Mastering Django:

Core

The Complete Guide to Django 1.8 LTS

Nigel George

Mastering Django: Core

The Complete Guide to Django 1.8 LTS

Nigel George

This book is for sale at http://leanpub.com/masteringdjangocore

This version was published on 2017-05-01

ISBN 978-0-9946168-0-7

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Tweet This Book!

Please help Nigel George by spreading the word about this book on Twitter!

The suggested hashtag for this book is #masteringdjangocore.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#masteringdjangocore

Contents

Acknowledgements

First and foremost, I would like to thank the original authors of the Django Book - Adrian Holovaty and Jacob Kaplan-Moss. They provided such a strong foundation that it has really been a delight writing this new edition.

Equal first in the shout out has to be the Django community. Vibrant and collaborative, the Django community is what really stood out to this cynical old businessman many years ago when I first discovered the "new kid on the web-framework block". It's your support that makes Django so great. Thank you.

About the Author

Nigel George is a business systems developer specializing in the application of Open Source technologies to solve common business problems. He has a broad range of experience in software development - from writing database apps for small business, to developing the backend and UI for a distributed sensor network at the University of Newcastle, Australia.

Nigel also has over 15 years' experience in technical writing for business. He has written several training manuals and hundreds of technical procedures for corporations and Australian government departments. He has been using Django since version 0.96 and has written applications in C, C#, C++, VB, VBA, HTML, JavaScript, Python and PHP.

He has another book on Django - *Beginning Django CMS* - published by Apress in December 2015.

Nigel lives in Newcastle, NSW, Australia.

Introduction

This year (2014) it will be 30 years since I plugged the 5.25" DOS 3.3 disk into my school's very first Apple IIe computer and discovered BASIC.

In the intervening years I have written more lines of code than I could guess in about a dozen languages. I still write code every week - although the list of languages, and number of lines are somewhat diminished these days.

Over the years I have seen plenty of horrible code and some really good stuff too. In my own work, I have written my fair share of good and bad. Interestingly, not once in my career have I been employed as a programmer. I had my own IT business for five years, and have been in businesses large and small - mostly in R&D, technical and operations management - but never working solely as a programmer.

What I have been is the guy that gets called up to **Get Stuff Done**.

Emphasized for good reason - business is all about Getting Stuff Done. When everything has to work yesterday, religious wars over curly braces and pontification over which language is best for what application become trivialities.

Having read dozens and dozens of textbooks on all the various programming languages I have used, I know why you are here reading the introduction, so let's get right to the point.

Why should you care about Django?

While it is a given that Django is not the only web framework that will allow you to Get Stuff Done, I can confidently say one thing - if you want to write clean, intelligible code and build high performance, good looking modern websites quickly, then you will definitely benefit from working through this book.

I have deliberately not rattled off comparisons with other languages and frameworks because that's not the point - all languages and the frameworks and tools built on them have strengths and weaknesses. However, having worked with many of them over the years, I am totally convinced that Django stands way out

in front for ease of use and ability to allow a programmer to produce robust, secure, and bug free code quickly.

Django is spectacularly good at getting out of your way when you just need to Get Something Done, but still exposes all the good stuff just under the surface when you want to dig down further.

Django is also built with Python, arguably the most intelligible and easy to learn programming language. Of course these strengths do bring one challenge. Because both Python and Django hide an enormous amount of power and functionality just below the surface, it can be a bit confusing for beginners. This is where this book comes in. It's designed to quickly get you moving on your own Django projects, and then ultimately teach you everything you need to know to successfully design, develop, and deploy a site that you'll be proud of.

Adrian and Jacob wrote the original Django Book because they firmly believed that Django makes Web development better. I think Django's longevity and exponential growth in the years since the publication of the original Django Book is testament to this belief. As per the original, this book is open source and all are welcome to improve it by either submitting comments and suggestions at the Mastering Django website[1], or sending me an email to nigel@masteringdjango.com. I, like many, get a great deal of pleasure out of working with Django - it truly is as exciting, fun and useful as Adrian and Jacob had hoped it would be!

About This Book

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. *Mastering Django: Core* is a completely revised and updated version of the Django Book - first published by Apress in 2007 as *The Definitive Guide to Django: Web Development Done Right* and then republished as *The Django Book* by the original authors in 2009. The latter publication was released as an Open Source project under the Gnu Free Documentation License (GFDL).

Mastering Django: Core could be considered an unofficial 3rd edition of the Django Book, although I will leave it up to Jacob and the Django community to

[1] http://masteringdjango.com

decide whether it deserves that honor. Personally, I just wanted to see it back out there because, like many Django programmers, the Django Book is where I got started. To retain Adrian and Jacob's original desire for the Django Book to be accessible as possible, the source code for *Mastering Django: Core* is freely available online on the Mastering Django website.

The main goal of this book is to make you a Django expert. The focus is twofold. First, I explain in depth what Django does and how to build Web applications with it. Second, I 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.

The secondary, but no less important, goal of this book is to provide a programmer's manual that covers the current Long Term Support (LTS) version of Django. Django has matured to the point where it is seeing many commercial and business critical deployments. As such, this book is intended to provide the definitive up-to-date resource for commercial deployment of Django 1.8 LTS. The electronic version of this book will be kept in sync with Django 1.8 right up until the end of extended support (2018).

How to Read This Book

In writing *Mastering Django: Core*, I have tried to maintain a similar balance between readability and reference as the first book, however Django has grown considerably since 2007 and with increased power and flexibility, comes some additional complexity. Django still has one of the shortest learning curves of all the web application frameworks, but there is still some solid work ahead of you if you want to become a Django expert. This book retains the same "learn by example" philosophy as the original book, however some of the more complex sections (database configuration for example) have been moved to later chapters. This is so that you can first learn how Django works with a simple, out-of-the-box configuration and then build on your knowledge with more advanced topics later.

With that in mind, I recommend that you read Chapters 1 through 13 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 6 are the "core curriculum," Chapters 7 through 12 cover more

advanced Django usage, and Chapter 13 covers deployment. The remaining chapters, 14 through 21, focus on specific Django features and can be read in any order. The appendices are for reference. They, along with the free documentation at the Django Project[2], 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, I 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, I recommend you read the official Python tutorial[3]. I also recommend Mark Pilgrim's free book *Dive Into Python*, available online at `http://www.diveintopython.net/` and published in print by Apress.

[2] http://www.djangoproject.com/
[3] http://docs.python.org/tut/

Required Django Version

This book covers Django 1.8 Long Term Support (LTS). This is the long term support version of Django, with full support until at least April 2018.

If you have an early version of Django, it is recommended that you upgrade to the latest version of Django 1.8 LTS. At the time of printing (July 2016), the most current production version of Django 1.8 LTS is 1.8.13.

If you have installed a later version of Django, please note that while Django's developers maintain backwards compatibility as much as possible, some backwards incompatible changes do get introduced occasionally. The changes in each release are always covered in the release notes, which you can find at `https://docs.djangoproject.com/en/dev/releases/`.

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.

Introduction to Django

Great open source software almost always comes about because one or more clever developers had a problem to solve and no viable or cost effective solution available. Django is no exception. Adrian and Jacob have long since "retired" from the project, but the fundamentals of what drove them to create Django live on. It is this solid base of real-world experience that has made Django as successful as it is. In recognition of their contribution, I think it best we let them introduce Django in their own words (edited and reformatted from the original book).

Introducing Django

By Adrian Holovaty and Jacob Kaplan-Moss - December 2009

In the early days, Web developers wrote every page by hand. Updating a Web site meant editing HTML; a "redesign" involved redoing every single page, one at a time. As Web sites grew and became more ambitious, it quickly became obvious that that approach was tedious, time-consuming, and ultimately untenable.

A group of enterprising hackers at NCSA (the National Center for Supercomputing Applications, where Mosaic, the first graphical Web browser, was developed) solved this problem by letting the Web server spawn external programs that could dynamically generate HTML. They called this protocol the Common Gateway Interface, or CGI, and it changed the Web forever. It's hard now to imagine what a revelation CGI must have been: instead of treating HTML pages as simple files on disk, CGI allows you to think of your pages as resources generated dynamically on demand.

The development of CGI ushered in the first generation of dynamic Web sites. However, CGI has its problems: CGI scripts need to contain a lot of repetitive "boilerplate" code, they make code reuse difficult, and they can be difficult for first-time developers to write and understand.

PHP fixed many of these problems, and it took the world by storm - it's now the most popular tool used to create dynamic Web sites, and dozens of similar

languages (ASP, JSP, etc.) followed PHP's design closely. PHP's major innovation is its ease of use: PHP code is simply embedded into plain HTML; the learning curve for someone who already knows HTML is extremely shallow.

But PHP has its own problems; it's very ease of use encourages sloppy, repetitive, ill-conceived code. Worse, PHP does little to protect programmers from security vulnerabilities, and thus many PHP developers found themselves learning about security only once it was too late.

These and similar frustrations led directly to the development of the current crop of "third-generation" Web development frameworks. With this new explosion of Web development comes yet another increase in ambition; Web developers are expected to do more and more every day.

Django was invented to meet these new ambitions.

Django's History

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.

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 5) 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.

Django lets you build deep, dynamic, interesting sites in an extremely short time. Django is designed to let you focus on the fun, interesting parts of your job 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 on 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.

We wrote this book because we firmly believe that Django makes Web development better. It's designed to quickly get you moving on your own Django projects, and then ultimately teach you everything you need to know to successfully design, develop, and deploy a site that you'll be proud of.

Chapter 1: Getting Started

There are two very important things you need to do to get started with Django:

1. Install Django (obviously); and
2. Get a good understanding of the Model-View-Controller (MVC) design pattern.

The first, installing Django, is really simple and detailed in the first part of this chapter. The second is just as important, especially if you are a new programmer or coming from using a programming language that does not clearly separate the data and logic behind your website from the way it is displayed. Django's philosophy is based on *loose coupling*, which is the underlying philosophy of MVC. We will be discussing loose coupling and MVC in much more detail as we go along, but if you don't know much about MVC, then you best not skip the second half of this chapter, because understanding MVC will make understanding Django *so* much easier.

Installing Django

Before you can start learning how to use Django, you must first install some software on your computer. Fortunately, this is a simple three step process:

1. Install Python.
2. Install a Python Virtual Environment.
3. Install Django.

If this does not sound familiar to you don't worry, in this chapter I assume you have never installed software from the command line before and will lead you through it step by step.

I have written this section for those of you running Windows. While there is a strong *nix and OSX user base for Django, most new users are on Windows. If

you are using Mac or Linux, there are a large number of resources on the Internet; with the best place to start being Django's own installation instructions[4].

For Windows users, your computer can be running any recent version of Windows (Vista, 7, 8.1 or 10). This chapter also assumes you're installing Django on a desktop or laptop computer and will be using the development server and SQLite to run all the example code in this book. This is by far the easiest, and best way to setup Django when you are first starting out.

If you do want to go to a more advanced installation of Django, your options are covered in Chapter 13 - Deploying Django, Chapter 20 - More on installing Django and Chapter 21 - Advanced Database Management.

If you are using Windows, I recommend that you try out Visual Studio for all your Django development. Microsoft have made a significant investment in providing support for Python and Django programmers. This includes full IntelliSense support for Python/Django and incorporation of all of Django's command line tools into the VS IDE.

Best of all it's entirely free. I know, who would have expected that from M$??, but it's true!

See Appendix G for a complete installation guide for Visual Studio Community 2015, as well as a few tips on developing Django in Windows.

Installing Python

Django itself is written purely in Python, so the first step in installing the framework is to make sure you have Python installed.

Python Versions

Django version 1.8 LTS works with Python version 2.7, 3.3, 3.4 and 3.5. For each version of Python, only the latest micro release (A.B.C) is supported.

If you are just trialling Django, it doesn't really matter whether you use Python 2 or Python 3. If, however, you are planning on eventually deploying code to a

[4] https://docs.djangoproject.com/en/1.8/topics/install/

live website, Python 3 should be your first choice. The Python wiki[5] puts the reason behind this very succinctly:

> Short version: Python 2.x is legacy, Python 3.x is the present and future of the language

Unless you have a very good reason to use Python 2 (e.g. legacy libraries), Python 3 is the way to go.

NOTE: All of the code samples in this book are written in Python 3

Installation

If you're on Linux or Mac OS X, you probably have Python already installed. Type `python` at a command prompt (or in Applications/Utilities/Terminal, in OS X). If you see something like this, then Python is installed:

```
Python 2.7.5 (default, June 27 2015, 13:20:20)
[GCC x.x.x] on xxx Type "help", "copyright", "credits"
or "license" for more information.
>>>
```

You can see that, in the above example, Python interactive mode is running Python 2.7. This is a trap for inexperienced users. On Linux and Mac OS X machines, it is common for both Python 2 and Python 3 to be installed. If your system is like this, you need to type `python3` in front of all your commands, rather than python to run Django with Python 3.

Assuming Python is not installed in your system, we first need to get the installer. Go to `https://www.python.org/downloads/` and click the big yellow button that says "Download Python 3.x.x".

At the time of writing, the latest version of Python is 3.5.1, but it may have been updated by the time you read this, so the numbers may be slightly different.

[5] https://wiki.python.org/moin/Python2orPython3

DO NOT download version 2.7.x as this is the old version of Python. All of the code in this book is written in Python 3, so you will get compilation errors if you try to run the code on Python 2.

Once you have downloaded the Python installer, go to your Downloads folder and double click the file "python-3.x.x.msi" to run the installer. The installation process is the same as any other Windows program, so if you have installed software before, there should be no problem here, however the is one extremely important customization you must make.

Do not forget this next step as it will solve most problems that arise from incorrect mapping of pythonpath (an important variable for Python installations) in Windows.

By default, the Python executable is not added to the Windows PATH statement. For Django to work properly, Python must be listed in the PATH statement. Fortunately, this is easy to rectify:

- In Python 3.4.x, When the installer opens the customization window, the option "Add python.exe to Path" is not selected, you must change this to "Will be installed on local hard drive" as shown in Figure 1-1.
- In Python 3.5.x you make sure "Add Python 3.5 to PATH" is checked before installing (Figure 1-2).

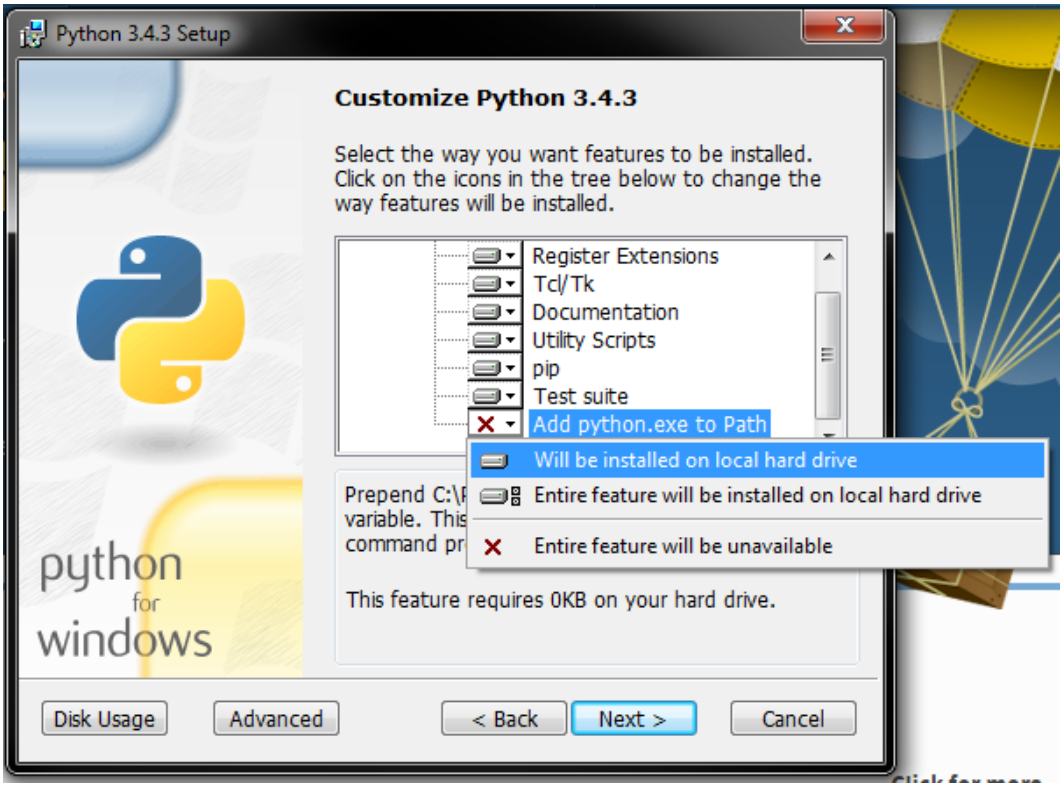

Figure 1-1: *Add Python to PATH (Version 3.4.x).*

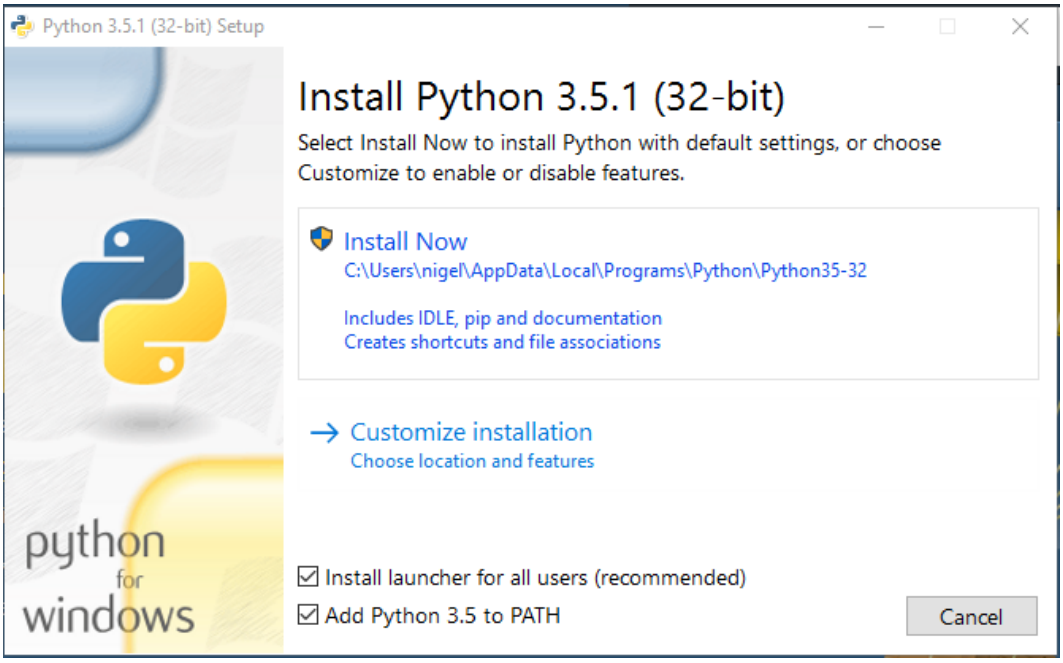

Figure 1-2: *Add Python to PATH (Version 3.5.x).*

Once Python is installed, you should be able to re-open the command window and type python at the command prompt and get something like this:

```
Python 3.5.1 (v3.5.1:37a07cee5969, Dec  6 2015,
01:38:48) [MSC v.1900 32 bit (Intel)] on win32 Type "help", "copyright", "credits"
or "license" for more information.
>>>
```

While you are at it, there is one more important thing to do. Exit out of Python with CTRL-C. At the command prompt type, the following and hit enter:

```
python -m pip install -U pip
```

The output will be something similar to this:

```
C:\Users\nigel>python -m pip install -U pip
Collecting pip
  Downloading pip-8.1.2-py2.py3-none-any.whl (1.2MB)
    100% |################################| 1.2MB 198kB/s
Installing collected packages: pip
Found existing installation: pip 7.1.2
  Uninstalling pip-7.1.2:
    Successfully uninstalled pip-7.1.2
Successfully installed pip-8.1.2
```

You don't need to understand exactly what this command does right now; put briefly `pip` is the Python package manager. It's used to install Python packages: pip is actually a recursive acronym for "Pip Installs Packages". Pip is important for the next stage of our install process, but first we need to make sure we are running the latest version of pip (8.1.2 at the time of writing), which is exactly what this command does.

Installing a Python Virtual Environment

If you are going to use Microsoft Visual Studio (VS), you can stop here and jump to Appendix G. VS only requires that you install Python, everything else VS does for you from inside the Integrated Development Environment (IDE).

All of the software on your computer operates interdependently - each program has other bits of software that it depends on (called dependencies) and settings that it needs to find the files and other software it needs to run (call environment variables).

When you are writing new software programs, it is possible (and common!) to modify dependencies and environment variables that your other software depends on. This can cause numerous problems, so should be avoided.

A Python virtual environment solves this problem by wrapping all the dependencies and environment variables that your new software needs into a file system separate from the rest of the software on your computer.

Some of you who have looked at other tutorials will note that this step is often described as optional. This is not a view I support, nor is it supported by a number of Django's core developers.

The advantages of developing Python applications (of which Django is one) within a virtual environment are manifest and not worth going through here. As a beginner, you just need to take my word for it - running a virtual environment for Django development is **not** optional.

The virtual environment tool in Python is called `virtualenv` and we install it from the command line using pip:

```
pip install virtualenv
```

The output from your command window should look something like this:

```
C:\Users\nigel>pip install virtualenv
  Collecting virtualenv
  Downloading virtualenv-15.0.2-py2.py3-none-any.whl (1.8MB)
    100% |################################| 1.8MB 323kB/s
Installing collected packages: virtualenv
Successfully installed virtualenv-15.0.2
```

Once `virtualenv` is installed, you need to create a virtual environment for your project by typing:

```
virtualenv env_mysite
```

Most examples on the Internet use "env" as your environment name. This is bad; principally because it's common to have several virtual environments installed to test different configurations, and "env" is not very descriptive. For example, you may be developing an application that must run on Python 2.7 and Python 3.4. Environments named "env_someapp_python27" and "env_someapp_python34" are going to be a lot easier to distinguish than if you had named them "env" and "env1".

