapp books
```

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

```
books/
    __init__.py
    models.py
    tests.py
    views.py
```

These files will contain the models and views for this app.

Have a look at models.py and views.py in your favorite text editor. Both files are empty, except for comments and an import in models.py. This is the blank slate for your Django app.

Defining Models in Python

As we discussed earlier in this chapter, the "M" in "MTV" stands for "Model." A Django model is a description of the data in your database, represented as Python code. It's your data layout – the equivalent of your 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 instead of in SQL?" Django works the way it does for several reasons:

- Introspection 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. The first way would be to explicitly describe the data in Python, and the second way would be to introspect the database at runtime to determine the data models.

  This second way seems cleaner, because the metadata about your tables lives in only one place, but it introduces a few problems. First, introspecting a database at runtime obviously requires overhead. If the framework had to introspect the database each time it processed a request, or even only when the Web server was initialized, this would incur an unacceptable level of overhead. (While some believe that level of overhead is acceptable, Django's developers aim to trim as much framework overhead as possible.) 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 to write SQL, then Python, and then SQL again is disruptive.

- 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 pragmatic to distribute a Python module that describes your data layout than separate sets of CREATE TABLE statements for MySQL, PostgreSQL, and SQLite.

  A drawback of this approach, however, is that it's possible for the Python code to get out of sync with what's actually in the database. If you make changes to a Django model, you'll need to make the same changes inside your database to keep your database consistent with the model. We'll discuss some strategies for handling this problem later in this chapter.

Finally, we should note that Django includes a utility that can generate models by introspecting an existing database. This is useful for quickly getting up and running with legacy data. We'll cover this in Chapter 18.

Your First Model

As an ongoing example in this chapter and the next chapter, we'll focus on a basic book/author/publisher data layout. We use this as our example because the conceptual relationships between books, authors, and publishers are well known, and this is a common data layout used in introductory SQL textbooks. You're also reading a book that was written by authors and produced by a publisher!

We'll suppose the following concepts, fields, and relationships:

- An author has a first name, a last name and an email address.
- 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-to-many 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.pyfile that was created by the startapp command, enter the following:

```
from django.db import models


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


class Author(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=40)
    email = models.EmailField()
```

```
class Book(models.Model):
    title = models.CharField(max_length=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. Believe it or not, this is all the code we need to write to have basic data access with Django.

Each model generally corresponds to a single database table, and each attribute on a model generally corresponds to a column in that database table. The attribute name corresponds to the column's name, and the type of field (e.g., CharField) corresponds to the database column type (e.g., varchar). For example, the Publisher model is equivalent to the following table (assuming PostgreSQL CREATE TABLE syntax):

```
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
);
```

Indeed, Django can generate that CREATE TABLE statement automatically, as we'll show you in a moment.

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.

For a full list of field types and model syntax options, see Appendix B.

Finally, note we haven't explicitly defined a primary key in any of these models. Unless you instruct it otherwise, Django automatically gives every model an auto-incrementing integer primary key field called id. Each Django model is required to have a single-column primary key.

Installing the Model

We've written the code; now let's 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',
    'django.contrib.messages',
    'django.contrib.staticfiles',
)
```

Temporarily comment out all six of those strings by putting a hash character (#) in front of them. (They're included by default as a common-case convenience, but we'll activate and discuss them in subsequent chapters.) While you're at it, comment out the default MIDDLEWARE_CLASSES setting, too; the default values in MIDDLEWARE_CLASSES depend on some of the apps we just commented out. Then, add 'books' to theINSTALLED_APPS list, so the setting ends up looking like this:

```
MIDDLEWARE_CLASSES = (
    # 'django.middleware.common.CommonMiddleware',
    # 'django.contrib.sessions.middleware.SessionMiddleware',
    # 'django.middleware.csrf.CsrfViewMiddleware',
```

```
    #
'django.contrib.auth.middleware.AuthenticationMiddlewar
e',

    #
'django.contrib.messages.middleware.MessageMiddleware',

)


INSTALLED_APPS = (

    # 'django.contrib.auth',

    # 'django.contrib.contenttypes',

    # 'django.contrib.sessions',

    # 'django.contrib.sites',

    'books',

)
```

(As we mentioned last chapter when setting TEMPLATE_DIRS, you'll need to be sure to include the trailing comma in INSTALLED_APPS, because it's a single-element tuple. By the way, this book's authors prefer to put a comma after *every* element of a tuple, regardless of whether the tuple has only a single element. This avoids the issue of forgetting commas, and there's no penalty for using that extra comma.)

'mysite.books' refers to the books app we're working on. Each app in INSTALLED_APPS is represented by its full Python path – that is, the path of packages, separated by dots, leading to the app package.

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 you don't, make sure you typed in the model code correctly. The error output should give you helpful information about what was wrong with the code.

Any time you think you have problems with your models, run python manage.py validate. It tends to catch all the common model problems.

If your models are valid, run the following command for Django to generate `CREATE TABLE` statements for your models in the `books` app (with colorful syntax highlighting available, if you're using Unix):

