

Brad Dayley

24 Proven One-hour Lessons

Sams **Teach Yourself**

Django

in **24**
Hours

SAMS



Sams Teach Yourself Django in 24 Hours

Copyright © 2008 by Sams Publishing

All rights reserved. No part of this book shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from the publisher. No patent liability is assumed with respect to the use of the information contained herein. Although every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions. Nor is any liability assumed for damages resulting from the use of the information contained herein.

ISBN-13: 9780672329593

ISBN-10: 067232959X

Library of Congress Cataloging-in-Publication Data:

Dayley, Brad.

Sams teach yourself Django in 24 hours / Brad Dayley.

p. cm.

ISBN 978-0-672-32959-3 (pbk.)

1. Web site development. 2. Django (Electronic resource) 3. Python (Computer program language) I. Title.

TK5105.888.D397 2008

006.7'6—dc22

2008001956

Printed in the United States of America

First Printing January 2008

Trademarks

All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. Sams Publishing cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

Warning and Disclaimer

Every effort has been made to make this book as complete and accurate as possible, but no warranty or fitness is implied. The information provided is on an "as is" basis. The author and the publisher shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book or from the use of the CD or programs accompanying it.

Bulk Sales

Sams Publishing offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales. For more information, please contact

U.S. Corporate and Government Sales

1-800-382-3419

corpsales@pearsontechgroup.com

For sales outside of the U.S., please contact

International Sales

international@pearsoned.com

This Book Is Safari Enabled

The Safari® Enabled icon on the cover of your favorite technology book means the book is available through Safari Bookshelf. When you buy this book, you get free access to the online edition for 45 days.

Safari Bookshelf is an electronic reference library that lets you easily search thousands of technical books, find code samples, download chapters, and access technical information whenever and wherever you need it.

To gain 45-day Safari Enabled access to this book:

- ▶ Go to <http://www.informit.com/onlineedition>.
- ▶ Complete the brief registration form.
- ▶ Enter the coupon code ERBC-2QCB-ZL1W-YDBJ-YZ42.

If you have difficulty registering on Safari Bookshelf or accessing the online edition, please email customer-service@safaribooksonline.com.

Editor-in-Chief

Mark Taub

Development Editor

Songlin Qiu

Managing Editor

Gina Kanouse

Project Editor

Jovana San Nicolas-Shirley

Copy Editor

Gayle Johnson

Indexer

Cheryl Lenser

Proofreader

Anne Goebel

Technical Editor

Timothy Boronczyk

Publishing Coordinator

Vanessa Evans

Book Designer

Gary Adair

Compositor

Nonie Ratcliff

Introduction

I have been working with the Django framework for about a year and a half, and I love it. Every so often you run into ideas that make absolute, complete sense, and Django is one of those. The folks at Django seem to be bent on making it the most elegant web framework available, and so far they are doing a great job.

This was a tough book to write. The Django framework is simple to implement, but you can accomplish so much with it. The format of this book is Teach Yourself in 24 Hours. The idea is that after spending 24 hours with this book and a Django installation, you should have a pretty good idea of how to use Django to build a full-featured production website.

Throughout this book, I use a fictitious website project called iFriends to illustrate the building blocks of a Django-powered website. The book has several “Try It Yourself” sections that take you through specific tasks of building the iFriends website. Actually *do* the “Try It Yourself” sections. They will help everything else make a lot more sense. They build on each other, so if you skip one, future “Try It Yourself” sections may not work properly.

When you have finished the “Try It Yourself” sections, you will have a mostly functional website. You should easily have enough skills by then that you could tweak and finish the website on your own in only a few hours if you wanted to. There just wasn’t enough room in the book to finish every component. I felt it was much more important to cover the topics that I did.

I do have one disclaimer: There is absolutely no CSS code in my HTML template examples. I would much rather have used CSS code to format my HTML templates than the classic HTML tags (some of them deprecated) that I used. I chose not to include CSS for two important reasons. The first reason is room. Adding CSS files to all the examples would have taken quite a bit more room, which I didn’t have. The second reason is that this book is designed for Python programmers as well as HTML programmers. Using CSS for someone who is not as familiar with it could provide a distraction. This book is about learning to implement the Django framework. CSS programming techniques belong in a different book.

When designing the content for this book, I tried to come up with the most relevant way to present the Django framework that will actually help programmers develop websites that are pertinent to real-world needs. I know that a few components and concepts have been left out. I welcome your comments and any suggestions on things that you feel need to be added to this book. If I get a chance, I will try to incorporate them into future revisions of the book. You can email any queries or suggestions to dayleybooks@yahoo.com.

I hope you enjoy the Django framework as much as I have and that the concepts in this book prove useful to you.

Who Should Read This Book

This book should be read by anyone who is developing or even considering developing websites. The Django framework saves web developers a lot of time and headaches. This book is designed for website developers who have at least some familiarity with the Python programming language. Don't worry if you are not very familiar with Python. You should be able to pick up on what is going on with a few visits to www.python.org.

How This Book Is Organized

This book is organized into four parts that help you quickly navigate the Django framework so that you will have the knowledge necessary to leverage the framework to build production websites. I tried to design the book to start slowly so that you will be able to build a good foundation for the Django framework. Then, as the hours (chapters) progress, the book delves deeper into different aspects of the Django framework.

- ▶ Part I, “Creating the Website Framework,” covers the basics of installing, configuring, and using the Django framework to build basic websites. You are introduced to the model, template, and view concepts that Django uses to implement websites.
- ▶ Part II, “Implementing the Website Interface,” covers building templates and views to build web pages. You will learn how to use templates and views to store, access, and retrieve data that is stored in the website's database.
- ▶ Part III, “Implementing a Full-Featured Website,” covers adding authentication, cookie handling, and other features necessary to implement a full production website. You will learn how to create users and groups and how to assign permissions to specific data.
- ▶ Part IV, “Implementing Advanced Website Components,” covers some of the advanced features of the Django framework that you will likely want to implement in production websites. You will learn how to implement middleware to enable advanced request and response handlers. You will also learn how to implement localized strings to add multiple-language capability to the website, implement caching to improve website performance, and deploy a Django website.

How to Use This Book

The *Teach Yourself in 24 Hours* series has several unique elements that will help you as you are trying to learn the Django framework. Throughout the book, I use the following elements to draw attention to specific concepts:

This element provides information about slightly off-topic tangents that may be beneficial to you but that are not necessarily directly related to the current section.

***Did you
Know?***

This element provides information that is directly related to the current section but that does not necessarily flow with the text. It discusses what is happening in the background or points that you may not easily pick up on but that are important.

***By the
Way***

This element notes important things that you need to know before proceeding through the book. It is important to read these sections to avoid problems with your website.

***Watch
Out!***

The “Try It Yourself” sections are designed to take you through the steps of actually performing the tasks that you have been reading about. Do not skip these sections. They usually provide additional information about the topic and are a great chance to practice the concepts.

At the end of each hour, you will find the following sections that are designed to help you solidify what you have read:

- ▶ The “Q&A” section poses questions and gives answers on concepts that are related to the hour but that fall outside what is covered in the book.
- ▶ The “Quiz” section provides questions and answers about the topics covered in each hour.
- ▶ The “Exercises” section lists activities that you can do to practice what you have learned during the hour. These exercises are a great way to strike out on your own a bit and get more confident with Django.

HOUR 2

Creating Your First Website

What You'll Learn in This Hour:

- ▶ How to begin creating a Django project
- ▶ How to start and stop the built-in web server
- ▶ The steps to configure Django to access the database
- ▶ How to create and install an application
- ▶ The steps to apply a model to an application
- ▶ The steps to activate a model in Django
- ▶ How to configure Django to accept specific URL requests
- ▶ How to create a simple view for a web browser

In Hour 1, “Understanding Django,” you learned some of the basics about the Django framework. This hour guides you through the steps of creating a functional website called iFriends. Although this website will be basic, it will be the basis for future hours to build on as you are guided through the various aspects of the Django framework.

Creating a Django Project

Let's begin the process of creating a working website by creating a Django project. A Django project is a collection of settings that define a specific instance of Django. These settings include things such as database configuration, URL configuration, and other options that you will learn about as the hours tick by.

▼ Try It Yourself

Create Your First Django Project

Creating a Django project is relatively simple to do from the command prompt. In this section, you create a project called iFriends.

1. From a command prompt, change to the directory where you want to store the code for the iFriends project.
2. Create a directory called iFriends. This will be the root directory for the iFriends project.
3. Change to the iFriends directory.
4. Type the following command to create the iFriends project:

```
python django-admin.py startproject iFriends
```

Watch Out!