In this example, I have kept it simple as we will only be using one virtual environment for our project, so I have used "env_mysite". The output from your command should look something like this:

```
C:\Users\nigel>virtualenv env_mysite Using base prefix 'c:\\users\\nigel\\appdata\\lo\
cal\\programs\\python\\python35-32'
New python executable in C:\Users\nigel\env_mysite\Scripts\python.exe
Installing setuptools, pip, wheel...done.
```

Once `virtualenv` has finished setting up your new virtual environment, open Windows Explorer and have a look at what `virtualenv` created for you. In your home directory, you will now see a folder called `\env_mysite` (or whatever name you gave the virtual environment). If you open the folder, you will see the following:

```
\Include
\Lib
\Scripts
\src
```

Virtualenv has created a complete Python installation for you, separate from your other software, so you can work on your project without affecting any of the other software on you system.

To use this new Python virtual environment, we have to activate it, so let's go back to the command prompt and type the following:

```
env_mysite\scripts\activate
```

This will run the activate script inside your virtual environment's `\scripts` folder. You will notice your command prompt has now changed:

```
(env_mysite) C:\Users\nigel>
```

The `(env_mysite)` at the beginning of the command prompt lets you know that you are running in the virtual environment. Our next step is to install Django.

Installing Django

Now that we have Python and are running a virtual environment, installing Django is super easy, just type the command:

```
pip install django==1.8.13
```

This will instruct pip to install Django into your virtual environment. Your command output should look like this:

```
(env_mysite) C:\Users\nigel>pip install django==1.8.13
Collecting django==1.8.13
  Downloading Django-1.8.13-py2.py3-none-any.whl (6.2MB)
    100% |################################| 6.2MB 107kB/s
Installing collected packages: django
Successfully installed django-1.8.13
```

In this case, we are explicitly telling pip to install Django 1.8.13, which is the latest version of Django 1.8 LTS at the time of writing. If you are installing Django, it's good practice to check the Django Project website for the latest version of Django 1.8 LTS.

In case you were wondering, typing in `pip install django` will install the latest stable release of Django. If you want information on installing the latest development release of Django, see Chapter 20.

For some post-installation positive feedback, take a moment to test whether the installation worked. At your virtual environment command prompt, start the Python interactive interpreter by typing `python` and hitting enter. If the installation was successful, you should be able to import the module `django`:

```
(env_mysite) C:\Users\nigel>python Python 3.5.1 (v3.5.1:37a07cee5969, Dec  6 2015,
01:38:48) [MSC v.1900 32 bit (Intel)] on win32 Type "help", "copyright", "credits"
or "license" for more information.
>>> import django >>> django.get_version()
'1.8.13'
```

Setting Up a Database

This step is not necessary in order to complete any of the examples in this book. Django comes with SQLite installed by default. SQLite requires no configuration on your part. If you would like to work with a "large" database engine like PostgreSQL, MySQL, or Oracle, see Chapter 21.

Starting a Project

Once you've installed Python, Django and (optionally) your database server/library, you can take the first step in developing a Django application by creating a *project*.

A project is a collection of settings for an instance of Django. If this is your first time using Django, you'll have to take care of some initial setup. Namely, you'll need to auto-generate some code that establishes a Django project: a collection of settings for an instance of Django, including database configuration, Django-specific options and application-specific settings.

I am assuming at this stage you are still running the virtual environment from the previous installation step. If not, you will have to start it again with `env_mysite\scripts\activate\`. From your virtual environment command line, run the following command:

```
django-admin startproject mysite
```

This will create a `mysite` directory in your current directory (in this case `\env_mysite\`). If you want to create your project in a directory other than the root, you can create a new directory, change into that directory and run the `startproject` command from there.

Warning! You'll need to avoid naming projects after built-in Python or Django components. In particular, this means you should avoid using names like "django" (which will conflict with Django itself) or "test" (which conflicts with a built-in Python package).

Let's look at what `startproject` created:

```
mysite/
  manage.py
  mysite/
    __init__.py
    settings.py
    urls.py
    wsgi.py
```

These files are:

- The outer `mysite/` root directory. It's just a container for your project. Its name doesn't matter to Django; you can rename it to anything you like.
- `manage.py`. A command-line utility that lets you interact with your Django project in various ways. You can read all the details about `manage.py` on the Django Project website[6].
- The inner `mysite/` directory. It's the Python package for your project. It's the name you'll use to import anything inside it (e.g. `mysite.urls`).
- `mysite/__init__.py`. An empty file that tells Python that this directory should be considered a Python package. (Read more about packages in the official Python docs[7] if you're a Python beginner.).
- `mysite/settings.py`. Settings/configuration for this Django project. Appendix D will tell you all about how settings work.
- `mysite/urls.py`. The URL declarations for this Django project; a "table of contents" of your Django-powered site. You can read more about URLs in Chapters 2 and 7.
- `mysite/wsgi.py`. An entry-point for WSGI-compatible web servers to serve your project. See Chapter 13 for more details.

Django Settings

Now, edit `mysite/settings.py`. It's a normal Python module with module-level variables representing Django settings. First step while you're editing `settings.py`, is to set `TIME_ZONE` to your time zone. Note the `INSTALLED_APPS` setting at the top of the file. That holds the names of all Django applications that are activated in this Django instance. Apps can be used in multiple projects,

[6] https://docs.djangoproject.com/en/1.8/ref/django-admin/
[7] https://docs.python.org/tutorial/modules.html#packages

and you can package and distribute them for use by others in their projects. By default, `INSTALLED_APPS` contains the following apps, all of which come with Django:

- `django.contrib.admin` – The admin site.
- `django.contrib.auth` – An authentication system.
- `django.contrib.contenttypes` – A framework for content types.
- `django.contrib.sessions` – A session framework.
- `django.contrib.messages` – A messaging framework.
- `django.contrib.staticfiles` – A framework for managing static files.

These applications are included by default as a convenience for the common case. Some of these applications makes use of at least one database table though, so we need to create the tables in the database before we can use them. To do that, run the following command:

```
python manage.py migrate
```

The `migrate` command looks at the `INSTALLED_APPS` setting and creates any necessary database tables according to the database settings in your `settings.py` file and the database migrations shipped with the app (we'll cover those later). You'll see a message for each migration it applies.

The Development Server

Let's verify your Django project works. Change into the outer `mysite` directory, if you haven't already, and run the following commands:

```
python manage.py runserver
```

You'll see the following output on the command line:

```
Performing system checks...

0 errors found June 12, 2016 - 08:48:58 Django version 1.8.13, using settings 'mysite\
.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

You've started the Django development server, a lightweight Web server written purely in Python. We've included this with Django so you can develop things rapidly, without having to deal with configuring a production server - such as Apache - until you're ready for production.

Now's a good time to note: **don't** use this server in anything resembling a production environment. **It's intended only for use while developing**.

Now that the server's running, visit `http://127.0.0.1:8000/` with your Web browser. You'll see a "Welcome to Django" page in pleasant, light-blue pastel (Figure 1-3). It worked!

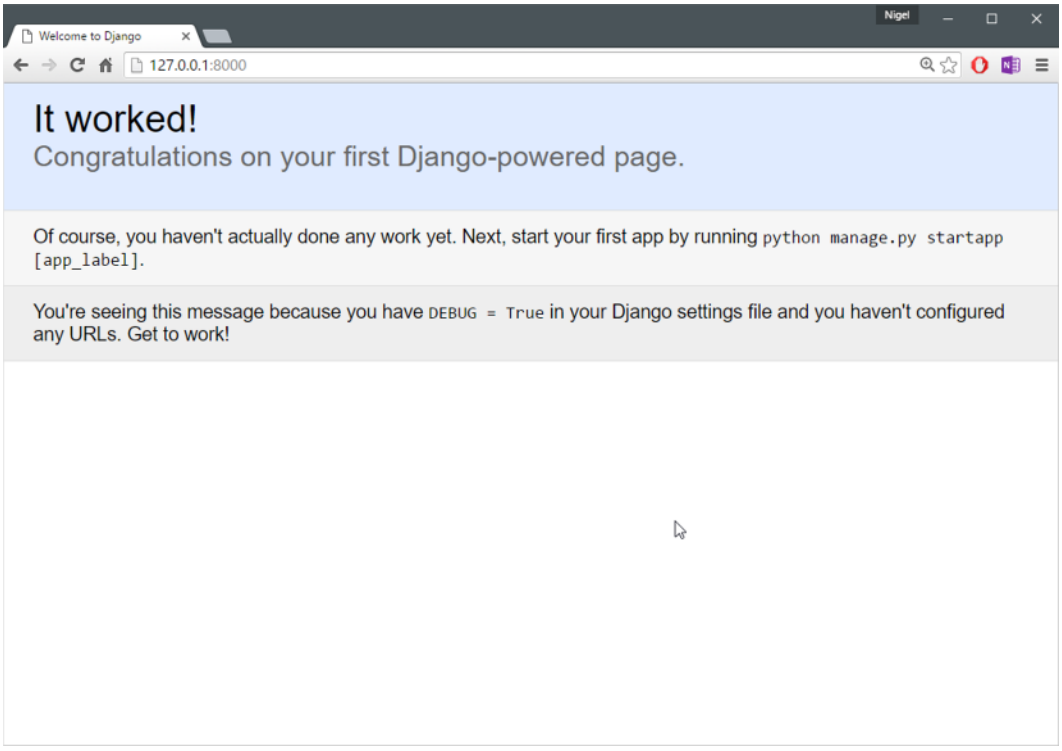

Figure 1-3: *Django's welcome page*

Automatic reloading of runserver

The development server automatically reloads Python code for each request as needed. You don't need to restart the server for code changes to take effect. However, some actions like adding files don't trigger a restart, so you'll have to restart the server in these cases.

The Model-View-Controller (MVC) Design Pattern

MVC has been around as a concept for a long time, but has seen exponential growth since the advent of the Internet because it is the best way to design client-server applications. All of the best web frameworks are built around the MVC concept. At the risk of starting a flame war, I contest that if you are not using MVC to design web apps, you are doing it wrong. As a concept, the MVC design pattern is really simple to understand:

- The **model(M)** is a model or representation of your data. It's not the actual data, but an interface to the data. The model allows you to pull data from your database without knowing the intricacies of the underlying database. The model usually also provides an *abstraction* layer with your database, so that you can use the same model with multiple databases.
- The **view(V)** is what you see. It's the presentation layer for your model. On your computer, the view is what you see in the browser for a Web app, or the UI for a desktop app. The view also provides an interface to collect user input.
- The **controller(C)** controls the flow of information between the model and the view. It uses programmed logic to decide what information is pulled from the database via the model and what information is passed to the view. It also gets information from the user via the view and implements business logic: either by changing the view, or modifying data through the model, or both.

Where it gets difficult is the vastly different interpretations of what actually happens at each layer - different frameworks implement the same functionality in different ways. One framework "guru" might say a certain function belongs in a view, while another might vehemently defend the need for it to be in the controller.

You, as a budding programmer who Gets Stuff Done, do not have to care about this because in the end, it *doesn't matter*. As long as you understand how Django implements the MVC pattern, you are free to move on and get some real work done. Although, watching a flame war in a comment thread can be a highly amusing distraction...

Django follows the MVC pattern closely, however it does use its own logic in the implementation. Because the "C" is handled by the framework itself and most of the excitement in Django happens in models, templates and views, Django is often referred to as an *MTV framework*. In the MTV development pattern:

- **M stands for "Model,"** the data access layer. This layer contains anything and everything about the data: how to access it, how to validate it, which behaviors it has, and the relationships between the data. We will be looking closely at Django's models in Chapter 4.
- **T stands for "Template,"** the presentation layer. This layer contains presentation-related decisions: how something should be displayed on a Web

page or other type of document. We will explore Django's templates in Chapter 3.

- **V stands for "View,"** the business logic layer. This layer contains the logic that accesses the model and defers to the appropriate template(s). You can think of it as the bridge between models and templates. We will be checking out Django's views in the next chapter.

This is probably the only unfortunate bit of naming in Django, because Django's view is more like the controller in MVC, and MVC's view is actually a Template in Django. It is a little confusing at first, but as a programmer getting a job done, you really won't care for long. It is only a problem for those of us who have to teach it. Oh, and to the flamers of course.

What's Next?

Now that you have everything installed and the development server running, you're ready to move on to Django's views and learning the basics of serving Web pages with Django.

Chapter 2: Views and URLconfs

In the previous chapter, I explained how to set up a Django project and run the Django development server. In this chapter, you'll learn the basics of creating dynamic Web pages with Django.

Your First Django-Powered Page: Hello World

As our first goal, let's create a Web page that outputs that famous example message: "Hello world." If you were publishing a simple "Hello world" Web page without a Web framework, you'd simply type "Hello world" into a text file, call it "`hello.html`", and upload it to a directory on a Web server somewhere. Notice in that process you've specified two key pieces of information about that Web page: its contents (the string `"Hello world"`) and its URL (e.g. `http://www.example.com/hello.html`). With Django, you specify those same two things, but in a different way. The contents of the page are produced by a *view function*, and the URL is specified in a *URLconf*. First, let's write our "Hello world" view function.

Your First View

Within the `mysite` directory that we created in the last chapter, create an empty file called `views.py`. This Python module will contain our views for this chapter. Our "Hello world" view is simple. Here's the entire function, plus import statements, which you should type into the `views.py` file:

```
from django.http import HttpResponse

def hello(request):
    return HttpResponse("Hello world")
```

Let's step through this code one line at a time:

- First, we import the class `HttpResponse`, which lives in the `django.http` module. We need to import this class because it's used later in our code.
- Next, we define a function called `hello`- the view function. Each view function takes at least one parameter, called `request` by convention. This is an object that contains information about the current Web request that has triggered this view, and is an instance of the class `django.http.HttpRequest`.

In this example, we don't do anything with `request`, but it must be the first parameter of the view nonetheless. Note that the name of the view function doesn't matter; it doesn't have to be named in a certain way in order for Django to recognize it. We're calling it `hello` here, because that name clearly indicates the gist of the view, but it could just as well be named `hello_wonderful_beautiful_world`, or something equally revolting. The next section, "Your First URLconf", will shed light on how Django finds this function.

The function is a simple one-liner: it merely returns an `HttpResponse` object that has been instantiated with the text `"Hello world"`.

The main lesson here is this: a view is just a Python function that takes an `HttpRequest` as its first parameter and returns an instance of `HttpResponse`. In order for a Python function to be a Django view, it must do these two things. (There are exceptions, but we'll get to those later.)

Your First URLconf

If, at this point, you ran `python manage.py runserver` again, you'd still see the "Welcome to Django" message, with no trace of our "Hello world" view anywhere. That's because our `mysite` project doesn't yet know about the `hello` view; we need to tell Django explicitly that we're activating this view at a particular URL. Continuing our previous analogy of publishing static HTML files, at this point we've created the HTML file but haven't uploaded it to a directory on the server yet.

To hook a view function to a particular URL with Django, we use a *URLconf*. A URLconf is like a table of contents for your Django-powered Web site. Basically, it's a mapping between URLs and the view functions that should be called for those URLs. It's how you tell Django, "For this URL, call this code, and for that URL, call that code."

For example, when somebody visits the URL `/foo/`, call the view function `foo_view()`, which lives in the Python module `views.py`. When you executed `django-admin startproject` in the previous chapter, the script created a URLconf for you automatically: the file `urls.py`.

By default, it looks something like this:

```
"""mysite URL Configuration

The urlpatterns list routes URLs to views. For more information please see:
    https://docs.djangoproject.com/en/1.8/topics/http/urls/
Examples:
Function views
    1. Add an import:  from my_app import views
    2. Add a URL to urlpatterns:  url(r'^$', views.home, name='home')
Class-based views
    1. Add an import:  from other_app.views import Home
    2. Add a URL to urlpatterns:  url(r'^$', Home.as_view(), name='home')
Including another URLconf
    1. Add an import:  from blog import urls as blog_urls
    2. Add a URL to urlpatterns:  url(r'^blog/', include(blog_urls))
"""
from django.conf.urls import include, url
from django.contrib import admin

urlpatterns = [
    url(r'^admin/', include(admin.site.urls)),
]
```

If we ignore the documentation comments at the top of the file, here's the essence of a URLconf:

```
from django.conf.urls import include, url
from django.contrib import admin

urlpatterns = [
    url(r'^admin/', include(admin.site.urls)),
]
```

Let's step through this code one line at a time:

- The first line imports two functions from the `django.conf.urls` module: `include` which allows you to include a full Python import path to another URLconf module, and `url` which uses a regular expression to pattern match the URL in your browser to a module in your Django project.
- The second line calls the function `admin` from the `django.contrib` module. This function is called by the `include` function to load the URLs for the Django admin site.
- The third line is `urlpatterns` - a simple list of `url()` instances.

The main thing to note here is the variable `urlpatterns`, which Django expects to find in your URLconf module. This variable defines the mapping between URLs and the code that handles those URLs. To add a URL and view to the URLconf, just add a mapping between a URL pattern and the view function. Here's how to hook in our `hello` view:

```
from django.conf.urls import include, url
from django.contrib import admin
from mysite.views import hello
urlpatterns = [
    url(r'^admin/', include(admin.site.urls)),
    url(r'^hello/$', hello),
]
```

We made two changes here:

- First, we imported the `hello` view from its module - `mysite/views.py`, which translates into `mysite.views` in Python import syntax. (This assumes `mysite/views.py` is on your Python path.)
- Next, we added the line `url(r'^hello/$', hello),` to `urlpatterns`. This line is referred to as a URLpattern. The `url()` function tells Django how to handle the url that you are configuring. The first argument is a pattern-matching string (a regular expression; more on this in a bit) and the second argument is the view function to use for that pattern. `url()` can take other optional arguments as well, which we'll cover in more depth in Chapter 7.

One more important detail we've introduced here is that 'r' character in front of the regular expression string. This tells Python that the string is a "raw string" - its contents should not interpret backslashes.

In normal Python strings, backslashes are used for escaping special characters - such as in the string "`\n`", which is a one-character string containing a newline. When you add the `r` to make it a raw string, Python does not apply its backslash escaping - so, "`r'\n'`" is a two-character string containing a literal backslash and a lowercase "n".

There's a natural collision between Python's usage of backslashes and the backslashes that are found in regular expressions, so it's best practice to use raw strings any time you're defining a regular expression in Django.

In a nutshell, we just told Django that any request to the URL `/hello/` should be handled by the `hello` view function.

It's worth discussing the syntax of this URLpattern, as it may not be immediately obvious. Although we want to match the URL `/hello/`, the pattern looks a bit different than that. Here's why:

- Django removes the slash from the front of every incoming URL before it checks the URLpatterns. This means that our URLpattern doesn't include the leading slash in `/hello/`. At first, this may seem unintuitive, but this requirement simplifies things - such as the inclusion of URLconfs within other URLconfs, which we'll cover in Chapter 7.
- The pattern includes a caret (^) and a dollar sign ($). These are regular expression characters that have a special meaning: the caret means "require that the pattern matches the start of the string," and the dollar sign means "require that the pattern matches the end of the string."

  This concept is best explained by example. If we had instead used the pattern `^hello/` (without a dollar sign at the end), then *any* URL starting with `/hello/` would match, such as `/hello/foo` and `/hello/bar`, not just `/hello/`.

  Similarly, if we had left off the initial caret character (i.e., `hello/$`), Django would match *any* URL that ends with `hello/`, such as `/foo/bar/hello/`.

  If we had simply used `hello/`, without a caret *or* dollar sign, then any URL containing `hello/` would match, such as `/foo/hello/bar`.

  Thus, we use both the caret and dollar sign to ensure that only the URL `/hello/` matches - nothing more, nothing less. Most of your URLpatterns will start with carets and end with dollar signs, but it's nice to have the flexibility to perform more sophisticated matches.

You may be wondering what happens if someone requests the URL `/hello` (that is, *without* a trailing slash). Because our URLpattern requires a trailing slash, that URL would *not* match. However, by default, any request to a URL that *doesn't* match a URLpattern and *doesn't* end with a slash will be redirected to the same URL with a trailing slash (This is regulated by the `APPEND_SLASH` Django setting, which is covered in Appendix D).

The other thing to note about this URLconf is that we've passed the `hello` view function as an object without calling the function. This is a key feature of Python (and other dynamic languages): functions are first-class objects, which means you can pass them around just like any other variables. Cool stuff, eh?

To test our changes to the URLconf, start the Django development server, as you did in Chapter 1, by running the command `python manage.py runserver`. (If you left it running, that's fine, too. The development server automatically detects changes to your Python code and reloads as necessary, so you don't have to restart the server between changes.) The server is running at the address `http://127.0.0.1:8000/`, so open up a Web browser and go to `http://127.0.0.1:8000/hello/`. You should see the text "Hello world" - the output of your Django view (Figure 2-1).

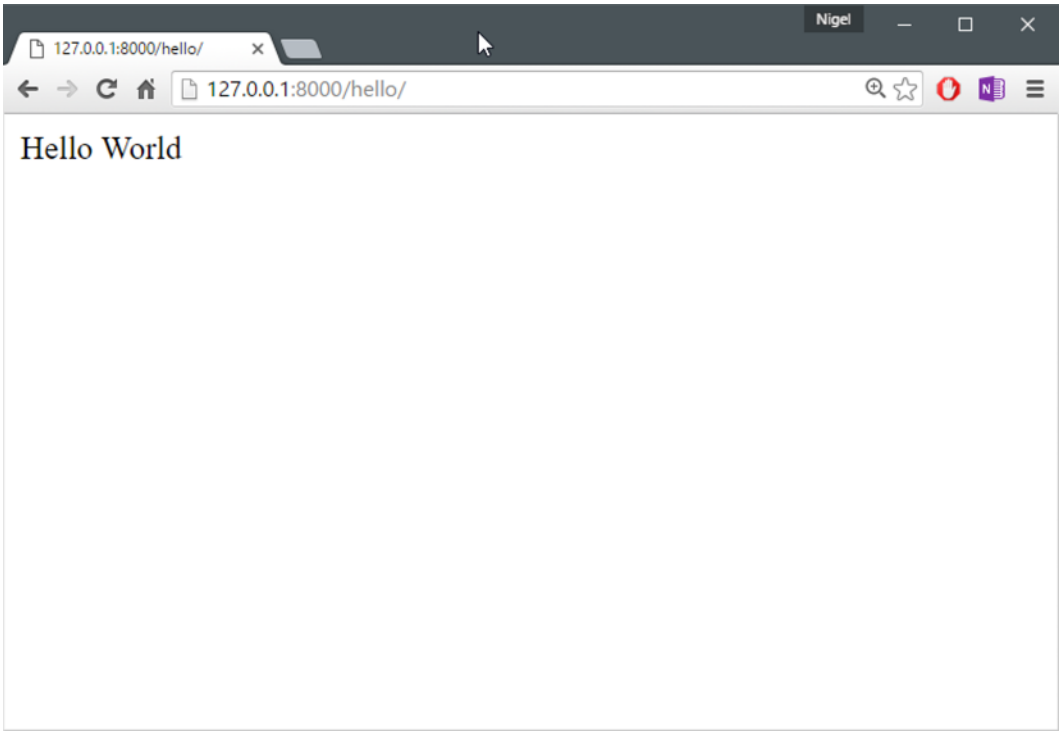

Figure 2-1: *Hooray! Your first Django view.*

Regular Expressions

Regular expressions (or *regexes*) are a compact way of specifying patterns in text. While Django URLconfs allow arbitrary regexes for powerful URL matching, you'll probably only use a few regex symbols in practice. Table 2-1 lists a selection of common symbols.

Table 2-1: *Common regex symbols*

Symbol	Matches
. (dot)	Any single character
\d	Any single digit
[A-Z]	Any character between A and Z (uppercase)
[a-z]	Any character between a and z (lowercase)
[A-Za-z]	Any character between a and z (case-insensitive)
+	One or more of the previous expression (e.g., \d+ matches one or more digits)
[^/]+	One or more characters until (and not including) a forward slash
?	Zero or one of the previous expression (e.g., \d? matches zero or one digits)
*	Zero or more of the previous expression (e.g., \d* matches zero, one or more than one digit)
{1,3}	Between one and three (inclusive) of the previous expression (e.g., \d{1,3} matches one, two or three digits)

For more on regular expressions, see the Python regex documentation[8].

A Quick Note About 404 Errors

At this point, our URLconf defines only a single URLpattern: the one that handles requests to the URL `/hello/`. What happens when you request a different URL? To find out, try running the Django development server and visiting a page such as `http://127.0.0.1:8000/goodbye/`. You should see a "Page not found" message (Figure 2-2). Django displays this message because you requested a URL that's not defined in your URLconf.

[8] https://docs.python.org/3.4/library/re.html

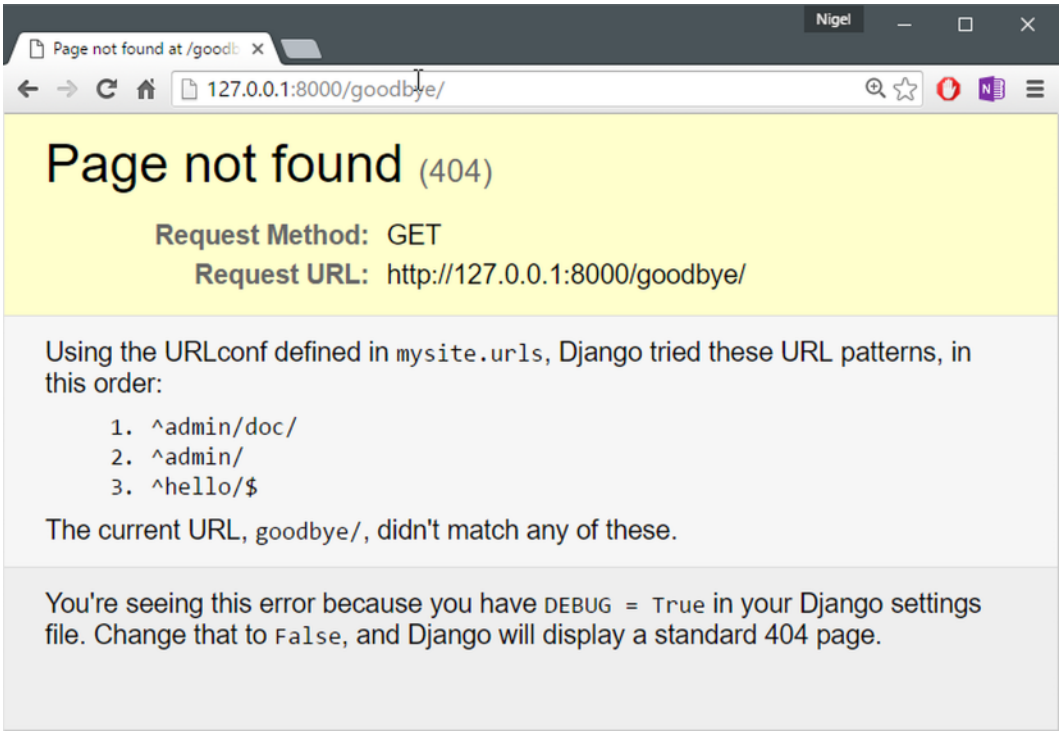

Figure 2-2: *Django's 404 page*

The utility of this page goes beyond the basic 404 error message. It also tells you precisely which URLconf Django used and every pattern in that URLconf. From that information, you should be able to tell why the requested URL threw a 404.

Naturally, this is sensitive information intended only for you, the Web developer. If this were a production site deployed live on the Internet, you wouldn't want to expose that information to the public. For that reason, this "Page not found" page is only displayed if your Django project is in *debug mode*.

I'll explain how to deactivate debug mode later. For now, just know that every Django project is in debug mode when you first create it, and if the project is not in debug mode, Django outputs a different 404 response.

A Quick Note About the Site Root

As explained in the last section, you'll see a 404 error message if you view the site root – `http://127.0.0.1:8000/`. Django doesn't magically add anything to the site root; that URL is not special-cased in any way.

It's up to you to assign it to a URLpattern, just like every other entry in your URLconf. The URLpattern to match the site root is a bit unintuitive, though, so it's worth mentioning.

When you're ready to implement a view for the site root, use the URLpattern "`^$`", which matches an empty string. For example:

```
from mysite.views import hello, my_homepage_view

urlpatterns = [
    url(r'^$', my_homepage_view),
    # ...
```

How Django Processes a Request

Before continuing to our second view function, let's pause to learn a little more about how Django works. Specifically, when you view your "Hello world" message by visiting `http://127.0.0.1:8000/hello/` in your Web browser, what does Django do behind the scenes? It all starts with the *settings file*.

When you run `python manage.py runserver`, the script looks for a file called `settings.py` in the inner `mysite` directory. This file contains all sorts of configuration for this particular Django project, all in uppercase: `TEMPLATE_DIRS`, `DATABASES`, etc. The most important setting is called `ROOT_URLCONF`. `ROOT_URLCONF` tells Django which Python module should be used as the URLconf for this Web site.

Remember when `django-admin startproject` created the files `settings.py` and `urls.py`? The auto-generated `settings.py` contains a `ROOT_URLCONF` setting that points to the auto-generated `urls.py`. Open the `settings.py` file and see for yourself; it should look like this:

```
ROOT_URLCONF = 'mysite.urls'
```

This corresponds to the file `mysite/urls.py`. When a request comes in for a particular URL - say, a request for `/hello/` - Django loads the URLconf pointed to by the `ROOT_URLCONF` setting. Then it checks each of the URLpatterns in that URLconf, in order, comparing the requested URL with the patterns one at a time, until it finds one that matches.

When it finds one that matches, it calls the view function associated with that pattern, passing it an `HttpRequest` object as the first parameter. (We'll cover the specifics of `HttpRequest` later.) As we saw in our first view example, a view function must return an `HttpResponse`.

Once it does this, Django does the rest, converting the Python object to a proper Web response with the appropriate HTTP headers and body (i.e., the content of the Web page). In summary:

1. A request comes in to `/hello/`.
2. Django determines the root URLconf by looking at the `ROOT_URLCONF` setting.
3. Django looks at all of the URLpatterns in the URLconf for the first one that matches `/hello/`.
4. If it finds a match, it calls the associated view function.
5. The view function returns an `HttpResponse`.
6. Django converts the `HttpResponse` to the proper HTTP response, which results in a Web page.

You now know the basics of how to make Django-powered pages. It's quite simple, really - just write view functions and map them to URLs via URLconfs.

Your Second View: Dynamic Content

Our "Hello world" view was instructive in demonstrating the basics of how Django works, but it wasn't an example of a *dynamic* Web page, because the content of the page is always the same. Every time you view `/hello/`, you'll see the same thing; it might as well be a static HTML file.

For our second view, let's create something more dynamic - a Web page that displays the current date and time. This is a nice, simple next step, because it doesn't involve a database or any user input - just the output of your server's

internal clock. It's only marginally more exciting than "Hello world," but it'll demonstrate a few new concepts. This view needs to do two things: calculate the current date and time, and return an `HttpResponse` containing that value. If you have experience with Python, you know that Python includes a `datetime` module for calculating dates. Here's how to use it:

```
>>> import datetime
>>> now = datetime.datetime.now()
>>> now
datetime.datetime(2015, 7, 15, 18, 12, 39, 2731)
>>> print (now)
2015-07-15 18:12:39.002731
```

That's simple enough, and it has nothing to do with Django. It's just Python code. (I want to emphasize that you should be aware of what code is "just Python" vs. code that is Django-specific. As you learn Django, I want you to be able to apply your knowledge to other Python projects that don't necessarily use Django.) To make a Django view that displays the current date and time, we just need to hook this `datetime.datetime.now()` statement into a view and return an `HttpResponse`. Here's what the updated `views.py` looks like:

```
from django.http import HttpResponse
import datetime

def hello(request):
    return HttpResponse("Hello world")

def current_datetime(request):
    now = datetime.datetime.now()
    html = "It is now %s." % now
    return HttpResponse(html)
```

Let's step through the changes we've made to `views.py` to accommodate the `current_datetime` view.

- We've added an `import datetime` to the top of the module, so we can calculate dates.
- The new `current_datetime` function calculates the current date and time, as a `datetime.datetime` object, and stores that as the local variable `now`.

- The second line of code within the view constructs an HTML response using Python's "format-string" capability. The `%s` within the string is a placeholder, and the percent sign after the string means "Replace the `%s` in the preceding string with the value of the variable `now`." The `now` variable is technically a `datetime.datetime` object, not a string, but the `%s` format character converts it to its string representation, which is something like `"2015-07-15 18:12:39.002731"`. This will result in an HTML string such as `"It is now 2015-07-15 18:12:39.002731."`.
- Finally, the view returns an `HttpResponse` object that contains the generated response - just as we did in `hello`.

After adding that to `views.py`, add the URLpattern to `urls.py` to tell Django which URL should handle this view. Something like `/time/` would make sense:

```
from django.conf.urls import include, url
from django.contrib import admin
from mysite.views import hello, current_datetime

    urlpatterns = [
        url(r'^admin/', include(admin.site.urls)),
        url(r'^hello/$', hello),
        url(r'^time/$', current_datetime),
    ]
```

We've made two changes here. First, we imported the `current_datetime` function at the top. Second, and more importantly, we added a URLpattern mapping the URL `/time/` to that new view. Getting the hang of this? With the view written and URLconf updated, fire up the `runserver` and visit `http://127.0.0.1:8000/time/` in your browser. You should see the current date and time. If you don't see your local time, it is likely because the default time zone in your `settings.py` is set to `'UTC'`.

URLconfs and Loose Coupling

Now's a good time to highlight a key philosophy behind URLconfs and behind Django in general: the principle of *loose coupling*. Simply put, loose coupling is a software-development approach that values the importance of making pieces

interchangeable. If two pieces of code are loosely coupled, then changes made to one of the pieces will have little or no effect on the other.

Django's URLconfs are a good example of this principle in practice. In a Django web application, the URL definitions and the view functions they call are loosely coupled; that is, the decision of what the URL should be for a given function, and the implementation of the function itself, reside in two separate places.

For example, consider our `current_datetime` view. If we wanted to change the URL for the application - say, to move it from `/time/` to `/current-time/`- we could make a quick change to the URLconf, without having to worry about the view itself. Similarly, if we wanted to change the view function - altering its logic somehow - we could do that without affecting the URL to which the function is bound. Furthermore, if we wanted to expose the current-date functionality at several URLs, we could easily take care of that by editing the URLconf, without having to touch the view code.

In this example, our `current_datetime` is available at two URLs. It's a contrived example, but this technique can come in handy:

```
urlpatterns = [
      url(r'^admin/', include(admin.site.urls)),
      url(r'^hello/$', hello),
      url(r'^time/$', current_datetime),
      url(r'^another-time-page/$', current_datetime),
]
```

URLconfs and views are loose coupling in action. I'll continue to point out examples of this important philosophy throughout the book.

Your Third View: Dynamic URLs

In our `current_datetime` view, the contents of the page - the current date/-time - were dynamic, but the URL (`/time/`) was static. In most dynamic Web applications though, a URL contains parameters that influence the output of the page. For example, an online bookstore might give each book its own URL, like `/books/243/` and `/books/81196/`. Let's create a third view that displays the current date and time offset by a certain number of hours. The goal is to craft a site in such a way that the page `/time/plus/1/` displays the date/time one hour

into the future, the page `/time/plus/2/` displays the date/time two hours into the future, the page `/time/plus/3/` displays the date/time three hours into the future, and so on. A novice might think to code a separate view function for each hour offset, which might result in a URLconf like this:

```
urlpatterns = [
    url(r'^time/$', current_datetime),
    url(r'^time/plus/1/$', one_hour_ahead),
    url(r'^time/plus/2/$', two_hours_ahead),
    url(r'^time/plus/3/$', three_hours_ahead),
]
```

Clearly, this line of thought is flawed.

Not only would this result in redundant view functions, but also the application is fundamentally limited to supporting only the predefined hour ranges - one, two or three hours.

If we decided to create a page that displayed the time *four* hours into the future, we'd have to create a separate view and URLconf line for that, furthering the duplication.

How, then do we design our application to handle arbitrary hour offsets? The key is to use *wildcard URLpatterns*. As I mentioned previously, a URLpattern is a regular expression; hence, we can use the regular expression pattern `\d+` to match one or more digits:

```
urlpatterns = [
    # ...
    url(r'^time/plus/\d+/$', hours_ahead),
    # ...
]
```

(I'm using the `# ...` to imply there might be other URLpatterns that have been trimmed from this example.) This new URLpattern will match any URL such as `/time/plus/2/`, `/time/plus/25/`, or even `/time/plus/100000000000/`. Come to think of it, let's limit it so that the maximum allowed offset is something reasonable.

In this example, we will set a maximum 99 hours by only allowing either one- or two-digit numbers - and in regular expression syntax, that translates into `\d{1,2}`:

```
url(r'^time/plus/\d{1,2}/$', hours_ahead),
```

Now that we've designated a wildcard for the URL, we need a way of passing that wildcard data to the view function, so that we can use a single view function for any arbitrary hour offset. We do this by placing parentheses around the data in the URLpattern that we want to save. In the case of our example, we want to save whatever number was entered in the URL, so let's put parentheses around the `\d{1,2}`, like this:

```
url(r'^time/plus/(\d{1,2})/$', hours_ahead),
```

If you're familiar with regular expressions, you'll be right at home here; we're using parentheses to *capture* data from the matched text. The final URLconf, including our previous two views, looks like this:

```
from django.conf.urls import include, url
from django.contrib import admin
from mysite.views import hello, current_datetime, hours_ahead
urlpatterns = [
    url(r'^admin/', include(admin.site.urls)),
    url(r'^hello/$', hello),
    url(r'^time/$', current_datetime),
    url(r'^time/plus/(\d{1,2})/$', hours_ahead),
]
```

If you're experienced in another Web development platform, you may be thinking, "Hey, let's use a query string parameter!" – something like `/time/plus?hours=3`, in which the hours would be designated by the hours parameter in the URL's query string (the part after the '?').

You *can* do that with Django (and I'll tell you how in Chapter 7), but one of Django's core philosophies is that URLs should be beautiful. The URL /time/plus/3/ is far cleaner, simpler, more readable, easier to recite to somebody aloud and . . . just plain prettier than its query string counterpart. Pretty URLs are a characteristic of a quality Web application. Django's URLconf system encourages pretty URLs by making it easier to use pretty URLs than *not* to.

With that taken care of, let's write the `hours_ahead` view. `hours_ahead` is very similar to the `current_datetime` view we wrote earlier, with a key difference: it takes an extra argument, the number of hours of offset. Here's the view code:

```
from django.http import Http404, HttpResponse
import datetime

def hours_ahead(request, offset):
    try:
        offset = int(offset)
    except ValueError:
        raise Http404()
    dt = datetime.datetime.now() + datetime.timedelta(hours=offset)
    html = "In %s hour(s), it will be  %s.
            " % (offset, dt)
    return HttpResponse(html)
```

Let's take a closer look at this code.

The view function, `hours_ahead`, takes two parameters: `request` and `offset`:

- `request` is an `HttpRequest` object, just as in `hello` and `current_datetime`. I'll say it again: each view always takes an `HttpRequest` object as its first parameter.
- `offset` is the string captured by the parentheses in the URLpattern. For example, if the requested URL were `/time/plus/3/`, then offset would be the string '3'. If the requested URL were `/time/plus/21/`, then offset would be the string '21'. Note that captured values will always be Unicode objects, not integers, even if the string is composed of only digits, such as '21'.

I decided to call the variable `offset`, but you can call it whatever you'd like, as long as it's a valid Python identifier. The variable name doesn't matter; all that matters is that it's the second argument to the function, after `request`. (It's also possible to use keyword, rather than positional, arguments in an URLconf. I cover that in Chapter 7.)

The first thing we do within the function is call `int()` on `offset`. This converts the Unicode string value to an integer.

Note that Python will raise a `ValueError` exception if you call `int()` on a value that cannot be converted to an integer, such as the string "foo". In this example, if we encounter the `ValueError`, we raise the exception `django.http.Http404`, which, as you can imagine, results in a 404 "Page not found" error.

Astute readers will wonder: how could we ever reach the `ValueError` case, anyway, given that the regular expression in our URLpattern - `(\d{1,2})` - captures only digits, and therefore `offset` will only ever be a string composed of digits? The answer is, we won't, because the URLpattern provides a modest but useful level of input validation, *but* we still check for the `ValueError` in case this view function ever gets called in some other way. It's good practice to implement view functions such that they don't make any assumptions about their parameters. Loose coupling, remember?

In the next line of the function, we calculate the current date/time and add the appropriate number of hours. We've already seen `datetime.datetime.now()` from the `current_datetime` view; the new concept here is that you can perform date/time arithmetic by creating a `datetime.timedelta` object and adding to a `datetime.datetime` object. Our result is stored in the variable `dt`.

This line also shows why we called `int()` on `offset` - the `datetime.timedelta` function requires the `hours` parameter to be an integer.

Next, we construct the HTML output of this view function, just as we did in `current_datetime`. A small difference in this line from the previous line is that it uses Python's format-string capability with *two* values, not just one. Hence, there are two `%s` symbols in the string and a tuple of values to insert: `(offset, dt)`.

Finally, we return an `HttpResponse` of the HTML.

With that view function and URLconf written, start the Django development server (if it's not already running), and visit `http://127.0.0.1:8000/time/plus/3/` to verify it works.

Then try `http://127.0.0.1:8000/time/plus/5/`.

Then `http://127.0.0.1:8000/time/plus/24/`.

Finally, visit `http://127.0.0.1:8000/time/plus/100/` to verify that the pattern in your URLconf only accepts one- or two-digit numbers; Django should display a "Page not found" error in this case, just as we saw in the section "A Quick Note About 404 Errors" earlier.

The URL `http://127.0.0.1:8000/time/plus/` (with *no* hour designation) should also throw a 404.

Django's Pretty Error Pages

Take a moment to admire the fine Web application we've made so far . . . now let's break it! Let's deliberately introduce a Python error into our `views.py` file by commenting out the `offset = int(offset)` lines in the `hours_ahead` view:

```
def hours_ahead(request, offset):
    # try:
    #     offset = int(offset)
    # except ValueError:
    #     raise Http404()
    dt = datetime.datetime.now() + datetime.timedelta(hours=offset)
    html = "In %s hour(s), it will be %s.
            " % (offset, dt)
    return HttpResponse(html)
```

Load up the development server and navigate to `/time/plus/3/`. You'll see an error page with a significant amount of information, including a `TypeError` message displayed at the very top: `"unsupported type for timedelta hours component: str"` (Figure 2-3).

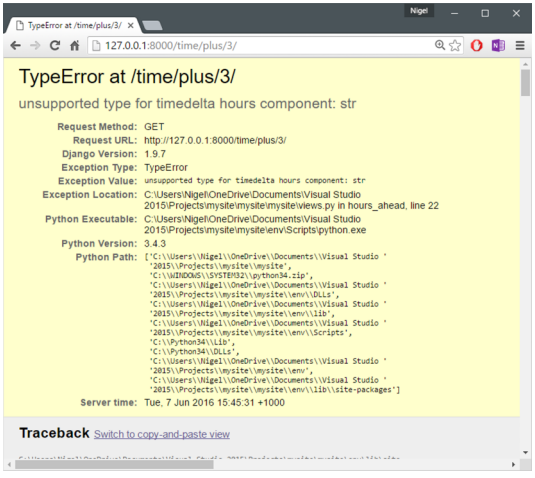

Figure 2-3: *Django's error page*

What happened? Well, the `datetime.timedelta` function expects the `hours` parameter to be an integer, and we commented out the bit of code that converted `offset` to an integer. That caused `datetime.timedelta` to raise the

`TypeError`. It's the typical kind of small bug that every programmer runs into at some point.

The point of this example was to demonstrate Django's error pages. Take some time to explore the error page and get to know the various bits of information it gives you. Here are some things to notice:

- At the top of the page, you get the key information about the exception: the type of exception, any parameters to the exception (the `"unsupported type"` message in this case), the file in which the exception was raised, and the offending line number.
- Under the key exception information, the page displays the full Python traceback for this exception. This is similar to the standard traceback you get in Python's command-line interpreter, except it's more interactive. For each level ("frame") in the stack, Django displays the name of the file, the function/method name, the line number, and the source code of that line.
- Click the line of source code (in dark gray), and you'll see several lines from before and after the erroneous line, to give you context. Click "Local vars" under any frame in the stack to view a table of all local variables and their values, in that frame, at the exact point in the code at which the exception was raised. This debugging information can be a great help.
- Note the "Switch to copy-and-paste view" text under the "Traceback" header. Click those words, and the traceback will switch to an alternate version that can be easily copied and pasted. Use this when you want to share your exception traceback with others to get technical support - such as the kind folks in the Django IRC chat room or on the Django users mailing list.
- Underneath, the "Share this traceback on a public Web site" button will do this work for you in just one click. Click it to post the traceback to dpaste[9], where you'll get a distinct URL that you can share with other people.
- Next, the "Request information" section includes a wealth of information about the incoming Web request that spawned the error: `GET` and `POST` information, cookie values, and meta information, such as CGI headers. Appendix F has a complete reference of all the information a request object contains.
- Below the "Request information" section, the "Settings" section lists all of the settings for this particular Django installation. All the available settings

are covered in detail in Appendix D.

The Django error page is capable of displaying more information in certain special cases, such as the case of template syntax errors. We'll get to those later, when we discuss the Django template system. For now, uncomment the `offset = int(offset)` lines to get the view function working properly again.

The Django error page is also really useful if you are the type of programmer who likes to debug with the help of carefully placed `print` statements.

At any point in your view, temporarily insert an `assert False` to trigger the error page. Then, you can view the local variables and state of the program. Here's an example, using the `hours_ahead` view:

```
def hours_ahead(request, offset):
    try:
        offset = int(offset)
    except ValueError:
        raise Http404()
    dt = datetime.datetime.now() + datetime.timedelta(hours=offset)
    assert False
    html = "In %s hour(s), it will be %s.
            " % (offset, dt)
    return HttpResponse(html)
```

Finally, it's obvious that much of this information is sensitive - it exposes the innards of your Python code and Django configuration - and it would be foolish to show this information on the public Internet. A malicious person could use it to attempt to reverse-engineer your Web application and do nasty things. For that reason, the Django error page is only displayed when your Django project is in debug mode. I'll explain how to deactivate debug mode in Chapter 13. For now, just know that every Django project is in debug mode automatically when you start it. (Sound familiar? The "Page not found" errors, described earlier in this chapter, work the same way.)

What's Next?

So far, we've been writing our view functions with HTML hard-coded directly in the Python code. I've done that to keep things simple while I demonstrated

core concepts, but in the real world, this is nearly always a bad idea. Django ships with a simple yet powerful template engine that allows you to separate the design of the page from the underlying code. We'll dive into Django's template engine in the next chapter.

Chapter 3: Templates

In the previous chapter, you may have noticed something peculiar in how we returned the text in our example views. Namely, the HTML was hard-coded directly in our Python code, like this:

```
def current_datetime(request):
    now = datetime.datetime.now()
    html = "It is now %s." % now
    return HttpResponse(html)
```

Although this technique was convenient for the purpose of explaining how views work, it's not a good idea to hard-code HTML directly into your views. Here's why:

- Any change to the design of the page requires a change to the Python code. The design of a site tends to change far more frequently than the underlying Python code, so it would be convenient if the design could change without needing to modify the Python code.
- This is only a very simple example. A common webpage template has hundreds of lines of HTML and scripts. Untangling and troubleshooting program code from this mess is a nightmare (*cough-PHP-cough*).
- Writing Python code and designing HTML are two different disciplines, and most professional Web development environments split these responsibilities between separate people (or even separate departments). Designers and HTML/CSS coders shouldn't be required to edit Python code to get their job done.
- It's most efficient if programmers can work on Python code and designers can work on templates at the same time, rather than one person waiting for the other to finish editing a single file that contains both Python and HTML.

For these reasons, it's much cleaner and more maintainable to separate the design of the page from the Python code itself. We can do this with Django's *template system*, which we discuss in this chapter.

Template System Basics

A Django template is a string of text that is intended to separate the presentation of a document from its data. A template defines placeholders and various bits of basic logic (template tags) that regulate how the document should be displayed. Usually, templates are used for producing HTML, but Django templates are equally capable of generating any text-based format.

Philosophy behind Django's templates

If you have a background in programming, or if you're used to languages which mix programming code directly into HTML, you'll want to bear in mind that the Django template system is not simply Python embedded into HTML.

This is by design: the template system is meant to express presentation, not program logic.

Let's start with a simple example template. This Django template describes an HTML page that thanks a person for placing an order with a company. Think of it as a form letter:

```
<html>
<head>
    <title>Ordering notice</title>
</head>
<body>

    <h1>Ordering notice</h1>

    <p>Dear {{ person_name }},</p>

    <p>Thanks for placing an order from {{ company }}. It's scheduled to ship on {{ s\
hip_date|date:"F j, Y" }}.</p>
    <p>Here are the items you've ordered:</p>
    <ul>
        {% for item in item_list %}
        <li>{{ item }}</li>{% endfor %}
    </ul>
```

```
    {% if ordered_warranty %}
    <p>Your warranty information will be included in the packaging.</p>
    {% else %}
    <p>
        You didn't order a warranty, so you're on your own when
        the products inevitably stop working.
    </p>
    {% endif %}

    <p>Sincerely,<br />{{ company }}</p>

</body>
</html>
```

This template is basic HTML with some variables and template tags thrown in. Let's step through it:

- Any text surrounded by a pair of braces (e.g., `{{ person_name }}`) is a *variable*. This means "insert the value of the variable with the given name." How do we specify the values of the variables? We'll get to that in a moment.
- Any text that's surrounded by curly braces and percent signs (e.g., `{% if ordered_warranty %}`) is a *template tag*. The definition of a tag is quite broad: a tag just tells the template system to "do something".
- This example template contains a `for` tag (`{% for item in item_list %}`) and an `if` tag (`{% if ordered_warranty %}`). A `for` tag works very much like a `for` statement in Python, letting you loop over each item in a sequence.

  An `if` tag, as you may expect, acts as a logical "if" statement. In this particular case, the tag checks whether the value of the `ordered_warranty` variable evaluates to `True`. If it does, the template system will display everything between the `{% if ordered_warranty %}` and `{% else %}`. If not, the template system will display everything between `{% else %}` and`{% endif %}`. Note that the `{% else %}` is optional.
- Finally, the second paragraph of this template contains an example of a *filter*, which is the most convenient way to alter the formatting of a variable. In this example, `{{ ship_date|date:"F j, Y" }}`, we're passing the `ship_date` variable to the `date` filter, giving the `date` filter the argument `"F j, Y"`. The `date` filter formats dates in a given format, as specified

by that argument. Filters are attached using a pipe character (|), as a reference to Unix pipes. Each Django template has access to several built-in tags and filters, many of which are discussed in the sections that follow. Appendix E contains the full list of tags and filters, and it's a good idea to familiarize yourself with that list so you know what's possible. It's also possible to create your own filters and tags; we'll cover that in Chapter 8.

Using the Template System

A Django project can be configured with one or several template engines (or even zero if you don't use templates). Django ships with a built-in backend for its own template system - the *Django Template Language (DTL)*. Django 1.8 also includes support for the popular alternative Jinja2[9]. If you don't have a pressing reason to choose another backend, you should use the DTL - especially if you're writing a pluggable application and you intend to distribute templates. Django's `contrib` apps that include templates, like `django.contrib.admin`, use the DTL. All of the examples in this chapter will use the DTL. For more advanced template topics, including configuring third-party template engines, see Chapter 8. Before we go about implementing Django templates in your view, lets first dig inside the DTL a little so you can see how it works. Here is the most basic way you can use Django's template system in Python code:

1. Create a `Template` object by providing the raw template code as a string.
2. Call the `render()` method of the `Template` object with a given set of variables (the context). This returns a fully rendered template as a string, with all of the variables and template tags evaluated according to the context. In code, here's what that looks like:

[9] http://jinja.pocoo.org/

```
>>> from django import template
>>> t = template.Template('My name is {{ name }}.')
>>> c = template.Context({'name': 'Nige'})
>>> print (t.render(c))
My name is Nige.
>>> c = template.Context({'name': 'Barry'})
>>> print (t.render(c))
My name is Barry.
```

The following sections describe each step in much more detail.

Creating Template Objects

The easiest way to create a `Template` object is to instantiate it directly. The `Template` class lives in the `django.template` module, and the constructor takes one argument, the raw template code. Let's dip into the Python interactive interpreter to see how this works in code. From the `mysite` project directory you created in Chapter 1, type `python manage.py shell` to start the interactive interpreter.

Let's go through some template system basics:

```
>>> from django.template import Template
>>> t = Template('My name is {{ name }}.')
>>> print (t)
```

If you're following along interactively, you'll see something like this:

```
<django.template.base.Template object at 0x030396B0>
```

That `0x030396B0` will be different every time, and it isn't relevant; it's a Python thing (the Python "identity" of the `Template` object, if you must know).

When you create a `Template` object, the template system compiles the raw template code into an internal, optimized form, ready for rendering. But if your template code includes any syntax errors, the call to `Template()` will cause a `TemplateSyntaxError` exception:

```
>>> from django.template import Template
>>> t = Template('{% notatag %}')
Traceback (most recent call last):
File "", line 1, in ?
...
django.template.base.TemplateSyntaxError: Invalid block tag: 'notatag'
```

The term "block tag" here refers to `{% notatag %}`. "Block tag" and "template tag" are synonymous. The system raises a `TemplateSyntaxError` exception for any of the following cases:

- Invalid tags
- Invalid arguments to valid tags
- Invalid filters
- Invalid arguments to valid filters
- Invalid template syntax
- Unclosed tags (for tags that require closing tags)

Rendering a Template

Once you have a `Template` object, you can pass it data by giving it a *context*. A context is simply a set of template variable names and their associated values. A template uses this to populate its variables and evaluate its tags. A context is represented in Django by the `Context` class, which lives in the `django.template` module. Its constructor takes one optional argument: a dictionary mapping variable names to variable values. Call the `Template` object's `render()` method with the context to "fill" the template:

```
>>> from django.template import Context, Template
>>> t = Template('My name is {{ name }}.')
>>> c = Context({'name': 'Stephane'})
>>> t.render(c)
'My name is Stephane.'
```

A special Python prompt

If you've used Python before, you may be wondering why we're running `python manage.py shell` instead of just `python` (or `python3`). Both commands will start the interactive interpreter, but the `manage.py shell` command has one key difference: before starting the interpreter, it tells Django which settings file to use.

Many parts of Django, including the template system, rely on your settings, and you won't be able to use them unless the framework knows which settings to use.

If you're curious, here's how it works behind the scenes. Django looks for an environment variable called `DJANGO_SETTINGS_MODULE`, which should be set to the import path of your `settings.py`. For example, `DJANGO_SETTINGS_MODULE` might be set to `'mysite.settings'`, assuming `mysite` is on your Python path.

When you run `python manage.py shell`, the command takes care of setting `DJANGO_SETTINGS_MODULE` for you. You will need to use `python manage.py shell` in these examples, or Django will throw an exception.

Dictionaries and Contexts

A Python dictionary is a mapping between known keys and variable values. A `Context` is similar to a dictionary, but a `Context` provides additional functionality, as covered in Chapter 8.

Variable names must begin with a letter (A-Z or a-z) and may contain more letters, digits, underscores, and dots. (Dots are a special case we'll get to in a moment.) Variable names are case sensitive. Here's an example of template compilation and rendering, using a template similar to the example in the beginning of this chapter:

```
>>> from django.template import Template, Context
>>> raw_template = """<p>Dear {{ person_name }},</p>
...
... <p>
    Thanks for placing an order from {{ company }}. It's scheduled to
    ... ship on {{
ship_date|date:"F j, Y"
    }}.
</p>
...
... {% if ordered_warranty %}
... <p>Your warranty information will be included in the packaging.</p>
... {% else %}
... <p>
    You didn't order a warranty, so you're on your own when
    ... the products inevitably stop working.
</p>
... {% endif %}
...
... <p>Sincerely,<br />{{ company }}</p>"""
>>> t = Template(raw_template)
>>> import datetime
>>> c = Context({'person_name': 'John Smith',
...     'company': 'Outdoor Equipment',
...     'ship_date': datetime.date(2015, 7, 2),
...     'ordered_warranty': False})
>>> t.render(c)
"<p>Dear John Smith,</p>\n\n<p>Thanks for placing an order from Outdoor Equipment. It\
's scheduled to\nship on July 2,2015.</p>\n\n\n<p>You didn't order a warranty, so you\
're on your own when\nthe products inevitably stop working.</p>\n\n\n<p>Sincerely,<br\
 />Outdoor Equipment</p>"
```

- First, we import the classes `Template` and `Context`, which both live in the module `django.template`.
- We save the raw text of our template into the variable `raw_template`. Note that we use triple quote marks to designate the string, because it wraps over multiple lines; in contrast, strings within single quote marks cannot be wrapped over multiple lines.
- Next, we create a template object, `t`, by passing `raw_template` to the `Template` class constructor.
- We import the `datetime` module from Python's standard library, because

we'll need it in the following statement.

- Then, we create a `Context` object, `c`. The `Context` constructor takes a Python dictionary, which maps variable names to values. Here, for example, we specify that the `person_name` is "`John Smith`", `company` is "`Outdoor Equipment`", and so forth.
- Finally, we call the `render()` method on our template object, passing it the context. This returns the rendered template - i.e., it replaces template variables with the actual values of the variables, and it executes any template tags. Note that the "You didn't order a warranty" paragraph was displayed because the `ordered_warranty` variable evaluated to `False`. Also note the date, `July 2, 2015`, which is displayed according to the format string "`F j, Y`". (We'll explain format strings for the `date` filter in a little while.)

If you're new to Python, you may wonder why this output includes newline characters ("`\n`") rather than displaying the line breaks. That's happening because of a subtlety in the Python interactive interpreter: the call to `t.render(c)` returns a string, and by default the interactive interpreter displays the representation of the string, rather than the printed value of the string. If you want to see the string with line breaks displayed as true line breaks rather than "`\n`" characters, use the print function: `print (t.render(c))`.

Those are the fundamentals of using the Django template system: just write a template string, create a `Template` object, create a `Context`, and call the `render()` method.

Multiple Contexts, Same Template

Once you have a `Template` object, you can render multiple contexts through it. For example:

```
>>> from django.template import Template, Context
>>> t = Template('Hello, {{ name }}')
>>> print (t.render(Context({'name': 'John'})))
Hello, John
>>> print (t.render(Context({'name': 'Julie'})))
Hello, Julie
>>> print (t.render(Context({'name': 'Pat'})))
Hello, Pat
```

Whenever you're using the same template source to render multiple contexts like this, it's more efficient to create the `Template` object *once*, and then call `render()` on it multiple times:

```
# Bad
for name in ('John', 'Julie', 'Pat'):
t = Template('Hello, {{ name }}')
print (t.render(Context({'name': name})))

# Good
t = Template('Hello, {{ name }}')
for name in ('John', 'Julie', 'Pat'):
print (t.render(Context({'name': name})))
```

Django's template parsing is quite fast. Behind the scenes, most of the parsing happens via a call to a single regular expression. This is in stark contrast to XML-based template engines, which incur the overhead of an XML parser and tend to be orders of magnitude slower than Django's template rendering engine.

Context Variable Lookup

In the examples so far, we've passed simple values in the contexts - mostly strings, plus a `datetime.date` example. However, the template system elegantly handles more complex data structures, such as lists, dictionaries, and custom objects. The key to traversing complex data structures in Django templates is the dot character (".").

Use a dot to access dictionary keys, attributes, methods, or indices of an object. This is best illustrated with a few examples. For instance, suppose you're passing a Python dictionary to a template. To access the values of that dictionary by dictionary key, use a dot:

```
>>> from django.template import Template, Context
>>> person = {'name': 'Sally', 'age': '43'}
>>> t = Template('{{ person.name }} is {{
person.age
}} years old.')
>>> c = Context({'person': person})
>>> t.render(c)
'Sally is 43 years old.'
```

Similarly, dots also allow access to object attributes. For example, a Python `datetime.date` object has `year`, `month`, and `day` attributes, and you can use a dot to access those attributes in a Django template:

```
>>> from django.template import Template, Context
>>> import datetime
>>> d = datetime.date(1993, 5, 2)
>>> d.year
1993
>>> d.month
5
>>> d.day
2
>>> t = Template('The month is {{ date.month }}
and the year is {{ date.year }}.')
>>> c = Context({'date': d})
>>> t.render(c)
'The month is 5 and the year is 1993.'
```

This example uses a custom class, demonstrating that variable dots also allow attribute access on arbitrary objects:

```
>>> from django.template import Template, Context
>>> class Person(object):
...     def __init__(self, first_name, last_name):
...         self.first_name, self.last_name =
first_name, last_name
>>> t = Template('Hello, {{ person.first_name }} {{ person.last_name }}.')
>>> c = Context({'person': Person('John', 'Smith')})
>>> t.render(c)
'Hello, John Smith.'
```

Dots can also refer to *methods* of objects. For example, each Python string has the methods `upper()` and `isdigit()`, and you can call those in Django templates using the same dot syntax:

```
>>> from django.template import Template, Context
>>> t = Template('{{ var }} -- {{ var.upper }} -- {{ var.isdigit }}')
>>> t.render(Context({'var': 'hello'}))
'hello -- HELLO -- False'
>>> t.render(Context({'var': '123'}))
'123 -- 123 -- True'
```

Note that you do *not* include parentheses in the method calls. Also, it's not possible to pass arguments to the methods; you can only call methods that have no required arguments. (I explain this philosophy later in this chapter.) Finally, dots are also used to access list indices, for example:

```
>>> from django.template import Template, Context
>>> t = Template('Item 2 is {{ items.2 }}.')
>>> c = Context({'items': ['apples', 'bananas',
'carrots']})
>>> t.render(c)
'Item 2 is carrots.'
```

Negative list indices are not allowed. For example, the template variable `{{ items.-1 }}` would cause a `TemplateSyntaxError`.

Python Lists

A reminder: Python lists have 0-based indices. The first item is at index 0, the second is at index 1, and so on.

Dot lookups can be summarized like this: when the template system encounters a dot in a variable name, it tries the following lookups, in this order:

- Dictionary lookup (e.g., `foo["bar"]`)
- Attribute lookup (e.g., `foo.bar`)
- Method call (e.g., `foo.bar()`)
- List-index lookup (e.g., `foo[2]`)

The system uses the first lookup type that works. It's short-circuit logic. Dot lookups can be nested multiple levels deep. For instance, the following example uses `{{ person.name.upper }}`, which translates into a dictionary lookup (`person['name']`) and then a method call (`upper()`):

```
>>> from django.template import Template, Context
>>> person = {'name': 'Sally', 'age': '43'}
>>> t = Template('{{ person.name.upper }} is {{
person.age
}} years old.')
>>> c = Context({'person': person})
>>> t.render(c)
'SALLY is 43 years old.'
```

Method Call Behavior

Method calls are slightly more complex than the other lookup types. Here are some things to keep in mind:

- If, during the method lookup, a method raises an exception, the exception will be propagated, unless the exception has an attribute `silent_variable_failure` whose value is `True`. If the exception *does* have a `silent_variable_failure` attribute, the variable will render as the value of the engine's `string_if_invalid` configuration option (an empty string, by default). For example:

```
>>>
t = Template("My name is {{ person.first_name }}.")
>>> class PersonClass3:
...     def first_name(self):
...         raise AssertionError("foo")
>>> p = PersonClass3()
>>> t.render(Context({"person": p}))
Traceback (most recent call last):
...
AssertionError: foo

>>> class SilentAssertionError(Exception):
...     silent_variable_failure = True
>>> class PersonClass4:
...     def first_name(self):
...         raise SilentAssertionError
>>> p = PersonClass4()
>>> t.render(Context({"person": p}))
'My name is .'
```

- A method call will only work if the method has no required arguments. Otherwise, the system will move to the next lookup type (list-index lookup).
- By design, Django intentionally limits the amount of logic processing available in the template, so it's not possible to pass arguments to method calls accessed from within templates. Data should be calculated in views and then passed to templates for display.
- Obviously, some methods have side effects, and it would be foolish at best, and possibly even a security hole, to allow the template system to access them.
- Say, for instance, you have a `BankAccount` object that has a `delete()` method. If a template includes something like `{{ account.delete }}`, where `account` is a `BankAccount` object, the object would be deleted when the template is rendered! To prevent this, set the function attribute `alters_data` on the method:

  ```
  def delete(self):
  # Delete the account
  delete.alters_data = True
  ```

  The template system won't execute any method marked in this way.

Continuing the above example, if a template includes `{{ account.delete }}` and the `delete()` method has the `alters_data=True`, then the `delete()` method will not be executed when the template is rendered, the engine will instead replace the variable with `string_if_invalid`.

NOTE: The dynamically-generated `delete()` and `save()` methods on Django model objects get `alters_data=true` set automatically.

How Invalid Variables Are Handled

Generally, if a variable doesn't exist, the template system inserts the value of the engine's `string_if_invalid` configuration option, which is an empty string by default. For example:

```
>>> from django.template import Template, Context
>>> t = Template('Your name is {{ name }}.')
>>> t.render(Context())
'Your name is .'
>>> t.render(Context({'var': 'hello'}))
'Your name is .'
>>> t.render(Context({'NAME': 'hello'}))
'Your name is .'
>>> t.render(Context({'Name': 'hello'}))
'Your name is .'
```

This behaviour is better than raising an exception because it's intended to be resilient to human error. In this case, all of the lookups failed because variable names have the wrong case or name. In the real world, it's unacceptable for a Web site to become inaccessible due to a small template syntax error.

Basic Template Tags and Filters

As we've mentioned already, the template system ships with built-in tags and filters. The sections that follow provide a rundown of the most common tags and filters.

Tags

if/else

The `{% if %}` tag evaluates a variable, and if that variable is "True" (i.e., it exists, is not empty, and is not a false Boolean value), the system will display everything between `{% if %}` and `{% endif %}`, for example:

```
{% if today_is_weekend %}
<p>Welcome to the weekend!</p>
{% endif %}

An `{% else %}` tag is optional:

{% if today_is_weekend %}
<p>Welcome to the weekend!</p>
{% else %}
<p>Get back to work.</p>
{% endif %}
```

The `if` tag may also take one or several `{% elif %}` clauses as well:

```
{% if athlete_list %}
Number of athletes: {{ athlete_list|length }}
{% elif athlete_in_locker_room_list %}
<p>Athletes should be out of the locker room soon! </p>
{% elif ...
...
{% else %}
<p>No athletes. </p>
{% endif %}
```

The `{% if %}` tag accepts and, or, or not for testing multiple variables, or to negate a given variable. For example:

```
{% if athlete_list and coach_list %}
<p>Both athletes and coaches are available. </p>
{% endif %}

{% if not athlete_list %}
<p>There are no athletes. </p>
{% endif %}

{% if athlete_list or coach_list %}
<p>There are some athletes or some coaches. </p>
{% endif %}

{% if not athlete_list or coach_list %}
<p>There are no athletes or there are some coaches. </p>
{% endif %}

{% if athlete_list and not coach_list %}
<p>There are some athletes and absolutely no coaches. </p>
{% endif %}
```

Use of both `and` and `or` clauses within the same tag is allowed, with `and` having higher precedence than `or` e.g.:

```
{% if athlete_list and coach_list or cheerleader_list %}
```

will be interpreted like:

```
if (athlete_list and coach_list) or cheerleader_list
```

NOTE: Use of actual parentheses in the if tag is invalid syntax.

If you need parentheses to indicate precedence, you should use nested if tags. The use of parentheses for controlling order of operations is not supported. If you find yourself needing parentheses, consider performing logic outside the template and passing the result of that as a dedicated template variable. Or, just use nested `{% if %}` tags, like this:

```
{% if athlete_list %}
    {% if coach_list or cheerleader_list %}
        <p>We have athletes, and either coaches or cheerleaders! </p>
    {% endif %}
{% endif %}
```

Multiple uses of the same logical operator are fine, but you can't combine different operators. For example, this is valid:

```
{% if athlete_list or coach_list or parent_list or teacher_list %}
```

Make sure to close each `{% if %}` with an `{% endif %}`. Otherwise, Django will throw a `TemplateSyntaxError`.

for

The `{% for %}` tag allows you to loop over each item in a sequence. As in Python's `for` statement, the syntax is `for X in Y`, where `Y` is the sequence to loop over and `X` is the name of the variable to use for a particular cycle of the loop. Each time through the loop, the template system will render everything between `{% for %}` and `{% endfor %}`. For example, you could use the following to display a list of athletes given a variable `athlete_list`:

```
<ul>
    {% for athlete in athlete_list %}
    <li>{{ athlete.name }}</li>
    {% endfor %}
</ul>
```

Add `reversed` to the tag to loop over the list in reverse:

```
{% for athlete in athlete_list reversed %}
...
{% endfor %}
```

It's possible to nest `{% for %}` tags:

```
{% for athlete in athlete_list %}
<h1>{{ athlete.name }}</h1>
<ul>
    {% for sport in athlete.sports_played %}
    <li>{{ sport }}</li>
    {% endfor %}
</ul>
{% endfor %}
```

If you need to loop over a list of lists, you can unpack the values in each sub list into individual variables.

For example, if your context contains a list of (x,y) coordinates called `points`, you could use the following to output the list of points:

```
{% for x, y in points %}
<p>There is a point at {{ x }},{{ y }}</p>
{% endfor %}
```

This can also be useful if you need to access the items in a dictionary. For example, if your context contained a dictionary `data`, the following would display the keys and values of the dictionary:

```
{% for key, value in data.items %}
{{ key }}: {{ value }}
{% endfor %}
```

A common pattern is to check the size of the list before looping over it, and outputting some special text if the list is empty:

```
{% if athlete_list %}

{% for athlete in athlete_list %}
<p>{{ athlete.name }}</p>
{% endfor %}

{% else %}
<p>There are no athletes. Only computer programmers.</p>
{% endif %}
```

Because this pattern is so common, the `for` tag supports an optional `{% empty %}` clause that lets you define what to output if the list is empty. This example is equivalent to the previous one:

```
{% for athlete in athlete_list %}
<p>{{ athlete.name }}</p>
{% empty %}
<p>There are no athletes. Only computer programmers.</p>
{% endfor %}
```

There is no support for "breaking out" of a loop before the loop is finished. If you want to accomplish this, change the variable you're looping over so that it includes only the values you want to loop over.

Similarly, there is no support for a "continue" statement that would instruct the loop processor to return immediately to the front of the loop. (See the section "Philosophies and Limitations" later in this chapter for the reasoning behind this design decision.)

Within each `{% for %}` loop, you get access to a template variable called `forloop`. This variable has a few attributes that give you information about the progress of the loop:

- `forloop.counter` is always set to an integer representing the number of times the loop has been entered. This is one-indexed, so the first time through the loop, `forloop.counter` will be set to `1`. Here's an example:

```
{% for item in todo_list %}

<p>{{ forloop.counter }}: {{ item }}</p>
{%
endfor %}
```

- `forloop.counter0` is like `forloop.counter`, except it's zero-indexed. Its value will be set to `0` the first time through the loop.
- `forloop.revcounter` is always set to an integer representing the number of remaining items in the loop. The first time through the loop, `forloop.revcounter` will be set to the total number of items in the sequence you're traversing. The last time through the loop, `forloop.revcounter` will be set to `1`.
- `forloop.revcounter0` is like `forloop.revcounter`, except it's zero-indexed. The first time through the loop, `forloop.revcounter0` will be set to the number of elements in the sequence minus 1. The last time through the loop, it will be set to `0`.
- `forloop.first` is a Boolean value set to `True` if this is the first time through the loop. This is convenient for special-casing:

```
{% for object in objects %}

{% if forloop.first %}<li class="first">{% else %}<li>
    {%
    endif %}
    {{ object }}
</li>
{% endfor %}
```

- `forloop.last` is a Boolean value set to `True` if this is the last time through the loop. A common use for this is to put pipe characters between a list of links:

```
{% for link in links %}
{{ link }}{% if not forloop.last %} | {% endif %}
{% endfor %}
```

The above template code might output something like this:

```
Link1 | Link2 | Link3 | Link4
```

Another common use for this is to put a comma between words in a list:

```
<p>Favorite places:</p>
    {% for p in places %}{{ p }}{% if not forloop.last %}, {% endif %}
    {% endfor %}
```

- `forloop.parentloop` is a reference to the `forloop` object for the *parent* loop, in case of nested loops. Here's an example:

```
{% for country in countries %}
<table>
    {% for city in country.city_list %}
    <tr>
        <td>Country #{{ forloop.parentloop.counter }}</td>
        <td>City #{{ forloop.counter }}</td>
        <td>{{ city }}</td>
    </tr>
    {% endfor %}
</table>
{% endfor %}
```

The `forloop` variable is only available within loops. After the template parser has reached `{% endfor %}`, `forloop` disappears.

Context and the forloop Variable

Inside the `{% for %}` block, the existing variables are moved out of the way to avoid overwriting the `forloop` variable. Django exposes this moved context in `forloop.parentloop`. You generally don't need to worry about this, but if you supply a template variable named `forloop` (though we advise against it), it will be named forloop.parentloop while inside the `{% for %}` block.

ifequal/ifnotequal

The Django template system is not a full-fledged programming language and thus does not allow you to execute arbitrary Python statements. (More on this idea in the section "Philosophies and Limitations").

However, it's quite a common template requirement to compare two values and display something if they're equal - and Django provides an `{% ifequal %}` tag for that purpose.

The `{% ifequal %}` tag compares two values and displays everything between `{% ifequal %}` and `{% endifequal %}` if the values are equal. This example compares the template variables `user` and `currentuser`:

```
{% ifequal user currentuser %}
<h1>Welcome!</h1>
{% endifequal %}
```

The arguments can be hard-coded strings, with either single or double quotes, so the following is valid:

```
{% ifequal section 'sitenews' %}
<h1>Site News</h1>
{% endifequal %}

{% ifequal section "community" %}
<h1>Community</h1>
{% endifequal %}
```

Just like `{% if %}`, the `{% ifequal %}` tag supports an optional `{% else %}`:

```
{% ifequal section 'sitenews' %}
<h1>Site News</h1>
{% else %}
<h1>No News Here</h1>
{% endifequal %}
```

Only template variables, strings, integers, and decimal numbers are allowed as arguments to `{% ifequal %}`. These are valid examples:

```
{% ifequal variable 1 %}
{% ifequal variable 1.23 %}
{% ifequal variable 'foo' %}
{% ifequal variable "foo" %}
```

Any other types of variables, such as Python dictionaries, lists, or Booleans, can't be hard-coded in `{% ifequal %}`. These are invalid examples:

```
{% ifequal variable True %}
{% ifequal variable [1, 2, 3] %}
{% ifequal variable {'key': 'value'} %}
```

If you need to test whether something is true or false, use the `{% if %}` tags instead of `{% ifequal %}`.

An alternative to the `ifequal` tag is to use the `if` tag and the "`==`" operator.

The `{% ifnotequal %}` tag is identical to the `ifequal` tag, except that it tests whether the two arguments are not equal. An alternative to the `ifnotequal` tag is to use the if tag and the "`!=`" operator.

Comments

Just as in HTML or Python, the Django template language allows for comments. To designate a comment, use `{# #}`:

```
{# This is a comment #}
```

The comment will not be output when the template is rendered. Comments using this syntax cannot span multiple lines. This limitation improves template parsing performance.

In the following template, the rendered output will look exactly the same as the template (i.e., the comment tag will not be parsed as a comment):

```
This is a {# this is not
a comment #}
test.
```

If you want to use multi-line comments, use the `{% comment %}` template tag, like this:

```
{% comment %}
This is a
multi-line comment.
{% endcomment %}
```

Comment tags cannot be nested.

Filters

As explained earlier in this chapter, template filters are simple ways of altering the value of variables before they're displayed. Filters use a pipe character, like this:

```
{{ name|lower }}
```

This displays the value of the `{{ name }}` variable after being filtered through the `lower` filter, which converts text to lowercase. Filters can be *chained* - that is, they can be used in tandem such that the output of one filter is applied to the next.

Here's an example that takes the first element in a list and converts it to uppercase:

```
{{ my_list|first|upper }}
```

Some filters take arguments. A filter argument comes after a colon and is always in double quotes. For example:

```
{{ bio|truncatewords:"30" }}
```

This displays the first 30 words of the `bio` variable.

The following are a few of the most important filters. Appendix E covers the rest.

- `addslashes`: Adds a backslash before any backslash, single quote, or double quote. This is useful for escaping strings. For example:`{{ value|addslashes }}`
- `date`: Formats a `date` or `datetime` object according to a format string given in the parameter. For example: `{{ pub_date|date:"F j, Y" }}`

  Format strings are defined in Appendix E.
- `length`: Returns the length of the value. For a list, this returns the number of elements. For a string, this returns the number of characters. If the variable is undefined, `length` returns `0`.

Philosophies and Limitations

Now that you've gotten a feel for the Django Template Language(DTL), it is probably time to explain the basic design philosophy behind the DTL. First and foremost, the **limitations to the DTL are intentional.**

Django was developed in the high volume, ever-changing environment of an online newsroom. The original creators of Django had a very definite set of philosophies in creating the DTL.

These philosophies remain core to Django today. They are:

1. Separate logic from presentation
2. Discourage redundancy
3. Be decoupled from HTML
4. XML is bad
5. Assume designer competence
6. Treat whitespace obviously
7. Don't invent a programming language
8. Ensure safety and security
9. Extensible

1. Separate logic from presentation

A template system is a tool that controls presentation and presentation-related logic - and that's it. The template system shouldn't support functionality that goes beyond this basic goal.

2. Discourage redundancy

The majority of dynamic Web sites use some sort of common site-wide design - a common header, footer, navigation bar, etc. The Django template system should make it easy to store those elements in a single place, eliminating duplicate code. This is the philosophy behind template inheritance.

3. Be decoupled from HTML

The template system shouldn't be designed so that it only outputs HTML. It should be equally good at generating other text-based formats, or just plain text.

4. XML should not be used for template languages

Using an XML engine to parse templates introduces a whole new world of human error in editing templates - and incurs an unacceptable level of overhead in template processing.

5. Assume designer competence

The template system shouldn't be designed so that templates necessarily are displayed nicely in WYSIWYG editors such as Dreamweaver. That is too severe of a limitation and wouldn't allow the syntax to be as nice as it is.

Django expects template authors are comfortable editing HTML directly.

6. Treat whitespace obviously

The template system shouldn't do magic things with whitespace. If a template includes whitespace, the system should treat the whitespace as it treats text - just display it. Any whitespace that's not in a template tag should be displayed.

7. Don't invent a programming language

The template system intentionally doesn't allow the following:

- Assignment to variables

- Advanced logic The goal is not to invent a programming language. The goal is to offer just enough programming-esque functionality, such as branching and looping, that is essential for making presentation-related decisions.

The Django template system recognizes that templates are most often written by *designers*, not *programmers*, and therefore should not assume Python knowledge.

8. Safety and security

The template system, out of the box, should forbid the inclusion of malicious code - such as commands that delete database records. This is another reason the template system doesn't allow arbitrary Python code.

9. Extensibility

The template system should recognize that advanced template authors may want to extend its technology. This is the philosophy behind custom template tags and filters.

Having worked with many different templating systems myself over the years, I whole-heartedly endorse this approach - the DTL and the way it has been designed is one of the major pluses of the Django framework.

When the pressure is on to Get Stuff Done, and you have both designers and programmers trying to communicate and get all the of the last minute tasks done, Django just gets out of the way and lets each team concentrate on what they are good at.

Once you have found this out for yourself through real-life practice, you will find out very quickly why Django really is the "framework for perfectionists with deadlines".

With all this in mind, Django is flexible - it does not require you to use the DTL. More than any other component of Web applications, template syntax is highly subjective, and programmers' opinions vary wildly. The fact that Python alone has dozens, if not hundreds, of open source template-language implementations supports this point. Each was likely created because its developer deemed all existing template languages inadequate.

Because Django is intended to be a full-stack Web framework that provides all the pieces necessary for Web developers to be productive, most times it's more convenient to use the DTL, but it's not a strict requirement in any sense.

Using Templates in Views

You've learned the basics of using the template system; now let's use this knowledge to create a view.

Recall the `current_datetime` view in `mysite.views`, which we started in the previous chapter. Here's what it looks like:

```
from django.http import HttpResponse
import datetime

def current_datetime(request):

now = datetime.datetime.now()
html = "<html><body>It is now %s</body></html>" % now
    return HttpResponse(html)
```

Let's change this view to use Django's template system. At first, you might think to do something like this:

```
from django.template import Template, Context
from django.http import HttpResponse
import datetime

def current_datetime(request):

    now = datetime.datetime.now()
    t = Template("<html><body>It is now {{ current_date }}.</body></html>")
    html = t.render(Context({'current_date': now}))
    return HttpResponse(html)
```

Sure, that uses the template system, but it doesn't solve the problems we pointed out in the introduction of this chapter. Namely, the template is still embedded in the Python code, so true separation of data and presentation isn't achieved. Let's fix that by putting the template in a *separate file*, which this view will load.

You might first consider saving your template somewhere on your filesystem and using Python's built-in file-opening functionality to read the contents of the template. Here's what that might look like, assuming the template was saved as the file `/home/djangouser/templates/mytemplate.html`:

```
from django.template import Template, Context
from django.http import HttpResponse
import datetime

def current_datetime(request):

    now = datetime.datetime.now()
    # Simple way of using templates from the filesystem.
    # This is BAD because it doesn't account for missing files!
    fp = open('/home/djangouser/templates/mytemplate.html')
    t = Template(fp.read())
    fp.close()

    html = t.render(Context({'current_date': now}))
    return HttpResponse(html)
```

This approach, however, is inelegant for these reasons:

- It doesn't handle the case of a missing file. If the file `mytemplate.html` doesn't exist or isn't readable, the `open()` call will raise an `IOError` exception.
- It hard-codes your template location. If you were to use this technique for every view function, you'd be duplicating the template locations. Not to mention it involves a lot of typing!
- It includes a lot of boring boilerplate code. You've got better things to do than to write calls to `open()`, `fp.read()`, and `fp.close()` each time you load a template. To solve these issues, we'll use *template loading* and *template directories*.

Template Loading

Django provides a convenient and powerful API for loading templates from the filesystem, with the goal of removing redundancy both in your template-loading calls and in your templates themselves. In order to use this template-loading API, first you'll need to tell the framework where you store your templates. The place to do this is in your settings file - the `settings.py` file that I mentioned last chapter, when I introduced the `ROOT_URLCONF` setting. If you're following along, open your `settings.py` and find the `TEMPLATES` setting. It's a list of configurations, one for each engine:

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [],
        'APP_DIRS': True,
        'OPTIONS': {
            # ... some options here ...
        },
    },
]
```

- `BACKEND` is a dotted Python path to a template engine class implementing Django's template backend API. The built-in backends are `django.template.backends.django.DjangoTemplates` and `django.template.backends.jinja2.Jinja2`. Since most engines load templates from files, the top-level configuration for each engine contains three common settings:
- `DIRS` defines a list of directories where the engine should look for template source files, in search order.
- `APP_DIRS` tells whether the engine should look for templates inside installed applications. By convention, when `APPS_DIRS` is set to `True`, `DjangoTemplates` looks for a "templates" subdirectory in each of the `INSTALLED_APPS`. This allows the template engine to find application templates even if `DIRS` is empty.
- `OPTIONS` contains backend-specific settings. While uncommon, it's possible to configure several instances of the same backend with different options. In that case you should define a unique `NAME` for each engine.

Template Directories

`DIRS`, by default, is an empty list. To tell Django's template-loading mechanism where to look for templates, pick a directory where you'd like to store your templates and add it to `DIRS`, like so:

```
'DIRS': [
            '/home/html/example.com',
            '/home/html/default',
        ],
```

There are a few things to note:

- Unless you are building a very simple program with no apps, you are better off leaving `DIRS` empty. The default settings file configures `APP_DIRS` to `True`, so you are better off having a "templates" subdirectory in your Django app.
- If you want to have a set of master templates at project root, e.g. `mysite/templates`, you *do* need to set `DIRS`, like so:

  ```
  'DIRS': [os.path.join(BASE_DIR, 'templates')],
  ```

- Your templates directory does not have to be called `'templates'`, by the way - Django doesn't put any restrictions on the names you use - but it makes your project structure much easier to understand if you stick to convention.

  If you don't want to go with the default, or can't for some reason, you can specify any directory you want, as long as the directory and templates within that directory are readable by the user account under which your Web server runs.
- If you're on Windows, include your drive letter and use Unix-style forward slashes rather than backslashes, as follows:

  ```
  'DIRS':
  [
  'C:/www/django/templates',
  ]
  ```

As we have not yet created a Django app, you will have to set `DIRS` to `[os.path.join(BASE_DIR,'templates')]` as per the example above for the code below to work as expected. With `DIRS` set, the next step is to change the view code to use Django's template-loading functionality rather than hard-coding the template paths. Returning to our `current_datetime` view, let's change it like so:

```
from django.template.loader import get_template
from django.template import Context
from django.http import HttpResponse
import datetime

def current_datetime(request):
    now = datetime.datetime.now()
    t = get_template('current_datetime.html')
    html = t.render(Context({'current_date': now}))
    return HttpResponse(html)
```

In this example, we're using the function `django.template.loader.get_template()` rather than loading the template from the filesystem manually. The `get_template()` function takes a template name as its argument, figures out where the template lives on the filesystem, opens that file, and returns a compiled `Template` object. Our template in this example is `current_datetime.html`, but there's nothing special about that `.html` extension. You can give your templates whatever extension makes sense for your application, or you can leave off extensions entirely.

To determine the location of the template on your filesystem, `get_template()` will look in order:

- If `APP_DIRS` is set to `True`, and assuming you are using the DTL, it will look for a "templates" directory in the current app.
- If it does not find your template in the current app, `get_template()` combines your template directories from `DIRS` with the template name that you pass to `get_template()` and steps through each of them in order until it finds your template. For example, if the first entry in your `DIRS` is set to `'/home/django/mysite/templates'`, the above `get_template()` call would look for the template `/home/django/mysite/templates/current_datetime.html`.
- If `get_template()` cannot find the template with the given name, it raises a `TemplateDoesNotExist` exception.

To see what a template exception looks like, fire up the Django development server again by running `python manage.py runserver` within your Django project's directory. Then, point your browser at the page that activates the

`current_datetime` view (e.g., `http://127.0.0.1:8000/time/`). Assuming your `DEBUG` setting is set to `True` and you haven't yet created a `current_datetime.html` template, you should see a Django error page highlighting the `TemplateDoesNotExist` error (Figure 3-1).

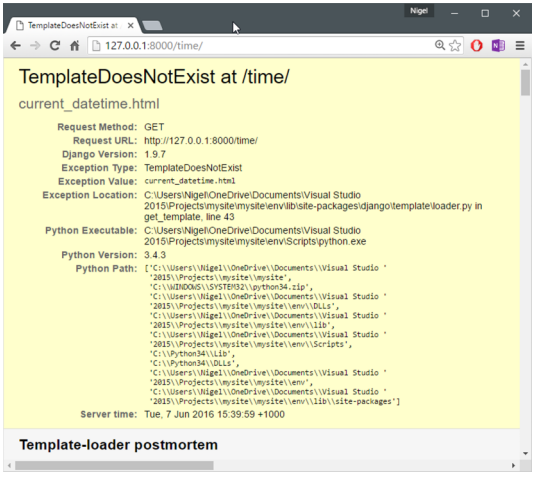

Figure 3-1: *Missing template error page.*

This error page is similar to the one I explained in Chapter 2, with one additional piece of debugging information: a "Template-loader postmortem" section. This section tells you which templates Django tried to load, along with the reason each attempt failed (e.g., "File does not exist"). This information is invaluable when you're trying to debug template-loading errors. Moving along, create the `current_datetime.html` file using the following template code:

```
It is now {{ current_date }}.
```

Save this file to `mysite/templates` (create the "templates" directory if you have not done so already). Refresh the page in your Web browser, and you should see the fully rendered page.

render()

So far, we've shown you how to load a template, fill a `Context` and return an `HttpResponse` object with the result of the rendered template. Next step was

to optimize it to use `get_template()` instead of hard-coding templates and template paths. I took you through this process to ensure you understood how Django templates are loaded and rendered to your browser.

