Jeff Forcier
Paul Bissex
Wesley Chun

Covers
Django 1.0

Python Web Development with Django®

Developer's Library

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States, please contact:

International Sales
international@pearson.com

Visit us on the Web: informit.com/aw

Library of Congress Cataloging-in-Publication Data:

Forcier, Jeff, 1982-
Python web development with Django / Jeff Forcier, Paul Bissex, Wesley Chun.
p. cm.
Includes index.
ISBN-10: 0-13-235613-9 (pbk. : alk. paper)
ISBN-13: 978-0-13-235613-8 (pbk. : alk. paper) 1. Web site development. 2. Django (Electronic resource) 3. Python (Computer program language) 4. Web sites—Authoring programs. I. Bissex, Paul. II. Chun, Wesley. III. Title.
TK5105.8885.D54F68 2009
006.7'6—dc22
2008037134

Copyright © 2009 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, write to:

Pearson Education, Inc
Rights and Contracts Department
501 Boylston Street, Suite 900
Boston, MA 02116
Fax (617) 671 3447

ISBN-13: 978-0-13-235613-8
ISBN-10: 0-13-235613-9
Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.
First printing October 2008

Editor-In-Chief
Mark Taub

Acquisitions Editor
Debra Williams Cauley

Development Editor
Michael Thurston

Managing Editor
Kristy Hart

Project Editor and Copy Editor
Jovana San Nicolas-Shirley

Indexer
Cheryl Lenser

Proofreader
Geneil Breeze

Publishing Coordinator
Kim Boedigheimer

Cover Designer
Gary Adair

Compositor
Jake McFarland

Preface

Welcome to Django!

Greetings, and welcome to Django! We're glad to have you along on our journey. You will discover a powerful Web application framework that lets you do everything rapidly—from designing and developing the original application to updating its features and functionality without requiring major changes to the codebase.

About This Book

Several Django books are already on the market, but ours differs from most in that we focus equally on three areas: Django basics, a variety of example applications, and advanced Django topics. Our intent is to make this the most well-rounded book on the subject, one you find useful regardless of background, and which will give you a complete picture of the framework and what you can do with it.

Chapter Guide

In Figure 0.1, you see recommended starting points for your reading depending on your Python and Django experience. Of course, we recommend a cover-to-cover treatment, but the diagram will help if time is not on your side. Regardless of your experience, you are always welcome to look at the applications because reading and studying code is one of the best ways to learn. We also provide this chapter-by-chapter reading guide to further help direct you to where you need to read.

Part I, "Getting Started"

Part I covers the basic material needed to introduce users new to Django and/or Python, although we recommend Chapter 3, "Starting Out," even to advanced readers.

Chapter 1, "Practical Python for Django"

This chapter is an introduction for readers who are new to Python. In one comprehensive chapter, we show you not only the syntax, but also go a bit more in-depth and expose you to Python's memory model and data types, especially constructs commonly used in Django.

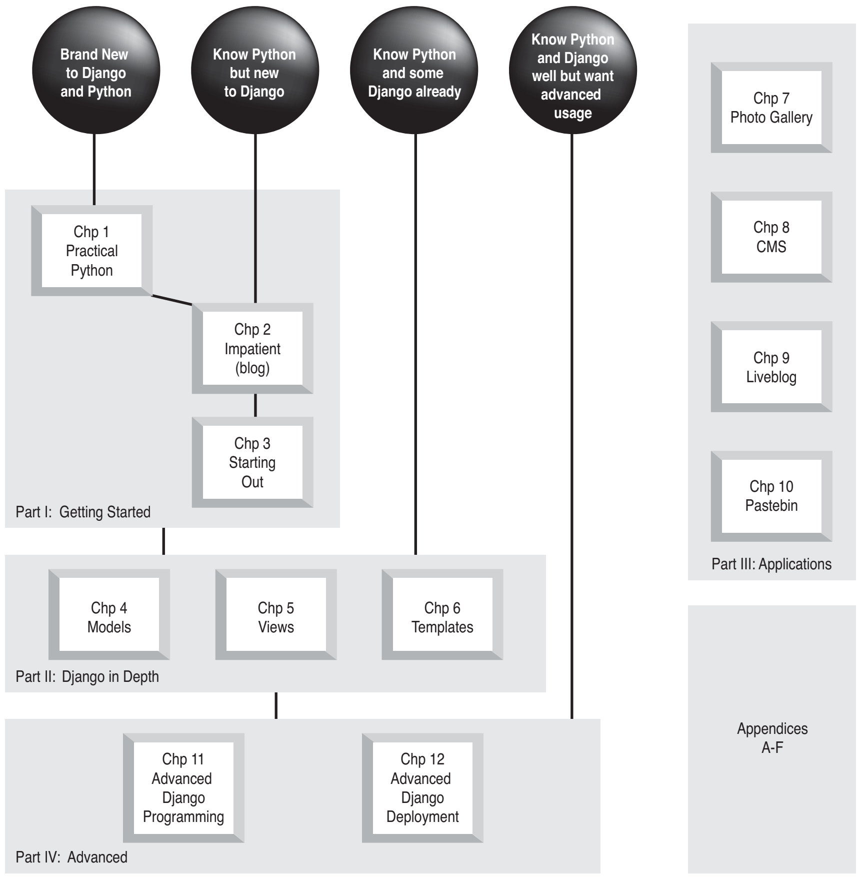


Figure 0.1 Suggested reading guide based on your Python and/or Django experience

Chapter 2, "Django for the Impatient: Building a Blog"

This is a chapter for those who want to skip any Python introduction and want to dive immediately into a Django application that can be completed in 15-20 minutes. It gives a good overview of what's possible with the framework.

Chapter 3, "Starting Out"

For those with a bit more patience, this chapter serves as an introduction to all the foundations of developing Web-based applications (useful both for newbies and experienced coders alike). Once the formalities are over, we describe how each concept fits into the world of Django as well as what its philosophies are and how it may differ from other Web application frameworks.

Part II, "Django in Depth"

Part II covers all the basic components of the framework, laying the foundation for the example applications in Part III, "Django Applications by Example."

Chapter 4, "Defining and Using Models"

In Chapter 4, learn how to define and work with your data model, including the basics of Django's object-relational mapper (ORM) from simple fields up to complex relations.

Chapter 5, "URLs, HTTP Mechanisms, and Views"

This chapter goes into detail on how Django handles URL processing and the rest of the HTTP protocol, including middleware layers, as well as how to use Django's time-saving generic views, and how to write custom or partially custom views from scratch.

Chapter 6, "Templates and Form Processing"

Chapter 6 covers the final major piece of the framework, where we explore Django's template language and its form-handling mechanisms. It covers how to display data to your users and get data back from them.

Part III, "Django Applications by Example"

In Part III, we create four distinct applications, each highlighting a different aspect or component of Django development, both to introduce new general ideas and to expand on the concepts found in Parts I and II.

Chapter 7, "Photo Gallery"

In Chapter 7, learn how to apply the "Don't Repeat Yourself" convention to your URL structure and create a new thumbnail-creating image form field, while you make a simple photo gallery application.

Chapter 8, "Content Management System"

Chapter 8 contains two related approaches to creating a CMS or CMS-like system and covers the use of a number of "contrib" Django applications.

Chapter 9, "Liveblog"

Chapter 9 covers writing a "liveblog"—a site that makes use of advanced JavaScript techniques, serves as a backdrop for applying AJAX to a Django project, and shows how easy it is to use any AJAX toolkit you want.

Chapter 10, "Pastebin"

In Chapter 10, learn the power of Django's generic views as we create a pastebin using almost no custom logic whatsoever.

Part IV, "Advanced Django Techniques and Features"
Part IV is a collection of advanced topics, ranging from customizing Django's admin application to writing command-line scripts that interface with your Django applications.

Chapter 11, "Advanced Django Programming"
Chapter 11 covers a number of topics related to fleshing out your own application's code, such as RSS generation, extending the template language, or making better use of the Django admin application.

Chapter 12, "Advanced Django Deployment"
In Chapter 12, learn a number of tricks related to deploying Django applications or working with your app from outside your Django project's core code, such as command-line scripts, cron jobs, testing, or data import.

Part V, "Appendices"
Part V fills in the remaining gaps or addresses topics relevant to the rest of the book but that don't fit in well as full chapters. Learn the basics of the Unix command line, Django installation and deployment strategies, tools for development, and more.

Appendix A, "Command Line Basics"
Appendix A is an introduction to the Unix command line for those who haven't been exposed to it before now. Trust us—it's useful!

Appendix B, "Installing and Running Django"
In Appendix B, learn how to install all the necessary components for running Django, including the various options for database and Web servers, as well as some tips on specific deployment strategies.

Appendix C, " Tools for Practical Django Development"
Appendix C outlines some basic development tools you may or may not be familiar with, including source control, text editors, and more.

Appendix D, "Finding, Evaluating, and Using Django Applications"
Good developers write code, but great developers reuse somebody else's code! In Appendix D, we share some tips on the where and how of finding reusable Django applications.

Appendix E, "Django on the Google App Engine"
Appendix E provides an exclusive look at how Google's new App Engine leverages Django, and you can also learn how to enable your Django applications to run under the App Engine framework.

Appendix F, "Getting Involved in the Django Project"
In Appendix F, learn how to contribute to Django and become a part of the community.

Conventions

Throughout this book, we use **bold** to introduce new or important terms, *italics* for emphasis, http://links/ for URLs, and `monospacing` to delineate Python and command line material such as variable names or commands. Multiline blocks of code or command examples are in monospaced blocks, like so:

```
>>> print "This is Python!"
This is Python!
```

We have made use of all three major platforms—Mac OS X, Linux, and Windows—when writing this book and the example applications. In addition, we've used all major browsers (although not all may be present in our screenshots), namely Firefox, Safari, Opera, and Internet Explorer.

Book Resources

You can contact the authors collectively at authors@withdjango.com. Our Web site, http://withdjango.com, contains a large amount of auxiliary material and is referenced in a number of places throughout the book.

Introduction

If you're a Web developer, a programmer who creates Web sites, then Django just might change your life. It has certainly changed ours.

Anyone with even a little experience building dynamic Web sites knows the pain of reinventing certain standard features over and over. You need to create database schemas. You need to get data into and out of the database. You need to parse URLs. You need to sanitize input. You need to provide content-editing tools. You need to attend to security and usability. And so on.

Where Web Frameworks Come From

At some point you realize the wastefulness of reimplementing all these features on every new project; you decide to code your own libraries from scratch to provide them—or, more likely, you extract those libraries from your latest and greatest creation. Thereafter, when you start a new project, the first thing you do is install your library code. It saves you work and time.

However, there are still rough spots. Clients want features that aren't provided by your library code, so you add these. Different clients need different things, so you end up with different versions of your library installed on different servers. Maintenance becomes hell.

So then, seasoned with experience, you take your base library and all the best add-ons from your various projects and combine them. For most projects you no longer have to tweak your library code directly; you alter a configuration file instead. Your codebase is bigger and more complicated, but it's also more powerful.

Congratulations, you've written a Web framework.

And as long as you (or your team, or your company, or your clients) keep on using it, you're responsible for keeping it working. Will the next upgrade of your OS, your Web server, or your programming language break it? Will it be flexible enough to accommodate future changes without great pain? Does it support difficult but important features like session management, localization, and database transactions? And how's your test coverage?

A Better Way

You have this book in your hands because you want something better. You want a powerful, flexible, elegant, well-tested framework *you don't have to maintain yourself.*

You want to write your code in a real programming language; one that is powerful, clean, mature, and extensively documented. You want it to have a great standard library and a huge selection of high-quality third-party packages for whatever needs arise, from generating a CSV or a pie chart to scientific computations or image file processing.

You want a framework that has a vibrant, helpful community of users and developers; one that is designed to function smoothly as an integrated stack, but whose components are loosely coupled, so you can make substitutions if circumstances require.

In short, you want Python, and you want Django. We wrote this book to help you learn and use Django in real-world settings as easily, quickly, and smartly as possible.

We're Not in Kansas Anymore

Django was originally written by Adrian Holovaty and Simon Willison at World Online, the Web arm of a family-owned media company in Lawrence, Kansas. It was born out of a need to quickly develop database-driven applications tied into news content.

After proving itself in the field, Django was released as an open source project in July 2005—a time, ironically, when it was widely felt that Python had far *too many* Web frameworks—and rapidly gained a strong following. Today, it is one of the leaders not just among Python frameworks, but among Web frameworks in general.

Django is still heavily used at World Online of course, and some of its core developers work there and use it daily. But since Django's open source release, companies and organizations around the world have picked it up for use in projects large and small. A partial list includes

- The *Washington Post*
- The *Lawrence Journal-World*
- Google
- EveryBlock
- Newsvine
- Curse Gaming
- Tabblo
- Pownce

There are, of course, thousands of other Django sites that are not (yet) household names. It's inevitable that as Django spreads and grows that an increasing number of popular sites will be powered by it. We hope that yours is one of them.

Web Development Is Better with Python and Django

Web development is generally messy business. You have to contend with browser incompatibilities, rogue bots, bandwidth and server limitations, and an overall architecture that seems to defy thorough testing.

Of course, we believe our book is an excellent introduction to the basics of Django, but we also aim to address many of those messy spots—the 20 percent of your work that can take 80 percent of your time. We've worked with, listened to, and helped many developers using Django and have kept their questions and challenges in mind while writing this book.

If we didn't think that Django and Python were great, we wouldn't have gone to the trouble of writing a whole book about them. But when there are limitations you should know about or sharp edges you should watch out for, we'll tell you. Our goal is to help you get things done.

2

Django for the Impatient: Building a Blog

Django bills itself as "the Web framework for perfectionists with deadlines." So let's put ourselves on deadline and see how fast we can produce a simple blog using Django. (We'll address your perfectionist side later.)

Note

This chapter assumes you've already installed Django on your system. If you haven't, consult Appendix B, "Installing and Running Django."