Because the project will act as a Python package, avoid using a project name that conflicts with any existing built-in Python packages. The documentation for built-in Python packages can be found at <http://www.python.org>.

By the Way

There is no need to put your project code in a directory in the web server's document base. The Django framework will be responsible for executing the code. In fact, it is a much better idea to store the code somewhere outside the web server's root. That way your code will be protected from being accessed directly from a web browser.

The `startproject` command first creates a directory called iFriends, and then it stores the basic set of Python files that are needed to begin the project in the iFriends directory. The `startproject` command creates the following files:

- ▶ `__init__.py` is an empty file that tells Python that the website directory should be treated as a Python package.
 - ▶ `manage.py` is the command-line utility that allows the administrator to start and manage the Django project.
 - ▶ `settings.py` is the configuration file that controls the behavior of the Django project.
 - ▶ `urls.py` is a Python file that defines the syntax and configures the behavior of the URLs that will be used to access the website.
- ▼

The basic purpose of these files is to set up a Python package that Django can use to define the website's structure and behavior. We will discuss these files a bit more in this hour and in subsequent hours as the website gets increasingly complex.

Starting the Development Server

After you have created the Django project, you should be able to start the development server to test it. The development server is a lightweight web server that is included with the Django project. It lets you develop and test your website without having to deal with all the configuration and management issues of a production web server.

Try It Yourself

Start the Development Server

In this section, you learn how to start the development server.

1. From a command prompt, change to the root directory for the iFriends project.
2. Enter the following command to start the development server, as shown in Figure 2.1:

```
python manage.py runserver
```



```
Command Prompt2 - manage.py runserver
C:\websites>cd iFriends
C:\websites\iFriends>manage.py runserver
Validating models...
0 errors found.

Django version 0.96-pre, using settings 'iFriends.settings'
Development server is running at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

FIGURE 2.1
Starting the development server from a command line.

The `manage.py` utility is copied into the root of your project by the `createproject` command discussed earlier in this hour. The `manage.py` utility first validates the project and reports any errors. If no critical errors are encountered, you are notified that the development server is running at `http://127.0.0.1:8000/`.

**By the
Way**

- ▼
3. Verify that the development server is working properly by opening a web browser and entering the following address:

`http://127.0.0.1:8000/`

If the development server starts properly (and you haven't changed the debug setting), you should see a page similar to the one shown in Figure 2.2.

FIGURE 2.2
Initial browser view of a Django website.



By the Way

You can tell the development server to use a different port than 8000 if that port is already being used by adding the port to the command line. The following example shows the syntax for configuring the development server to run on port 8008:

```
manage.py runserver 8008
```

Did you Know?

To stop the development server, press Ctrl+Break or Ctrl+C.



Configuring the Database

After you have verified that you can start and stop the development server, it is time to configure access to the database. This section takes you through the process of creating and configuring access to the database that will be used in the sample project.

By the Way

Django can dynamically serve web pages without using a database to store information. However, one of the best aspects of Django is its ability to implement database-backed websites.

Configuring the database involves three major steps. The first is to create the database and assign rights. The second is to modify the `settings.py` file to specify the database type, name, location, and access credentials. The third step is to synchronize the Django project with the database to create the initial tables necessary for the Django engine.

Django supports several different types of database engines. The project used in this book uses a MySQL database. This section assumes that you have already installed, configured, and started a database engine and that it is accessible from the development server.

The MySQL database does not allow you to use case sensitive names when creating tables. If you want to define objects in your project that have uppercase characters, then you will need to turn off case sensitivity in the Django framework by using the following setting in the `<django installation path>/django/db/backends/___init___py` file:

```
uses_case_insensitive_names = True
```

**Watch
Out!**

Try It Yourself

Create the Database and Grant Rights

This section takes you through the steps of creating the database, creating an admin user, and granting rights from your SQL database command console. You will also modify the `uses_case_insensitive_names` setting in the Django framework so that you can name objects with uppercase characters. This step will be critical for some of the other Try it Yourself sections.

1. From your SQL database command console, enter the following command to create a database called `iFriends`:

```
CREATE DATABASE iFriendsDB;
```

2. Enter the following command to begin using the `iFriends` database:

```
USE iFriendsDB;
```

3. Enter the following command to create an administrative user named `dAdmin` with a password called `test`:

```
CREATE USER 'dAdmin'@'localhost' IDENTIFIED BY 'test';
```

- ▼
4. Enter the following command to grant all rights on the iFriends database to the dAdmin user:

```
GRANT ALL ON *.* TO 'dAdmin'@'localhost';
```

By the Way

If your database engine has a graphical interface that allows you to manage databases and users, you can use that interface as well to create the database and admin user and to assign rights.

5. Open the `<django installation path>/django/db/backends/___init___py` file in an editor.
6. Add the following setting to the file to disable case sensitivity for the MySQL database:

```
uses_case_insensitive_names = True
```

- ▲
7. Save the `___init___py` file.
-

Configuring Database Access in settings.py

After the database has been created and a user account set up for Django, you need to configure the `settings.py` file in your Django project to access that database. Each Django project has its own `settings.py` file. The `settings.py` file is a Python script that configures various project settings.

Django uses the following settings in the `settings.py` file to control access to the database:

- ▶ `DATABASE_ENGINE` is the type of database engine. Django accepts `postgresql_psycopg2`, `postgresql`, `mysql`, `mysql_old`, `sqlite3`, and `ado_mssql`.
- ▶ `DATABASE_NAME` is the name of the database. For SQLite, you need to specify the full path.
- ▶ `DATABASE_USER` is the user account to use when connecting to the database. No user is used with SQLite.
- ▶ `DATABASE_PASSWORD` is the password for `DATABASE_USER`. No password is used with SQLite.
- ▶ `DATABASE_HOST` is the host on which the database is stored. This can be left empty for localhost. No host is specified with SQLite.

- ▶ `DATABASE_PORT` is the port to use when connecting to the database. This can be left empty for the default port. No port is specified with SQLite.

Try It Yourself

Configure Django to Access the iFriends Database

The following section takes you through the steps to modify the settings in the `settings.py` file for the database and user created in the preceding section (a MySQL database named `iFriendsDB`, and a username of `dAdmin` with a password of `test` running on the localhost and default port). Open the `iFriends/settings.py` file in a text editor.

1. Find the `DATABASE_ENGINE` setting, and change the value to the following:

```
DATABASE_ENGINE = 'mysql'
```

If you have elected to use an SQL database other than MySQL, you need to use that database type here instead of `mysql`.

**By the
Way**

2. Change the value of the `DATABASE_NAME` setting to the following:

```
DATABASE_NAME = 'iFriendsDB'
```

3. Change the value of the `DATABASE_USER` setting to the following:

```
DATABASE_USER = 'dAdmin'
```

4. Change the value of the `DATABASE_PASSWORD` setting to the following:

```
DATABASE_PASSWORD = 'test'
```

5. Verify that the `DATABASE_HOST` and `DATABASE_PORT` settings have no value:

```
DATABASE_HOST = ''  
DATABASE_PORT = ''
```

When the `DATABASE_HOST` and `DATABASE_PORT` settings are left blank, they default to the localhost and default port. If the database is on a remote server or is running on a nondefault port, these options need to be set accordingly.

**By the
Way**



Synchronizing the Project to the Database

After you have configured access to the database in the `settings.py` file, you can synchronize your project to the database. Django's synchronization process creates the tables necessary in the database to support your project.

The tables are created based on what applications are specified in the `INSTALLED_APPS` setting of the `settings.py` file. The following are the default settings already specified in the `INSTALLED_APPS` setting:

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

The following list describes the default applications that get installed in the Django project:

- ▶ `django.contrib.auth` is the default authentication system included with Django.
- ▶ `django.contrib.contenttypes` is the framework of types of content.
- ▶ `django.contrib.sessions` is the framework used to manage sessions.
- ▶ `django.contrib.sites` is the framework used to manage multiple sites using a single Django installation.



Try It Yourself

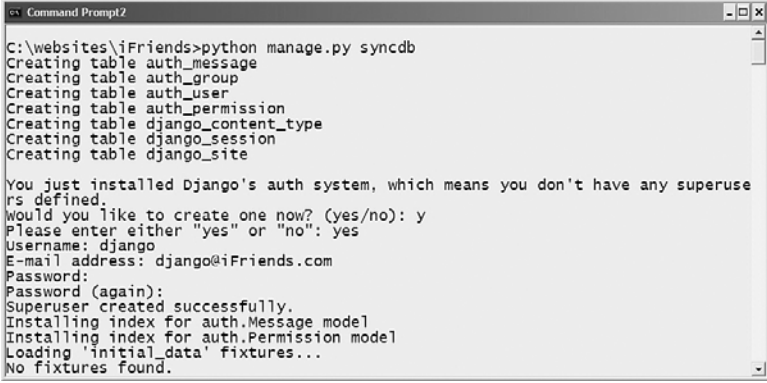
Synchronize the iFriends Project to the iFriends Database

This section guides you through the steps to synchronize the Django project to the database. During the process, Django creates the default tables and prompts you to input the name, email address, and password for a website administration account. The username and password you specify allow you to access the Django authentication system.

1. Make certain that the development server has stopped by pressing `Ctrl+Break` from the console prompt.
2. Change to the root directory of the iFriends project.
3. Enter the following command at the console prompt to begin the synchronization, as shown in Figure 2.3:

```
python manage.py syncdb
```





```
C:\websites\iFriends>python manage.py syncdb
Creating table auth_message
Creating table auth_group
Creating table auth_user
Creating table auth_permission
Creating table django_content_type
Creating table django_session
Creating table django_site

You just installed Django's auth system, which means you don't have any superusers defined.
Would you like to create one now? (yes/no): y
Please enter either "yes" or "no": yes
Username: django
E-mail address: django@iFriends.com
Password:
Password (again):
Superuser created successfully.
Installing index for auth.Message model
Installing index for auth.Permission model
Loading 'initial_data' fixtures...
No fixtures found.
```

FIGURE 2.3
Synchronizing the initial Django project with the database from a command line.

4. At the prompt, enter a username for the website administrator's account.
5. At the prompt, enter a password for the website administrator's account.

The database now has the appropriate tables configured to allow Django to use its authentication, content, session, and site frameworks correctly.

Installing an Application

After you have configured and synchronized the database, you can begin installing applications in it. Installing applications is simply a matter of creating an application directory, defining a model, and then activating the application so that Django can access it in the database.

Try It Yourself

Create Your First Application

The first application you will create is an application called People, which will be used to keep track of the individuals who use the website.

1. From a console prompt, change to the root directory of the iFriends project.
2. Enter the following command to create a blank application called People:

```
python manage.py startapp People
```

▼ The `startapp` command creates a `People` directory within the `iFriends` directory and then populates it with the following files:

- ▶ `__init__.py` is a necessary file for the application to be used as a Python package.
- ▶ `models.py` contains the Python code that defines the model.
- ▶ `views.py` contains the Python code that defines the views for the model.

▲ The files in the application directory define how information for the application will be stored and accessed in the database. They also define how information in the model will be viewed when accessed from the website.

Creating a Model

After the application has been created, you need to create a model for the data that will be stored in the application. A model is simply a definition of the classes, attributes, and relationships of objects in the application.

To create a model, you need to modify the `models.py` file located in the application directory. The `models.py` file is a Python script that is used to define the tables that will be added to the database to store objects in the model.

The `models.py` file initially has only one line, which imports the `models` object from the `django.db` package. To define the model, you need to define one or more classes. Each class represents an object type in the database.

▼ Try It Yourself

Create a Model for the People Application

In this section, you create the class `Person` in the `People` model by modifying the Python script, `models.py`, for the `People` application. Initially, the script is blank. This section takes you through adding the Python code to define classes in the model.

1. Open the `iFriends\People\models.py` file in an editor.
2. Add the following line of code to the file to import the Django `models` package into the application:

▼

```
from django.db import models
```

3. Add the following code snippet to define the Person class with name, email, headshot, and text attributes:

```
class Person(models.Model):
    name = models.CharField('name', maxlength=200)
    email = models.EmailField('Email', blank=True)
    headshot = models.ImageField(upload_to='img', blank=True)
    text = models.TextField('Desc', maxlength=500, blank=True)
    def __str__(self):
        return '%s' % (self.name)
```

4. Save the file.

Listing 2.1 shows the complete code for the `iFriends\People\models.py` file.

LISTING 2.1 Full Contents of the `iFriends\People\models.py` File

```
from django.db import models

class Person(models.Model):
    name = models.CharField('name', max_length=200)
    text = models.TextField('Desc', max_length=500, blank=True)

    def __str__(self):
        return '%s' % (self.name)
```

The definition for `__str__` defines a string representation of the object that can be used in views or other Python scripts. Django uses the `__str__` method in several places to display objects as well.

**By the
Way**

Try It Yourself

Activate the Person Model

This section takes you through the process of activating the Person model by adding it to the `INSTALLED_APPS` setting in the `settings.py` file and then synchronizing the database.

1. Open the `iFriends\settings.py` file in an editor.
2. Find the `INSTALLED_APPS` setting, and add the `iFriends.People` application to it, as shown in the following snippet:

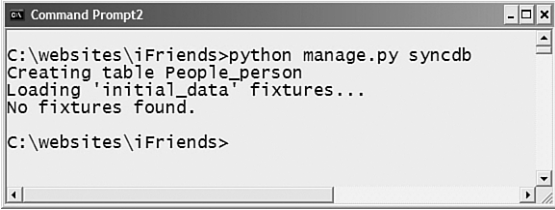
```
INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    'iFriends.People',
)
```


3. Save the file.
4. Synchronize the People application into the iFriends database by using the following command from the root of the iFriends project, as shown in Figure 2.4:

```
python manage.py syncdb
```

FIGURE 2.4

Synchronizing the new People application with the database from a command line.



```
Command Prompt2
C:\websites\iFriends>python manage.py syncdb
Creating table People_person
Loading 'initial_data' fixtures...
No fixtures found.
C:\websites\iFriends>
```

The syncdb command creates the necessary tables in the iFriends database for the People application. The model is now active, and data can be added to and retrieved from the database using Django at this point.

Adding Data Using the API

This section briefly describes how to use the Django shell interface and database API to quickly add a single Person object to the People table. The Django shell is a Python shell that gives you access to the database API included with Django. The database API is a set of Python methods that allow you to access the project database from the data model.

Try It Yourself

Add a Person Object to the iFriends Database

Open the Django shell, and follow these steps to add yourself as a Person object in the People model of the iFriends database.

1. From a console prompt, change to the root directory of the iFriends project.
2. Enter the following command to invoke the Django shell:

```
python manage.py shell
```

3. From the shell prompt, enter the following to import the Person class from the People package:

```
from iFriends.People.models import Person
```

4. Enter the following command to create a Person object named p:

```
p = Person(name="<your name>", email="<your eMail>")
```

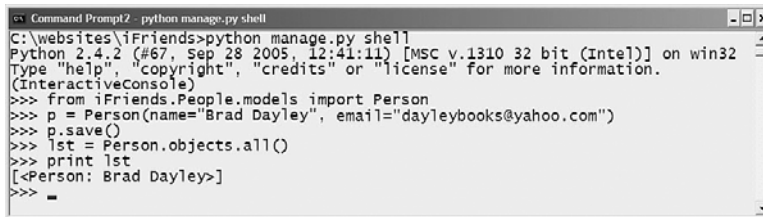
5. Save the Person object you just created using the following command:

```
p.save()
```

6. Verify that the object was created by using the Person.objects.all() function, which returns a list of all Person objects, and then print the list:

```
lst = Person.objects.all()
print lst
```

Figure 2.5 shows these commands.