In practice, Django provides a much easier way to do this. Django's developers recognized that because this is such a common idiom, Django needed a shortcut that could do all this in one line of code. This shortcut is a function called `render()`, which lives in the module `django.shortcuts`.

Most of the time, you'll be using `render()` rather than loading templates and creating `Context` and `HttpResponse` objects manually - unless your employer judges your work by total lines of code written, that is.

Here's the ongoing `current_datetime` example rewritten to use `render()`:

```
from django.shortcuts import render
import datetime

def current_datetime(request):
    now = datetime.datetime.now()
    return render(request, 'current_datetime.html', {'current_date': now})
```

What a difference! Let's step through the code changes:

- We no longer have to import `get_template`, `Template`, `Context`, or `HttpResponse`. Instead, we import `django.shortcuts.render`. The `import datetime` remains.
- Within the `current_datetime` function, we still calculate `now`, but the template loading, context creation, template rendering, and `HttpResponse` creation are all taken care of by the `render()` call. Because `render()` returns an `HttpResponse` object, we can simply `return` that value in the view.

The first argument to `render()` is the request, the second is the name of the template to use. The third argument, if given, should be a dictionary to use in creating a `Context` for that template. If you don't provide a third argument, `render()` will use an empty dictionary.

Template Subdirectories

It can get unwieldy to store all of your templates in a single directory. You might like to store templates in subdirectories of your template directory, and that's fine.

In fact, I recommend doing so; some more advanced Django features (such as the generic views system, which we cover in Chapter 10) expect this template layout as a default convention.

Storing templates in subdirectories of your template directory is easy. In your calls to `get_template()`, just include the subdirectory name and a slash before the template name, like so:

```
t = get_template('dateapp/current_datetime.html')
```

Because `render()` is a small wrapper around `get_template()`, you can do the same thing with the second argument to `render()`, like this:

```
return render(request, 'dateapp/current_datetime.html', {'current_date': now})
```

There's no limit to the depth of your subdirectory tree. Feel free to use as many subdirectories as you like.

Windows users, be sure to use forward slashes rather than backslashes. `get_template()` assumes a Unix-style file name designation.

The include Template Tag

Now that we've covered the template-loading mechanism, we can introduce a built-in template tag that takes advantage of it: `{% include %}`.

This tag allows you to include the contents of another template. The argument to the tag should be the name of the template to include, and the template name can be either a variable or a hard-coded (quoted) string, in either single or double quotes.

Anytime you have the same code in multiple templates, consider using an `{% include %}` to remove the duplication. These two examples include the contents of the template `nav.html`. The examples are equivalent and illustrate that either single or double quotes are allowed:

```
{% include 'nav.html' %}
{% include "nav.html" %}
```

This example includes the contents of the template `includes/nav.html`:

```
{% include 'includes/nav.html' %}
```

This example includes the contents of the template whose name is contained in the variable `template_name`:

```
{% include template_name %}
```

As in `get_template()`, the file name of the template is determined by either adding the path to the "templates" directory in the current Django app (if `APPS_DIR` is `True`) or by adding the template directory from `DIRS` to the requested template name. Included templates are evaluated with the context of the template that's including them.

For example, consider these two templates:

```
# mypage.html

<html>
<body>
    {% include "includes/nav.html" %}
    <h1>{{ title }}</h1>
</body>
</html>

# includes/nav.html

<div id="nav">
    You are in: {{ current_section }}
</div>
```

If you render `mypage.html` with a context containing `current_section`, then the variable will be available in the "included" template, as you would expect.

If, in an `{% include %}` tag, a template with the given name isn't found, Django will do one of two things:

- If `DEBUG` is set to `True`, you'll see the `TemplateDoesNotExist` exception on a Django error page.
- If `DEBUG` is set to `False`, the tag will fail silently, displaying nothing in the place of the tag.

There is no shared state between included templates - each include is a completely independent rendering process.

Blocks are evaluated *before* they are included. This means that a template that includes blocks from another will contain blocks that have *already been evaluated and rendered* - not blocks that can be overridden by, for example, an extending template.

Template Inheritance

Our template examples so far have been tiny HTML snippets, but in the real world, you'll be using Django's template system to create entire HTML pages. This leads to a common Web development problem: across a Web site, how does one reduce the duplication and redundancy of common page areas, such as site wide navigation?

A classic way of solving this problem is to use server-side includes, directives you can embed within your HTML pages to "include" one Web page inside another. Indeed, Django supports that approach, with the `{% include %}` template tag just described.

But the preferred way of solving this problem with Django is to use a more elegant strategy called *template inheritance*. In essence, template inheritance lets you build a base "skeleton" template that contains all the common parts of your site and defines "blocks" that child templates can override. Let's see an example of this by creating a more complete template for our `current_datetime` view, by editing the `current_datetime.html` file:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">

<html lang="en">

<head>
    <title>The current time</title>
</head>
<body>
    <h1>My helpful timestamp site</h1>
    <p>It is now {{ current_date }}.</p>

    <hr>
    <p>Thanks for visiting my site.</p>
</body>
</html>
```

That looks just fine, but what happens when we want to create a template for another view - say, the `hours_ahead` view from Chapter 2? If we want again to make a nice, valid, full HTML template, we'd create something like:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">

<html lang="en">

<head>
    <title>Future time</title>
</head>
<body>
    <h1>My helpful timestamp site</h1>
    <p>
        In {{ hour_offset }} hour(s), it will be {{ next_time }}.
    </p>

    <hr>
    <p>Thanks for visiting my site.</p>
</body>
</html>
```

Clearly, we've just duplicated a lot of HTML. Imagine if we had a more typical site, including a navigation bar, a few style sheets, perhaps some JavaScript - we'd end up putting all sorts of redundant HTML into each template.

The server-side include solution to this problem is to factor out the common bits in both templates and save them in separate template snippets, which are then included in each template. Perhaps you'd store the top bit of the template in a file called `header.html`:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">

<html lang="en">

<head>
```

And perhaps you'd store the bottom bit in a file called `footer.html`:

```
    <hr>
    <p>Thanks for visiting my site.</p>
    </body>
</html>
```

With an include-based strategy, headers and footers are easy. It's the middle ground that's messy. In this example, both pages feature a title - "My helpful timestamp site" - but that title can't fit into `header.html` because the title on both pages is different. If we included the h1 in the header, we'd have to include the title, which wouldn't allow us to customize it per page.

Django's template inheritance system solves these problems. You can think of it as an "inside-out" version of server-side includes. Instead of defining the snippets that are common, you define the snippets that are different.

The first step is to define a base template - a skeleton of your page that child templates will later fill in. Here's a base template for our ongoing example:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">

<html lang="en">

<head>
    <title>{% block title %}{% endblock %}</title>
</head>
<body>
    <h1>My helpful timestamp site</h1>
    {% block content %}{% endblock %}

    {% block footer %}
    <hr>
    <p>Thanks for visiting my site.</p>
    {% endblock %}
</body>
</html>
```

This template, which we'll call `base.html`, defines a simple HTML skeleton document that we'll use for all the pages on the site.

It's the job of child templates to override, or add to, or leave alone the contents of the blocks. (If you're following along, save this file to your template directory as `base.html`.)

We're using a template tag here that you haven't seen before: the `{% block %}` tag. All the `{% block %}` tags do is tell the template engine that a child template may override those portions of the template.

Now that we have this base template, we can modify our existing `current_datetime.html` template to use it:

```
{% extends "base.html" %}

{% block title %}The current time{% endblock %}

{% block content %}
    <p>It is now {{ current_date }}.</p>
{% endblock %}
```

While we're at it, let's create a template for the `hours_ahead` view from Chapter 3. (If you're following along with code, I'll leave it up to you to change `hours_`-

ahead to use the template system instead of hard-coded HTML.) Here's what that could look like:

```
{% extends "base.html" %}

{% block title %}Future time{% endblock %}

{% block content %}
<p>
    In {{ hour_offset }} hour(s), it will be {{ next_time }}.
</p>
{% endblock %}
```

Isn't this beautiful? Each template contains only the code that's *unique* to that template. No redundancy needed. If you need to make a site-wide design change, just make the change to `base.html`, and all of the other templates will immediately reflect the change.

Here's how it works. When you load the template `current_datetime.html`, the template engine sees the `{% extends %}` tag, noting that this template is a child template. The engine immediately loads the parent template - in this case, `base.html`.

At that point, the template engine notices the three `{% block %}` tags in `base.html` and replaces those blocks with the contents of the child template. So, the title we've defined in `{% block title %}` will be used, as will the `{% block content %}`.

Note that since the child template doesn't define the footer block, the template system uses the value from the parent template instead. Content within a `{% block %}` tag in a parent template is always used as a fall-back.

Inheritance doesn't affect the template context. In other words, any template in the inheritance tree will have access to every one of your template variables from the context. You can use as many levels of inheritance as needed. One common way of using inheritance is the following three-level approach:

1. Create a `base.html` template that holds the main look and feel of your site. This is the stuff that rarely, if ever, changes.
2. Create a `base_SECTION.html` template for each "section" of your site (e.g., `base_photos.html` and `base_forum.html`). These templates extend `base.html` and include section-specific styles/design.

3. Create individual templates for each type of page, such as a forum page or a photo gallery. These templates extend the appropriate section template.

This approach maximizes code reuse and makes it easy to add items to shared areas, such as section-wide navigation.

Here are some guidelines for working with template inheritance:

- If you use `{% extends %}` in a template, it must be the first template tag in that template. Otherwise, template inheritance won't work.
- Generally, the more `{% block %}` tags in your base templates, the better. Remember, child templates don't have to define all parent blocks, so you can fill in reasonable defaults in a number of blocks, and then define only the ones you need in the child templates. It's better to have more hooks than fewer hooks.
- If you find yourself duplicating code in a number of templates, it probably means you should move that code to a `{% block %}` in a parent template.
- If you need to get the content of the block from the parent template, use `{{ block.super }}`, which is a "magic" variable providing the rendered text of the parent template. This is useful if you want to add to the contents of a parent block instead of completely overriding it.
- You may not define multiple `{% block %}` tags with the same name in the same template. This limitation exists because a block tag works in "both" directions. That is, a block tag doesn't just provide a hole to fill, it also defines the content that fills the hole in the *parent*. If there were two similarly named `{% block %}` tags in a template, that template's parent wouldn't know which one of the blocks' content to use.
- The template name you pass to `{% extends %}` is loaded using the same method that `get_template()` uses. That is, the template name is appended to your `DIRS` setting, or the "templates" folder in the current Django app.
- In most cases, the argument to `{% extends %}` will be a string, but it can also be a variable, if you don't know the name of the parent template until runtime. This lets you do some cool, dynamic stuff.

What's Next?

You now have the basics of Django's template system under your belt. What's next? Most modern Web sites are *database-driven*: the content of the Web site

is stored in a relational database. This allows a clean separation of data and logic (in the same way views and templates allow the separation of logic and display.) The next chapter covers the tools Django gives you to interact with a database.

Chapter 4: Models

In Chapter 2, we covered the fundamentals of building dynamic Web sites with Django: setting up views and URLconfs. As we explained, a view is responsible for doing *some arbitrary logic*, and then returning a response. In one of the examples, our arbitrary logic was to calculate the current date and time.

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

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

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

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

The "Dumb" Way to Do Database Queries in Views

Just as Chapter 2 detailed a "dumb" way to produce output within a view (by hard-coding the text directly within the view), there's a "dumb" way to retrieve data from a database in a view. It's simple: just use any existing Python library to execute an SQL query and do something with the results. In this example

view, we use the `MySQLdb` library to connect to a MySQL database, retrieve some records, and feed them to a template for display as a Web page:

```
from django.shortcuts import render
import MySQLdb

def book_list(request):
    db = MySQLdb.connect(user='me', db='mydb',  passwd='secret', host='localhost')
    cursor = db.cursor()
    cursor.execute('SELECT name FROM books ORDER BY name')
    names = [row[0] for row in cursor.fetchall()]
    db.close()
    return render(request, 'book_list.html', {'names': names})
```

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

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

As you might expect, Django's database layer solves these problems.

Configuring the Database

With all of that philosophy in mind, let's start exploring Django's database layer. First, let's explore the initial configuration that was added to `settings.py` when we created the application:

```
# Database
#...
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    }
}
```

The default setup is pretty simple. Here's a rundown of each setting.

- `ENGINE` tells Django which database engine to use. As we are using SQLite in the examples in this book, we will leave it to the default `django.db.backends.sqlite3`.
- `NAME` tells Django the name of your database. For example: `'NAME': 'mydb',`

Since we're using SQLite, `startproject` created a full filesystem path to the database file for us.

This is it for the default setup - you don't need to change anything to run the code in this book, I have included this simply to give you an idea of how simple it is to configure databases in Django. For a detailed description on how to set up the various databases supported by Django, see Chapter 21.

Your First App

Now that you've verified the connection is working, it's time to create a *Django app* - a bundle of Django code, including models and views, that live together in a single Python package and represent a full Django application. It's worth explaining the terminology here, because this tends to trip up beginners. We've already created a *project*, in Chapter 1, so what's the difference between a *project* and an *app*? The difference is that of configuration vs. code:

- A project is an instance of a certain set of Django apps, plus the configuration for those apps. Technically, the only requirement of a project is that it supplies a settings file, which defines the database connection information, the list of installed apps, the `DIRS`, and so forth.

- An app is a portable set of Django functionality, usually including models and views, that live together in a single Python package. For example, Django comes with a number of apps, such as the automatic admin interface. A key thing to note about these apps is that they're portable and reusable across multiple projects.

There are very few hard-and-fast rules about how you fit your Django code into this scheme. If you're building a simple Web site, you may use only a single app. If you're building a complex Web site with several unrelated pieces such as an e-commerce system and a message board, you'll probably want to split those into separate apps so that you'll be able to reuse them individually in the future.

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

However, there's one requirement regarding the app convention: if you're using Django's database layer (models), you must create a Django app. Models must live within apps. Thus, in order to start writing our models, we'll need to create a new app.

Within the `mysite` project directory (this is the directory where your `manage.py` file is, **not** the `mysite` app directory), type this command to create a `books` app:

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

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

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

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

Defining Models in Python

As we discussed earlier in Chapter 1, the "M" in "MTV" stands for "Model." A Django model is a description of the data in your database, represented as Python code. It's your data layout - the equivalent of your SQL `CREATE TABLE` statements - except it's in Python instead of SQL, and it includes more than just database column definitions.

Django uses a model to execute SQL code behind the scenes and return convenient Python data structures representing the rows in your database tables. Django also uses models to represent higher-level concepts that SQL can't necessarily handle.

If you're familiar with databases, your immediate thought might be, "Isn't it redundant to define data models in Python instead of in SQL?" Django works the way it does for several reasons:

- Introspection requires overhead and is imperfect. In order to provide convenient data-access APIs, Django needs to know the database layout *somehow*, and there are two ways of accomplishing this. The first way would be to explicitly describe the data in Python, and the second way would be to introspect the database at runtime to determine the data models.

  This second way seems cleaner, because the metadata about your tables lives in only one place, but it introduces a few problems. First, introspecting a database at runtime obviously requires overhead. If the framework had to introspect the database each time it processed a request, or even only when the Web server was initialized, this would incur an unacceptable level of overhead.(While some believe that level of overhead is acceptable, Django's developers aim to trim as much framework overhead as possible.) Second, some databases, notably older versions of MySQL, do not store sufficient metadata for accurate and complete introspection.
- Writing Python is fun, and keeping everything in Python limits the number of times your brain has to do a "context switch." It helps productivity if you keep yourself in a single programming environment/mentality for as long as possible. Having to write SQL, then Python, and then SQL again is disruptive.

- Having data models stored as code rather than in your database makes it easier to keep your models under version control. This way, you can easily keep track of changes to your data layouts.
- SQL allows for only a certain level of metadata about a data layout. Most database systems, for example, do not provide a specialized data type for representing email addresses or URLs. Django models do. The advantage of higher-level data types is higher productivity and more reusable code.
- SQL is inconsistent across database platforms. If you're distributing a Web application, for example, it's much more pragmatic to distribute a Python module that describes your data layout than separate sets of `CREATE TABLE` statements for MySQL, PostgreSQL, and SQLite. A drawback of this approach, however, is that it's possible for the Python code to get out of sync with what's actually in the database. If you make changes to a Django model, you'll need to make the same changes inside your database to keep your database consistent with the model. I'll show you how to handle this problem when we discuss *migrations* later in this chapter.

Finally, you should note that Django includes a utility that can generate models by introspecting an existing database. This is useful for quickly getting up and running with legacy data. We'll cover this in Chapter 21.

Your First Model

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

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

- An author has a first name, a last name and an email address.
- A publisher has a name, a street address, a city, a state/province, a country, and a Web site.
- A book has a title and a publication date. It also has one or more authors (a many-to-many relationship with authors) and a single publisher (a one-to-many relationship - aka foreign key - to publishers).

The first step in using this database layout with Django is to express it as Python code. In the `models.py` file that was created by the `startapp` command, enter the following:

```
from django.db import models

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

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

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

Let's quickly examine this code to cover the basics. The first thing to notice is that each model is represented by a Python class that is a subclass of `django.db.models.Model`. The parent class, `Model`, contains all the machinery necessary to make these objects capable of interacting with a database – and that leaves our models responsible solely for defining their fields, in a nice and compact syntax.

Believe it or not, this is all the code we need to write to have basic data access with Django. Each model generally corresponds to a single database table, and each attribute on a model generally corresponds to a column in that database table. The attribute name corresponds to the column's name, and the type of field (e.g., `CharField`) corresponds to the database column type (e.g., `varchar`). For example, the `Publisher` model is equivalent to the following table (assuming PostgreSQL `CREATE TABLE` syntax):

```
CREATE TABLE "books_publisher" (
    "id" serial NOT NULL PRIMARY KEY,
    "name" varchar(30) NOT NULL,
    "address" varchar(50) NOT NULL,
    "city" varchar(60) NOT NULL,
    "state_province" varchar(30) NOT NULL,
    "country" varchar(50) NOT NULL,
    "website" varchar(200) NOT NULL
);
```

Indeed, Django can generate that `CREATE TABLE` statement automatically, as we'll show you in a moment. The exception to the one-class-per-database-table rule is the case of many-to-many relationships. In our example models, `Book` has a `ManyToManyField` called `authors`. This designates that a book has one or many authors, but the `Book` database table doesn't get an `authors` column. Rather, Django creates an additional table - a many-to-many "join table" - that handles the mapping of books to authors.

For a full list of field types and model syntax options, see Appendix B. Finally, note we haven't explicitly defined a primary key in any of these models. Unless you instruct it otherwise, Django automatically gives every model an auto-incrementing integer primary key field called `id`. Each Django model is required to have a single-column primary key.

Installing the Model

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

```
INSTALLED_APPS = (
'django.contrib.admin',
'django.contrib.auth',
'django.contrib.contenttypes',
'django.contrib.sessions',
'django.contrib.messages',
'django.contrib.staticfiles',
)
```

To register our "books" app, add `'books'` to `INSTALLED_APPS`, so the setting ends up looking like this (`'books'` refers to the "books" app we're working on):

```
INSTALLED_APPS = (
'django.contrib.admin',
'django.contrib.auth',
'django.contrib.contenttypes',
'django.contrib.sessions',
'django.contrib.messages',
'django.contrib.staticfiles',
'books',
)
```

Each app in `INSTALLED_APPS` is represented by its full Python path – that is, the path of packages, separated by dots, leading to the app package. Now that the Django app has been activated in the settings file, we can create the database tables in our database. First, let's validate the models by running this command:

```
python manage.py check
```

The `check` command runs the Django system check framework – a set of static checks for validating Django projects. If all is well, you'll see the message `System check identified no issues (0 silenced)`. If you don't, make sure you typed in the model code correctly. The error output should give you helpful information about what was wrong with the code. Any time you think you have problems with your models, run `python manage.py check`. It tends to catch all the common model problems.

If your models are valid, run the following command to tell Django that you have made some changes to your models (in this case, you have made a new one):

```
python manage.py makemigrations books
```

You should see something similar to the following:

```
Migrations for 'books':
  0001_initial.py:
    - Create model Author
    - Create model Book
    - Create model Publisher
    - Add field publisher to book
```

Migrations are how Django stores changes to your models (and thus your database schema) - they're just files on disk. In this instance, you will find a file names `0001_initial.py` in the 'migrations' folder of the `books` app. The `migrate` command will take your latest migration file and update your database schema automatically, but first, let's see what SQL that migration would run. The `sqlmigrate` command takes migration names and returns their SQL:

python manage.py sqlmigrate books 0001 You should see something similar to the following (reformatted for readability):

```
BEGIN;

CREATE TABLE "books_author" (
    "id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,
    "first_name" varchar(30) NOT NULL,
    "last_name" varchar(40) NOT NULL,
    "email" varchar(254) NOT NULL
);
CREATE TABLE "books_book" (
    "id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,
    "title" varchar(100) NOT NULL,
    "publication_date" date NOT NULL
);
CREATE TABLE "books_book_authors" (
    "id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,
    "book_id" integer NOT NULL REFERENCES "books_book" ("id"),
    "author_id" integer NOT NULL REFERENCES "books_author" ("id"),
    UNIQUE ("book_id", "author_id")
);
CREATE TABLE "books_publisher" (
```