All the work in this chapter is done on the command line in your shell of choice (`bash`, `tcsh`, `zsh`, Cygwin, or what have you). So open your terminal and `cd` to a directory that is on your `PYTHONPATH` environment variable. On a Unix-based system such as Linux, Mac OS X, FreeBSD, and so on, you can issue an `echo $PYTHONPATH` command to see its contents; from a Win32 Command window, type `echo %PYTHONPATH%`.You can read more about paths in both the installation and Python chapters.

We recommend you try to follow along and actually build the blog as you go. If that's not practical—if you aren't near a computer, or you're just impatient—simply reading it is illuminating too.That's especially true if you have experience with one or more other modern Web frameworks, since many of the basic concepts are familiar.

If you *are* following along on your own computer, and you reach a point where the results you're getting don't match what you see here, stop and re-examine the step you just completed, and then review the two or three steps before that. Look for a spot where you could have skipped over a seemingly unimportant detail or didn't understand a specific instruction. If no light bulbs come on, delete your sample project and start over.The authors used this method when learning Django; in addition to being faster than staring blankly at error messages for hours, the repetition of the steps leading up to your trouble spot really help with your retention!

Creating the Project

The easiest way to organize your Django code when you are starting out is to use what Django calls a **project**: A directory of files that constitute, usually, a single Web site. Django comes with a utility called `django-admin.py` to streamline tasks such as the creation of these project directories. On Unix, it has a default installation into the `/usr/bin` directory, and if you're on Win32, it goes into the `Scripts` folder right in your Python installation, for example, `C:\Python25\Scripts`. In either case, you need to make sure that `django-admin.py` is in your `PATH` so it can be executed from the command line.

To create the project directory for your blog project, issue this `django-admin.py` command:

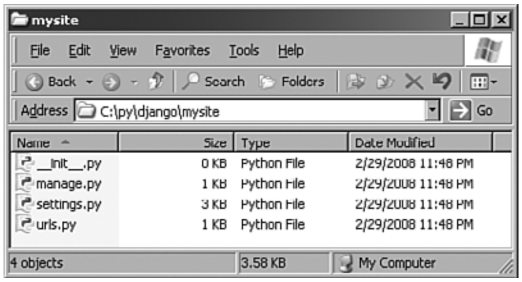
```
django-admin.py startproject mysite
```

On a Win32 box, you need to open a DOS Command window first. It can be accessed via `Start -> Programs -> Accessories -> Command Prompt`. Also, instead of a `$`, you see something like `C:\WINDOWS\system32>` as a shell prompt.

Now take a look at the contents of the directory to see what this command has created for you. It should look something like this on Unix:

```
$ cd mysite
$ ls -l
total 24
-rw-r--r--    1 pbx  pbx     0 Jun 26 18:51 __init__.py
-rwxr-xr-x    1 pbx  pbx   546 Jun 26 18:51 manage.py
-rw-r--r--    1 pbx  pbx  2925 Jun 26 18:51 settings.py
-rw-r--r--    1 pbx  pbx   227 Jun 26 18:51 urls.py
```

If you were developing on a Win32 platform, opening an Explorer window to that folder looks something like Figure 2.1, if we created a folder named `C:\py\django` with the intention of putting our project there.



C:\py\django\mysite

Name	Size	Type	Date Modified
__init__.py	0 KB	Python File	2/29/2008 11:48 PM
manage.py	1 KB	Python File	2/29/2008 11:48 PM
settings.py	3 KB	Python File	2/29/2008 11:48 PM
urls.py	1 KB	Python File	2/29/2008 11:48 PM

4 objects | 3.58 KB | My Computer

Figure 2.1 `mysite` folder on Win32

Note

As you probably know if you're an advanced Python user, that `__init__.py` file makes this project directory a Python **package**—a collection of related Python modules. Its status as a package enables us to use Python's dot-notation to address individual pieces of our project, such as `mysite.urls`. (You can read more about packages in Chapter 1, "Practical Python for Django.")

Besides `__init__.py`, the `startproject` command has created three other files.

- `manage.py` is a utility for working with this Django project. You can see from its permissions flags in the directory listing that it is executable. We run it in a moment.
- `settings.py` is a file containing default settings for your project. These include database information, debugging flags, and other important variables. Any value in this file is available to any of your project's installed apps—we show you the usefulness of that as we progress through this chapter.
- `urls.py` is what's known in Django as a **URLconf**, a configuration file that maps URL patterns to actions your applications perform. URLconfs are an exciting and powerful feature of Django.

Note

Every file created by the `startproject` command is Python source code. There's no XML, no .ini files, and no funky configuration syntax. Django pursues a "pure Python" philosophy wherever possible. This gives you a lot of flexibility without adding complexity to the framework. For example, if you want your settings file to import settings from some *other* file or to calculate a value instead of having it hardcoded, there's no barrier—it's just Python.

Running the Development Server

At this point, you haven't built your blog application yet, but nonetheless there are some Django conveniences in place for your use. One of the handiest is Django's built-in Web server. It's a server designed not for deploying public sites, but for quick development. Advantages of using it include

- You don't need to install Apache, Lighttpd, or whatever other Web server software you'd use in actual production—great if you're working on a fresh server or a non-server development machine or just playing around.
- It automatically detects when you make changes to your Python source files and reloads those modules. This is a *huge* time-saver compared to manually restarting your Web server every time you edit your code, which is what's required with most Python Web server setups.
- It knows how to find and display static media files for the admin application, so you can work with it right away.

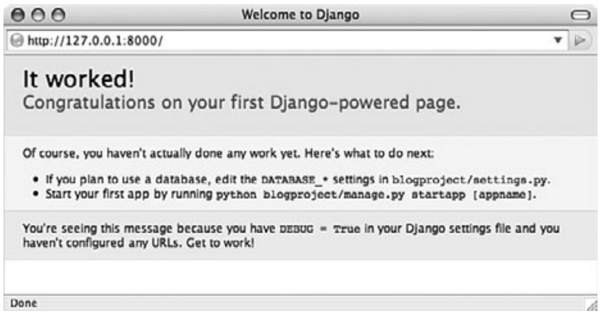
Running the development (or "dev") server is as simple as issuing a single command. We're going to use our project's `manage.py` utility, a thin wrapper script that saves us the work of telling `django-admin.py` to use our specific project settings file. The command to run the dev server is

```
./manage.py runserver       # or ".\manage.py runserver" on win32
```

You should see something like the following with a slight difference for Win32 platforms where the quit key combination is `CTRL-BREAK` instead of `CONTROL-C`:

```
Validating models...
0 errors found.

Django version 1.0, using settings 'mysite.settings'
Development server is running at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

Open that link in your browser, and you should see Django's "It Worked!" screen, as shown in Figure 2.2.



It worked!
Congratulations on your first Django-powered page.

Of course, you haven't actually done any work yet. Here's what to do next:

- If you plan to use a database, edit the `DATABASE_*` settings in `blogproject/settings.py`.
- Start your first app by running `python blogproject/manage.py startapp [appname]`.

You're seeing this message because you have `DEBUG = True` in your Django settings file and you haven't configured any URLs. Get to work!

Figure 2.2 Django's initial It worked! screen

Meanwhile, if you look in your terminal session, you see the dev server has logged your GET request.

```
[07/Dec/2007 10:26:37] "GET / HTTP/1.1" 404 2049
```

The four chunks of the log line are from left to right: timestamp, request, HTTP response code, and byte count. (Your byte count is likely to be slightly different.) The response code is 404 ("Not Found") because your project has no URLs defined yet. The It Worked! page is Django's friendly way of telling you that.

Tip

If your server isn't working at this point, retrace your steps. Be ruthless! It's probably easier to delete your whole project and start following this chapter again from the beginning than it is to laboriously check every file and every line of code.

When you've successfully got the server running, we can move on to setting up your first Django application.

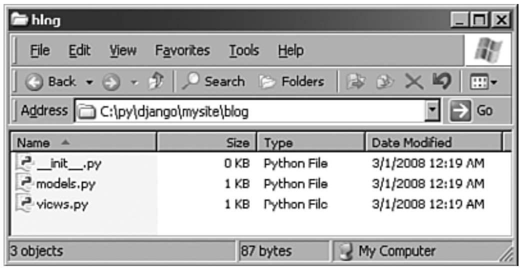
Creating the Blog Application

Now that we have a project, we can create applications (or "apps" in Django-speak) within it. To create our blog app, we'll use `manage.py` again.

```
./manage.py startapp blog   # or ".\manage.py startapp blog" on win32
```

It's just as simple as starting a project. Now we have a `blog` directory inside our project directory. Here's what's in it, first in Unix format, and then in a screenshot of Windows Explorer (see Figure 2.3).

```
$ ls -l blog/
total 16
-rw-r--r--   1 pbx  pbx   0 Jun 26 20:33 __init__.py
-rw-r--r--   1 pbx  pbx  57 Jun 26 20:33 models.py
-rw-r--r--   1 pbx  pbx  26 Jun 26 20:33 views.py
```


blog
File Edit View Favorites Tools Help
Back Search Folders
Address C:\py\django\mysite\blog
Go

Name	Size	Type	Date Modified
__init__.py	0 KB	Python File	3/1/2008 12:19 AM
models.py	1 KB	Python File	3/1/2008 12:19 AM
views.py	1 KB	Python File	3/1/2008 12:19 AM

3 objects | 87 bytes | My Computer

Figure 2.3 `mysite\blog` folder on Win32

Like your project, your app is a package too. The `models.py` and `views.py` files have no real code in them; they're merely placeholders. For our simple blog, in fact, we don't need to touch the dummy `views.py` file at all.

To tell Django this new app is part of your project, you need to edit `settings.py` (which we can also refer to as your "settings file"). Open your settings file in your editor and find the `INSTALLED_APPS` tuple near the bottom. Add your app in dotted module form to that tuple in a line that looks like this (note the trailing comma):

```
'mysite.blog',
```

Django uses `INSTALLED_APPS` to determine the configuration of various parts of the system, including the automatic admin application and the testing framework.

Designing Your Model

We've now arrived at the core of your Django-based blog application: the `models.py` file. This is where we'll define the data structures of the blog. Following the principle of Don't Repeat Yourself (DRY), Django gets a lot of mileage out of the model information you provide for your application. Let's create a basic model, and then see all the stuff Django does for us using that information.

Open up `models.py` in your favorite text editor (bonus points if it has a Python mode with syntax coloring). You see this placekeeper text:

```
from django.db import models

# Create your models here.
```

Delete the comment, and then add the following lines:

```
class BlogPost(models.Model):
    title = models.CharField(max_length=150)
    body = models.TextField()
    timestamp = models.DateTimeField()
```

That's a complete model, representing a "BlogPost" object with three fields. (Actually, strictly speaking it has four fields—Django automatically creates an auto-incrementing, unique `id` field for each model by default.)

You can see our newly minted class, BlogPost, is a subclass of `django.db.models.Model`. That's Django's standard base class for data models, which is the core of Django's powerful object-relational mapping system. Also, you notice our fields are defined like regular class attributes with each one being an instance of a particular field class. Those field classes are also defined in `django.db.models`, and there are many more types—ranging from BooleanField to XMLField—than the three we're using here.

Setting Up the Database

If you don't have a database server installed and running, we recommend SQLite as the fastest and easiest way to get going. It's fast, widely available, and stores its database as a single file in the filesystem. Access controls are simply file permissions. For more on how to set up a a database for use with Django, see Appendix B.

If you *do* have a database server—PostgreSQL, MySQL, Oracle, MSSQL—and want to use it rather than SQLite, then use your database's administration tools to create a new database for your Django project. We name this database "djangodb" in our examples, but you can name it whatever you like.

Either way, with your (empty) database in place, all that remains is to tell Django how to use it. This is where your project's `settings.py` file comes in.

Using a Database Server

Many people use Django with a relational database server such as PostgreSQL or MySQL. There are six potentially relevant settings here (though you may need only two): `DATABASE_ENGINE`, `DATABASE_NAME`, `DATABASE_HOST`, `DATABASE_PORT`, `DATABASE_USER`, and `DATABASE_PASSWORD`. Their names make their respective purposes pretty obvious. Just plug in the correct values corresponding to the database server you are using with Django. For example, settings for MySQL look something like this:

```
DATABASE_ENGINE = "mysql"
DATABASE_NAME = "djangodb"
DATABASE_HOST = "localhost"
DATABASE_USER = "paul"
DATABASE_PASSWORD = "pony"  # secret!
```

> **Note**
>
> We haven't specified `DATABASE_PORT` because that's only needed if your database server is running on a nonstandard port. For example, MySQL's server uses port 3306 by default. Unless you've changed the setup, you don't have to specify `DATABASE_PORT` at all.

For details on creating a new database and database user (which is required for database servers), see Appendix B.

Using SQLite

SQLite is a popular choice for testing and even for deployment in scenarios where there isn't a great deal of simultaneous writing going on. No host, port, user, or password information is needed because SQLite uses the local filesystem for storage and the native filesystem permissions for access control. So only two settings are needed to tell Django to use your SQLite database.

```
DATABASE_ENGINE = "sqlite3"
DATABASE_NAME = "/var/db/django.db"
```

> **Note**
>
> When using SQLite with a real Web server such as Apache, you need to make sure the account owning the Web server process has write access both for the database file itself *and the directory containing that database file*. When working with the dev server like we are here, permissions are typically not an issue because the user (you) running the dev server also owns the project files and directories.

SQLite is also one of the most popular choices on Win32 platforms because it comes free with the Python distribution. Given we have already created a `C:\py\django` directory with our project (and application), let's create a `db` directory as well.