```
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_author" (
    "id" serial NOT NULL PRIMARY KEY,
    "first_name" varchar(30) NOT NULL,
    "last_name" varchar(40) NOT NULL,
    "email" varchar(75) 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") DEFERRABLE INITIALLY DEFERRED,

    "publication_date" date NOT NULL

)

;

CREATE TABLE "books_book_authors" (

    "id" serial NOT NULL PRIMARY KEY,

    "book_id" integer NOT NULL REFERENCES "books_book"
("id") DEFERRABLE INITIALLY DEFERRED,

    "author_id" integer NOT NULL REFERENCES
"books_author" ("id") DEFERRABLE INITIALLY DEFERRED,

    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 lowercase name of the model (publisher, book, and author). You can override this behavior, as detailed in Appendix B.
- 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, too.
- The foreign key relationship is made explicit by a REFERENCES statement.
- These CREATE TABLE statements are tailored to the database you're using, so database-specific 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.

  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. If you wanted to, you could copy and paste this SQL into your database client, or use Unix pipes to pass

it directly (e.g.,`python manage.py sqlall books | psql mydb`). However, Django provides an easier way of committing the SQL to the database: the `syncdb` command:

```
python manage.py syncdb
```

Run that command, and you'll see something like this:

```
Creating table books_publisher

Creating table books_author

Creating table books_book

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. (More on this in the "Making Changes to a Database Schema" section toward the end of this chapter.)

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`. Ergo, it's always safe to run `python manage.py syncdb` – it won't clobber things.

If you're interested, take a moment to dive into your database server's command-line client and see the database tables Django created. You can manually run the command-line client (e.g., `psql` for PostgreSQL) or you can run the command `python manage.py dbshell`, which will figure out which command-line client to run, depending on your `DATABASE_SERVER` setting. The latter is almost always more convenient.

Basic Data Access

Once you'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 typing the following:

```
>>> from books.models import Publisher

>>> p1 = Publisher(name='Apress', address='2855
Telegraph Avenue',

...     city='Berkeley', state_province='CA',
country='U.S.A.',

...     website='http://www.apress.com/')

>>> p1.save()
```

```
>>> p2 = Publisher(name="O'Reilly", address='10 Fawcett
St.',

...     city='Cambridge', state_province='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>]
```

These few lines of code accomplish quite a bit. Here are the highlights:

- First, we import our Publisher model class. This lets us interact with the database table that contains publishers.
- We create a Publisher object by instantiating it with values for each field – name, address, etc.
- To save the object to the database, call its save() method. Behind the scenes, Django executes an SQL INSERT statement here.
- To retrieve publishers from the database, use the attribute Publisher.objects, which you can think of as a set of all publishers. Fetch a list of *all* Publisher objects in the database with the statementPublisher.objects.all(). Behind the scenes, Django executes an SQL SELECT statement here.

One thing is worth mentioning, in case it wasn't clear from this example. When you're creating objects using the Django model API, Django doesn't save the objects to the database until you call the save() method:

```
p1 = Publisher(...)

# At this point, p1 is not saved to the database yet!

p1.save()

# Now it is.
```

If you want to create an object and save it to the database in a single step, use the objects.create()method. This example is equivalent to the example above:

```
>>> p1 = Publisher.objects.create(name='Apress',

...     address='2855 Telegraph Avenue',
```

```
...     city='Berkeley', state_province='CA',
country='U.S.A.',

...     website='http://www.apress.com/')

>>> p2 = Publisher.objects.create(name="O'Reilly",

...     address='10 Fawcett St.', city='Cambridge',

...     state_province='MA', country='U.S.A.',

...     website='http://www.oreilly.com/')

>>> publisher_list = Publisher.objects.all()

>>> publisher_list
```

Naturally, you can do quite a lot with the Django database API – but first, let's take care of a small annoyance.

Adding Model String Representations

When we printed out the list of publishers, all we got was this unhelpful display that 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 __unicode__() to our Publisher class.
A __unicode__()method tells Python how to display the "unicode" representation of an object.
You can see this in action by adding a __unicode__() method to the three models:

```
from django.db import models


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

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


class Author(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=40)
    email = models.EmailField()


    def __unicode__(self):
        return u'%s %s' % (self.first_name,
self.last_name)


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


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

As you can see, a __unicode__() method can do whatever it needs to do in order to return a representation of an object. Here, the __unicode__() methods for Publisher and Book simply return the object's name and title, respectively, but the __unicode__() for Author is slightly more complex – it pieces together the first_name and last_name fields, separated by a space.