```
Command Prompt2 - python manage.py shell
C:\websites\iFriends>python manage.py shell
Python 2.4.2 (#67, Sep 28 2005, 12:41:11) [MSC v.1310 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>> from iFriends.People.models import Person
>>> p = Person(name="Brad Dayley", email="dayleybooks@yahoo.com")
>>> p.save()
>>> lst = Person.objects.all()
>>> print lst
[<Person: Brad Dayley>]
>>> _
```

FIGURE 2.5

Using the Python shell to add an object to the database.

A Person object has now been created in the iFriends database. We will discuss accessing the database and using the database API in more depth later.

Setting Up the URLConf File

This section discusses configuring the URLConf file to define how installed applications are accessed from the web. The URLConf file is a Python script that allows you to define specific views that are accessed based on the URL that is sent by the web browser. When the Django server receives an URL request, it parses the request based on the patterns that are contained in the URLConf file. The parsed request is translated into a specific Python function that is executed in the views.py file, discussed in a moment.

**By the
Way**

The location of the URLConf file is defined by the `ROOT_URLCONF` setting in the `settings.py` file. The default location is the name of the project's root directory. In the case of the `iFriends` project, the value of `ROOT_URLCONF` would be set to the following value, where `'iFriends.urls'` equates to `iFriends/urls.py`:

```
ROOT_URLCONF = 'iFriends.urls'
```

**Try It Yourself****Add an URL Pattern to Use for a People View**

In this example, you set up a simple URL pattern for the `People` application by modifying the `urlpatterns` setting in the `iFriends/urls.py` file.

1. Open the `iFriends\urls.py` file in an editor.
2. Find the `urlpatterns` setting, and add the `iFriends.People.views.index` pattern to it:

```
urlpatterns = patterns('',  
    (r'^People/$', 'iFriends.People.views.index')  
)
```

3. Save the file.

**By the
Way**

In the preceding code snippet, `iFriends.People.views.index` refers to the `index()` function located in the `iFriends/People/views.py` file, which is discussed next.

**Creating a Simple View**

After you have configured the URLConf file, you need to add the views to the application. The application's views are stored as functions in the `views.py` file in the application directory. When the Django server receives an URL request, it parses the request based on the patterns that are contained in the URLConf file and determines which function to execute to generate the web view.

Try It Yourself

Create the Index View for the People Application

This section guides you through the steps of creating an index view stub for the People application in the iFriends project. After the view is created, you start the development server and view the web page that is generated.

1. Open the `iFriends/People/views.py` file in an editor. The `views.py` file is empty at first.
2. Use the editor to add the following code snippet to the file:

```
from django.shortcuts import HttpResponseRedirect
from iFriends.People.models import Person

