Chapter 1: Introduction to Django

This book is about Django, a Web development framework that saves you time and makes Web development a joy. Using Django, you can build and maintain high-quality Web applications with minimal fuss.

At its best, Web development is an exciting, creative act; at its worst, it can be a repetitive, frustrating nuisance. Django lets you focus on the fun stuff – the crux of your Web application – while easing the pain of the repetitive bits. In doing so, it provides high-level abstractions of common Web development patterns, shortcuts for frequent programming tasks, and clear conventions for how to solve problems. At the same time, Django tries to stay out of your way, letting you work outside the scope of the framework as needed.

The goal of this book is to make you a Django expert. The focus is twofold. First, we explain, in depth, what Django does and how to build Web applications with it. Second, we discuss higher-level concepts where appropriate, answering the question "How can I apply these tools effectively in my own projects?" By reading this book, you'll learn the skills needed to develop powerful Web sites quickly, with code that is clean and easy to maintain.

What Is a Web Framework?

Django is a prominent member of a new generation of *Web frameworks* – but what does that term mean, precisely?

To answer that question, let's consider the design of a Web application written in Python *without* a framework. Throughout this book, we'll take this approach of showing you basic ways of getting work done*without* shortcuts, in the hope that you'll recognize why shortcuts are so helpful. (It's also valuable to know how to get things done without shortcuts because shortcuts aren't always available. And most importantly, knowing *why* things work the way they do makes you a better Web developer.)

One of the simplest, most direct ways to build a Python Web app from scratch is to use the Common Gateway Interface (CGI) standard, which was a popular technique circa 1998. Here's a high-level explanation of how it works: just create a Python script that outputs HTML, then save the script to a Web server with a ".cgi" extension and visit the page in your Web browser. That's it.

Here's an example Python CGI script that displays the ten most recently published books from a database. Don't worry about syntax details; just get a feel for the basic things it's doing:

```
#!/usr/bin/env python


import MySQLdb


print "Content-Type: text/html\n"

print "<html><head><title>Books</title></head>"
```

```
print "<body>"

print "<h1>Books</h1>"

print "<ul>"


connection = MySQLdb.connect(user='me',
passwd='letmein', db='my_db')

cursor = connection.cursor()

cursor.execute("SELECT name FROM books ORDER BY
pub_date DESC LIMIT 10")


for row in cursor.fetchall():

    print "<li>%s</li>" % row[0]


print "</ul>"

print "</body></html>"


connection.close()
```

First, to fulfill the requirements of CGI, this code prints a "Content-Type" line, followed by a blank line. It prints some introductory HTML, connects to a database and runs a query to retrieve the names of the latest ten books. Looping over those books, it generates an HTML list of the titles. Finally, it prints the closing HTML and closes the database connection.

With a one-off page like this one, the write-it-from-scratch approach isn't necessarily bad. For one thing, this code is simple to comprehend – even a novice developer can read these 16 lines of Python and understand everything it does, from start to finish. There's nothing else to learn, no other code to read. It's also simple to deploy: just save this code in a file that ends with ".cgi", upload that file to a Web server, and visit that page with a browser.

But despite its simplicity, this approach has a number of problems and annoyances. Ask yourself these questions:

- What happens when multiple parts of your application need to connect to the database? Surely that database-connecting code shouldn't need to be duplicated in each individual CGI script. The pragmatic thing to do would be to refactor it into a shared function.

- Should a developer really have to worry about printing the "Content-Type" line and remembering to close the database connection? This sort of boilerplate reduces programmer productivity and introduces opportunities for mistakes. These setup- and teardown-related tasks would best be handled by some common infrastructure.
- What happens when this code is reused in multiple environments, each with a separate database and password? At this point, some environment-specific configuration becomes essential.
- What happens when a Web designer who has no experience coding Python wishes to redesign the page? One wrong character could crash the entire application. Ideally, the logic of the page – the retrieval of book titles from the database – would be separate from the HTML display of the page, so that a designer could edit the latter without affecting the former.

These problems are precisely what a Web framework intends to solve. A Web framework provides a programming infrastructure for your applications, so that you can focus on writing clean, maintainable code without having to reinvent the wheel. In a nutshell, that's what Django does.