The only requirement for __unicode__() is that it return a Unicode object.
If __unicode__() doesn't return a Unicode object – if it returns, say, an integer – then Python will raise a TypeError with a message like"coercing to Unicode: need string or buffer, int found".

Unicode objects

What are Unicode objects?

You can think of a Unicode object as a Python string that can handle more than a million different types of characters, from accented versions of Latin characters to non-Latin characters to curly quotes and obscure symbols.

Normal Python strings are *encoded*, which means they use an encoding such as ASCII, ISO-8859-1 or UTF-8. If you're storing fancy characters (anything beyond the standard 128 ASCII characters such as 0-9 and A-Z) in a normal Python string, you have to keep track of which encoding your string is using, or the fancy characters might appear messed up when they're displayed or printed. Problems occur when you have data that's stored in one encoding and you try to combine it with data in a different encoding, or you try to display it in an application that assumes a certain encoding. We've all seen Web pages and e-mails that are littered with "??? ??????" or other characters in odd places; that generally suggests there's an encoding problem.

Unicode objects, however, have no encoding; they use a consistent, universal set of characters called, well, "Unicode." When you deal with Unicode objects in Python, you can mix and match them safely without having to worry about encoding issues.

Django uses Unicode objects throughout the framework. Model objects are retrieved as Unicode objects, views interact with Unicode data, and templates are rendered as Unicode. Generally, you won't have to worry about making sure your encodings are right; things should just work.

Note that this has been a *very* high-level, dumbed down overview of Unicode objects, and you owe it to yourself to learn more about the topic. A good place to start ishttp://www.joelonsoftware.com/articles/Unicode.html .

For the `__unicode__()` 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: Apress>, <Publisher: O'Reilly>]
```

Make sure any model you define has a `__unicode__()` method – not only for your own convenience when using the interactive interpreter, but also because Django uses the output of `__unicode__()` in several places when it needs to display objects.

Finally, note that `__unicode__()` is a good example of adding *behavior* to models. A Django model describes more than the database table layout for an object; it also describes any functionality that object knows how to do. `__unicode__()` is one example of such functionality – a model knows how to display itself.

Inserting and Updating Data

You've already seen this done: to insert a row into your database, first create an instance of your model using keyword arguments, like so:

```
>>> p = Publisher(name='Apress',
...           address='2855 Telegraph Ave.',
...           city='Berkeley',
...           state_province='CA',
...           country='U.S.A.',
...           website='http://www.apress.com/')
```

As we noted above, this act of instantiating a model class does *not* touch the database. The record isn't saved into the database until you call save(), like this:

```
>>> p.save()
```

In SQL, this can roughly be translated into the following:

```
INSERT INTO books_publisher

    (name, address, city, state_province, country,
website)

VALUES

    ('Apress', '2855 Telegraph Ave.', 'Berkeley', 'CA',

     'U.S.A.', 'http://www.apress.com/');
```

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 = 'Apress Publishing'

>>> p.save()
```

The preceding save() statement will result in roughly the following SQL:

```
UPDATE books_publisher SET
    name = 'Apress Publishing',
    address = '2855 Telegraph Ave.',
    city = 'Berkeley',
    state_province = 'CA',
    country = 'U.S.A.',
    website = 'http://www.apress.com'
WHERE id = 52;
```

Yes, note that *all* of the fields will be updated, not just the ones that have been changed. Depending on your application, this may cause a race condition. See "Updating Multiple Objects in One Statement" below to find out how to execute this (slightly different) query:

```
UPDATE books_publisher SET
    name = 'Apress Publishing'
WHERE id=52;
```

Selecting Objects

Knowing how to create and update database records is essential, but chances are that the Web applications you'll build will be doing more querying of existing objects than creating new ones. We've already seen a way to retrieve *every* record for a given model:

```
>>> Publisher.objects.all()
[<Publisher: Apress>, <Publisher: O'Reilly>]
```

This roughly translates to this SQL:

```
SELECT id, name, address, city, state_province,
country, website
FROM books_publisher;
```

Note

Notice that Django doesn't use SELECT * when looking up data and instead lists all fields explicitly. This is by design: in certain circumstances SELECT * can be slower, and (more important) listing fields more closely follows one tenet of the Zen of Python: "Explicit is better than implicit."