```
DATABASE_ENGINE = 'sqlite3'
DATABASE_NAME = r'C:\py\django\db\django.db'
```

If you are new to Python, you notice the subtle difference in the first example; we used double quotes around `sqlite3`, whereas in the Win32 version, we used single quotes. Rest assured it has nothing to do with differing platforms—Python does not have a character type, so single quotes and double quotes are treated the same. Just make sure you open and close a string with the same type of quote!

You should also have noticed a small "`r`" in front of the folder name. If you've read Chapter 1, then you know this means to designate the object as a "raw string," or one that takes all characters of a string verbatim, meaning do not translate special character combinations. For example, `\n` usually means a newline character, but in a raw string, it means (literally) two characters: a backslash followed by an `n`. So the purpose of a raw string is specifically for DOS file paths, telling Python to not translate special characters (if there are any).

Creating the Tables

Now you tell Django to use the connection information you've given it to connect to the database and set up the tables your application needs. The command to do this is simply:

```
./manage.py syncdb          # or ".\manage.py syncdb" on win32
```

You see some output that starts like this as Django sets up the database:

```
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
Creating table blog_blogpost
```

When you issue the `syncdb` command, Django looks for a `models.py` file in each of your `INSTALLED_APPS`. For each model it finds, it creates a database table. (There are exceptions to this later when we get into fancy stuff such as many-to-many relations, but it's true for this example. If you are using SQLite, you also notice the `django.db` database file is created exactly where you specified.)

The other items in `INSTALLED_APPS`, the ones that were there by default, all have models too. The output from `manage.py syncdb` confirms this, as you can see Django is creating one or more tables for each of those apps.

That's not all the output you got from the syncdb command, though. You also got some interactive queries related to the `django.contrib.auth` app.

```
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): yes
Username (Leave blank to use 'pbx'):
E-mail address: pb@e-scribe.com
```

```
Password:
Password (again):
Superuser created successfully.
Installing index for auth.Message model
Installing index for auth.Permission model
```

Now you've got one superuser (hopefully yourself) in the auth system. This comes in handy in a moment, when we add in Django's automatic admin application.

Finally, the process wraps up with a couple lines relating to a feature called **fixtures**, which we come back to in Chapter 4, "Defining and Using Models." These enable you to preload data in a freshly created application. For now, we're not using that feature, so Django moves on.

```
Loading 'initial_data' fixtures...
No fixtures found.
```

Your initial database setup is now complete. The next time you run the `syncdb` command on this project (which you do any time you add an application or model), you see a bit less output because it doesn't need to set up any of those tables a second time or prompt you to create a superuser.

Setting Up the Automatic admin Application

The automatic back-end application, or admin, has been described as Django's "crown jewel." For anyone who has tired of creating simple "CRUD" (Create, Read, Update, Delete) interfaces for Web applications, it's a godsend. We get much deeper into the admin in "Customizing the Admin" in Chapter 11, "Advanced Django Programming." For now, let's just turn it on and poke around.

Because it's an optional part of Django, you need to specify in your `settings.py` file you're using it—just like you did with your own blog app. Open `settings.py` and add the following line to the `INSTALLED_APPS` tuple, just underneath `'django.contrib.auth'`.

```
'django.contrib.admin',
```

Every time you add a new application to your project, you should run the `syncdb` command to make sure the tables it needs have been created in your database. Here we can see adding the admin app to `INSTALLED_APPS` and running `syncdb` triggers the creation of one more table in our database:

```
$ ./manage.py syncdb
Creating table django_admin_log
Installing index for admin.LogEntry model
Loading 'initial_data' fixtures...
No fixtures found.
```

Now that the app is set up, all we need to do is give it a URL so we can get to it. You should have noticed these two lines in your automatically generated `urls.py`.

```
# Uncomment this for admin:
#     (r'^admin/', include('django.contrib.admin.urls')),
```

Remove the # character from the second line (and you can remove the first, comment-only line at the same time) and save the file. You've told Django to load up the default admin site, which is a special object used by the contrib admin application.

Finally, your applications need to tell Django which models should show up for editing in the admin screens. To do so, you simply need to define the default admin site mentioned previously and register your `BlogPost` model with it. Open the `mysite/blog/models.py` file, make sure the admin application is imported, and then add a line registering your model at the bottom.

```
from django.db import models
from django.contrib import admin

class BlogPost(models.Model):
    title = models.CharField(max_length=150)
    body = models.TextField()
    timestamp = models.DateTimeField()

admin.site.register(BlogPost)
```

This simple use of the admin is the tip of the iceberg; it's possible to specify many different admin-related options by making a special Admin class for a given model, and then registering the model with that class. We do this shortly, and you also see examples of advanced admin use in later chapters, especially in Parts III, "Django Applications by Example," and IV, "Advanced Django Techniques and Features."

Trying Out the admin

Now that we've set up our Django site with the admin app and registered our model with it, we can take it for a spin. Issue the `manage.py runserver` command again. Now, go to http://127.0.0.1:8000/admin/ in your Web browser. (Don't worry if your dev server address is different; just add an `admin/` onto it, whatever it is.) You should see a login screen, as shown in Figure 2.4.

Type the "superuser" name and password you created earlier. Once you've logged in, you see the admin home page, as shown in Figure 2.5.

We'll tour this interface later in the book; for now, just confirm your application, Blog, is showing up as seen in the screenshot. If it's not, recheck the previous steps.

Tip

The three most common causes for "My app doesn't show up in the admin," problems are 1) forgetting to register your model class with `admin.site.register`, 2) errors in the app's `models.py`, and 3) forgetting to add the app to the `INSTALLED_APPS` tuple in your `settings.py` file.

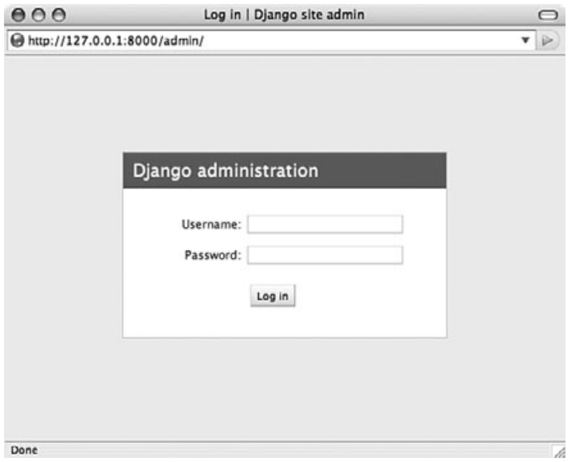


Figure 2.4 The admin login screen

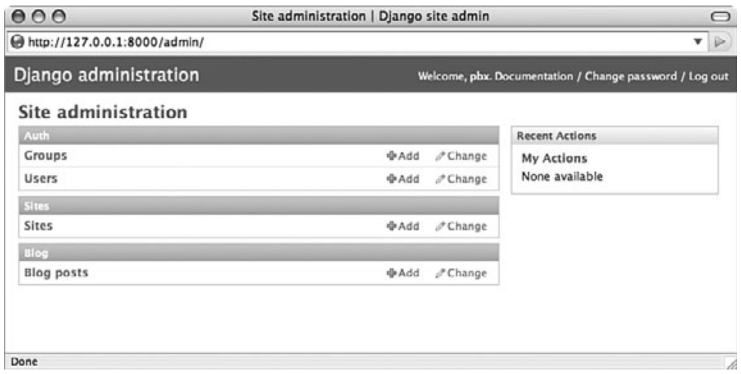


Figure 2.5 The admin home page

What's a blog without content? Click the Add button to the right of Blog Posts. The admin presents a form for adding a new post, as shown in Figure 2.6.

Give your post a title and some scintillating content. For the timestamp, you can click the Today and Now shortcut links to fill in the current date and time. You can also click the calendar or clock icons to pull up handy date and time pickers.

When you're done writing your masterpiece, click the Save button. You see a screen with a confirmation message ("The blog post 'BlogPost object' was added successfully") and a list of all your blog posts—a grand total of one at this point, as shown in Figure 2.7.

Why is the post given the awkward name of "BlogPost object"? Django is designed to flexibly handle an infinite variety of content types, so it doesn't take guesses about what

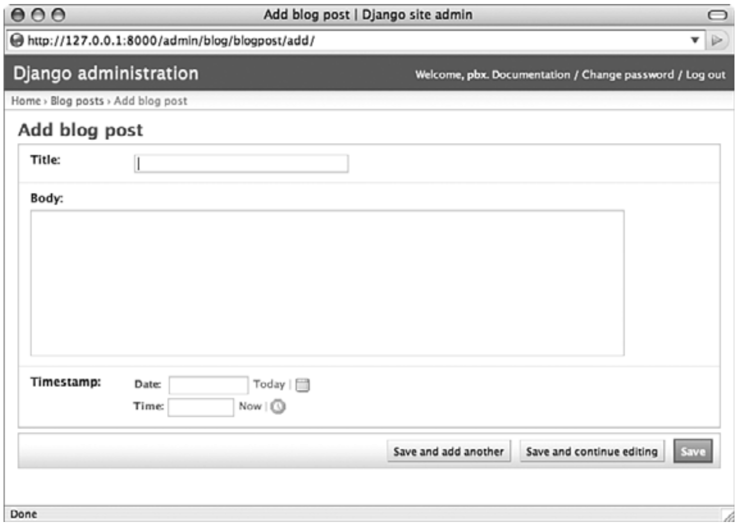
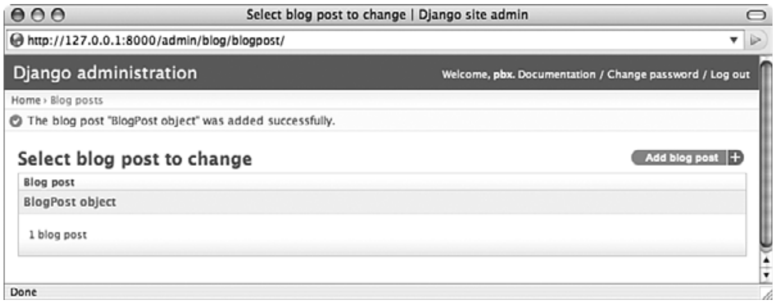

Add blog post | Django site admin
http://127.0.0.1:8000/admin/blog/blogpost/add/
Django administration
Welcome, pbx. Documentation / Change password / Log out
Home › Blog posts › Add blog post
Add blog post
Title:
Body:
Timestamp:
Date:
Today |
Time:
Now |
Save and add another
Save and continue editing
Save
Done

Figure 2.6 Adding new content via the admin


Select blog post to change | Django site admin
http://127.0.0.1:8000/admin/blog/blogpost/
Django administration
Welcome, pbx. Documentation / Change password / Log out
Home › Blog posts
Select blog post to change
Add blog post
Blog post
Done

The blog post "BlogPost object" was added successfully.

BlogPost object

1 blog post

Figure 2.7 Successfully saving your first blog entry

field can be the best handle for a given piece of content. Throughout Part 3's example applications, you see examples of defining how to specify a particular field, or specially constructed string, to be used for your objects' default labels.

Now go ahead and add a second post with different content by clicking on the Add Blog Post + button to the upper-right. When you are returned to the list view, you just see another BlogPost row added to the page. If you refresh the page or go away and come back to your application, the output has not improved any—you just do not feel satisfied with seeing all the entries generically labeled as "BlogPost object," as shown in Figure 2.8. You are not alone if you're thinking, "There has got to be a way to make it look more useful!"

However, we don't have to wait until then to clean up the list display in our admin view. Previously, we enabled the admin tool with the bare minimum of configuration,

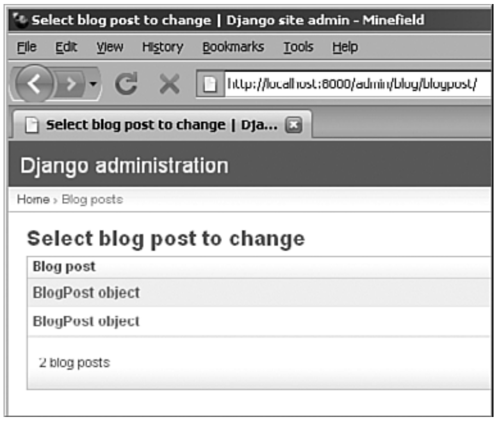

Select blog post to change | Django site admin - Minefield
File Edit View History Bookmarks Tools Help
http://localhost:8000/admin/blog/blogpost/
Select blog post to change | Dja...
Django administration
Home › Blog posts
Select blog post to change
Blog post
BlogPost object
BlogPost object
2 blog posts

Figure 2.8 Not the most useful summary page

namely registering our model with the admin app all by itself. However, with an extra two lines of code and a modification of the registration call, we can make the presentation of the listing much nicer and more usable. Update your `mysite/blog/models.py` file with a new `BlogPostAdmin` class and add it to the registration line, so your `models.py` looks like this:

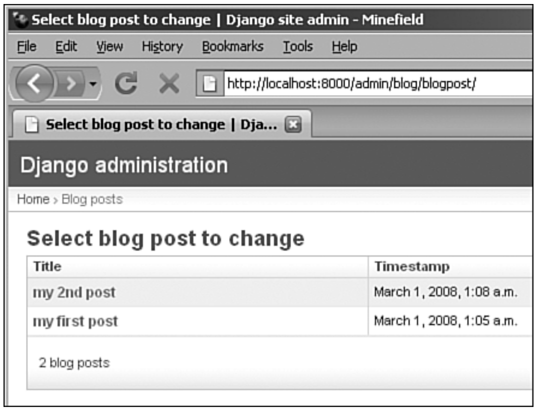
```
from django.db import models
from django.contrib import admin

class BlogPost(models.Model):
    title = models.CharField(max_length=150)
    body = models.TextField()
    timestamp = models.DateTimeField()

class BlogPostAdmin(admin.ModelAdmin):
    list_display = ('title', 'timestamp')

admin.site.register(BlogPost, BlogPostAdmin)
```

The development server notices your changes and reloads your models file. If you are monitoring your command shell, you see some output to this effect.

If you refresh the page, you now see much more useful output based on the new `list_display` variable you added to your `BlogPostAdmin` class (see Figure 2.9).

Try clicking on the Title and Timestamp column headers that have appeared—each one affects how your items are sorted. For example, click once on Title to sort in ascending order by title; click the Title header a second time to change to *descending* order.

The admin has many other useful features that can be activated with just a line or two of code: searching, custom ordering, filters, and more. As we've mentioned a few times already, Parts III and IV cover or demonstrate many of these topics in greater detail.


Select blog post to change | Django site admin - Minefield
File Edit View History Bookmarks Tools Help
http://localhost:8000/admin/blog/blogpost/
Select blog post to change | Dja...
Django administration
Home › Blog posts

Select blog post to change

Title	Timestamp
my 2nd post	March 1, 2008, 1:08 a.m.
my first post	March 1, 2008, 1:05 a.m.

2 blog posts

Figure 2.9 Much better

Making Your Blog's Public Side

With the database and admin side of our application taken care of, it's time to turn to the public-facing pages. A page, from Django's perspective, has three typical components:

- A **template** that displays information passed to it (in a Python-dictionary-like object called a `Context`)
- A **view** function that fetches information to be displayed, typically from a database
- A **URL pattern** that matches an incoming request with your view function, optionally passing parameters to the view as well

We'll tackle these three in that order. In a sense this is building from the inside out—when Django processes a request, it starts with the URL patterns, then calls the view, and then returns the data rendered into a template.

Creating a Template

Django's template language is easy enough to read that we can jump right in to example code. This is a simple template for displaying a single blog post:

```
<h2>{{ post.title }}</h2>
<p>{{ post.timestamp }}</p>
<p>{{ post.body }}</p>
```

It's just HTML (though Django templates can be used for any kind of textual output) plus special template tags in curly braces. These are **variable tags**, which display data passed to the template. Inside a variable tag, you can use Python-style dot-notation to access attributes of the objects you pass to your template. For example, this template assumes you have passed it a BlogPost object called "post." The three lines of the template fetch the BlogPost object's `title`, `timestamp`, and `body` fields, respectively.

Let's enhance the template a bit so it can be used to display multiple blog posts, using Django's `for` template tag.