The MVC Design Pattern

Let's dive in with a quick example that demonstrates the difference between the previous approach and a Web framework's approach. Here's how you might write the previous CGI code using Django. The first thing to note is that that we split it over four Python files (`models.py`, `views.py`, `urls.py`) and an HTML template (`latest_books.html`):

```
# models.py (the database tables)


from django.db import models


class Book(models.Model):
    name = models.CharField(max_length=50)
    pub_date = models.DateField()



# views.py (the business logic)


from django.shortcuts import render
from models import Book
```

```
def latest_books(request):
    book_list = Book.objects.order_by('-pub_date')[:10]
    return render(request, 'latest_books.html',
{'book_list': book_list})



# urls.py (the URL configuration)

from django.conf.urls.defaults import *
import views

urlpatterns = patterns('',
    (r'^latest/$', views.latest_books),
)



# latest_books.html (the template)

<html><head><title>Books</title></head>
<body>
<h1>Books</h1>
<ul>
{% for book in book_list %}
<li>{{ book.name }}</li>
{% endfor %}
</ul>
```

```
</body></html>
```

Again, don't worry about the particulars of syntax; just get a feel for the overall design. The main thing to note here is the *separation of concerns*:

- The `models.py` file contains a description of the database table, represented by a Python class. This class is called a *model*. Using it, you can create, retrieve, update and delete records in your database using simple Python code rather than writing repetitive SQL statements.
- The `views.py` file contains the business logic for the page. The `latest_books()` function is called a *view*.
- The `urls.py` file specifies which view is called for a given URL pattern. In this case, the URL `/latest/`will be handled by the `latest_books()` function. In other words, if your domain is example.com, any visit to the URL http://example.com/latest/ will call the `latest_books()` function.
- The `latest_books.html` file is an HTML template that describes the design of the page. It uses a template language with basic logic statements – e.g., `{% for book in book_list %}`.

Taken together, these pieces loosely follow a pattern called Model-View-Controller (MVC). Simply put, MVC is way of developing software so that the code for defining and accessing data (the model) is separate from request-routing logic (the controller), which in turn is separate from the user interface (the view). (We'll discuss MVC in more depth in Chapter 5.)

A key advantage of such an approach is that components are *loosely coupled*. Each distinct piece of a Django-powered Web application has a single key purpose and can be changed independently without affecting the other pieces. For example, a developer can change the URL for a given part of the application without affecting the underlying implementation. A designer can change a page's HTML without having to touch the Python code that renders it. A database administrator can rename a database table and specify the change in a single place, rather than having to search and replace through a dozen files.

In this book, each component of MVC gets its own chapter. Chapter 3 covers views, Chapter 4 covers templates, and Chapter 5 covers models.

Django's History

Before we dive into more code, we should take a moment to explain Django's history. We noted above that we'll be showing you how to do things *without* shortcuts so that you more fully understand the shortcuts. Similarly, it's useful to understand *why* Django was created, because knowledge of the history will put into context why Django works the way it does.

If you've been building Web applications for a while, you're probably familiar with the problems in the CGI example we presented earlier. The classic Web developer's path goes something like this:

1. Write a Web application from scratch.
2. Write another Web application from scratch.
3. Realize the application from step 1 shares much in common with the application from step 2.
4. Refactor the code so that application 1 shares code with application 2.
5. Repeat steps 2-4 several times.
6. Realize you've invented a framework.

This is precisely how Django itself was created!

Django grew organically from real-world applications written by a Web development team in Lawrence, Kansas, USA. It was born in the fall of 2003, when the Web programmers at the *Lawrence Journal-World*newspaper, Adrian Holovaty and Simon Willison, began using Python to build applications.

The World Online team, responsible for the production and maintenance of several local news sites, thrived in a development environment dictated by journalism deadlines. For the sites – including LJWorld.com, Lawrence.com and KUsports.com – journalists (and management) demanded that features be added and entire applications be built on an intensely fast schedule, often with only days' or hours' notice. Thus, Simon and Adrian developed a time-saving Web development framework out of necessity – it was the only way they could build maintainable applications under the extreme deadlines.

In summer 2005, after having developed this framework to a point where it was efficiently powering most of World Online's sites, the team, which now included Jacob Kaplan-Moss, decided to release the framework as open source software. They released it in July 2005 and named it Django, after the jazz guitarist Django Reinhardt.