```
    "id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,
    "name" varchar(30) NOT NULL,
    "address" varchar(50) NOT NULL,
    "city" varchar(60) NOT NULL,
    "state_province" varchar(30) NOT NULL,
    "country" varchar(50) NOT NULL,
    "website" varchar(200) NOT NULL
);
CREATE TABLE "books_book__new" (
    "id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,
    "title" varchar(100) NOT NULL,
    "publication_date" date NOT NULL,
    "publisher_id" integer NOT NULL REFERENCES
    "books_publisher" ("id")
);

INSERT INTO "books_book__new" ("id", "publisher_id", "title",
"publication_date") SELECT "id", NULL, "title", "publication_date" FROM
"books_book";

DROP TABLE "books_book";

ALTER TABLE "books_book__new" RENAME TO "books_book";

CREATE INDEX "books_book_2604cbea" ON "books_book" ("publisher_id");

COMMIT;
```

Note the following:

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

These `CREATE TABLE` statements are tailored to the database you're using, so database-specific field types such as `auto_increment` (MySQL), `serial` (Post-

greSQL), or `integer primary key` (SQLite) are handled for you automatically. The same goes for quoting of column names (e.g., using double quotes or single quotes). This example output is in PostgreSQL syntax.

The `sqlmigrate` command doesn't actually create the tables or otherwise touch your database - it just prints output to the screen so you can see what SQL Django would execute if you asked it. If you wanted to, you could copy and paste this SQL into your database client, however Django provides an easier way of committing the SQL to the database: the `migrate` command:

```
python manage.py migrate
```

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

```
Operations to perform:
  Apply all migrations: books
Running migrations:
  Rendering model states... DONE

  # ...

  Applying books.0001_initial... OK

  # ...
```

In case you were wondering what all the extras are (commented out above), the first time you run migrate, Django will also create all the system tables that Django needs for the inbuilt apps. Migrations are Django's way of propagating changes you make to your models (adding a field, deleting a model, etc.) into your database schema. They're designed to be mostly automatic, however there are some caveats. For more information on migrations, see Chapter 21.

Basic Data Access

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

```
>>> from books.models import Publisher
>>> p1 = Publisher(name='Apress', address='2855 Telegraph Avenue',
...     city='Berkeley', state_province='CA', country='U.S.A.',
...     website='http://www.apress.com/')
>>> p1.save()
>>> p2 = Publisher(name="O'Reilly", address='10 Fawcett St.',
...     city='Cambridge', state_province='MA', country='U.S.A.',
...     website='http://www.oreilly.com/')
>>> p2.save()
>>> publisher_list = Publisher.objects.all()
>>> publisher_list
[<Publisher: Publisher object>, <Publisher: Publisher object>]
```

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

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

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

```
p1 = Publisher(...)
# At this point, p1 is not saved to the database yet!
p1.save()
# Now it is.
```

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

```
>>> p1 = Publisher.objects.create(name='Apress',
...     address='2855 Telegraph Avenue',
...     city='Berkeley', state_province='CA', country='U.S.A.',
...     website='http://www.apress.com/')
>>> p2 = Publisher.objects.create(name="O'Reilly",
...     address='10 Fawcett St.', city='Cambridge',
...     state_province='MA', country='U.S.A.',
...     website='http://www.oreilly.com/')
>>> publisher_list = Publisher.objects.all()
>>> publisher_list
[<Publisher: Publisher object>, <Publisher: Publisher object>]
```

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

Adding Model String Representations

When we printed out the list of publishers, all we got was this unhelpful display that makes it difficult to tell the `Publisher` objects apart:

```
[<Publisher: Publisher object>, <Publisher: Publisher object>]
```

We can fix this easily by adding a method called `__str__()` to our `Publisher` class. A `__str__()` method tells Python how to display a human-readable representation of an object. You can see this in action by adding a `__str__()` method to the three models:

```
from django.db import models

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

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

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

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

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

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

As you can see, a `__str__()` method can do whatever it needs to do in order to return a representation of an object. Here, the `__str__()` methods for `Publisher` and `Book` simply return the object's name and title, respectively, but the `__str__()` for `Author` is slightly more complex - it pieces together the `first_name` and `last_name` fields, separated by a space. The only requirement for `__str__()` is that it return a string object. If `__str__()` doesn't return a string object - if it returns, say, an integer - then Python will raise a `TypeError` with a message like:

```
TypeError: __str__ returned non-string (type int).
```

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

```
>>> from books.models import Publisher
>>> publisher_list = Publisher.objects.all()
>>> publisher_list
[<Publisher: Apress>, <Publisher: O'Reilly>]
```

Make sure any model you define has a `__str__()` method - not only for your own convenience when using the interactive interpreter, but also because Django

uses the output of `__str__()` in several places when it needs to display objects. Finally, note that `__str__()` is a good example of adding *behavior* to models. A Django model describes more than the database table layout for an object; it also describes any functionality that object knows how to do. `__str__()` is one example of such functionality - a model knows how to display itself.

Inserting and Updating Data

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

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

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

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

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

```
INSERT INTO books_publisher
    (name, address, city, state_province, country, website)
VALUES
    ('Apress', '2855 Telegraph Ave.', 'Berkeley', 'CA',
     'U.S.A.', 'http://www.apress.com/');
```

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

```
>>> p.id
52    # this will differ based on your own data Subsequent calls to `save()` will sav\
e the record in place, without creating a new record (i.e.,
performing an SQL `UPDATE` statement instead of an `INSERT`):

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

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

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

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

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

Selecting Objects

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

```
>>> Publisher.objects.all()
[<Publisher: Apress>, <Publisher: O'Reilly>]
This roughly translates to this SQL:

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

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

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

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

- First, we have the model we defined, `Publisher`. No surprise here: when you want to look up data, you use the model for that data.
- Next, we have the `objects` attribute. This is called a *manager*. Managers are discussed in detail in Chapter 09. For now, all you need to know is that managers take care of all "table-level" operations on data including, most important, data lookup. All models automatically get an `objects` manager; you'll use it any time you want to look up model instances.
- Finally, we have `all()`. This is a method on the `objects` manager that returns all the rows in the database. Though this object *looks* like a list, it's actually a *QuerySet* - an object that represents a specific set of rows from the database. Appendix C deals with QuerySets in detail. For the rest of this chapter, we'll just treat them like the lists they emulate.

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

Filtering Data

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

```
>>> Publisher.objects.filter(name='Apress')
[<Publisher: Apress>]
```

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

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

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

```
>>> Publisher.objects.filter(country="U.S.A.",
state_province="CA")
[<Publisher: Apress>]
```

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

```
SELECT id, name, address, city, state_province, country, website
FROM books_publisher
WHERE country = 'U.S.A.'
AND state_province = 'CA';
```

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

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

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

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

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

Retrieving Single Objects

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

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

Instead of a list (rather, `QuerySet`), only a single object is returned. Because of that, a query resulting in multiple objects will cause an exception:

```
>>> Publisher.objects.get(country="U.S.A.")
Traceback (most recent call last):
    ...
MultipleObjectsReturned: get() returned more than one Publisher -- it returned 2! Loo\
kup parameters were {'country': 'U.S.A.'}
```

A query that returns no objects also causes an exception:

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

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

```
try:
    p = Publisher.objects.get(name='Apress')
except Publisher.DoesNotExist:
    print ("Apress isn't in the database yet.")
else:
    print ("Apress is in the database.")
```

Ordering Data

As you play around with the previous examples, you might discover that the objects are being returned in a seemingly random order. You aren't imagining things; so far we haven't told the database how to order its results, so we're simply getting back data in some arbitrary order chosen by the database. In your Django applications, you'll probably want to order your results according to a certain value - say, alphabetically. To do this, use the `order_by()` method:

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

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

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

You can order by any field you like:

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

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

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

```
>>> Publisher.objects.order_by("state_province",
"address")
 [<Publisher: Apress>, <Publisher: O'Reilly>]
```

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

```
>>> Publisher.objects.order_by("-name")
[<Publisher: O'Reilly>, <Publisher: Apress>]
```

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

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

    def __str__(self):
        return self.name

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

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

Chaining Lookups

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

```
>>> Publisher.objects.filter(country="U.S.A.").order_by("-name")
[<Publisher: O'Reilly>, <Publisher: Apress>]
```

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

```
SELECT id, name, address, city, state_province, country, website
FROM books_publisher
WHERE country = 'U.S.A'
ORDER BY name DESC;
```

Slicing Data

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

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

This translates roughly to:

```
SELECT id, name, address, city, state_province, country, website
FROM books_publisher
ORDER BY name
LIMIT 1;
```

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

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

This returns two objects, translating roughly to:

```
SELECT id, name, address, city, state_province, country, website
FROM books_publisher
ORDER BY name
OFFSET 0 LIMIT 2;
```

Note that negative slicing is *not* supported:

```
>>> Publisher.objects.order_by('name')[-1]
Traceback (most recent call last):
  ...
AssertionError: Negative indexing is not supported.
```

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

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

Updating Multiple Objects in One Statement

We pointed out in the "Inserting and Updating Data" section that the model `save()` method updates *all* columns in a row. Depending on your application, you may want to update only a subset of columns. For example, let's say we want to update the Apress `Publisher` to change the name from `'Apress'` to `'Apress Publishing'`. Using `save()`, it would look something like this:

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

This roughly translates to the following SQL:

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

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

(Note that this example assumes Apress has a publisher ID of `52`.) You can see in this example that Django's `save()` method sets *all* of the column values, not just the `name` column. If you're in an environment where other columns of the database might change due to some other process, it's smarter to change *only* the column you need to change. To do this, use the `update()` method on `QuerySet` objects. Here's an example:

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

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

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

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

```
>>> Publisher.objects.all().update(country='USA')
2
```

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

Deleting Objects

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

```
>>> p = Publisher.objects.get(name="O'Reilly")
>>> p.delete()
>>> Publisher.objects.all()
[<Publisher: Apress Publishing>]
```

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

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

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

```
>>> Publisher.objects.delete()
Traceback (most recent call last):
  File "", line 1, in
AttributeError: 'Manager' object has no attribute 'delete'
```

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

```
>>> Publisher.objects.all().delete()
If you're just deleting a subset of your data, you don't need to include `all()`.
```

To repeat a previous example:

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

What's Next?

Having read this chapter, you have enough knowledge of Django models to be able to write basic database applications. Chapter 9 will provide some information on more advanced usage of Django's database layer. Once you've defined your models, the next step is to populate your database with data. You might have legacy data, in which case Chapter 21 will give you advice about integrating with legacy databases. You might rely on site users to supply your data, in which case Chapter 6 will teach you how to process user-submitted form data. But in some cases, you or your team might need to enter data manually, in which case it would be helpful to have a Web-based interface for entering and managing data. The next chapter covers Django's admin interface, which exists precisely for that reason.

Appendix G: Developing Django with Visual Studio

Regardless of what you might hear trolling around the Internet, Microsoft Visual Studio (VS) has always been an extremely capable and powerful Integrated Development Environment (IDE). As a developer for multiple platforms, I have dabbled in just about everything else out there and have always ended up back with VS.

The biggest barriers to wider uptake of VS in the past have been (in my opinion):

1. Lack of good support for languages outside of Microsoft's ecosystem (C++, C# and VB).
2. Cost of the fully featured IDE. Previous incarnations of Microsoft 'free' IDE's have fallen a bit short of being useful for professional development.

With the release of Visual Studio Community Editions a few years ago and the more recent release of Python Tools for Visual Studio (PTVS), this situation has changed dramatically for the better. So much so that I now do *all* my development in VS - both Microsoft technologies and Python and Django.

I am not going to go on with the virtues of VS, lest I begin to sound like a commercial for Microsoft, so let's assume that you have at least decided to give VS and PTVS a go.

Firstly, I will explain how to install VS and PTVS on your Windows box and then I will give you a quick overview of all the cool Django and Python tools that you have at your disposal.

Installing Visual Studio

Before you start

Because it's still Microsoft, we can't get past the fact that VS is a **big** install. To minimize the chances of grief, please:

1. Turn off your antivirus for the duration of the install
2. Make sure you have a good Internet connection. Wired is better than wireless
3. Pause other memory/disk hogs like OneDrive and Dropbox
4. Close every application that doesn't have to be open

Once you have taken careful note of the warning above, jump on to the Visual Studio website[10] and download the free Visual Studio Community Edition 2015 (Figure G-1).

[10] https://www.visualstudio.com/

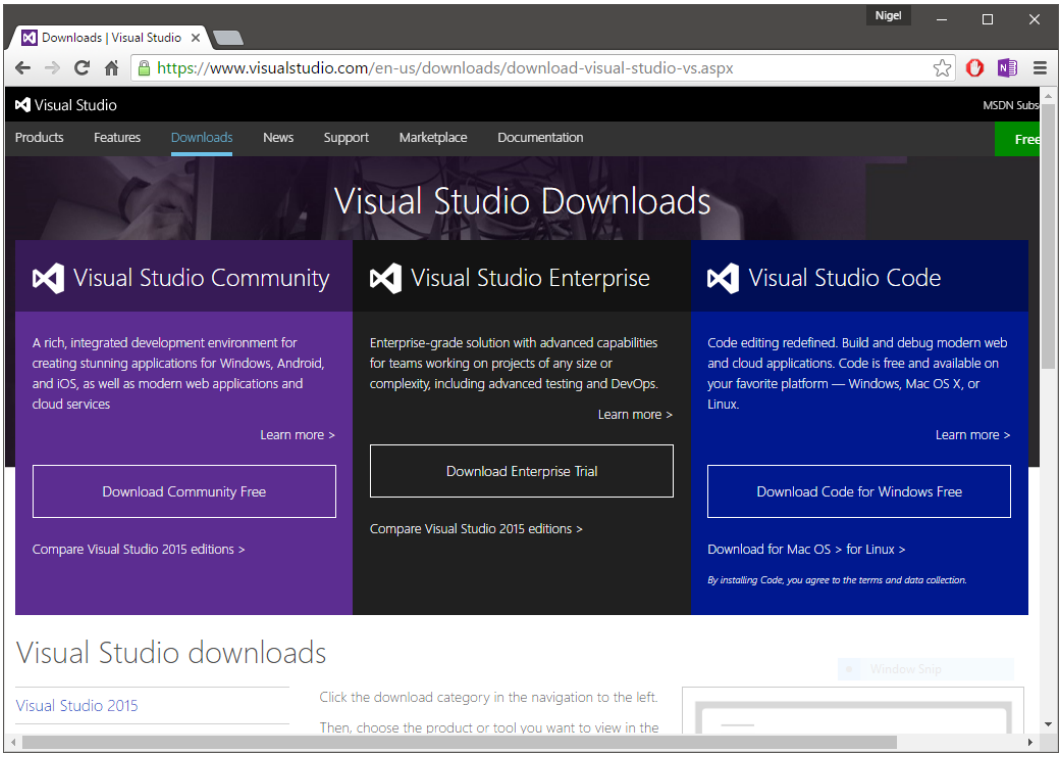

Figure G-1: *Visual Studio Downloads*

Launch the downloaded installer file, make sure the default install option is selected (Figure G-2) and click install.

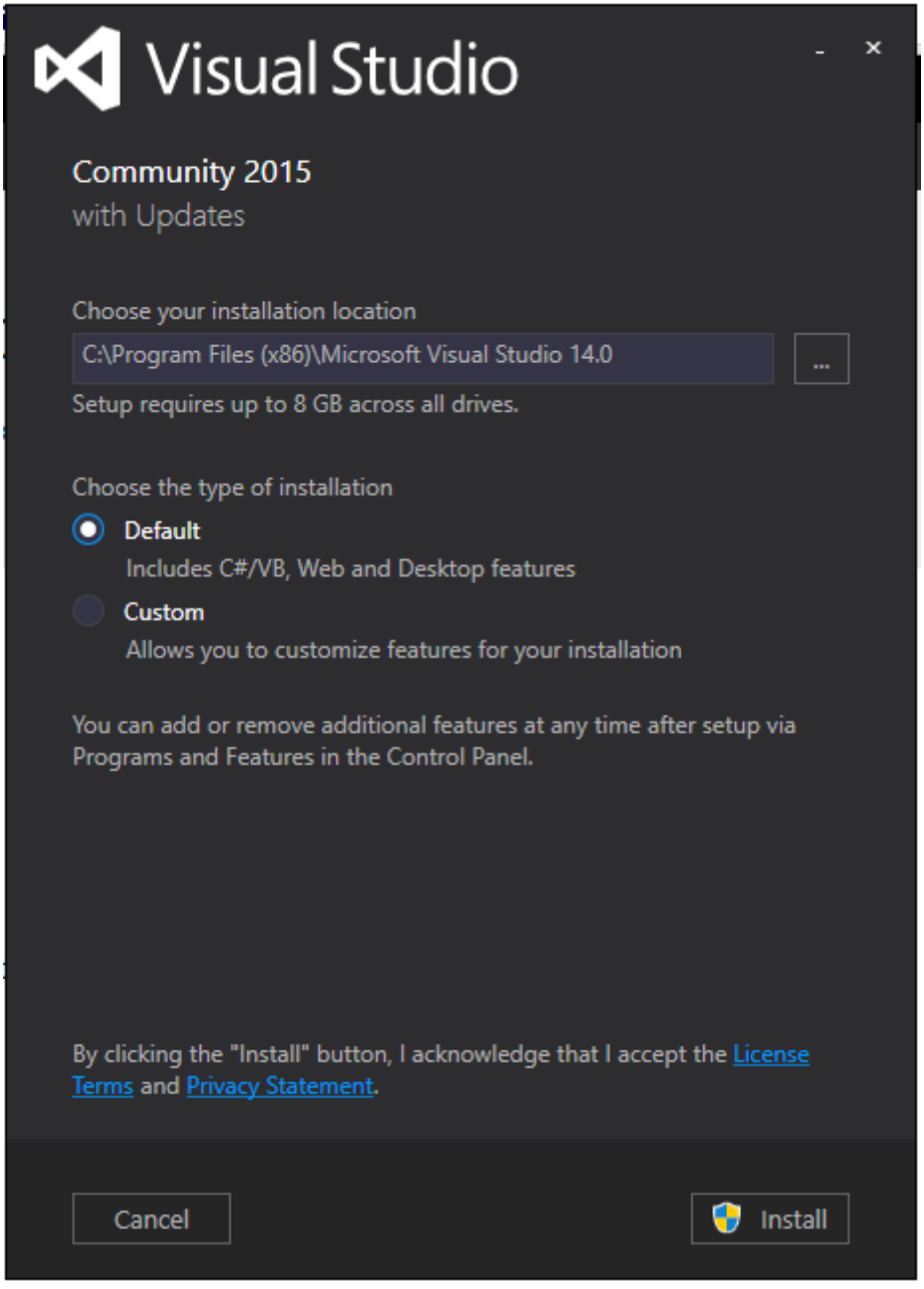

Figure G-2: *Visual Studio's default install*

Now's the time to go make yourself a coffee. Or seven. Microsoft, remember - it's going to take a while. Depending on your Internet connection this can take anywhere from 15 minutes to more than an hour.

In a few rare cases it will fail. This is always (in my experience) either forgetting to turn antivirus off or a momentary dropout in your Internet connection. Luckily VS's recovery process is pretty robust and I have found rebooting and restarting the install after a failure works every time. VS will even remember where it's up to, so you don't have to start all over again.

Install PTVS and Web Essentials

Once you have installed VS, it's time to add Python Tools for Visual Studio (PTVS) and Visual Studio Web Essentials. From the top menu, select `Tools > Extensions and Updates` (Figure G-3).

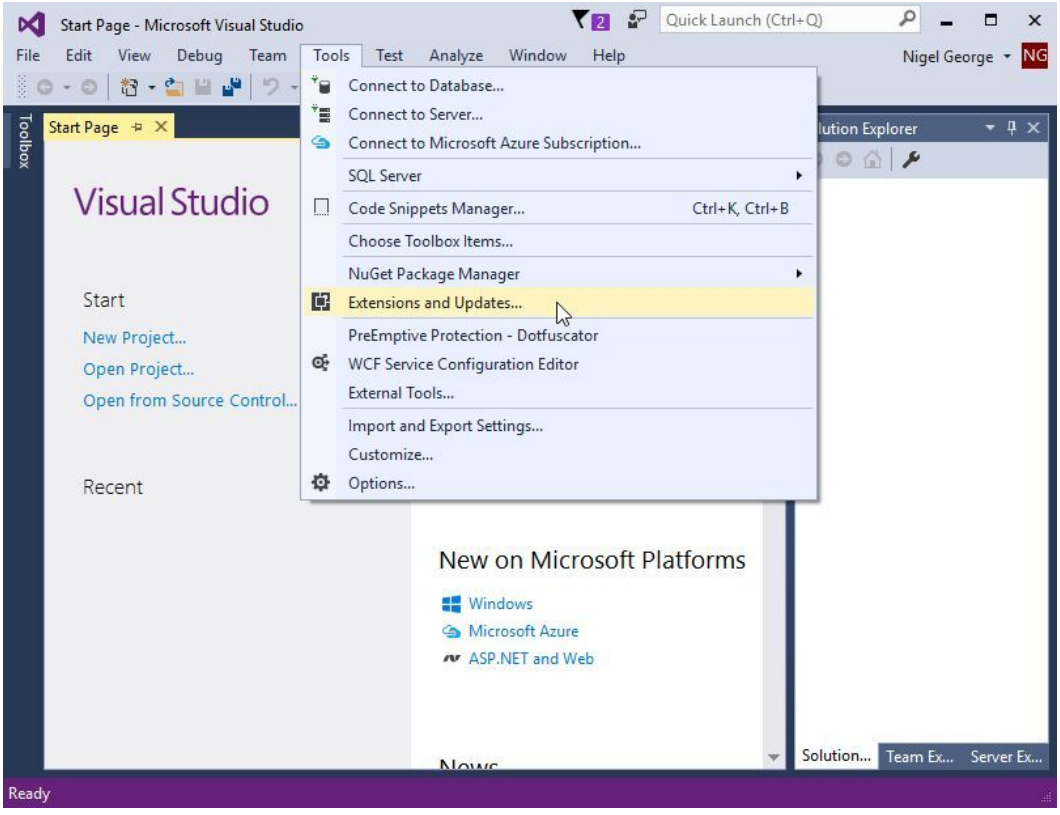

Figure G-3: *Install Visual Studio's extension*

Once the Extensions and Updates window opens, select "online" from the dropdown on the left to go to the VS online application gallery. Type "Python" in the search box on the top right and the PTVS extension should appear on the top of the list (Figure G-4).