```
{% for post in posts %}
<h2>{{ post.title }}</h2>
<p>{{ post.timestamp }}</p>
<p>{{ post.body }}</p>
{% endfor %}
```

The original three lines are unchanged; we've simply added a **block tag** called `for` that renders a template section once for each of a sequence of items. The syntax is deliberately similar to Python's loop syntax. Note that unlike variable tags, block tags are enclosed in `{% ... %}` pairs.

Save this simple five-line template in a file called `archive.html`, and put that file in a directory called `templates` inside your `blog` app directory. That is, the path to your template file should be:

```
mysite/blog/templates/archive.html
```

The name of the template itself is arbitrary (we could have called it `foo.html`), but the `templates` directory name is mandatory. By default, when searching for templates, Django looks for a `templates` directory inside each of your installed applications.

Creating a View Function

Now we'll write a simple view function that fetches all our blog posts from the database and displays them using our template. Open up the `blog/views.py` file and type the following:

```
from django.template import loader, Context
from django.http import HttpResponse
from mysite.blog.models import BlogPost

def archive(request):
    posts = BlogPost.objects.all()
    t = loader.get_template("archive.html")
    c = Context({ 'posts': posts })
    return HttpResponse(t.render(c))
```

Skipping over the import lines for the moment (they just load up the function and classes we need), here's the breakdown of the view function, line by line:

- Line 5: Every Django view function takes a `django.http.HttpRequest` object as its first argument. It can also take other arguments that get passed in via the URLconf, which is a feature you are using a lot.
- Line 6: When we created our `BlogPost` class as a subclass of `django.db.models.Model`, we inherited the full power of Django's object-relational mapper. This line is a simple example of using the **ORM** (Object-Relational Mapper; see Chapters 3, "Starting Out," and 4 for more) to get all the BlogPost objects in the database.

- Line 7: To create our template object `t`, we only need to tell Django the name of the template. Because we've stored it in the `templates` directory of our app, Django can find it without further instruction.
- Line 8: Django templates render data that is provided to them in a **context**, a dictionary-like object. Our context `c` has only a single key and value.
- Line 9: Every Django view function returns a `django.http.HttpResponse` object. In the simplest case, we pass the constructor a string. The template `render` method returns a string, conveniently.

Creating a URL Pattern

Only one more piece is needed for our page to work—like anything else on the Web, it needs a URL.

We could create the needed URL pattern directly inside `mysite/urls.py`, but that creates a messy coupling between our project and our app. We can use our blog app somewhere else, so it would be nice if it were responsible for its own URLs. We do this in two simple steps.

The first step is much like enabling the admin. In `mysite/urls.py`, there's a commented example line that is almost what we need. Edit it so it looks like this:

```
url(r'^blog/', include('mysite.blog.urls')),
```

This catches any requests that begin with `blog/` and passes them to a new URLconf you're about to create.

The second step is to define URLs inside the blog application package itself. Make a new file, `mysite/blog/urls.py`, containing these lines:

```
from django.conf.urls.defaults import *
from mysite.blog.views import archive

urlpatterns = patterns('',
    url(r'^$', archive),
)
```

It looks a lot like our base URLconf. The action happens in line 5. First, note the `blog/` part of the request URL, which our root URLconf was matching, is stripped—our blog application is reusable and shouldn't care if it's mounted at `blog/` or `news/` or `what/i/had/for/lunch/`. The regular expression in line 5 matches a bare URL, such as `/blog/`.

The view function, `archive`, is provided in the second part of the pattern tuple. (Note we're not passing a string that names the function, but an actual first-class function object. Strings can be used as well, as you see later.)

Let's see it in action! Is the dev server still running? If not, fire it up with `manage.py runserver`, and then go to http://127.0.0.1:8000/blog/ in your browser. You should see a simple, bare-bones rendering of any blog posts you have entered, complete with title, timestamp, and post body.

Finishing Touches

Using the key concepts laid out so far, you could go forward and refine this primitive blog engine in a number of ways. Let's step through a few of them to make this project feel just a little more polished.

Template Niceties

Our template is plain to say the least. Because this is a book on Web programming not Web design, we leave the aesthetic touches to you, but template inheritance is another feature of the template system that can make your life easier, especially as your page styles proliferate.

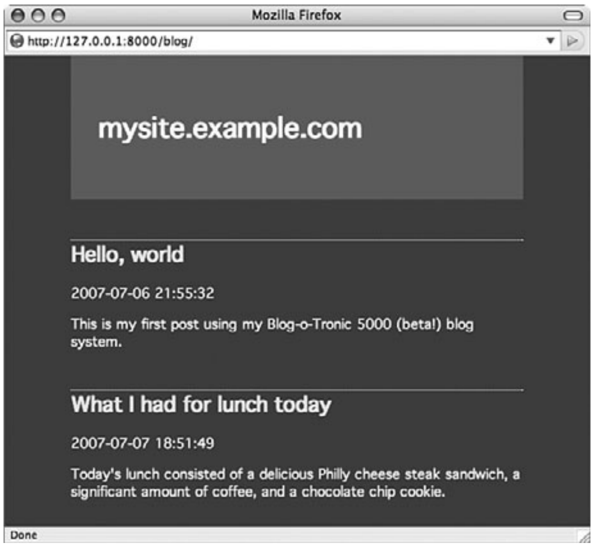
Our simple template is completely self-contained. But what if our site had a blog, a photo archive, and a links page, and we wanted all these to be based on a common base? Experience tells you the *wrong* way to do this would be to copy and paste your way to three kind-of-identical self-contained templates. The *right* way in Django is to create a base template, and then *extend* this template to generate the other, specific templates. In your `mysite/blog/templates` directory, create a template called `base.html` containing the following:

```
<html>
<style type="text/css">
body { color: #efd; background: #453; padding: 0 5em; margin: 0 }
h1 { padding: 2em 1em; background: #675 }
h2 { color: #bf8; border-top: 1px dotted #fff; margin-top: 2em }
p { margin: 1em 0 }
</style>
<body>
<h1>mysite.example.com</h1>
{% block content %}
{% endblock %}
</body>
</html>
```

Not exactly valid XHTML Strict, but it'll do. The detail to notice is the `{% block ... %}` tag. This defines a named area that subtemplates can change. To make your blog app use this template, change your `archive.html` template so it references this new base template and its "content" block.

```
{% extends "base.html" %}
{% block content %}
{% for post in posts %}
<h2>{{ post.title }}</h2>
<p>{{ post.timestamp }}</p>
<p>{{ post.body }}</p>
{% endfor %}
{% endblock %}
```

The `{% extends ... %}` tag tells Django to look for a template named `base.html`, and plug the content of any named blocks in *this* template into the corresponding blocks in *that* template. You should now see something like Figure 2.10 (hopefully your blog posts are more exciting, though).



mysite.example.com

Hello, world

2007-07-06 21:55:32

This is my first post using my Blog-o-Tronic 5000 (beta!) blog system.

What I had for lunch today

2007-07-07 18:51:49

Today's lunch consisted of a delicious Philly cheese steak sandwich, a significant amount of coffee, and a chocolate chip cookie.

Figure 2.10 The blog, lightly styled

Date-Based Ordering

You should have noticed your blog posts are not being presented in traditional reverse-chronological order. It's easy for us to tell Django to do that; in fact, we have a choice as to where we want to tell it to do so. We can add a default ordering to our model, or we can add it to the `BlogPost.objects.all()` query in our view code. In this case the model is a better location because we most often want posts ordered reverse chronologically. If we set our preferred ordering in the model, any part of Django that accesses our data uses that ordering.

To set default ordering for your model, give it an inner class called `Meta` and set the `ordering` attribute in that class.

```
class Meta:
    ordering = ('-timestamp',)
```

Take a look at your blog home page (`/blog/`). The newest post should now be on top. The string `-timestamp` is a concise way of telling Django, "order by the 'timestamp' field, and do it in descending order." (If we omitted the "-", they'd be presented in ascending date order instead.)

> **Note**
>
> Don't omit the trailing comma inside the parentheses! It makes this a single-item tuple, rather than just a parenthesized string. Django expects a tuple here; you're allowed to specify as many fields for ordering as you want. If you added `'title'` after the comma, and you had two posts titled "A" and "B" with the same timestamp, post "A" would come first.

Timestamp Formatting Via a Template Filter

That timestamp is handy, but its ISO8601 format is a little nerdy. Let's humanize it a bit by using a cool feature of the Django template system: filters.

Because this is a presentation detail, not a data structure or business logic detail, the appropriate place for it is in the template. Open your `archive.html` file and change the "post.timestamp" line to

```
<p>{{ post.timestamp|date }}</p>
```

To apply a filter to a variable, you simply tack it on to the end of the variable name—inside the curly brackets—using a vertical bar, or "pipe," character. Reload your blog home page. Now your dates appear in a more liberal-arts-friendly form ("July 7 ").

If the default style of the `date` filter isn't to your liking, you can pass it an argument using `strftime`-type formatting. However, rather than using the conversion codes from Python's `time` module, it uses the same formatting directives as PHP's `date` function. For example, if you want to display the day of the week but omit the year, change the line to pass an argument to the date filter.