def index(request):
    html = "<H1>People</H1><HR>"
    return HttpResponseRedirect(html)
```

3. Save the file.
4. From a command prompt, change to the root directory for the iFriends project.
5. Enter the following command to start the development server:
`python manage.py runserver`
6. Access the `http://127.0.0.1:8000/People` URL. You should see a web page similar to the one shown in Figure 2.6.

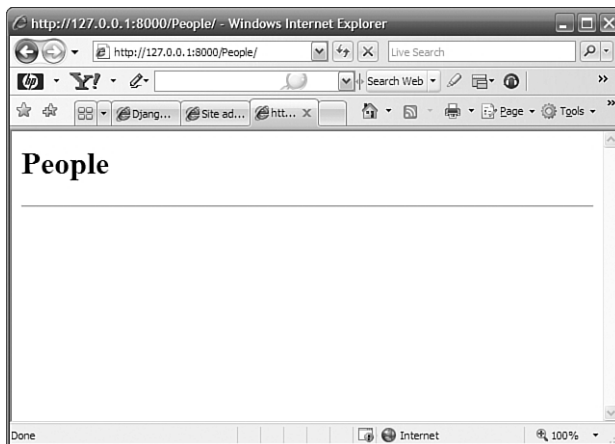


FIGURE 2.6 Accessing a custom index view in the Django project from a web browser.

Summary

In this hour, you created a Django project called iFriends. You configured access to a MySQL database for the project. You created an application called People, added a Person class, and populated the database with one Person object. You then configured the URL behavior to support an index view and added the necessary code in that view to display a list of objects in the Person class.

The steps you took during this hour helped demonstrate how easy it is to set up a website using the Django framework. Subsequent hours will build on this framework to implement a full-featured website.

Q&A

Q. *How do I modify a model after it has been synced to the database?*

A. Currently, Django cannot update models reliably. The safest and easiest method to modify an existing model is to make changes to the model and then delete all tables related to the model in the database using the SQL drop command. Finally, use the syncdb command to sync the model with the database.

Q. *Is there a way to check for errors in my model before trying to sync to the database?*

A. Django has a utility to validate the contents of models before syncing to the database. From the root directory of the project, enter `python manage.py validate`. The validate utility checks the model's syntax and logic and reports any problems.

Workshop

The workshop consists of a set of questions and answers designed to solidify your understanding of the material covered in this hour. Try answering the questions before looking at the answers.

Quiz

1. What file contains the information that Django uses to connect to the database?
2. What default file contains the configuration that Django uses to parse the location URLs?
3. What file contains the code that implements an index view for the People application in the iFriends project?

Quiz Answers

1. settings.py
2. urls.py
3. iFriends/People/views.py

Exercises

Try your hand at creating and activating a simple application. Create an application called Comments. Add a class to the model called Note, with two CharField attributes called Title and Text. Then activate the model by adding it to the INSTALLED_APPS setting in the settings.py file. Synchronize the model to the database. Test your application by adding an object to the database using the Django shell.

Index

Symbols

\$ (dollar sign), in URL patterns, 110
| (pipe), in filters, 156
__init__() function, 385
__init__.py file, 20, 28
__str__ method, 29

A

access

- limiting to one site, 441
- from multiple sites, defining models, 439-440

accessing

- add forms, 13
- admin interface, enabling in URLconf file, 38-40
- change forms, 12
- change list view, 12
- databases, configuring in settings.py file, 24-25
- languages in templates, 411
- list items in reverse order, 140
- model view, 11

- normalized data in forms, 215-216
- objects with HTML templates, 123-126
- settings.py file, 9
- sites from views, 442

activating models, 29-30

- in admin interface, 40-42

add forms, explained, 13-14

add() function, 309

admin change list view. See change list view

Admin class, 41

admin interface

- adding to URLconf file, 38-40
- change list view, explained, 12-13
- contents of, 365-366
- customizing
 - admin templates, overriding, 366-368, 372-377
 - admin templates, overriding block tags in, 373-374
 - admin URL patterns, overriding, 378-379
 - admin views, creating custom, 368-372
- explained, 11

admin interface

- form views, explained, 12-14
- installing, 37-38
- model change list view, customizing, 347-356
- model forms, customizing, 356-359
- model view, explained, 11-12
- models, activating, 40-42
- multiple models, viewing inline, 359-362
- navigating, 42-43
- navigation links in, 45
- objects, adding/modifying, 43-45
- admin media files, adding to servers, 467**
- admin model view. See model view**
- admin template directory, path for, 118**
- admin templates**
 - block tags, overriding, 373-374
 - overriding, 366-368, 372-377
 - storage location, 365
- admin URL patterns**
 - overriding, 378-379
 - storage location, 365
- admin views**
 - creating custom, 368-372
 - storage location, 366
- all() function, 86-88**
- allow_empty argument (generic views), 233**
- allow_future argument (generic views), 233**
- anonymous users, 297**
- Apache**
 - deploying Django with mod_python, 465
 - implementing multiple Django installations, 469-470
 - web server page, 477
- application level, overriding admin templates, 372-377**
- applications. See also middleware applications**
 - default, list of, 26
 - installing, 27-28
 - activating models, 29-30
 - creating models, 28-29
 - loading templates from, 269-270
 - views, creating, 32-33
- archive_day view, 247**
- archive_detail view, 248**
- archive_index view, 245**
- archive_month view, 246**
- archive_today view, 248**
- archive_week view, 247**
- archive_year view, 246**
- argument names in expressions, 73-74**
- arguments. See also options**
 - for date and time field types, 47
 - for file fields, 47
 - for ForeignKey field, 56
 - for form fields, 188
 - for image fields, 47
 - for ManyToMany field, 57
 - passing
 - field options as, 50-52
 - to filters, 156
 - positional arguments, 73
 - for text fields, 46
 - for URLfields, 48
 - to view function, expression values as, 72-73
- arguments dictionary for generic views, 232-234**
- assigning content to sites, 439-440**
- as_p() function, 191**
- as_table() function, 190**
- as_ul() function, 190-191**
- attributes of HttpRequest class, 64-65**
- authenticate() function, 314**
- authentication**
 - generic views, limiting access to, 330
 - groups, creating, 304-307
 - login process, 313-318
 - logout process, 318-319
 - permissions, setting, 307-310
 - permissions, verifying
 - in templates, 326-330
 - in views, 325-326
 - User objects
 - anonymous users, 297
 - changing passwords in views, 298
 - creating in views, 297-304
 - explained, 295-296
 - fields of, 296
 - verifying
 - in templates, 321-325
 - in views, 320
- AuthenticationMiddleware application, 403**
- auto_now_add argument (date and time field types), 47**
- auto_now argument (date and time field types), 47**
- availability of cookies, verifying, 336-339**

B

- backends**
 - configuring caching, 451-452
 - database backend, 452-453
 - dummy backend, 454
 - file system backend, 453
 - local memory backend, 454
 - memcached backend, 455
 - simple backend, 454
 - database backend, caching, 455

basic permissions, setting, 307-308

blank field option, 50-51

block tags, 129

creating custom, 277-278

overriding in admin

templates, 373-374

blocktrans tag, 410

BooleanField object, 187, 481

bound forms, 188-189

built-in middleware applications,

list of, 402-403

built-in Python packages, 20

C

cache API, caching and retrieving specific objects, 459-460

cached pages

allowing to vary based on

headers, 461-462

controlling with cache-control

header, 462-463

CacheMiddleware application, 403

CACHE_MIDDLEWARE_

ANONYMOUS_ONLY, 456

CACHE_MIDDLEWARE_KEY_

PREFIX, 456

CACHE_MIDDLEWARE_SECONDS, 456

caching

configuring caching backends, 451-452

database backend,

452-453

dummy backend, 454

file system backend, 453

local memory

backend, 454

memcached backend, 455

simple backend, 454

implementing

per-object caches,

458-459

per-site caching, 456-457

per-view caches, 457-458

at site level, 457

optimizing Django

deployment, 472

specific objects, using

cache API, 459-460

upstream caching, managing,

460-461

calling view functions, 113

capfirst filter, 157

case sensitivity

disabling, 23

in MySQL databases, 23

of URL dispatcher

algorithm, 69

case of text, changing, 157

chaining QuerySets, 93-94

change forms, explained, 12-13

change list view

customizing, 347-348

date hierarchy, adding,

352-353

fields, displaying, 348-349

filters, adding, 350-351

ordering objects in, 353

search fields, adding,

354-356

explained, 12-13

changefreq member (sitemap

classes), 425

CharField object, 187, 481-482

CharField type, 46

check_password() function, 314

child templates, 129-133

ChoiceField object, 187, 482

choices field option, 51

class relationships. See

relationships

cleaned_data() function,

215-216

clear() function, 309

cloning objects in database, 90

commands in manage.py utility, 10

CommonMiddleware application, 402-403

compilation functions, creating

custom tags, 276

compile-messages.py utility, 414

compiling message files, 414

conditions. See if logic

configuring

caching backends, 451-452

database backend,

452-453

dummy backend, 454

file system backend, 453

local memory

backend, 454

memcached backend, 455

simple backend, 454

databases, 22-23

granting rights, 23-24

settings.py file, setting

database access, 24-25

synchronizing project to database, 26-27

Django projects, 466

eggs with mod_python, 471

session manager, 334-336

with settings.py file, 9-10

template directory, 118

URLConf file, 31-32, 63-64

URLpatterns, 68-70

contains (field lookup type), 92

content

assigning to sites, 439-440

modifying based on site,

443-448

context_processors argument

(generic views), 233

context variables in generic

views, 235-237

contexts (template)

RequestContext objects as,

289-290

retrieving/setting variables in,

278-284

cookies

cookies

- drawbacks to, 334
- setting/retrieving, 344
- verifying availability of, 336-339

core field option, 51

create() function, 82

create_object view, 254-258

create_user() function,

- passwords in, 297

CurrentSiteManager class,

- 442-443

custom admin views, creating,

- 368-372

custom filters

- creating, 272-275
- extending templates, 270-272

custom permissions, setting, 308

custom tags

- creating, 275-277
 - block tags, 277-278
 - retrieving/setting variables in context, 278-284
 - simple_tag filter, 277

- extending templates, 270-272

custom variables, passing to views, 232

customizing

- admin interface
 - admin templates, overriding, 366-368, 372-377
 - admin templates, overriding block tags in, 373-374
 - admin URL patterns, overriding, 378-379
 - admin views, creating custom, 368-372

- forms, 200-205

- in templates, 202-203