For more on the Zen of Python, try typing import this at a Python prompt.

Let's take a close look at each part of this Publisher.objects.all() line:

- First, we have the model we defined, Publisher. No surprise here: when you want to look up data, you use the model for that data.
- Next, we have the objects attribute. This is called a *manager*. Managers are discussed in detail in Chapter 10. For now, all you need to know is that managers take care of all "table-level" operations on data including, most important, data lookup.

  All models automatically get a objects manager; you'll use it any time you want to look up model instances.
- Finally, we have all(). This is a method on the objects manager that returns all the rows in the database. Though this object *looks* like a list, it's actually a *QuerySet* – an object that represents a specific set of rows from the database. Appendix C deals with QuerySets in detail. For the rest of this chapter, we'll just treat them like the lists they emulate.

  Any database lookup is going to follow this general pattern – we'll call methods on the manager attached to the model we want to query against.

Filtering Data

Naturally, it's rare to want to select *everything* from a database at once; in most cases, you'll want to deal with a subset of your data. In the Django API, you can filter your data using the filter() method:

```
>>> Publisher.objects.filter(name='Apress')

[<Publisher: Apress>]
```

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_province,
country, website

FROM books_publisher

WHERE name = 'Apress';
```

You can pass multiple arguments into filter() to narrow down things further:

```
>>> Publisher.objects.filter(country="U.S.A.",
state_province="CA")

[<Publisher: Apress>]
```

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 books_publisher

WHERE country = 'U.S.A.'

AND state_province = 'CA';
```

Notice that by default the lookups use the SQL = operator to do exact match lookups. Other lookup types are available:

```
>>> Publisher.objects.filter(name__contains="press")

[<Publisher: Apress>]
```

That's a *double* underscore there between name and contains. Like Python itself, Django uses the double underscore to signal that something "magic" is happening – here, the __contains part gets translated by Django into a SQL LIKE statement:

```
SELECT id, name, address, city, state_province,
country, website

FROM books_publisher

WHERE name LIKE '%press%';
```

Many other types of lookups are available, including icontains (case-insensitive LIKE), startswith andendswith, and range (SQL BETWEEN queries). Appendix C describes all of these lookup types in detail.

Retrieving Single Objects

The filter() examples above all returned a QuerySet, which you can treat like a list. Sometimes it's more convenient to fetch only a single object, as opposed to a list. That's what the get() method is for:

```
>>> Publisher.objects.get(name="Apress")
```

```
<Publisher: Apress>
```

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="U.S.A.")

Traceback (most recent call last):

    ...

MultipleObjectsReturned: get() returned more than one
Publisher --

    it returned 2! Lookup parameters were {'country':
'U.S.A.'}
```

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.
```

The DoesNotExist exception is an attribute of the model's class – Publisher.DoesNotExist. In your applications, you'll want to trap these exceptions, like this:

```
try:

    p = Publisher.objects.get(name='Apress')

except Publisher.DoesNotExist:

    print "Apress isn't in the database yet."

else:

    print "Apress is in the database."
```

Ordering Data

As you play around with the previous examples, you might discover that the objects are being returned in a seemingly random order. You aren't imagining things; so far we haven't told the database how to order its results, so we're simply getting back data in some arbitrary order chosen by the database.

In your Django applications, you'll probably want to order your results according to a certain value – say, alphabetically. To do this, use the order_by() method:

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

[<Publisher: Apress>, <Publisher: O'Reilly>]
```

This doesn't look much different from the earlier all() example, but the SQL now includes a specific ordering:

```
SELECT id, name, address, city, state_province,
country, website

FROM books_publisher

ORDER BY name;
```

You can order by any field you like:

```
>>> Publisher.objects.order_by("address")

[<Publisher: O'Reilly>, <Publisher: Apress>]


>>> Publisher.objects.order_by("state_province")

[<Publisher: Apress>, <Publisher: O'Reilly>]
```

To order by multiple fields (where the second field is used to disambiguate ordering in cases where the first is the same), use multiple arguments:

```
>>> Publisher.objects.order_by("state_province",
"address")

 [<Publisher: Apress>, <Publisher: O'Reilly>]
```

You can also specify reverse ordering by prefixing the field name with a - (that's a minus character):

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

[<Publisher: O'Reilly>, <Publisher: Apress>]
```

While this flexibility is useful, using order_by() all the time can be quite repetitive. Most of the time you'll have a particular field you usually want to order by. In these cases, Django lets you specify a default ordering in the model:

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

    def __unicode__(self):
        return self.name

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

Here, we've introduced a new concept: the `class Meta`, which is a class that's embedded within the `Publisher` class definition (i.e., it's indented to be within `class Publisher`). You can use this `Meta` class on any model to specify various model-specific options. A full reference of `Meta` options is available in Appendix B, but for now, we're concerned with the `ordering` option. If you specify this, it tells Django that unless an ordering is given explicitly with `order_by()`, all `Publisher` objects should be ordered by the `name` field whenever they're retrieved with the Django database API.