```
<p>{{ post.timestamp|date:"l, F jS" }}</p>
```

This particular format string gives you dates in the style "Friday, July 6th." Make sure you don't leave any space on either side of that colon—the Django template engine is particular about this.

Summary

Of course, we could continue adding features to our blog engine forever (many people do!), but hopefully you've seen enough to give you a taste of the power of Django. In the course of building this skeletal blog app you've seen a number of Django's elegant, labor-saving features:

- The built-in Web server, which makes your development work more self-contained and automatically reloads your code if you edit it
- The pure-Python approach to data model creation, which saves you from having to write or maintain SQL code or XML description files
- The automatic admin application, which provides full-fledged content-editing features even for nontechnical users
- The template system, which can be used to produce HTML, CSS, JavaScript, or any textual output format

- Template filters, which can alter the presentation of your data (such as dates) without messing with your application's business logic
- The URLconf system, which gives you great flexibility in URL design while keeping application-specific portions of URLs in the application, where they belong

Just to give you an idea of what's ahead, the following are some things we could proceed to do to our blog using Django's built-in features:

- Publish Atom or RSS feeds of our latest posts (see Chapter 11)
- Add a search feature so that users can locate blog posts containing specific terms (see the CMS example app in Chapter 8, "Content Management System")
- Adopt Django's "generic views" to avoid having to write any code in views.py at all (see the Pastebin example app in Chapter 10, "Pastebin")

You've completed your whirlwind tour of Django basics. Chapter 3, fleshes out with a broad look at Django's key components and the philosophies behind them, as well as provides a recap of some Web development principles central not only to Django itself, but to the lessons we offer in later parts of the book. Chapter 4, takes you down into the details of the framework, where you find answers to the questions of "how, why, and what about ...?" that probably arose as you walked through the previous examples. After Chapter 4, you have a solid enough understanding to follow along and build several example applications: a content management system, a pastebin, a photo gallery, and an Ajax-powered "live blog."

Index

Symbols

& (ampersand) operator (Django), 109

>> append operator, 291

* (asterisk)
  - in functions (Python), 40-42, 132
  - repetition/duplication, 16

@ (at sign), function decorators, 44

^ (carat), in URLconfs, 118

: (colon), sequence slicing, 15

, (comma), in tuples, 24

$ (dollar sign)
  - in URLconfs, 118
  - variable (jQuery), 214

** (double asterisk), in functions (Python), 40-42, 132

# (hash mark), comments, 10

% (percent sign), string format operator, 22-24

| (pipe)
  - operators (Django), 109, 290
  - template filters, 137

+ (plus sign), concatenation, 16

~Q, in keyword composition (Django), 109-110

< redirection operator, 291

> redirection operator, 291

A

ab (benchmarking tool), 265

abstract base classes (Django), 97-99

access control. *See* authentication decorators (Python), 240

accessing App Engine objects in Django applications, 330

add-ons for Python, 300
- Easy Install, 300
- IPython, 300-301

admin application (Django)
- changing list display, 68-69
- creating blog posts, 67-69
- customizing, 235-236
  - with authentication decorators, 239-240
  - base templates, extending, 237-238
  - with fieldsets setting, 236-237
  - views, creating, 238-239
- login, 66-67
- model options, 101-102
- setup, 65-66
- troubleshooting, 66

admin views in CMS example application, 193-195

Ajax (Asynchronous Javascript And XML), 205. *See also* JavaScript
- advantages of, 206
- explained, 205-206
- libraries (liveblog example application)
  - installing, 213-214
  - selecting, 207-208
  - setup and testing, 214-215
- requirements for usage, 212
- XML versus JSON, 212-213

all function (Python), 25-26

all method (Django), 104

alternative templating (Django), 258-260

ampersand (&) operator (Django), 109

animation with Ajax. *See* Ajax

anonymous functions (Python), 38
- in Django, 39
- lambda keyword, 38-39

any function (Python), 25-26

Apache Web server
- hooking Django into, 304-305
- installing, 304
- serving static files, 305-306

App Engine. *See* Google App Engine

append method (Python), 18

append operator (>>), 291

applications (Django)
- blog project example
  - admin application, 65-69
  - applications, creating, 61
  - databases, setup, 62-65
  - development runserver, running, 59-61
  - models, 62, 74
  - projects, creating, 58-59
  - templates, creating, 70-75
  - URL patterns, creating, 72
  - view functions, creating, 71-72
- CMS example application, 185
  - admin views in, 193-195
  - custom managers for, 189
  - displaying stories, 198-199
  - enhancement suggestions, 202-203
  - imports for, 188
  - Markdown usage in, 190-192

model definition for, 186-189
search functionality in, 199-201
templates for, 196-198
urls.py file, 192-193
user management, 201
views.py file, 196
workflow management, 202
converting to App Engine applications, 328
accessing App Engine objects, 330
adding data, 331-332
copying App Engine code, 328
integrating App Engine Helper for Django, 329-330
linking to App Engine code, 329
testing, 330
creating, 61
for Google App Engine, 333-334
evaluating, 322-323
finding, 321-322
Flatpages, 182
enabling, 182-184
error handling, 184
templates, creating, 184
testing, 184-185
installing, 323
liveblog example application
Ajax libraries, installing, 213-214
Ajax libraries, selecting, 207-208
Ajax libraries, setup and testing, 214-215
Ajax requirements, 212
directory structure of, 208-211
planning, 206-207
view function, creating, 216-217
view function, using via JavaScript, 217-218
XML versus JSON, 212-213
pastebin example application, 221-222
cleaning up with cron jobs, 231-232
limited number of items displayed, 229-230
models.py file, 222-223
syntax highlighting in, 230-231
templates, creating, 223-225
testing, 226-229
URLs, designing, 225
photo gallery example application, 159
file uploads, preparing for, 161-162
ImageField class, testing, 163-165
models.py file, 160-161
PIL (Python Imaging Library), installing, 162-163
templates, 173-179
ThumbnailImageField class, building, 164-169
ThumbnailImageField class, usage, 169
URLs setup, 169-173
sharing, 323
testing, 274-279
doctest module (Python), 275
models, testing, 276-278
running tests, 276
tools for, 278-279
unittest module (Python), 276

args/kwargs convention. *See* * (asterisk); **(double asterisk)

arguments for commands, 286, 288-289

arithmetic operators (Python), 11, 13

arrays. *See* lists (Python)

AssertionError exception (Python), 32

assigning environment variable values, 291

associative arrays. *See* dictionaries (Python)

asterisk (*)

- in functions (Python), 40-42, 132
- repetition/duplication, 16

Asynchronous Javascript And XML. *See* Ajax

at sign (@), function decorators, 44

AttributeError exception (Python), 32

attributes (Django)

- adding to fields, 167-168
- calculated attributes, 90
- of request objects, 124-125

augmented assignment operators (Python), 11, 14

authentication decorators (Python) for admin application, 239-240

B

backend types for caching (Django), 272

- db, 273
- file, 273
- memcached, 273-274

base classes, 44

base templates

- creating, 73-74
- extending, 237-238

baselines (server performance), determining, 265-266

base_site.html template (admin application), extending, 238

benchmarking server performance, 265-266

BigTable system, 326

binary directories, 293

bitwise operators (Python), 14

block tag (Django), 71, 139-141

block template tag (Django), 73

blog project example

- admin application
  - changing list display, 68-69
  - creating blog posts, 67-69
  - login, 66-67
  - setup, 65-66
  - troubleshooting, 66
- applications, creating, 61
- databases, setup, 62-65
- development runserver, running, 59-61
- models
  - default ordering, setting, 74
  - designing, 62
- projects, creating, 58-59
- templates
  - base templates, creating, 73-74
  - creating, 70-71
  - filters, applying, 75
- URL patterns, creating, 72
- view functions, creating, 71-72

blogs. *See also* blog project example
- Django blog aggregator, 338
- liveblogs, defined, 205

bool function (Python), 12

Boolean data type (Python), 12

Boolean operators (Python), 11

BooleanField class (Django), 92

bound forms (Django), 147-149

branches (version control systems), 314

BSD/MIT-style licenses, GPL (GNU Public License) versus, 321

built-in functions (Python), 14

C

CACHE_BACKEND setting, arguments for, 272

cache_control decorator (Django), 269

cache_page decorator (Django), 268

cache-related headers (Django), controlling, 268-269

cache template tag (Django), 271

cache type (Django), setting, 266

caching (Django), 226, 265
- backend types, 272
  - db, 273
  - file, 273
  - memcached, 273-274
- baseline, determining, 265-266
- cache type, setting, 266
- middleware, adding, 266
- strategies for, 267
  - cache-related headers, controlling, 268-269
  - cache template tag, 271
  - object cache, 269-270
  - per-view caching, 268
  - site-wide caching, 267
- testing, 266-267

calculated attributes, 90

calendar module (Python), 254

callable views (Django), 121-122

calling functions (Python), 34-35
- * and ** in, 40
- by reference/value. *See* mutability
- keyword arguments in, 35

carat (^), in URLconfs, 118

cascades (SQL), support for, 113

Cascading Style Sheets (CSS), 79

casting Python data types. *See* numeric functions (Python)

centralized version control systems, 315

changes (in version control systems)
- committing, 317
- merging, 314

changesets, recording, 317

CharField class (Django), 91

charts, generating, 246-248

CherryPy, 306

choosing. *See* selecting

class keyword (Python), 44

classes (Django)
- abstract base classes, 97-99
- BooleanField, 92
- CharField, 91
- EmailField, 91
- Field, 91
- FileField, 92

ForeignKey, 93
Form, 142
Http404, 131
ImageField, testing, 163-165. *See also* photo gallery example
ImageFieldFile, 167-168
IPAddressField, 91
Manager, 104-105
ManyToManyField, 95
Meta, 100-101
ModelChoiceField, 146
ModelForm, 143
ModelMultipleChoiceField, 146
NullBooleanField, 92
OneToOneField, 96
Q, 109-110
QueryDict, 123
QuerySet, 104-105
as database query, 105-106
joining databases, 108-109
keyword composition with Q and ~Q, 109-110
as list container, 106
modifying SQL queries, 111-112
as nested queries, 106-107
removing duplicates, 108
sorting query results, 108
TextField, 91
URLField, 91
Widget, 152

classes (Python)
creating, 44-45
Django models and, 46-47
dynamic instance attributes, 52-53
inner classes, 46
instantiating, 45, 52
subclassing, 46

clean up. *See* cron jobs

clear method (Python), 28

CMS (Content Management System)
custom example application, 185
admin views in, 193-195
custom managers for, 189
displaying stories, 198-199
enhancement suggestions, 202-203
imports for, 188
Markdown usage in, 190-192
model definition for, 186-189
search functionality in, 199-201
templates for, 196-198
urls.py file, 192-193
user management, 201
views.py file, 196
workflow management, 202
explained, 181
Flatpages application, 182
enabling, 182-184
error handling, 184
templates, creating, 184
testing, 184-185

codebase (Django), testing, 279-281

coding style (Python), 53

- documentation strings, 54-55
- indentation, 53
- single-line suites, 54
- spaces versus tabs, 53

colon (:), sequence slicing, 15

columns (in database tables), 78

comma (,), in tuples, 24

comma-separated values (CSV) files, generating, 245-246

command line. *See also* utility scripts

- arguments and options, 288-289
- environment variables, 291-293
- input and output, 289-291
- Mac OS X, using in, 285
- operational overview, 285-287
- paths, 293-294
- Unix program names, 287-288
- Windows, using in, 285

commands

- arguments and options, 286-289
- Django
  - dumpdata, 104
  - loaddata, 104
  - manage.py shell, 10
  - runserver, 60
  - sql, 103
  - sqlall, 103
  - sqlclear, 103
  - sqlcustom, 103, 113
  - sqlindexes, 103
  - sqlrest, 103
  - startapp, 61
  - startproject, 58
  - syncdb, 64, 103
- Python
  - del, 27
  - print, 8-9

commenting system (Django), 203

comments (Python), 10

committing changes in version control systems, 317

communication methods (in Web development), 78

comparison operators (Python), 11

complex data type (Python), 13

concatenating sequences (Python), 16-17

conditionals (Python), 29

configuration files (Nagios), generating, 243-244

configuring

- feeds (Django), 240-241
- pure Google App Engine applications, 326

constraining relationships, 96

constructors. *See* instantiation (Python)

contact information (vCard generation), 244-245

containers, QuerySet class (Django) as, 106

Content Management System. *See* CMS

context processors (Django), 136

contexts for templates (Django), 135-136

contributing to Django community, 337-338

controllers (in MVC architecture), 80

converting

- Markdown content to HTML, 191
- Python data types. *See* numeric functions (Python)

cookies, 124

COOKIES data structure (Django), 124

copying

- App Engine code to projects, 328
- objects (Python), mutability and, 51-52
- repositories (version control systems), 317
- sequences (Python), 16

create_update.create_object generic view (Django), 130

create_update.update_object generic view (Django), 130

cron jobs

- in pastebin example application, 231-232
- utility script for, 262-263

CRUD operations, 128

CSS (Cascading Style Sheets), 79

CSV (comma-separated values) files, generating, 245-246

csv module (Python), 245-246

custom classes (photo gallery example)

- building, 164-169
- using, 169

custom datatypes (SQL), support for, 113

custom filters (Django), creating, 256-258

custom functions (SQL), support for, 113

custom managers (Django), 248

- for CMS example application, 189
- default object set, changing, 248-249
- methods
  - creating, 249-250
  - naming, 249

custom queries (SQL), support for, 115

custom template tags (Django), creating, 250-253

custom views (Django), 131-132

customizing

- admin application (Django), 235-236
  - with authentication decorators, 239-240
  - base templates, extending, 237-238
  - with fieldsets setting, 236-237
  - views, creating, 238-239
- codebase (Django), 264
- generic views (Django) with dictionaries, 225

CVS (Concurrent Versions System), 313

Cygwin, 285

D

data aggregation, 90

data caching (Django), 226

data import/export, utility script for, 263-264

data normalization in forms (Django), 150

data storage methods (in Web development), 78

data types (Python)
  Boolean, 12
  complex, 13
  conversion. *See* numeric functions (Python)