- widgets, changing, 200-202

- model change list view, 347-348

- date hierarchy, adding, 352-353

- fields, displaying, 348-349

- filters, adding, 350-351

- ordering objects in, 353

- search fields, adding, 354-356

- model forms, 356-359

cycle logic, adding to tables in templates, 151

D

data, adding with database API, 30-31

data models. See models

data validation, 46

database API, adding data with, 30-31

database-abstraction API, 81

objects

- adding to database, 81-85

- deleting from database, 90

- retrieving from database, 86-89

- updating in database, 89-90

- queries, performing in database, 90-97

QuerySets

- chaining, 93-94

- ordering objects in, 94-95

- retrieving from database, 90-97

- views, displaying database data in, 97-100

database backend

- configuring caching, 452-453
- enabling for caching, 455

databases

- adding objects to, 81-85

- configuring, 22-23

- granting rights, 23-24

- settings.py file, setting database access, 24-25

- synchronizing project to database, 26-27

- deleting objects from, 90

- displaying data in views, 97-100

- performing queries in, 90-97

retrieving

- objects from, 86-89

- QuerySets from, 90-97

- saving form data to, 219-228

- SQL databases as Django

- prerequisite, 14

- synchronizing, 179

- updating objects in, 89-90

DATABASE_ENGINE setting (settings.py file), 24

DATABASE_HOST setting (settings.py file), 24

DATABASE_NAME setting (settings.py file), 24

DATABASE_PASSWORD setting (settings.py file), 24

DATABASE_PORT setting (settings.py file), 25

DATABASE_USER setting (settings.py file), 24

date and time fields, 46-47

date-based objects, displaying in generic views, 245-254

date_field argument (generic views), 233

date filter, 174-175

date formatting, filters for, 174-181

- date filter, 174-175

- now filter, 176

- time filter, 175-176

- timesince filter, 176-178

- timeuntil filter, 177-178

date_joined field (User object), 296

date_list context variable (generic views), 235

- date hierarchy, adding to change list view, 352-353
- DateField object, 187, 482-483
- DateField type, 46
- dates, format characters for, 491
- DateTimeField object, 187, 483
- DateTimeField type, 46
- day (field lookup type), 93
- day argument (generic views), 233
- day context variable (generic views), 235
- day_format argument (generic views), 233
- dbshell command, 10
- DecimalField object, 187, 483-484
- decorator functions, registering filters with, 273
- default applications, list of, 26
- default field option, 51
- defining models, 46
 - adding fields to, 48-50
 - field options, 50-55
 - field types, 46-48
- delete() function, 90
- delete_object view, 263-267
- delete_test_cookie() function, 336
- deleting objects from database, 90
- deploying Django, 471
 - adding RAM, 472
 - to Apache with mod_python, 465
 - caching, 472
 - middleware, 472
 - moving databases/media to separate server, 471
 - projects, 468-469
- details() function, 104-105
- details() view
 - creating, 69-70
 - updating, 74-76
- development server
 - explained, 10
 - ports, changing, 22
 - starting, 11, 21-22
 - stopping, 11, 22
- development version of Django
 - downloading, 14
 - installing, 15-16
- dictionaries
 - arguments dictionary for generic views, 232-234
 - sorting, filters for, 173-174
- dictsort filter, 173-174
- directories, creating template directory, 118
- direct_to_template view, 237
- disabling case sensitivity, 23
- displaying
 - database data in views, 97-100
 - date-based objects in generic views, 245-254
 - fields in change list view, 348-349
 - object details in generic views, 242-244
 - object lists in generic views, 238-241
 - web pages with HttpResponse class, 66-68
- Django
 - default applications, list of, 26
 - defined, 8
 - installing, 14-16
 - online resources, 477-479
 - pronunciation, 8
 - reasons for using, 8
- Django project home page, 477
- DoesNotExist exception, 88
- dollar sign (\$), in URL patterns, 110
- downloading development version of Django, 14
- dummy backend, configuring caching, 454
- dynamic links in templates, 147-150
- dynamic web pages, defined, 8

E

- edit_inline argument (ForeignKey field), 56
- editable field option, 52
- eggs, configuring with mod_python, 471
- email fields, 47, 296
- EmailField object, 187, 484
- EmailField type, 47
- embedding templates, 133-135
- enabling
 - admin access in URLconf file, 38-40
 - session framework, 334
 - sitemaps, 425-431
- endswith (field lookup type), 92
- error handling POST request data, 215
- escape filter, 158
- escaping HTML code, 158
- exact (field lookup type), 91
- exception postprocessors, implementing, 398-402
- exceptions, raising from get() function, 87
- exclude() function, 91
- expressions
 - adding to URL patterns, 71-72
 - argument names in, 73-74
 - list of, 71-72
 - named expressions, adding to URL patterns, 74-76
 - in views, 72-73
- extending templates, 129-133
 - with custom tags/filters, 270-272
- extra_context argument (generic views), 232-233
- extra options dictionary, 103-108

FastCGI web page

F

FastCGI web page, 478

field lookups, list of, 91-93

field objects

list of, 481-487

mapping form field objects
to model field objects,
488-490

field options, 50

adding to models, 53-55

list of, 50-52

field types, 46

adding field options to, 50-55

date and time fields, 46-47

email fields, 47

file fields, 47

text fields, 46

URL fields, 48

fields, 185

adding to models, 48-50

displaying in change list view,
348-349

form fields, list of, 186-188

search fields, adding to
change list view, 354-356

for User objects, 296

file fields, 47

file system backend, configuring
caching, 453

FormField object, 187, 484

FormField type, 47

FILES attribute (HttpRequest
class), 65

filter() function, 90-91

filtering change list view, 12

filters

adding to change list view,
350-351

custom filters

creating, 272-275

extending templates,
270-272

explained, 155-156

formatting dates, 174,
178-181

date filter, 174-175

now filter, 176

time filter, 175-176

timesince filter, 176-178

timeuntil filter, 177-178

formatting text, 157, 163-166

applying line breaks, 159

changing text case, 157

determining length of
objects, 159

escaping HTML code, 158

pluralizing words, 161-162

removing HTML tags, 160

string formatting, 159-160

testing true/false

variables, 162

wordwrapping, 160

list management, 166

join filter, 168

make_list filter, 167-168

random filter, 168-173

slice filter, 166-167

registering with decorator
functions, 273

sorting dictionaries, 173-174

filter_interface argument
(ManyToMany field), 57

first_name field (User object), 296

first_on_page context variable
(generic views), 235flatpages, creating sitemaps
for, 429

flup wiki page, 478

for loops in templates, 139-143

ForeignKey field, 55-56

forloop variable, fields in, 141

form context variable (generic
views), 235form field objects, mapping to
model field objects, 488-490

form fields, list of, 186-188

form views, explained, 12-14

format characters for
dates/times, 491-492

formatting

dates, filters for, 174,
178-181

date filter, 174-175

now filter, 176

time filter, 175-176

timesince filter, 176-178

timeuntil filter, 177-178

text, filters for, 157, 163-166

applying line breaks, 159

changing text case, 157

determining length of
objects, 159

escaping HTML code, 158

pluralizing words, 161-162

removing HTML tags, 160

string formatting, 159-160

testing true/false

variables, 162

wordwrapping, 160

forms, 185

creating form instances, 186

bound/unbound forms,
188-189

field types, 186-188

customizing, 200-205

in templates, 202-203

widgets, changing,
200-202GET requests, handling,
209-213model forms, customizing,
356-359normalized data, accessing,
215-216object-creation views,
254-258object-deletion views,
263-267

object-update views, 258-262

POST requests

error handling, 215

handling, 209-213

retrieving data from, 214

validating data from,
214-219

- rendering, 189-194
 - as lists, 190-191
 - from models, 194-200
 - from objects, 195-196
 - as paragraphs, 191
 - partial forms from models, 196-197
 - as tables, 190
 - saving data to database, 219-228
- form_for_instance() function, 195-196**
- form_for_model() function, 194-195**
- forms library, 185-186**

G

- generic templates, 237**
- generic views, 231-232**
 - arguments dictionary, 232-234
 - context variables, 235-237
 - date-based objects, displaying, 245-254
 - generic templates, 237
 - limiting access to, 330
 - object-creation views, 254-258
 - object-deletion views, 263-267
 - object details, displaying, 242-244
 - object lists, displaying, 238-241
 - object-update views, 258-262
 - simple generic views, 237-238
 - sitemaps of, creating, 428-431
 - URL patterns, 235
- GET attribute (HttpRequest class), 65**

- GET requests, handling, 209-213**
- get() function, 87-94**
- gettext() function, 408**
- gettext_lazy() function, 408-409**
- gettext_noop() function, 409**
- gettext package, installing, 413**
- Google, pinging, 434**
- granting rights, 23-24**
- groups**
 - creating, 304-307
 - permissions, creating, 308-309
- groups field (User object), 296**
- gt (field lookup type), 92**
- gte (field lookup type), 92**
- GZipMiddleware application, 403**

H

- has_module_perms() function, 326**
- has_next context variable (generic views), 236**
- has_perm() function, 326**
- has_previous context variable (generic views), 236**
- headers**
 - allowing cached pages to vary, 461-462
 - cache-control headers, 462-463
- height_field argument (image fields), 47**
- help_text argument (form fields), 188**
- hits context variable (generic views), 236**
- home_view() function, 459-460**
- HTML code, escaping, 158**
- HTML forms. See forms**
- HTML tags, removing, 160**

- HTML templates**
 - accessing objects with, 123-126
 - creating, 120-123
- HTTP response, rendering templates as, 127-128**
- httpd.conf file, 467-468**
- HttpRequest class, 64-66**
- HttpResponse class, 66-68**

I

- icontains (field lookup type), 92**
- iendswith (field lookup type), 92**
- iexact (field lookup type), 91**
- if logic in templates**
 - if tag, 144
 - ifchanged tag, 144-145
 - ifequal tag, 145-147
 - ifnotequal tag, 145-146
- ImageField object, 187, 485**
- ImageField type, 47**
- implementing**
 - caching at site level, 457
 - exception postprocessors, 398-402
 - middleware applications, 384-385
 - multiple Django installations in Apache, 469-470
 - per-object caches, 458-459
 - per-site caches, 456-457
 - per-view caches, 457-458
 - request preprocessors, 385-389
 - response postprocessors, 394-398
 - sites framework, 437
 - view preprocessors, 390-394
- importing view functions, 113**