Now, several years later, Django is a well-established open source project with tens of thousands of users and contributors spread across the planet. Two of the original World Online developers (the "Benevolent Dictators for Life," Adrian and Jacob) still provide central guidance for the framework's growth, but it's much more of a collaborative team effort.

This history is relevant because it helps explain two key things. The first is Django's "sweet spot." Because Django was born in a news environment, it offers several features (such as its admin site, covered in Chapter 6) that are particularly well suited for "content" sites – sites like Amazon.com, craigslist.org, and washingtonpost.com that offer dynamic, database-driven information. Don't let that turn you off, though – although Django is particularly good for developing those sorts of sites, that doesn't preclude it from being an effective tool for building any sort of dynamic Web site. (There's a difference between being *particularly effective* at something and being *ineffective* at other things.)

The second matter to note is how Django's origins have shaped the culture of its open source community. Because Django was extracted from real-world code, rather than being an academic exercise or commercial product, it is acutely focused on solving Web development problems that Django's developers themselves have faced – and continue to face. As a result, Django itself is actively improved on an almost daily basis. The framework's maintainers have a vested interest in making sure Django saves developers time, produces applications that are easy to maintain and performs well under load. If nothing else, the developers are motivated by their own selfish desires to save themselves time and enjoy their jobs. (To put it bluntly, they eat their own dog food.)

How to Read This Book

In writing this book, we tried to strike a balance between readability and reference, with a bias toward readability. Our goal with this book, as stated earlier, is to make you a Django expert, and we believe the best way to teach is through prose and plenty of examples, rather than providing an exhaustive but bland catalog of Django features. (As the saying goes, you can't expect to teach somebody how to speak a language merely by teaching them the alphabet.)

With that in mind, we recommend that you read Chapters 1 through 12 in order. They form the foundation of how to use Django; once you've read them, you'll be able to build and deploy Django-powered Web sites. Specifically, Chapters 1 through 7 are the "core curriculum," Chapters 8 through 11 cover more advanced Django usage, and Chapter 12 covers deployment. The remaining chapters, 13 through 20, focus on specific Django features and can be read in any order.

The appendixes are for reference. They, along with the free documentation athttp://www.djangoproject.com/, are probably what you'll flip back to occasionally to recall syntax or find quick synopses of what certain parts of Django do.

Required Programming Knowledge

Readers of this book should understand the basics of procedural and object-oriented programming: control structures (e.g., `if`, `while`, `for`), data structures (lists, hashes/dictionaries), variables, classes and objects.

Experience in Web development is, as you may expect, very helpful, but it's not required to understand this book. Throughout the book, we try to promote best practices in Web development for readers who lack this experience.

Required Python Knowledge

At its core, Django is simply a collection of libraries written in the Python programming language. To develop a site using Django, you write Python code that uses these libraries. Learning Django, then, is a matter of learning how to program in Python and understanding how the Django libraries work.

If you have experience programming in Python, you should have no trouble diving in. By and large, the Django code doesn't perform a lot of "magic" (i.e., programming trickery whose implementation is difficult to explain or understand). For you, learning Django will be a matter of learning Django's conventions and APIs.

If you don't have experience programming in Python, you're in for a treat. It's easy to learn and a joy to use! Although this book doesn't include a full Python tutorial, it highlights Python features and functionality where appropriate, particularly when code doesn't immediately make sense. Still, we recommend you read the official Python tutorial, available online at http://docs.python.org/tut/. We also recommend Mark Pilgrim's free book *Dive Into Python*, available at http://www.diveintopython.net/ and published in print by Apress.

Required Django Version

This book covers Django 1.4.

Django's developers maintain backwards compatibility as much as possible, but occasionally introduce some backwards incompatible changes. The changes in each release are always covered in the release notes, which you can find here: https://docs.djangoproject.com/en/dev/releases/1.X

Getting Help

One of the greatest benefits of Django is its kind and helpful user community. For help with any aspect of Django – from installation, to application design, to database design, to deployment – feel free to ask questions online.

- The django-users mailing list is where thousands of Django users hang out to ask and answer questions. Sign up for free at http://www.djangoproject.com/r/django-users.
- The Django IRC channel is where Django users hang out to chat and help each other in real time. Join the fun by logging on to #django on the Freenode IRC network.