  Decimal, 13
  dictionaries, 26-28
  float, 12
  int, 12
  iterables, 14
  long, 12
  numeric, 12-13
  sequences, 14-15
    concatenation, 16-17
    copying, 16
    functions for, 25-26
    lists, 14, 17-19
    operations on, 16
    slicing, 15-16
    strings, 14, 19-24
    tuples, 15, 24-25
  for variables, 10-11
databases, 78
  Django
    creating and updating, 103-104
    non-Django SQL features, usage of, 112-115
    querying, 104-112
    synchronization, 103
  MySQL, 310-311
  Oracle, 311
  PostgreSQL, 309-310
  setup, 62
    for SQLite, 63-64
    table creation, 64-65
  SQLite, 308
  support for, 311
DATABASE_ENGINE setting, 63
DATABASE_HOST setting, 63
DATABASE_NAME setting, 63
DATABASE_PASSWORD setting, 63
DATABASE_PORT setting, 63
DATABASE_USER setting, 63
date-based ordering of models, setting, 74
date_based.* generic views (Django), 130
db cache type (Django), 272-273
decentralized version control systems, 315-316
Decimal data type (Python), 13
declaring functions (Python), 34
  * and ** in, 41-42
  default arguments in, 35-36
decorators
  Django
    cache_control, 269
    cache_page, 268
    stringvalue, 258
  Python
    authentication decorators for admin application, 239-240
    for functions, 39, 42-44
deep copying (Python), 52
default arguments in functions (Python), 35-36

default object set in ORM system (Django), changing, 248-249

default ordering (in models), setting, 74

default widgets (Django), overriding, 153-154

defining

forms (Django), 142-143

initial SQL files, 113

models (Django), 91

CMS example application, 186-189

pastebin example application, 222-223

del command (Python), 27

delete method (Django), 168

order of operations, 169

deleting

directories, 287

duplicates in query results (Django), 108

files, 286

older items (pastebin example application), 231-232

thumbnail images (photo gallery example), 168-169

designing URLs (Django) for pastebin example application, 225

Development Console, 328

development runserver, 303

running, 59-61

development version of Django, 302

dictionaries (Python), 26-28

COOKIES data structure, 124

as default function arguments, 36

in Django, 28

customizing generic views with, 225

FILES data structure, 125

GET and POST data structures, 123-124

META data structure, 125

REQUEST data structure, 124

session data structure, 124

directories

binary directories, 293

listing contents of, 286

liveblog example application, 208-211

preparing for file uploads (photo gallery example), 161-162

removing, 287

shared applications directory, creating, 323

site-packages directory, installing applications to, 323

display of admin application, changing, 236-237

displaying forms (Django), 150-152

distinct method (Django), 108

division operators (Python), 13

Django. *See also* Django applications

admin application

changing list display, 68-69

creating blog posts, 67-69

customizing, 235-240

login, 66-67

setup, 65-66

troubleshooting, 66

anonymous functions (Python) in, 39

architectural overview, 82
block tags, 71
blog aggregator, 338
caching, 226, 265
  backend types, 272-274
  baseline, determining, 265-266
  cache type, setting, 266
  middleware, adding, 266
  strategies for, 267-271
  testing, 266-267
codebase
  customizing, 264
  testing, 279-281
community, contributing to, 337-338
core philosophies of, 82
  DRY (Don't Repeat Yourself), 84
  modularity, 84-85
  as Pythonic, 84
  rapid development, 85
custom managers, 248
  default object set, changing, 248-249
  methods, creating, 249-250
  methods, naming, 249
databases, setup, 62-65
decorators
  cache_control, 269
  cache_page, 268
  stringvalue, 258
described, 7
development of, 2
dictionaries in, 28, 225
documentation, 338
downloadable file generation, 243
  charts and graphs, 246-248
  CSV (comma-separated values) files, 245-246
  Nagios configuration files, 243-244
  vCards, 244-245
downloading
  development version, 302
  packaged releases, 302
exception handling, 33
filters, 229
fixtures, 65, 113-115
forms, 142
  data normalization, 150
  defining, 142-143
  displaying, 150-152
  filling out, 147-149
  model-based forms, 143-146
  subclassing, 146-147
  validating, 149-150
  widgets, 152-154
function objects (Python) in, 37-38
hooking into Apache Web server with mod_python, 304-305
imports (CMS example application), 188
installing, 302-303
interactive interpreter (Python) usage, 9-10
keyword arguments (Python) in, 42
managers, 249

models
  admin options, 101–102
  classes (Python) and, 46–47
  CMS example application, 186–189
  creating and updating databases, 103–104
  default ordering, setting, 74
  defining, 91
  designing, 62
  encapsulation of methods, 89–90
  enumerate function (Python) in, 30
  field types, 91–92
  inheritance, 97–100
  Meta class, 100–101
  non-Django SQL features, usage of, 112–115
  ORM (Object-Relational Mapper), advantages of, 89–91
  photo gallery example, 160–161
  portability, 90
  primary keys, 92–93
  query syntax, ease of use, 90
  querying databases, 104–112
  registering, 66
  relationships, 93–96
  security, 90
  uniqueness, enforcing, 93
MVC (Model-View-Controller) architecture in, 80
projects, creating, 58–59
raw strings, 22
superusers, creating, 65
Syndication application, 240
  feeds, configuring, 240–241
  feeds, URLs for, 242
  Universal Feed Parser, 242–243
templates, 70, 135
  base templates, creating, 73–74
  for CMS example application, 196–198
  contexts, 135–136
  creating, 70–71
  embedding JavaScript actions in, 215
  extending, 250–260
  filters, 75, 137–138
  for Flatpages application, 184
  inclusion, 141
  inheritance, 139–141
  language syntax, 136–137
  liveblog example application, 209
  photo gallery example, 173–179
  tags, 136, 138–141
tuples in, 24–25
URL patterns, 70
  creating, 72
URLs, 117
  callable views, 121–122
  HTTP request-response model, 122–127
  include function, 120–121
  multiple patterns objects, 119–120
  url method, 119
  URLconfs, 59, 117–118

utility scripts, 261
cron jobs, 262-263
data import/export, 263-264
variable tags, 70
view functions, 70
creating, 71-72
liveblog example application, 216-218
views, 127
custom views, 131-132
generic views, 128-131, 221. *See also* pastebin example
as Python functions, 128

Django applications.
blog project example
admin application, 65-69
applications, creating, 61
databases, setup, 62-65
development runserver, running, 59-61
models, 62, 74
projects, creating, 58-59
templates, creating, 70-75
URL patterns, creating, 72
view functions, creating, 71-72
CMS example application, 185
admin views in, 193-195
custom managers for, 189
displaying stories, 198-199
enhancement suggestions, 202-203
imports for, 188
Markdown usage in, 190-192
model definition for, 186-189
search functionality in, 199-201
templates for, 196-198
urls.py file, 192-193
user management, 201
views.py file, 196
workflow management, 202
converting to App Engine applications, 328
accessing App Engine objects, 330
adding data, 331-332
copying App Engine code, 328
integrating App Engine Helper for Django, 329-330
linking to App Engine code, 329
testing, 330
creating, 61
for Google App Engine, 333-334
evaluating, 322-323
finding, 321-322
Flatpages, 182
enabling, 182-184
error handling, 184
templates, creating, 184
testing, 184-185
installing, 323
liveblog example application
Ajax libraries, installing, 213-214
Ajax libraries, selecting, 207-208
Ajax libraries, setup and testing, 214-215
Ajax requirements, 212

directory structure of, 208-211
planning, 206-207
view function, creating, 216-217
view function, using via JavaScript, 217-218
XML versus JSON, 212-213
pastebin example application, 221-222
cleaning up with cron jobs, 231-232
limited number of items displayed, 229-230
models.py file, 222-223
syntax highlighting in, 230-231
templates, creating, 223-225
testing, 226-229
URLs, designing, 225
photo gallery example application, 159
file uploads, preparing for, 161-162
ImageField class, testing, 163-165
models.py file, 160-161
PIL (Python Imaging Library), installing, 162-163
templates, 173-179
ThumbnailImageField class, building, 164-169
ThumbnailImageField class, usage, 169
URLs setup, 169-173
sharing, 323
testing, 274-279
doctest module (Python), 275
models, testing, 276-278
running tests, 276
tools for, 278-279
unittest module (Python), 276

django-admin.py utility, directory location of, 58

django.shortcuts module (Python), 131-132

djangogigs.com, 338

djangopeople.net, 338

Djangopluggables.com, 322

DjangoResources page (Django project wiki), 322

DjangoSites.org, 322

__doc__ attribute (Python), 54

docstrings. *See* documentation strings (Python)

doctest module (Python), 275

documentation (Django), 338

documentation strings (Python), 54-55

Dojo, 207

dollar sign ($)
in URLconfs, 118
variable (jQuery), 214

double asterisk (**), in functions (Python), 40-42, 132

downloadable files, generating, 243
charts and graphs, 246-248
CSV (comma-separated values) files, 245-246
Nagios configuration files, 243-244
vCards, 244-245

downloading
Django
development version, 302
packaged releases, 302
Google App Engine Helper for Django, 327
Google App Engine SDK, 327
Python, 295
Python-Markdown module, 190

dpaste.com, 222

DRY (Don't Repeat Yourself) principle, 62, 84
- URLs setup (photo gallery example), 169-173

dummy cache type (Django), 272

dumpdata command (Django), 104

duplicates in query results (Django), removing, 108

dynamic content, 78

dynamic instance attributes (Python), 45, 52-53

dynamic URLs (Django), 118

dynamic Web development. *See* Web development

E

Easy Install (Python add-on), 300

Eclipse IDE, 320

editing
- INSTALLED_APPS tuple, 61
- settings.py file, 61, 65
- urls.py file, 65

editors. *See* text editors

elif statement (Python), 29

else statement (Python), 29

Emacs, 319

EmailField class (Django), 91

embedding
- Django applications, 323
- JavaScript actions, 215

enabling Flatpages application, 182-184

encapsulation of methods (Django), 89-90

encoding variable (request objects), 125

enumerate function (Python), 25-26, 29
- in Django models, 30

environment (command line), 291

environment variables, 291-293

error handling in Flatpages application, 184. *See also* exception handling

escaping HTML, 197

evaluating Django applications, 322-323

example applications
- blog project
  - admin application, 65-69
  - applications, creating, 61
  - databases, setup, 62-65
  - development runserver, running, 59-61
  - models, 62, 74
  - projects, creating, 58-59
  - templates, creating, 70-75
  - URL patterns, creating, 72
  - view functions, creating, 71-72
- CMS, 185
  - admin views in, 193-195
  - custom managers for, 189
  - displaying stories, 198-199
  - enhancement suggestions, 202-203
  - imports for, 188
  - Markdown usage in, 190-192
  - model definition for, 186-189
  - search functionality in, 199-201
  - templates for, 196-198
  - urls.py file, 192-193
  - user management, 201
  - views.py file, 196
  - workflow management, 202

liveblog
- Ajax libraries, installing, 213-214
- Ajax libraries, selecting, 207-208
- Ajax libraries, setup and testing, 214-215
- Ajax requirements, 212
- directory structure of, 208-211
- planning, 206-207
- view function, creating, 216-217
- view function, using via JavaScript, 217-218
- XML versus JSON, 212-213

pastebin, 221-222
- cleaning up with cron jobs, 231-232
- limited number of items displayed, 229-230
- models.py file, 222-223
- syntax highlighting in, 230-231
- templates, creating, 223-225
- testing, 226-229
- URLs, designing, 225

photo gallery, 159
- file uploads, preparing for, 161-162
- ImageField class, testing, 163-165
- models.py file, 160-161
- PIL (Python Imaging Library), installing, 162-163
- templates, 173-179
- ThumbnailImageField class, building, 164-169
- ThumbnailImageField class, usage, 169
- URLs setup, 169-173

except clause (Python), 30-31

exception handling (Python)
- in Django, 33
  - Http404 class, 131
- exceptions, list of, 32-33
- raise statement, 32-33
- try-except blocks, 30-31
- try-finally blocks, 31-32

exclude method (Django), 104

expanding environment variable values, 292

exporting data, utility script for, 263-264

expressions (Python), statements (Python) versus, 38. *See also* regular expressions (Python)

extend method (Python), 51

extending
- base templates (admin application), 237-238
- template system (Django), 250
  - alternative templating, 258-260
  - custom filters, 256-258
  - custom tags, 250-253
  - inclusion tags, 253-256

extends tag (Django), 74, 139-141

extra method (Django), 111-112

F

factory functions (Python), 14

False value (Python), 12

FastCGI, 307-308

feeds (Django)
- configuring, 240-241
- Universal Feed Parser, 242-243
- URLs for, 242

feeds.py file, configuring, 240-241

Field class (Django), 91

field types (Django), 91-92

fields (Django), adding attributes to, 167-168

fieldsets setting (admin application), 236-237

file cache type (Django), 272-273

file extensions (Python), 10

file uploads, preparing for (photo gallery example), 161-162

FileField class (Django), 92

files (Python)
- initialization, 166-167
- methods for, 33-34
- removing, 286

FILES data structure (Django), 125

filling out forms (Django), 147-149

filter method (Django), 104

filtering
- lists (Django) in pastebin example application, 229-230
- queries (Django), 106-107
- QuerySets (Django) in generic views, 129

filters (Django)
- custom filters, creating, 256-258
- safe, 197
- slice, 229
- in templates, 75, 137-138

finally clause (Python), 31-32

finding Django applications, 321-322

fixtures (Django), 65, 113-115

Flatpages application, 182
- enabling, 182-184
- error handling, 184
- templates, creating, 184
- testing, 184-185

flexibility of Django, 84-85

float data type (Python), 12

floor division (Python), 13

flow control (Python), 28
- conditionals, 29
- loops, 29

flup module, 307-308

for loops (Python), 29

for template tag (Django), 71

foreign keys (Django), 93-94

ForeignKey class (Django), 93

Form class (Django), 142

form display options in admin application (Django), 102

formatting
- with filters, 75
- strings (Python), 22-24

forms (Django), 142
- data normalization, 150
- defining, 142-143
- displaying, 150-152
- filling out, 147-149
- model-based forms, 143
  - modifying, 145-146
  - saving, 143-145
- subclassing, 146-147
- validating, 149-150
- widgets, 152-154

fromkeys method (Python), 28

functions (Django)
- get_list_or_404, 131
- get_object_or_404, 131
- include, 120-121
- patterns, 117
- render_to_response, 131
- serialize, 216

functions (JavaScript), setInterval, 217

functions (Python), 34. *See also* methods (Python)
- * and ** in, 40-42, 132
- anonymous functions, 38
  - in Django, 39
  - lambda keyword, 38-39
- as objects, 36
  - in Django, 37-38
  - references, 36-37
- bool, 12
- calling, 34-35
- declaring, 34
- decorators, 39, 42-44
- default arguments in, 35-36
- enumerate, 29
  - in Django models, 30
- help, 54
- keyword arguments in, 35
- match, 48
- numeric functions, 14