- in (field lookup type), 92**
- include() function, 110-113**
- inclusion tags, creating, 284-289**

index() function

index() function, 103-104

indexes for sitemaps, creating, 432-433

initial argument (form fields), 188

inline, viewing multiple models, 359-362

INSTALLED_APPS setting (settings.py file), 26

installing

- admin interface, 37-38
- applications, 27-30
- Django, 14-16
- gettext package, 413
- middleware applications, 384
- sites framework, 438

instances (of forms), creating, 186-189

IntegerField object, 187, 485

internationalization

- language files, creating, 412-415
- language preferences, setting, 416-421
- localizing strings, 407
 - adding to view function, 409
 - lazy translation method, 408-409
 - no-op string translation method, 409
 - standard translation method, 408
 - in templates, 410-412

IPIAddressField object, 187, 485

iregex (field lookup type), 93

is_active field (User object), 296

is_paginated context variable (generic views), 236

is_staff field (User object), 296

is_superuser field (User object), 296

is_valid() function, 214-215

ISNULL (field lookup type), 93

startswith (field lookup type), 92

items() method (sitemap classes), 424

J-K-L

join filter, 168

label argument (form fields), 188

language files, creating, 412-415

language preferences, setting, 416-421

languages, accessing in templates, 411

lastmod member (sitemap classes), 425

last_login field (User object), 296

last_name field (User object), 296

last_on_page context variable (generic views), 236

latest context variable (generic views), 236

lazy translation method, 408-409

len() function, 86

length filter, 159

length of objects, determining, 159

line breaks, applying, 159

linebreaks filter, 159

links. *See also* relationships

- dynamic links in templates, 147-150
- navigation links, generating with inclusion tags, 285-289

Linux distributions, Django included with, 14

list() function, 86-87

list items

- accessing in reverse order, 140
- cycling through, 151

list management, filters for, 166

- join filter, 168
- make_list filter, 167-168
- random filter, 168-173
- slice filter, 166-167

lists, rendering forms as, 190-191

loading templates from applications, 269-270

local memory backend, configuring caching, 454

LocaleMiddleware application, 403

localizing strings, 407

- adding to view function, 409
- lazy translation method, 408-409
- no-op string translation method, 409
- standard translation method, 408
- in templates, 410-412

locals() function, 127

location member (sitemap classes), 424

login() function, 314

login process, 313-318

login_required argument (generic views), 233

login_required() function, 320

logout process, 318-319

loops in templates, 139-143

lower filter, 157

lt (field lookup type), 92

lte (field lookup type), 92

M

make_list filter, 167-168

make-messages.py utility, 413-414

make_object_list argument (generic views), 233

manage.py utility, 20

- commands, 10
- explained, 10
- starting development server, 21

managing upstream caching, 460-461

many-to-many relationships, 56-60

many-to-one relationships, 55-60

- ManyToMany field, 56-57
 - mapping form field objects to model field objects, 488-490
 - max_length argument
 - text fields, 46
 - URL fields, 48
 - media files, serving on same server, 470
 - memcached backend, configuring caching, 455
 - message files, building, 413-414
 - message-make.py utility, 413-414
 - messages, translating, 414
 - META attribute (HttpRequest class), 65
 - method attribute (HttpRequest class), 64
 - Microsoft SQL database server website, 479
 - middleware applications, 334, 383
 - built-in applications, list of, 402-403
 - implementing, 384-385
 - exception postprocessors, 398-402
 - request preprocessors, 385-389
 - response postprocessors, 394-398
 - view preprocessors, 390-394
 - installing, 384
 - mimetype argument (generic views), 233
 - model argument (generic views), 233
 - model change list view, customizing, 347-348
 - date hierarchy, adding, 352-353
 - fields, displaying, 348-349
 - filters, adding, 350-351
 - ordering objects in, 353
 - search fields, adding, 354-356
 - Model class, 8
 - model field objects, mapping form field objects to, 488-490
 - model forms, customizing, 356-359
 - model view, explained, 11-12
 - Model View Controller (MVC) architecture, 8
 - models, 185
 - activating, 29-30
 - adding relationships to, 55
 - many-to-many relationships, 56-60
 - many-to-one relationships, 55-60
 - one-to-one relationships, 57
 - admin interface. *See* admin interface
 - allowing access from multiple sites, 439-440
 - creating, 28-29
 - defining, 46
 - adding fields to, 48-50
 - field options, 50-55
 - field types, 46-48
 - explained, 8-9
 - limiting access to one site, 441
 - multiple models, viewing inline, 359-362
 - rendering forms from, 194-200
 - models.django.db package, 46
 - models.py file, 28
 - modifying
 - objects in admin interface, 43-45
 - URLconf file, named expressions in URL patterns, 74-76
 - view behavior and content based on sites, 443-448
 - mod_python, 466
 - configuring eggs, 471
 - deploying Django to Apache, 465
 - web page, 477
 - month (field lookup type), 93
 - month argument (generic views), 233
 - month context variable (generic views), 236
 - month_format argument (generic views), 233
 - multiple models, viewing inline, 359-362
 - MultipleChoiceField object, 187, 486
 - MVC (Model View Controller) architecture, 8
 - MySQL databases, case sensitivity and, 23
 - MySQL project home page, 478
- ## N
- named expressions, adding to URL patterns, 74-76
 - navigating admin interface, 42-43
 - navigation links
 - in admin interface, 45
 - generating with inclusion tags, 285-289
 - newforms library. *See* forms library
 - next context variable (generic views), 236
 - next_day context variable (generic views), 236
 - next_month context variable (generic views), 236
 - no-op string translation method, 409
 - normalized data, accessing in forms, 215-216
 - now filter, 176
 - null field option, 50-51
 - NullBooleanField object, 187, 486
 - num_latest argument (generic views), 233
 - numeric format, expression values, 72

object context variable (generic views)

O

object context variable (generic views), 236

object details, displaying in generic views, 242-244

object level, overriding admin template, 372-377

object lists, displaying in generic views, 238-241

object-creation generic views, 254-258

object-deletion generic views, 263-267

object-update generic views, 258-262

ObjectDoesNotExist exception, 88

objects

- accessing with HTML templates, 123-126
- adding/modifying in admin interface, 43-45
- date-based objects, displaying in generic views, 245-254
- ordering in change list view, 353
- referencing in templates, 150-151
- rendering forms from, 195-196

object_detail view, 242-244

object_id argument (generic views), 234

object_list context variable (generic views), 236

object_list view, 238-241

one-to-one relationships, 57

OneToOne field, 57

online resources, list of, 477-479

optimizing Django deployment, 471

- adding RAM, 472
- caching, 472
- middleware, 472
- moving database/media to separate server, 471

options, passing to views, 103-108. *See also* arguments

Oracle database website, 479

ordering objects

- in change list view, 353
- in QuerySets, 94-95

order_by() function, 94-95

overriding

- admin templates, 366-368, 372-377
- admin URL patterns, 378-379
- block tags in admin templates, 373-374

P

packages, built-in Python packages, 20

page argument (generic views), 234

page context variable (generic views), 236

pages context variable (generic views), 236

paginate_by argument (generic views), 234

paragraphs, rendering forms as, 191

params context variable (generic views), 236

parent templates, 129-133

parse() function, 277

partial forms, rendering from models, 196-197

passing

- custom variables to views, 232
- options to views, 103-108

password field (User object), 296

passwords

- changing in views, 298
- in **create_user()** function, 297

path attribute (HttpRequest class), 64

paths

- for admin template directory, 118
- specifying for file fields, 47

patterns. *See* URL patterns

patterns() function, 108-109

per-object caches, implementing, 458-459

per-site cache, implementing, 456-457

per-view caches, implementing, 457-458

permissions, setting, 307-310

persistent data

- cookies
 - setting/retrieving, 344
 - verifying availability of, 336-339
- session framework, 333-334
 - configuring session manager, 334-336
 - enabling, 334
 - setting/retrieving session data, 339-343

PIL (Python Imaging Library)

- project home page, 477

ping_google() function, 434

pinging Google, 434

pipe (**|**), in filters, 156

placeholders in templates, 119

pluralize filter, 161-162

pluralizing words, 161-162

ports for development server, changing, 22

positional arguments, 73

POST attribute (HttpRequest class), 65

POST requests

- error handling, 215
- handling, 209-213
- retrieving data from, 214
- validating data from, 214-219

post_delete_redirect argument (generic views), 234

post_save_redirect argument (generic views), 234

PostgreSQL project home page, 478

preferences, setting language preferences, 416-421

prefixes, view prefixes, 108-110

prerequisites, installing Django, 14

previous context variable (generic views), 236

previous_day context variable (generic views), 236

previous_month context variable (generic views), 236

primary_key field option, 52

primary keys, updating objects, 90

priority member (sitemap classes), 425

process_exception() function, 384, 398

process_request() function, 384-385

process_response() function, 384, 394

process_view() function, 384, 390

program flow. *See if logic projects*

- code storage location, 20
- configuring, 466
- creating, 19-21
- defined, 19
- deploying, 468-469
- placing in PYTHONPATH, 466
- synchronizing to database, 26-27

Python

- built-in packages, 20
- as Django prerequisite, 14

Python eggs, loading templates from, 270

Python Imaging Library (PIL) project home page, 477

Python project home page, 477

PYTHONPATH, placing projects in, 466

Q-R

queries, performing in database, 90-97

queryset argument (generic views), 234

QuerySets, 86

- chaining, 93-94
- ordering objects in, 94-95
- retrieving from database, 90-97

raising exceptions from get() function, 87

RAM, optimizing Django deployment, 472

random filter, 168-173

randomized QuerySets, 95

range (field lookup type), 93

recursive relationships, 56-57

redirect_to view, 238

referencing objects in templates, 150-151

regex (field lookup type), 93

RegexField object, 187, 486

registration