Chaining Lookups

You've seen how you can filter data, and you've seen how you can order it. Often, of course, you'll need to do both. In these cases, you simply "chain" the lookups together:

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

[<Publisher: O'Reilly>, <Publisher: Apress>]
```

As you might expect, this translates to a SQL query with both a `WHERE` and an `ORDER BY`:

```
SELECT id, name, address, city, state_province,
country, website
```

```
FROM books_publisher

WHERE country = 'U.S.A'

ORDER BY name DESC;
```

Slicing Data

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

```
>>> Publisher.objects.order_by('name')[0]

<Publisher: Apress>
```

This translates roughly to:

```
SELECT id, name, address, city, state_province,
country, website

FROM books_publisher

ORDER BY name

LIMIT 1;
```

Similarly, you can retrieve a specific subset of data using Python's range-slicing syntax:

```
>>> Publisher.objects.order_by('name')[0:2]
```

This returns two objects, translating roughly to:

```
SELECT id, name, address, city, state_province,
country, website

FROM books_publisher

ORDER BY name

OFFSET 0 LIMIT 2;
```

Note that negative slicing is *not* supported:

```
>>> Publisher.objects.order_by('name')[-1]

Traceback (most recent call last):
```

```
  ...
AssertionError: Negative indexing is not supported.
```

This is easy to get around, though. Just change the order_by() statement, like this:

```
>>> Publisher.objects.order_by('-name')[0]
```

Updating Multiple Objects in One Statement

We pointed out in the "Inserting and Updating Data" section that the model save() method updates *all* columns in a row. Depending on your application, you may want to update only a subset of columns.

For example, let's say we want to update the Apress Publisher to change the name from 'Apress' to 'Apress Publishing'. Using save(), it would look something like this:

```
>>> p = Publisher.objects.get(name='Apress')
>>> p.name = 'Apress Publishing'
>>> p.save()
```

This roughly translates to the following SQL:

```
SELECT id, name, address, city, state_province,
country, website
FROM books_publisher
WHERE name = 'Apress';


UPDATE books_publisher SET
    name = 'Apress Publishing',
    address = '2855 Telegraph Ave.',
    city = 'Berkeley',
    state_province = 'CA',
    country = 'U.S.A.',
    website = 'http://www.apress.com'
```

```
WHERE id = 52;
```

(Note that this example assumes Apress has a publisher ID of 52.)

You can see in this example that Django's save() method sets *all* of the column values, not just the namecolumn. If you're in an environment where other columns of the database might change due to some other process, it's smarter to change *only* the column you need to change. To do this, use the update() method on QuerySet objects. Here's an example:

```
>>> Publisher.objects.filter(id=52).update(name='Apress
Publishing')
```

The SQL translation here is much more efficient and has no chance of race conditions:

```
UPDATE books_publisher

SET name = 'Apress Publishing'

WHERE id = 52;
```

The update() method works on any QuerySet, which means you can edit multiple records in bulk. Here's how you might change the country from 'U.S.A.' to USA in each Publisher record:

```
>>> Publisher.objects.all().update(country='USA')

2
```

The update() method has a return value – an integer representing how many records changed. In the above example, we got 2.

Deleting Objects

To delete an object from your database, simply call the object's delete() method:

```
>>> p = Publisher.objects.get(name="O'Reilly")

>>> p.delete()

>>> Publisher.objects.all()

[<Publisher: Apress Publishing>]
```

You can also delete objects in bulk by calling delete() on the result of any QuerySet. This is similar to theupdate() method we showed in the last section:

```
>>> Publisher.objects.filter(country='USA').delete()
```

```
>>> Publisher.objects.all().delete()

>>> Publisher.objects.all()

[]
```

Be careful deleting your data! As a precaution against deleting all of the data in a particular table, Django requires you to explicitly use all() if you want to delete *everything* in your table. For example, this won't work:

```
>>> Publisher.objects.delete()

Traceback (most recent call last):

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

AttributeError: 'Manager' object has no attribute
'delete'
```

But it'll work if you add the all() method:

```
>>> Publisher.objects.all().delete()
```

If you're just deleting a subset of your data, you don't need to include all(). To repeat a previous example:

```
>>> Publisher.objects.filter(country='USA').delete()
```