- reversed, 18
- search, 47
- for sequences, 25-26
- sorted, 18
- views (Django) as, 128

G

generator expressions (Python), 19

generic views (Django), 128-130, 221. *See also* pastebin application
- CMS example application, 196
- customizing with dictionaries, 225
- modifying, 130-131

GET data structure (Django), 123-124

get method (Django), 104

get method (Python), 27-28

getJSON method (jQuery), 217

get_list_or_404 function (Django), 131

get_object_or_404 function (Django), 131

Git, 315

GitHub.com, 322

Google App Engine, 325. *See also* Google App Engine Helper for Django
- advantages of using, 325-326
- converting Django applications to App Engine applications, 328
  - accessing App Engine objects, 330
  - adding data, 331-332
  - copying App Engine code, 328
  - integrating App Engine Helper for Django, 329-330
  - linking to App Engine code, 329
  - testing, 330
- creating Django applications for, 333-334
- limitations of, 326-327
- objects, accessing in Django applications, 330
- pure applications for, 326
- tutorial, 327
- Web sites for information, 334-335

Google App Engine Helper for Django, 327-328
downloading, 327
integrating into Django applications, 329-330
Google App Engine SDK, downloading, 327
Google Code, 322
Google SiteSearch box, 199
Gorbatchev, Alex, 230
GPL (GNUPublic License), BSD/MIT-style licenses versus, 321
graphs, generating, 246-248
GUI animation with Ajax. *See* Ajax

H

handlers. *See* exception handling (Python)
hash mark (#), comments, 10
hash tables. *See* dictionaries (Python)
headers (Django), cache-related, controlling, 268-269
help function (Python), 54
help system for command line, 287
highlighted syntax in pastebin example application, 230-231
Holovaty, Adrian, 2
HTML (HyperText Markup Language), 79
converting Markdown content to, 191
escaping, 197
HTTP (HyperText Transfer Protocol), 78
HTTP request-response model (Django), 122
middleware, 126-127
request objects, 123-125
response objects, 125-126
Http404 class (Django), 131
Http404 error, raising, 33
HttpRequest object (Django), 123-125
HttpResponse object (Django), 125-126
changing MIME type in, 245

I

IBM DB2, 311
id attribute (Python), 50
IDs, timestamps versus, 216
if statement (Python), 29
IIS, 306
ImageField class (Django), testing, 163-165. *See also* photo gallery example
ImageFieldFile class (Django), 167-168
immutable variables (Python), as default function arguments, 36
ImportError exception (Python), 32
importing
data, utility script for, 263-264
modules (Python), 48-49
imports (Django), CMS example application, 188
imports (Python), 166
improper slicing, sequences (Python), 16
in operator (Python), 16
include function (Django), 120-121
include tag (Django), 141
inclusion of templates (Django), 141
inclusion tags (Django), creating, 253-256
indentation (Python), 9, 11, 53
IndentationError exception (Python), 32
index pages, defining URLs for, 118
IndexError exception (Python), 32

inheritance (Django), 97
with abstract base classes, 97-99
multi-table inheritance, 99-100
of templates, 139-141
__init__ initializer (Python), 45, 52
__init__.py file, 59
initial SQL files, defining, 113
initializing
files (Python), 166-167
objects (Python), 52
inner classes (Python), 46
input on command line, 289-291
insert method (Python), 18
INSTALLED_APPS tuple, editing, 61
installing
Ajax libraries (liveblog example application), 213-214
Apache Web server with mod_python, 304
Django, 302-303
applications, 323
testing installation, 303
PIL (Python Imaging Library), photo gallery example, 162-163
Python
on Mac OS X, 296
testing installation, 299
on Unix/Linux, 296
on Windows, 296-298
instantiation (Python), 45, 52
int data type (Python), 12
integrating App Engine Helper for Django into Django applications, 329-330
interactive interpreter (Python), 8-9
Django usage with, 9-10
IOError exception (Python), 32
IPAddressField class (Django), 91
IPython (Python add-on), 300-301
IRC channel for Django information, 338
isupper method (Python), 21
is_valid method (Django), 149
items method (Python), 28
iterables (Python), 14
iteritems method (Python), 28
iterkeys method (Python), 28
itervalues method (Python), 28

J

JavaScript, 79. *See also* Ajax
actions, embedding, 215
libraries. *See* Ajax libraries
using view functions via (liveblog example application), 217-218
JavaScript Object Notation. *See* JSON
joining
databases, 108-109
Django community, 337-338
jQuery, 207
installing (liveblog example application), 213-214
setup and testing (liveblog example application), 214-215
JSON (JavaScript Object Notation), 79, 212
Python interoperability with, 213
XML versus, 212-213

K

key argument (Python), 38

key collision (Python), 27

KeyboardInterrupt exception (Python), 32

KeyError exception (Python), 32

keys method (Python), 28

keyword arguments (Python), 41
- in Django, 42
- in functions, 35

keyword composition (Django), 109-110

L

lambda keyword (Python), 38-39
- in authentication decorators, 240

landing pages, defining URLs for, 118

language syntax for templates (Django), 136-137

layout of admin application, changing, 236-237

len function (Python), 25-26

libraries (Ajax), liveblog example application
- installing, 213-214
- selecting, 207-208
- setup and testing, 214-215

licensing, BSD/MIT-style licenses versus GPL (GNU Public License), 321

lighttpd, 306

limit_choices_to argument (Django), 96

line termination characters in files (Python), 34

linking App Engine code to projects, 329

Linux, installing Python on, 296

list comprehensions (Python), 18-19

list display in admin application, changing, 68-69

list formatting options in admin application (Django), 102

list function (Python), 25

listcomps. *See* list comprehensions (Python)

listing directory contents, 286

lists (Django)
- filtering in pastebin example application, 229-230
- QuerySet class as, 106

lists (Python), 14, 17-18
- as default function arguments, 36
- generator expressions, 19
- list comprehensions, 18-19
- sorting, 18, 51

list_detail.object_detail generic view (Django), 129

list_detail.object_list generic view (Django), 129

liveblog example
- Ajax libraries
  - installing, 213-214
  - selecting, 207-208
  - setup and testing, 214-215
- Ajax requirements, 212
- directory structure of, 208-211
- planning, 206-207
- view function
  - creating, 216-217
  - using via JavaScript, 217-218
- XML versus JSON, 212-213

liveblogs, defined, 205
load tag (Django templates), 253
loaddata command (Django), 104
loading modules (Python), 49
locmem cache type (Django), 266, 272
login for admin application, 66-67
long data type (Python), 12
loops (Python), 29
loose coupling of Django, 84-85

M

Mac OS X
  installing Python on, 296
  Unix command-line tools, using in, 285
mailing lists for Django information, 338
Mako templating engine, 259-260
man system (command-line help), 287
manage.py shell command (Django), 10
manage.py utility, 59
  applications, creating, 61
  databases, creating and updating, 103-104
  development runserver, running, 60
Manager class (Django), 104-105
managers (Django)
  custom manager, 248
    default object set, changing, 248-249
    methods, creating, 249-250
    methods, naming, 249
  objects, 249
many-to-many relationships, 94-95
many-to-one relationships, 93
ManyToManyField class (Django), 95
mapping types (Python), dictionaries, 26-28
Markdown (light markup language) in CMS example application, 190-192
match function (Python), 48
matching, searching versus (Python regular expressions), 48
max function (Python), 25-26
membership in sequences (Python), 16
memcached cache type (Django), 272-274
Mercurial, 315
  operational overview, 316-318
merging changes in version control systems, 314
Meta class (Django), 100-101
META data structure (Django), 125
Meta inner class (Python), 46
method variable (request objects), 125
methods (Django)
  all, 104
  for custom managers
    creating, 249-250
    naming, 249
  delete, 168
    order of operations, 169
  distinct, 108
  encapsulation, 89-90
  exclude, 104
  extra, 111-112
  filter, 104
  get, 104
  is_valid, 149

order_by, 108
process_request, 126-127
process_response, 127
reverse, 108
save, 143-145, 168
select_related, 108-109
url, 119
values, 108
values_list, 108

methods (jQuery), getJSON, 217

methods (Python). ***See also*** **functions (Python)**
for dictionaries, 27-28
for files, 33-34
for lists, 18
mutability and, 50-51
for strings, 19, 21

Microsoft SQL Server, 311

middleware (Django), 126-127
for caching, adding, 266

MIME type in HttpResponse object, changing, 245

min function (Python), 25-26

MochiKit, 207

model-based forms (Django), 143
modifying, 145-146
saving, 143-145

Model-View-Controller. ***See*** **MVC architecture**

ModelChoiceField class (Django), 146

ModelForm class (Django), 143

ModelMultipleChoiceField class (Django), 146

models (Django)
admin options, 101-102
classes (Python) and, 46-47
databases
creating and updating, 103-104
non-Django SQL features, usage of, 112-115
querying, 104-112
default ordering, setting, 74
defining, 91
CMS example application, 186-189
pastebin example application, 222-223
designing, 62
encapsulation of methods, 89-90
enumerate function (Python) in, 30
field types, 91-92
inheritance, 97
with abstract base classes, 97-99
multi-table inheritance, 99-100
Meta class, 100-101
ORM (Object-Relational Mapper), advantages of, 89-91
photo gallery example, 160-161
portability, 90
primary keys, 92-93
query syntax, ease of use, 90
registering, 66
relationships, 93
constraining relationships, 96
foreign keys, 93-94
many-to-many relationships, 94-95
one-to-one relationships, 96
security, 90
testing, 276-278
uniqueness, enforcing, 93

models (in MVC architecture), 79-81

models.py file, 61
- liveblog example application, 209
- models, designing, 62
- pastebin example application, 222-223
- photo gallery example, 160-161

modifying
- generic views (Django), 130-131
- model-based forms (Django), 145-146

modularity of Django, 84-85

modules (Python)
- calendar, 254
- csv, 245-246
- django.shortcuts, 131-132
- doctest, 275
- importing, 48-49
- loading, 49
- operator, 110
- packages, 49-50
- Python-Markdown module, downloading, 190
- re, 47
- StringIO, 245
- unittest, 276

mod_python module
- hooking Django into, 304-305
- installing, 304
- serving static files, 305-306

mod_wsgi module, 306-307

MooTools, 207

multi-table inheritance (Django), 99-100

multiple patterns objects (Django), 119-120

mutability of objects (Python), 50
- copying objects and, 51-52
- method calls and, 50-51

mutable variables (Python), as default function arguments, 36

MVC (Model-View-Controller) architecture, 79-80
- in Django, 80
- models in, 80-81
- templates in, 81-82
- views in, 81

Myghty templating engine, 259

MySQL, 310-311

N

Nagios configuration files, generating, 243-244

named arguments, positional arguments versus, 119

NameError exception (Python), 32

namespace. *See* environment (command line)

naming conventions
- managers/manager methods (Django), 249
- partial templates (Django), 255

negative indices in Python sequences, 15

nested classes. *See* inner classes (Python)

nested queries (Django), 106-107

Nginx, 306

non-Django SQL features, usage of, 112-115

None value (Python), 12

nonmembership in sequences (Python), 16

normalization. *See* data normalization
not in operator (Python), 16
NULL. *See* None value (Python)
NullBooleanField class (Django), 92
numeric data types (Python), 12-13
numeric functions (Python), 14
numeric operators (Python), 13-14

O

object cache (Django), 269-270
object-oriented programming (OOP) in Python, 44
  class definitions, 44–45
  Django models and, 46–47
  dynamic instance attributes, 52–53
  inner classes, 46
  instantiation, 45, 52
  subclassing, 46
Object-Relational Mapper. *See* ORM
objects (Django), printing, 137
objects (Python)
  dynamic instance attributes, 52–53
  functions as, 36
    in Django, 37–38
    references, 36–37
  initializing, 52
  mutability, 50
    copying objects and, 51–52
    method calls and, 50–51
objects manager (Django), 249
one-to-one relationships, 96
OneToOneField class (Django), 96
online resources. *See* Web sites
OOP (object-oriented programming) in Python, 44
  class definitions, 44–45
  Django models nad, 46–47
  dynamic instance attributes, 52–53
  inner classes, 46
  instantiation, 45, 52
  subclassing, 46
open method (Python), 33
operator module (Python), 110
operators (Python), 11
  numeric, 13–14
  for sequences, 16
options for commands, 286, 288-289
Oracle, 311
order of operations for delete method (Django), 169
ordering of models, setting, 74
order_by method (Django), 108
ORM (Object-Relational Mapper), 78
  advantages of, 89–91
  custom managers (Django), 248
    default object set, changing, 248–249
    methods, creating, 249–250
    methods, maning, 249
output on command line, 289-291
overriding default widgets (Django), 153-154

P

packaged releases of Django, 302
packages (Python), 49-50, 59
pagination, Django support for, 202

partial templates (Django), naming, 255
pastebin example, 221-222
cleaning up with cron jobs, 231-232
limiting number of items displayed, 229-230
models.py file, 222-223
syntax highlighting in, 230-231
templates, creating, 223-225
testing, 226-229
URLs, designing, 225
path variable (request objects), 125
paths, 293-294
binary directories, 293
updating for Python, 296-298
patterns function (Django), 117
patterns objects (Django), using multiple, 119-120
per-view caching (Django), 268
percent sign (%), string format operator, 22-24
permissions, database access, 63
photo gallery example, 159
file uploads, preparing for, 161-162
ImageField class, testing, 163-165
models.py file, 160-161
PIL (Python Imaging Library), installing, 162-163
templates, 173-179
ThumbnailImageField class
building, 164-169
usage, 169
URLs setup, 169-173
PIL (Python Imaging Library), installing (photo gallery example), 162-163
pipe (|)
operator (Django), 109, 290
template filters (Django), 137
pipes (command line), 289-291
plain text templating (Django), 258
planning liveblog example application, 206-207
plus sign (+), concatenation, 16
pop method (Python), 28
popitem method (Python), 28
portability of models (Django), 90
positional arguments, named arguments versus, 119
POST data structure (Django), 123-124
PostgreSQL, 309-310
pound sign (#), comments, 10
presentation methods (in Web development), 79
primary keys (Django), 92-93
print command (Python), 8-9
printing
environment state, 291
objects (Django), troubleshooting, 137
process_request method (Django), 126-127
process_response method (Django), 127
program names in Unix, 287-288
project management software, 318-319
projects (Django), creating, 58-59
prompts (command line), 286
Prototype, 207
.py file extension, 10
PyCha, 246-248
PyDev, 320

Python
add-ons, 300
Easy Install, 300
IPython, 300-301
classes
creating, 44-45
Django models and, 46-47
dynamic instance attributes, 52-53
inner classes, 46
instantiating, 45, 52
subclassing, 46