- custom tags, 276
- filters with decorator functions, 273
- generic views, limiting access to, 330
- groups, creating, 304-307
- login process, 313-318
- logout process, 318-319
- permissions, setting, 307-310
- User objects
 - anonymous users, 297
 - changing passwords in views, 298
 - creating in views, 297-304
 - explained, 295-296
 - fields of, 296
 - verifying authentication
 - in templates, 321-325
 - in views, 320
 - verifying permissions
 - in templates, 326-330
 - in views, 325-326

related_name argument

- ForeignKey field, 56
- ManyToMany field, 57

relationships, adding to models, 55

- many-to-many relationships, 56-60
- many-to-one relationships, 55-60
- one-to-one relationships, 57

released version of Django, installing, 15

remove() function, 309

removetags filter, 160

removing HTML tags, 160

render() function, 120

render_to_response() function, 127-128

renderer objects, creating custom tags, 275

rendering

- forms, 189, 191-194
 - as lists, 190-191
 - from models, 194-200
 - from objects, 195-196
 - as paragraphs, 191
 - partial forms from models, 196-197
 - as tables, 190
 - templates as HTTP response, 127-128

REQUEST attribute (HttpRequest class), 65

request preprocessors, implementing, 385-389

RequestContext objects in templates, 289-290

requests, retrieving information from, 64-66

required argument (form fields), 188

requirements. *See prerequisites*

response postprocessors

response postprocessors,
 implementing, 394-398

results_per_page context variable
 (generic views), 236

retrieving
 cookies, 344
 objects from database, 86-89
 POST request data, 214
 QuerySets from database,
 90-97
 session data, 339-343
 specific objects with
 cache API, 459-460
 variables in template context,
 278-284

reverse order, accessing list items
 in, 140

rights, granting, 23-24

ROOT_URLCONF setting
 (settings.py file), 32, 63-64

runserver command, 10

S

save() function, 82, 89-90

saving form data to database,
 219-228

search (field lookup type), 92

search fields, adding to change
 list view, 354-356

security. *See* authentication

servers
 adding admin media files
 to, 467
 development server. *See*
 development server
 serving media files on same
 server, 470

session framework, 333-334
 enabling, 334
 session manager, configuring,
 334-336
 setting/retrieving session
 data, 339-343

session manager, configuring,
 334-336

SessionMiddleware
 application, 403

sessions, setting language
 preferences, 417

settings.py file, 20
 explained, 9-10
 modifying, 118
 setting database access,
 24-25

set_cookie() function, 344

set_test_cookie() function, 336

shell, creating templates in,
 119-120

shell command, 10

shell interface, accessing
 database API to add data,
 30-31

simple backend, configuring
 caching, 454

simple generic views, 237-238

simple_tag filter, 277

site level, overriding admin
 templates, 366-368

site objects, creating, 438-439

sitemap classes, creating,
 424-425

sitemaps, 423-424
 creating, 424
 enabling sitemap,
 425-428
 sitemap class, creating,
 424-425
 indexes, creating, 432-433
 of generic views, creating,
 428-431
 notifying Google of
 changes, 434

sites
 accessing from views, 442
 assigning content to sites,
 439-440
 defining models that allow
 access from multiple sites,
 439-440

defining models that limit
 access to one site, 441
 modifying view behavior and
 content, 443-448

sites framework, 437-438

slice filter, 166-167

slug_field argument (generic
 views), 234

sorting change list view, 12

sorting dictionaries, filters for,
 173-174

split() function, 276

split_contents() function, 276

SQL databases, as Django
 prerequisite, 14

SQLite project home page, 479

startapp command, 10, 28

starting development server, 11,
 21-22

startproject command, 10, 20

startswith (field lookup type), 92

stopping development server,
 11, 22

string format, expression
 values, 72

stringformat filter, 159-160

strings
 formatting, 159-160
 localizing, 407
 adding to view
 function, 409
 lazy translation method,
 408-409
 no-op string translation
 method, 409
 standard translation
 method, 408
 in templates, 410-412

striptags filter, 160

Subversion version control
 system, 14, 479

superuser, creating, 38

symmetrical argument
 (ManyToMany field), 57

syncdb command, 10, 30, 170

synchronizing

- database, 179
- project to database, 26-27

T**tables**

- rendering forms as, 190
- in templates, adding cycle logic to, 151

tags

- block tags, overriding in admin templates, 373-374
- custom tags
 - creating, 275-284
 - extending templates, 270-272
- inclusion tags, creating, 284-289

template argument (generic views), 234**templates**

- accessing languages in, 411
- admin templates
 - overriding, 366-368, 372-377
 - overriding block tags in, 373-374
 - storage location, 365
- advantages of, 117
- creating in shell, 119-120
- custom filters, creating, 272-275
- custom tags, creating, 275-284
- customizing forms in, 202-203
- directory for, creating, 118
- dynamic links in, 147-150
- embedding, 133-135
- explained, 9
- extending, 129-133, 270-272

filters

- explained, 155-156
- formatting dates, 174-181
- formatting text, 157-166
- list management, 166-173
- sorting dictionaries, 173-174

for loops in, 139-143

- generic templates, 237
- generic views. *See* generic views

HTML templates

- accessing objects with, 123-126
- creating, 120-123

if logic in

- if tag, 144
- ifchanged tag, 144-145
- ifequal tag, 145-147
- ifnotequal tag, 145-146

inclusion tags, creating, 284-289**loading from applications, 269-270****localizing strings in, 410-412****placeholders in, 119****referencing objects in, 150-151****rendering as HTTP response, 127-128****RequestContext objects in, 289-290****tables, adding cycle logic to, 151****verifying**

- authentication in, 321-325
- permissions in, 326-330

template_loader argument (generic views), 234**template_name argument (generic views), 234****template_object_name argument (generic views), 234****test_cookie_worked() function, 336****test cookies, 336-339****testing true/false variables, 162****text fields, 46****text formatting, filters for, 157, 163-166**

- applying line breaks, 159
- changing text case, 157
- determining length of objects, 159
- escaping HTML code, 158
- pluralizing words, 161-162
- removing HTML tags, 160
- string formatting, 159-160
- testing true/false variables, 162
- wordwrapping, 160

TextField type, 46**time filter, 175-176****TimeField object, 187, 487****TimeField type, 46****times, format characters for, 492****timesince filter, 176-178****timeuntil filter, 177-178****title filter, 157****to_field argument (ForeignKey field), 56****trans tag, 410****translating messages, 414****translation. *See* internationalization****translation hooks, 411****U****unbound forms, 188-189****unique field option, 52****unique_for_date field option, 52****unique_for_month field option, 52****unique_for_year field option, 52****update_object view, 258-262**

updating

updating

- database data, 219-221
- details() view, 74-76
- objects in database, 89-90

upload_to argument (file fields), 47

upper filter, 157

upstream caching, managing, 460-461

url argument (generic views), 234

URL dispatcher algorithm, case sensitivity of, 68-69

URL dispatcher algorithm, operational overview, 68

URL fields, 48

URL patterns

- \$ (dollar sign) in, 110
- adding
 - expressions to, 71-72
 - named expressions to, 74-76
- admin URL patterns
 - overriding, 378-379
 - storage location, 365
- configuring, 68-70
- for date-based views, 248
- for generic views, 235
- view prefixes, 108-110

url tag, adding dynamic links to templates, 147-150

URLconf file

- adding
 - admin interface to, 38-40
 - URL patterns to, 69-70
 - URLconf files to, 110-113
- configuring, 31-32, 63-64
- importing view functions, 113
- location of, 32
- modifying named expressions in URL patterns, 74-76
- passing options to views, 103-108
- view prefixes, 108-110

URLField object, 187, 487

URLField type, 48

urlpatterns variable, 68

urls.py file, 20, 38

user attribute (HttpRequest class), 65

User objects. *See also*

- authentication
 - anonymous users, 297
 - changing passwords in views, 298
 - creating in views, 297-304
 - explained, 295-296
 - fields of, 296
 - groups, creating, 304-307
 - login process, 313-318
 - logout process, 318-319
 - permissions, setting, 307-310
- user_permissions field (User object), 296
- username field (User object), 296

V

validating

- data, 46
- POST request data, 214-219

variables

- context variables in generic views, 235-237
- retrieving/setting in template context, 278-284
- testing true/false, 162

vary_on_cookie(), 462

verify_exists argument (URL fields), 48

verifying

- authentication
 - in templates, 321-325
 - in views, 320
- availability of cookies, 336-339

permissions

- in templates, 326-330
- in views, 325-326

view function, 64

- calling directly, 113
- expression values as arguments to, 72-73
- HttpRequest class as argument, 64-66

view preprocessors, implementing, 390-394

viewing multiple models inline, 359-362

views

- accessing sites from, 442
- admin views
 - creating custom, 368-372
 - storage location, 366
- change list view, explained, 12-13
- changing passwords in, 298
- creating, 32-33
- creating User objects in, 297-304
- details() view
 - creating, 69-70
 - updating, 74-76
- displaying database data in, 97-100
- explained, 9
- expression values in, 72-73
- form views, explained, 12-14
- generic views. *See* generic views
- GET requests, handling, 209-213
- model view, explained, 11-12
- modifying behavior based on sites, 443-448
- passing custom variables to, 232
- passing options to, 103-108
- POST requests, handling, 209-213
- prefixes, 108-110

URL patterns. See URL patterns
 verifying
 authentication in, 320
 permissions in, 325-326
views.py file, **28, 32**
 passing options to, 103-108

X-Y-Z

year (field lookup type), **93**
year argument (generic views), **234**
year context variable (generic views), **236**
yesno filter, **162**

W

web pages, displaying with
 HttpResponse class, **66-68**
web requests. See **requests**
web servers, project code storage
 location, **20**
web sites, list of online resources,
 477-479
week argument (generic views), **234**
week context variable (generic views), **236**
widget argument (form fields), **188**
Widget objects, list of, **488**
widgets, **186**
 customizing, **200-202**
width_field argument (image fields), **47**
with tag, referencing objects in
 templates, **150-151**
words, pluralizing, **161-162**
wordwrap filter, **160**
wordwrapping, enabling, **160**
wrapper views, limiting access to
 generic views with, **330**