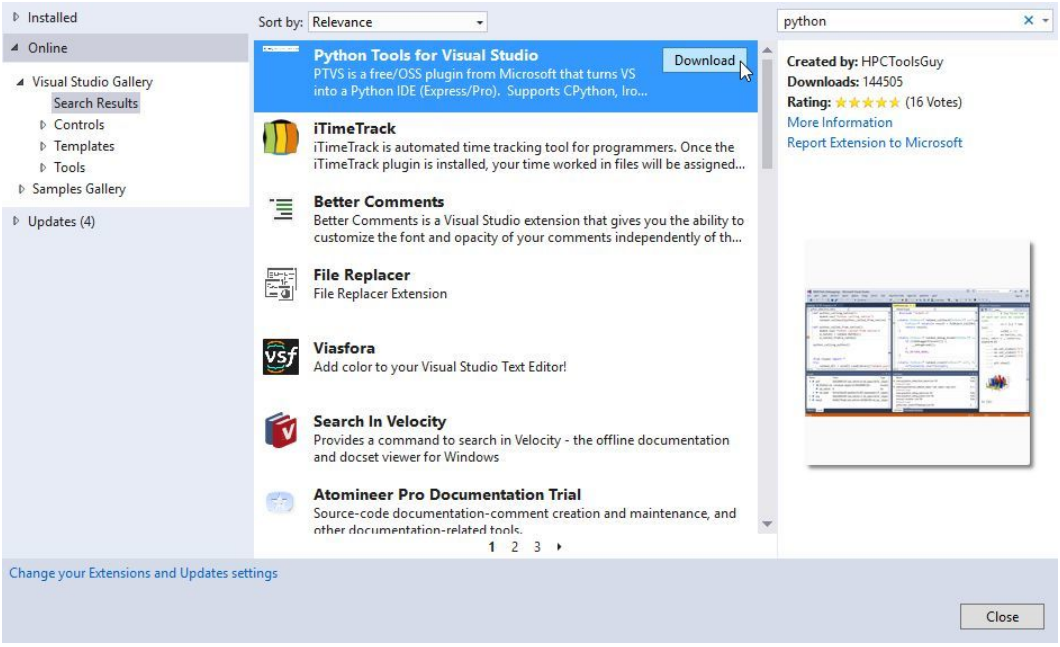

Figure G-4: *Install PTVS extension*

Repeat the same process for VS Web Essentials (Figure G-5). Note that, depending on the VS build and what extensions have been installed previously, Web Essentials may already be installed. If this is the case, the "download" button will be replaced with a green tick icon.

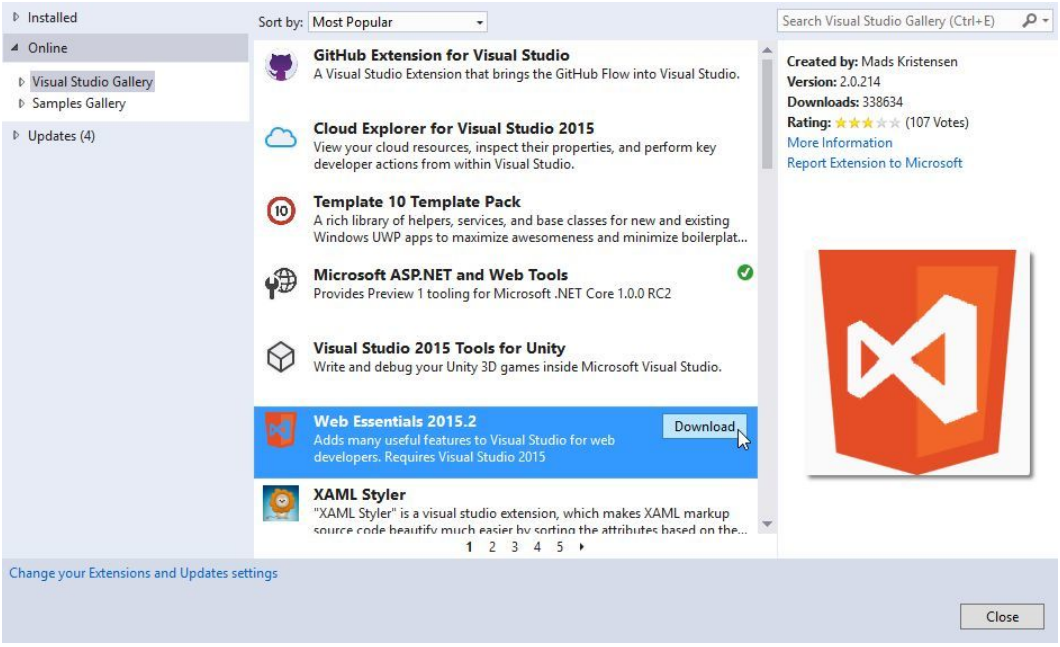

Figure G-5: *Install Web Essentials extension*

Creating A Django Project

One of the great things about using VS for Django development is that the only thing you need to install other than VS is Python. So if you followed the instructions in Chapter 1 and have installed Python, there is nothing else to do - VS takes care of the virtual environment, installing any Python modules you need and even has all of Django's management commands built in to the IDE.

To demonstrate these capabilities, lets create our `mysite` project from Chapter 1, but this time we will do it all from inside VS.

Start A Django Project

Select `File > New > Project` from the top menu and then select a Python web project from the dropdown on the left. You should see something like Figure G-6. Select a Blank Django Web Project, give your project a name and then click OK.

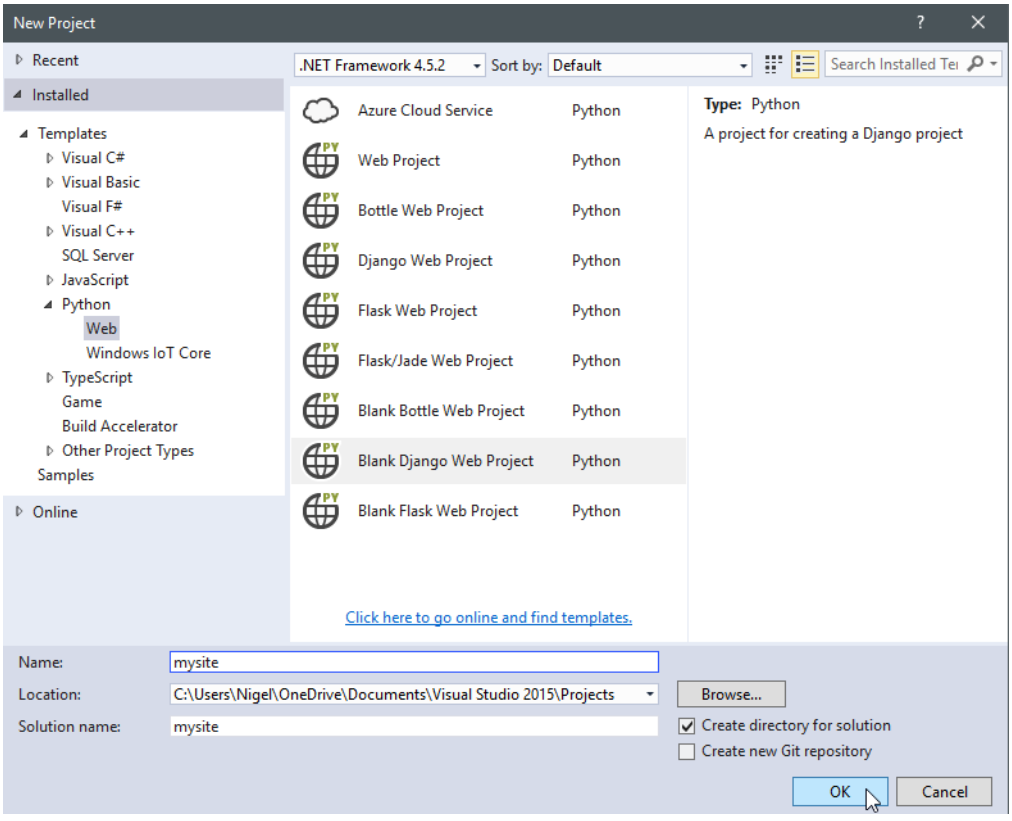

Figure G-6: *Create a blank Django project*

Visual Studio will then display a popup window saying that this project requires external packages (Figure G-7). The simplest option here is to install directly into a virtual environment (option 1), but this will install the latest version of Django, which at the time of writing is 1.9.7. As this book is for the 1.8 LTS version we want to select option 3 "I will install them myself" so we can make the necessary changes to the `requirements.txt` file.

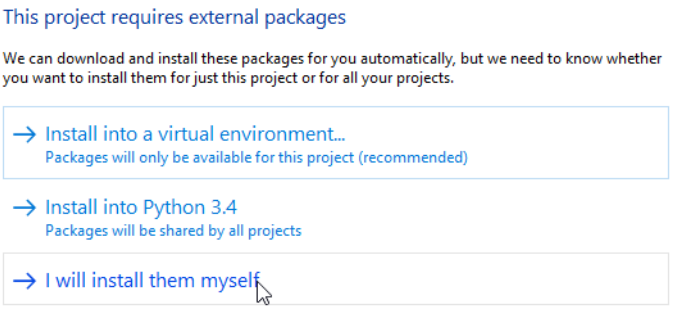

Figure G-7: *Install external packages*

Once the project has installed, you will notice in Solution Explorer on the right of the VS screen the complete Django project structure has been created for you. Next step is to add a virtual environment running Django 1.8. At the time of writing the latest version is 1.8.13, so we have to edit our `requirements.txt` file so the first line reads:

```
django==1.8.13
```

Save the file and then right click "Python Environments" in your Solution Explorer and select "Add Virtual Environment" (Figure G-8).

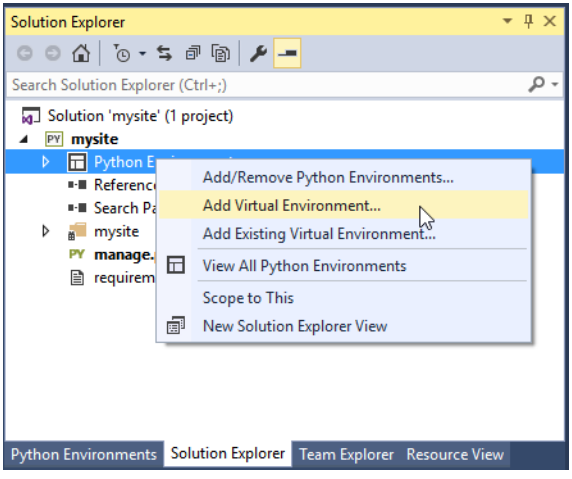

Figure G-8: *Add virtual environment*

In the popup window, change the default environment name from "env" to something more meaningful (if you are following on from the example in Chapter 1, use "`env_mysite`"). Click "Create" and VS will create a virtual environment for you (Figure G-9).

You don't have to explicitly activate a virtual environment when using VS - any code you run will automatically run in the active virtual environment in Solution Explorer.

This is really useful for cases like testing code against Python 2.7 and 3.4 - you just have to right click and activate whichever environment you want to run.

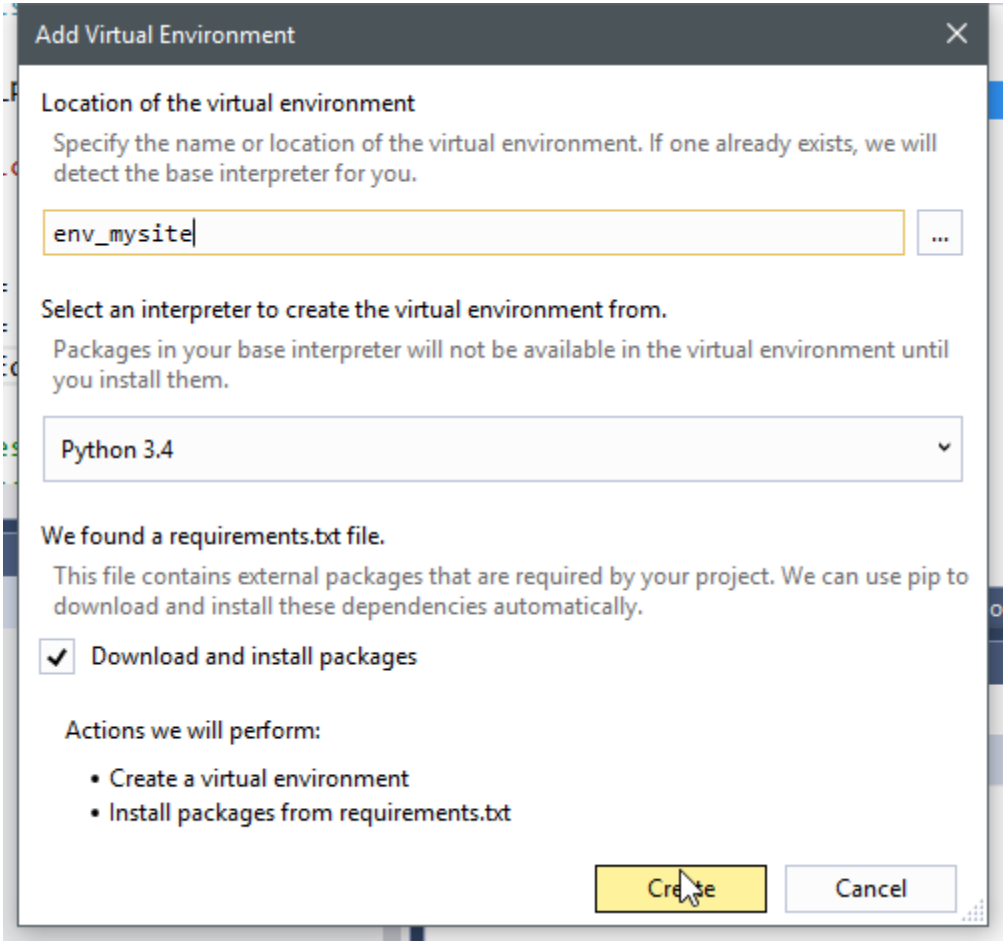

Figure G-9: *Create the virtual environment*

Django Development in Visual Studio

Microsoft have put a lot of effort into ensuring developing Python applications in VS is as simple and headache free as possible. The killer feature for beginning programmers is full IntelliSense for all Python and Django modules. This will accelerate your learning more than any other feature as you don't have to go through documentation looking for module implementations.

The other major aspects of Python/Django programming that VS makes really simple are:

1. Integration of Django management commands.
2. Easy installation of Python packages.
3. Easy installation of new Django apps.

Integration of Django Management Commands

All of Django's common management commands are available from the Project menu (Figure G-10).

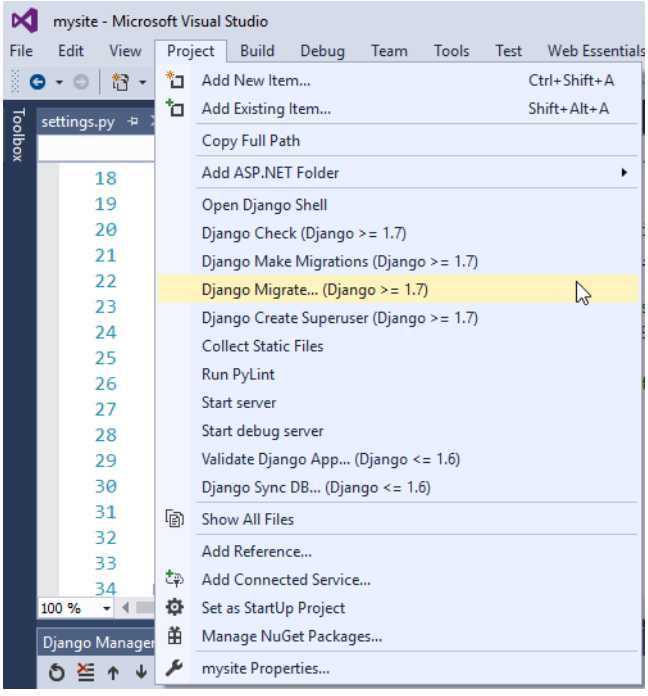

Figure G-10: *Common Django commands on Project menu*

From this menu you can run migrations, create superusers, open the Django shell and run the development server.

Easy Installation of Python Packages

Python packages can be installed directly into any virtual environment from Solution Explorer, just right click on the environment and select "Install Python Package..." (Figure G-11).

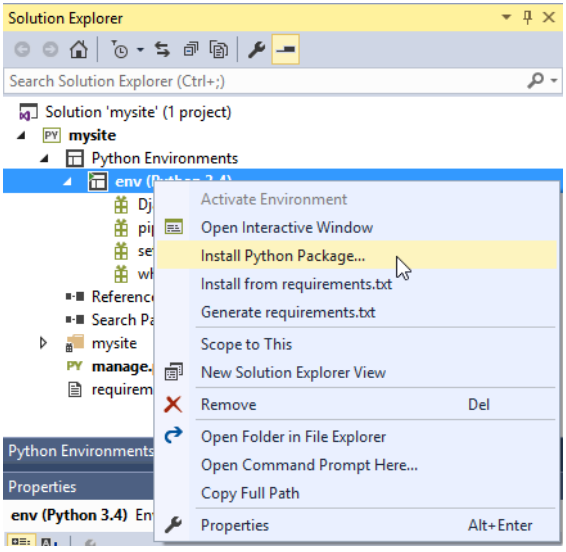

Figure G-11: *Install Python package*

Packages can be installed with either `pip` or `easy_install`.

Easy Installation of New Django Apps

And finally, adding a new Django app to your project is as simple as right clicking on your project and selecting `Add > Django app...` (Figure G-12). Give your app a name and click "OK" and VS will add a new app to your project.

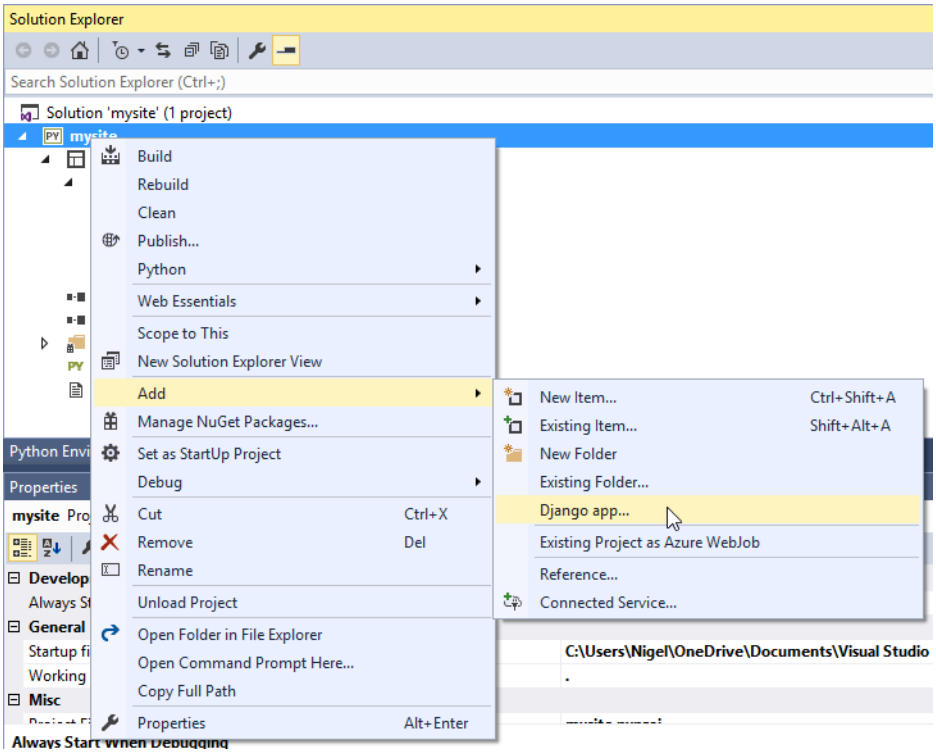

Figure G-12: *Add a Django app*

This is only a quick overview of the things you can do with Visual Studio; just to get you started. Other things worth exploring are:

- VS's repository management including full integration with local Git repos and GitHub.
- Deployment to Azure with a free MSDN developer account (only supports MySQL and SQLite and the time of writing).
- Inbuilt mixed-mode debugger. For example, debug Django and JavaScript in the same debugger.
- Inbuilt support for testing.
- Did I mention full IntelliSense support?

License & Copyright

GNU Free Documentation License

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is

released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A “Transparent” copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not “Transparent” is called “Opaque”.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The “Title Page” means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, “Title Page” means the text near the most prominent appearance of the work’s title, preceding the beginning of the body of the text.

The “publisher” means any person or entity that distributes copies of the Document to the public.

A section “Entitled XYZ” means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as “Acknowledgements”, “Dedications”, “Endorsements”, or “History”.) To “Preserve the Title” of such a section when you modify the Document means that it remains a section “Entitled XYZ” according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along

with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
- C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
- D. Preserve all the copyright notices of the Document.
- E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

- F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- H. Include an unaltered copy of this License.
- I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
- N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
- O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the

Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various partiesâ€"for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise

combine any sections Entitled “Acknowledgements”, and any sections Entitled “Dedications”. You must delete all sections Entitled “Endorsements”.

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an “aggregate” if the copyright resulting from the compilation is not used to limit the legal rights of the compilation’s users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document’s Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in

addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled “Acknowledgements”, “Dedications”, or “History”, the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  YEAR  YOUR NAME.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.3
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.
A copy of the license is included in the section entitled "GNU
Free Documentation License".
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the “with â€¦ Texts.” line with this:

```
with the Invariant Sections being LIST THEIR TITLES, with the
Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.