coding style, 53
documentation strings, 54-55
indentation, 53
single-line suites, 54
spaces versus tabs, 53
comments, 10
data types
Boolean, 12
complex, 13
Decimal, 13
dictionaries, 26-28
float, 12
int, 12
iterables, 14
lists, 14, 17-19
long, 12
numeric, 12-13
sequences, 14-17, 25-26
strings, 14, 19-24
tuples, 15, 24-25
described, 7
dictionaries
COOKIES data structure, 124
FILES data structure, 125
GET and POST data structures, 123-124
META data structure, 125
REQUEST data structure, 124
session data structure, 124
downloading, 295
exception handling
in Django, 33
exceptions, list of, 32-33
raise statement, 32-33
try-except blocks, 30-31
try-finally blocks, 31-32
expressions, statements versus, 38
file extensions, 10
files
initialization, 166-167
methods for, 33-34
flow control, 28
conditionals, 29
loops, 29
functions, 34
* and ** in, 40-42, 132
anonymous functions, 38-39
calling, 34-35
declaring, 34
decorators, 39, 42-44
default arguments in, 35-36
keyword arguments in, 35
numeric functions, 14
as objects, 36-38
views (Django) as, 128

indentation, 9, 11
installing
  on Mac OS X, 296
  testing installation, 299
  on Unix/Linux, 296
  on Windows, 296-298
interactive interpreter, 8-9
  Django usage with, 9-10
JSON interoperability with, 213
modules
  calendar, 254
  csv, 245-246
  importing, 48-49
  loading, 49
  packages, 49-50
  StringIO, 245
object-oriented programming (OOP) in, 44
  class definitions, 44-45
  Django models and, 46-47
  dynamic instance attributes, 52-53
  inner classes, 46
  instantiation, 45, 52
  subclassing, 46
objects
  dynamic instance attributes, 52-53
  initializing, 52
  mutability, 50-52
operators, 11
  numeric operators, 13-14
packages, 59
regexes in URLconfs, 118
regular expressions, 47
  re module, 47
  searching versus matching, 48
sequences
  tuples, 48
  unpacking in functions, 40
source code, examining, 165
strings, raw strings, 64
symbols, lack of, 11
tuples, two-tuples, 236
updating path to, 296-298
variables, 10-11
versions of, 8, 295
viewing version number, 295

Python Extensions for Windows, 296

Python Imaging Library (PIL), installing (photo gallery example), 162-163

Python-Markdown module, downloading, 190

python-mode (Emacs), 319

Pythonic, Django as, 84

PYTHONPATH environment variable, 57

Q

Q class (Django), 109-110

~Q, in keyword composition (Django), 109-110

queries (Django), building dynamically, 42

query syntax
  of models (Django), ease of use, 90
  of databases (Django), 104-112

QueryDict class (Django), 123

QuerySet class (Django), 104-105

as database query, 105-106

joining databases, 108-109

keyword composition with Q and ~Q, 109-110

as list container, 106

modifying SQLqueries, 111-112

as nested queries, 106-107

removing duplicates, 108

sorting query results, 108

QuerySets (Django), filtering in generic views, 129

quote marks, types of, 64

R

raise statement (Python), 32-33

raising Http404 error, 33

range function (Python), 25-26

rapid development in Django, 85

raw strings (Django), 22

raw strings (Python), 22, 64

in regular expressions, 47

raw_post_data variable (request objects), 125

RCS (Revision Control System), 313

re module (Python), 47

read method (Python), 34

readlines method (Python), 34

recording changesets, 317

redirection (command line), 289-291

redirection operators (<>), 291

references of function objects (Python), 36-37

regexes (Python), in URLconfs, 118

registering

as Django developer, 338

models (Django), 66

regression testing, 279

regular expressions (Python), 47

re module, 47

searching versus matching, 48

relationships between models (Django), 93

constraining relationships, 96

foreign keys, 93-94

many-to-many relationships, 94-95

one-to-one relationships, 96

removing. *See* deleting

render_to_response function (Django), 131

repetition/duplication of sequences (Python), 16

repositories (version control systems), copying, 317

repr function (Python), 25

REQUEST data structure (Django), 124

request middleware (Django), 126-127

request objects (Django), 123-125

requests (Web development), 78

response middleware (Django), 127

response objects (Django), 125-126

responses (Web development), 78

reverse method (Django), 108

reverse method (Python), 51

reversed function (Python), 18, 26, 51

root URLs (Django), regexes for, 118

rows (in database tables), 78

rstrip method (Python), 21, 34
running development runserver, 59-61
runserver command (Django), 60

S

safe filter (Django), 197
save method (Django), 143-145, 168
saving
  model-based forms (Django), 143-145
  thumbnail images (photo gallery example), 168-169
SCCS (Source Code Control System), 313
schema definition with initial SQL files, 113
scripts. *See* utility scripts (Django)
search function (Python), 47
search functionality in CMS example application, 199-201
searching, matching versus (Python), 48
security of models (Django), 90
selecting Ajax libraries (liveblog example application), 207-208
select_related method (Django), 108-109
Selenium, 279
Selenium Core, 279
Selenium IDE, 279
Selenium RC, 279
self keyword (Python), 44
sequences (Python), 14-15
  concatenation, 16-17
  copying, 16
  functions for, 25-26
  lists, 14, 17-18
    generator expressions, 19
    list comprehensions, 18-19
    sorting, 18, 51
  operations on, 16
  slicing, 15-16
  strings, 14, 19-21
    formatting, 22-24
    string designators, 22
  tuples, 15, 24
    in Django, 24-25
    single-element tuples, 48
  unpacking in functions, 40
serialize function (Django), 216
server performance, benchmarking, 265-266
server variables for request objects (Django), 124-125
session data structure (Django), 124
sessions, 124
setdefault method (Python), 27-28
setInterval function (JavaScript), 217
settings.py file, 59
  database settings, 63
  editing, 61, 65
setup of Ajax libraries (liveblog example application), 214-215
shallow copies (Python), 16, 51
shared applications directory, creating, 323
sharing Django applications, 323
shell (Python). *See* interactive interpreter (Python)

shortcuts for custom views (Django), 131-132

simple.direct_to_template generic view (Django), 129

single-element tuples (Python), 48

single-line suites (Python), 54

site-packages directory, installing applications to, 323

site-wide caching (Django), 267

slice filter (Django), 229

slicing sequences (Python), 15-16

sort method (Python), 18, 51

sorted function (Python), 18, 26, 51

sorting
- lists (Python), 18
- query results (Django), 108

source code (Python), examining, 165

source control. *See* version control systems

spaces (Python), tabs (Python) versus, 53

spam in pastebins, 228

Sphinx, 203

split method (Python), 21

splitlines method (Python), 21

SQL (Structured Query Language), 78
- non-Django SQL features, usage of, 112-115

sql command (Django), 103

SQL databases. *See* databases

sqlall command (Django), 103

sqlclear command (Django), 103

sqlcustom command (Django), 103, 113

sqlindexes command (Django), 103

SQLite, 308
- setup for blog project example, 63-64

sqlrest command (Django), 103

ssi tag (Django), 141

startapp command (Django), 61

startproject command (Django), 58

statements (Python)
- for conditionals, 29
- expressions (Python) versus, 38
- for loops, 29
- raise, 32-33

static class members, 44

static content, 78

static files, serving in Apache Web server, 305-306

stderr stream, 290

stdin stream, 290

stdout stream, 290

str function (Python), 25

string designators (Python), 22

StringIO module (Python), 245

strings in URLs, 192

strings (Python), 14, 19-21
- concatenation, 16-17
- formatting, 22-24
- raw strings, 64
  - in regular expressions, 47
- string designators, 22

stringvalue decorator (Django), 258

strip method (Python), 21

Structured Query Language. *See* SQL

style of coding. *See* coding style (Python)

subclassing
  forms (Django), 146-147
  in Python, 46
Subversion, 314-315
sum function (Python), 26
superusers (Django), creating, 65
symbolic groups (Python), 118
symbols (Python), lack of, 11
syncdb command (Django), 64, 103
synchronization of databases (Django), 103
Syndication application (Django), 240
  feeds
    configuring, 240-241
    Universal Feed Parser, 242-243
    URLs for, 242
Syntax Highlighter, 230
syntax highlighting (pastebin example application), 230-231
SyntaxError exception (Python), 33

T

tables (in databases), 78
  creating, 64-65
tabs (Python), spaces (Python) versus, 53
tag libraries (Django), 252
tags (Django templates), 136, 138-139
  block, 139-141
  extends, 139-141
  include, 141
  ssi, 141
template languages, 79
template system (Django), extending, 250
  alternative templating, 258-260
  custom filters, 256-258
  custom tags, 250-253
  inclusion tags, 253-256
template tags (Django), cache, 271
templates (Django), 70, 135
  base templates
    creating, 73-74
    extending, 237-238
  CMS example application, 196-198
  contexts, 135-136
  creating, 70-71
    for pastebin example application, 223-225
  embedding JavaScript actions in, 215
  filters, 137-138
    applying, 75
  Flatpages application, 184
  inclusion, 141
  inheritance, 139-141
  language syntax, 136-137
  liveblog example application, 209
  partial templates, naming, 255
  photo gallery example, 173-179
  tags, 136, 138-139
    block, 139-141
    extends, 139-141
    include, 141
    ssi, 141
templates (in MVC architecture), 81-82
templatetags directory (Django), 252

Terminal application, 285

testing

Ajax libraries (liveblog example application), 214-215

applications (Django), 274-275

doctest module (Python), 275

models, testing, 276-278

running tests, 276

tools for, 278-279

unittest module (Python), 276

caching (Django), 266-267

codebase (Django), 279-281

converted App Engine applications, 330

feeds (Django), 242-243

Flatpages application, 184-185

ImageField class (Django), photo gallery example, 163-165

installation (Django), 303

pastebin example application, 226-229

Python installation, 299

text editors, 319-320

TextField class (Django), 91

TextMate, 320

third-party applications. *See* Django applications

third-party tools. *See* add-ons

throwing exceptions (Python), raise statement, 32-33

ThumbnailImageField class (photo gallery example)

building, 164-169

usage, 169

timestamps, IDs versus, 216

toolkits (Ajax). *See* libraries (Ajax)

tools for testing applications (Django), 278-279

Trac, 319

triggers (SQL), support for, 113

triple quotes, in Python strings, 22-24

troubleshooting. *See also* testing

admin application problems, 66

printing objects (Django), 137

True value (Python), 12

trunk (version control systems), 314

try-except blocks (Python), 30-31

try-finally blocks (Python), 31-32

tuple function (Python), 25

tuples (Python), 15, 24

in Django, 24-25

single-element tuples, 48

two-tuples, 236

Twill, 279

two-tuples (Python), 236

type attribute (Python), 50

TypeError exception (Python), 33

types (Python). *See* data types (Python)

U

unbound forms (Django), 147-149

UnboundLocalError exception (Python), 33

unicode function (Python), 25

Unicode strings (Python), 22

uniqueness of models (Django), enforcing, 93

unittest module (Python), 276

Universal Feed Parser, 242-243

Unix

command line. *See* command line

installing Python on, 296

program names explained, 287–288

unnamed functions. *See* anonymous functions (Python)

unpacking sequences (Python) in functions, 40

update method (Python), 28, 51

updating path for Python, 296-298

uploading files, preparing for (photo gallery example), 161-162

upper method (Python), 18

url method (Django), 119

URL patterns (Django), 70

CMS example application, 192–193

creating, 72

URLconfs (Django), 59, 117-118

URLField class (Django), 91

URLs (Django), 117

callable views, 121–122

designing for pastebin example application, 225

for feeds, 242

HTTP request-response model, 122

middleware, 126–127

request objects, 123–125

response objects, 125–126

include function, 120–121

multipled patterns objects, 119–120

setup (photo gallery example), 169–173

strings in, 192

url method, 119

URLconfs, 59, 117–118

urls.py file, 59

CMS example application, 192–193

editing, 65

liveblog example application, 208

user management in CMS example application, 201

user variable (request objects), 125

utility scripts (Django), 261

cron jobs, 262–263

data import/export, 263–264

V

"vacuuming." *See* cron jobs

validating forms (Django), 149-150

value attribute (Python), 50

ValueError exception (Python), 33

values method (Django), 108

values method (Python), 28

values_list method (Django), 108

varargs (Python), 41

variable arguments (Python), 41

variable tags (Django), 70

variables (Python), 10-11

environment variables, 57, 291–293

mutable/immutable variables, as default function arguments, 36

vCards, generating, 244-245

version control systems, 313-314
centralized version control, 315
decentralized version control, 315-316
merging changes in, 314
operational overview, 316-318
trunk and branches approach, 314
version numbers (Python), viewing, 295
view functions (Django), 70
creating, 71-72
liveblog example application, 216-217
using via JavaScript (liveblog example application), 217-218
viewing Python version number, 295
views (Django), 127
callable views, 121-122
CMS example application
admin views, 193-195
generic views, 196
creating for admin application, 238-239
custom views, 131-132
generic views, 128-130, 221. *See also* pastebin example
customizing with dictionaries, 225
modifying, 130-131
as Python functions, 128
views (in MVC architecture), 79-81
views (SQL), support for, 113
views.py file, 61
in CMS example application, 196
Vim, 320
vObject module, 244
void. *See* None value (Python)

W

Web development
communication methods, 78
data storage methods, 78
Django core philosophies, 82
DRY (Don't Repeat Yourself), 84
modularity, 84-85
as Pythonic, 84
rapid development, 85
MVC (Model-View-Controller) architecture, 79-80
in Django, 80
models in, 80-81
templates in, 81-82
views in, 81
presentation methods, 79
Web frameworks, development of, 1
Web Server Gateway Interface (WSGI), 306-307
Web servers
Apache with mod_python
hooking Django into, 304-305
installing, 304
serving static files, 305-306
development runserver, 303
running, 59-61
FastCGI, 307-308
WSGI (Web Server Gateway Interface), 306-307
Web sites
finding Django applications, 321-322
Google App Engine information, 334-335
to register as Django developer, 338

while loops (Python), 29

Widget class (Django), 152

widgets (Django), 152-154

Willison, Simon, 2

Windows

installing Python on, 296-298

Unix command-line tools, using in, 285

workflow management in CMS example application, 202

wrappers. *See* decorators

write method (Python), 34

writelines method (Python), 34

WSGI (Web Server Gateway Interface), 306-307

X–Z

XHTML, 79

XML, 79

JSON versus, 212-213

xrange function (Python), 26

Yahoo! User Interface (YUI), 208

YAML file, 326

YUI (Yahoo! User Interface), 208

zip function (Python), 26