Covers Django 2 & 3

Mastering Django

The original, best-selling programmer's reference, now completely rewritten for Django 2 and 3

Nigel George

Mastering Django

The original, best-selling programmer's reference completely rewritten for Django 2 and 3.

Nigel George

Mastering Django

Published by GNW Independent Publishing, Hamilton NSW, Australia

ISBN: 978-0-6488844-1-5 (PRINT)

22 21 20 1 2 3 4 5 6 7 8 9

Acknowledgments

As always, my thanks go out to my family for putting up with the months of absences and insanity that comes with completing a project like this around working for a living.

To the Django community, without your input and enthusiasm, I would never find the motivation to even complete a web tutorial, let alone a 600-odd page textbook.

In particular, I would like to make special mention of my beta testers, who were (in no particular order):

> Didier Clapasson, Dominic Bühler, Maria Hynes, James Bellis, Rick Colgan, Simon Schliesky, Lourens Grové, Derrick Kearney, Adrian German, Raphael Thanhoffer, Jan Gondol, David Price, Jaap Onderwaater, Georges Samaha, Bogdan Górski, Hans Hendrick, Martijn Dekker, Alberto Nordmann G., Peter Boers, Robert Helewka, Phil Moose, Jean-Patrick Simard, Gerald Brown, Daniel Coughlin and Hermann Kass.

Thank you all—your feedback and suggestions have helped me make this a far better book than it would have been otherwise.

Table of Contents

Chapter 18 — Odds and Ends 579

1

Introducing Mastering Django

When Django 2 was released, a lot changed in the Django world.

Not just because we finally got rid of the complications with having to deal with both Python 2 and 3, but also because of the many new features, tweaks, updates and optimizations that ensure Django keeps getting better and better. Django 3 continues the tradition of continual improvement in Django's codebase.

What many of you may not know, is the original *Mastering Django:Core* was an update of the original book written by two of the creators of Django—Adrian Holovaty and Jacob Kaplan-Moss. Given the original book had been around since Django 1.1, it got dated. There are also a lot of similarities to the Django docs in several chapters.

(Funny aside: I have had a couple of people email me and take me to task for copying the docs. Lol! Given that the guys who wrote the original book also wrote the original docs, that will happen folks!)

I have decided to start with a clean sheet of paper for *Mastering Django*. This means you don't just get an update; you get a new book, written from scratch to meet the needs of today's programmers.

First and foremost, the book remains a plain-English, easy to follow deep-dive into Django's commonly used core functionalities. It covers both Django 2 and Django 3.

Second, the book complements the existing docs; it doesn't just reproduce them in a different format. I have removed all the original material from Jacob and Adrian's book and all the content from the Django documentation. There are lots of topics not covered adequately in the docs, which provide ample opportunities for me to create resources that will make you a better Django programmer. I have done my best to include them in this book.

Third, the book takes a more practical approach, with less emphasis on theory and more exploration of working code. It's impossible to put all Django's functions into a single book, but I have done my best to include the functions, classes and tools you will use regularly. With each major function or class, I expand them into functioning code examples. Source code is also available for *Mastering Django*.

And finally, while I will not be releasing the book as open-source, the early chapters will remain free to access and read on djangobook.com. As with the previous book, all income from sales support the Django Book project, allowing me to keep the core content ad-free and accessible to all.

Exciting times ahead! :)

All the best with your programming journey!

Cheers,

Big Nige

Who This Book is For

This book is a programmer's manual targeted at intermediate to advanced programmers wishing to gain an in-depth understanding of Django.

In saying that, it doesn't mean beginners can't get value out of the book. Since the publication of the first edition of the book, roughly half of the readers I have spoken to identified as being a beginner when they started out with the book.

The way I write—building on simple concepts and explaining every step—is highly accessible to beginners, so if you are a beginner, you will still learn a great deal about Django from the book.

Where the book can challenge beginners is all the peripheral stuff—HTML, CSS, Python and web development in general—that I don't explain in any detail. The book is big enough as it is, without me trying to teach you all the stuff you need to know that isn't Django-related!

I do, however, give you lots of references so you can easily get more information if you need it.

Structure of the Book

I've broken the book up into two parts:

- **PART 1: Fundamentals**—A high-level overview of Django, how it's structured, and its core components, so you can grasp how Django brings together each element to create powerful and scalable web applications.
- **PART 2: Essentials**—This is the meaty part of the book where we take a deep and detailed dive into the core modules of Django so you can gain a thorough and practical understanding of all the most commonly used modules in Django.

Throughout the book, I will use snippets of code from a fictitious website for a social or sporting club called MyClub.

It was my original plan for this book to end up with a complete website for MyClub. However, it was apparent halfway through that a book that teaches you as much as

possible about the popular parts of Django has very different goals to a book that teaches you how to create a complex, professional website.

In the latter case, no more so than the need to write tests. Tests, while necessary for developing a professional application, distract from learning the core functions of Django. If I wrote proper tests for all the code in this book, I would double the size of the codebase and add little to your learning. For these reasons, I have kept the code in this book to illustrative snippets.

Required Programming Knowledge

The book assumes you have little to no experience with Django.

I expect you to have a basic understanding of web technologies like HTML and CSS, and a basic understanding of how to structure code. You should also be familiar with your OS's terminal or shell program. Absolute beginners shouldn't be too concerned, as it's easy to learn the basics, and that's all you need to get value out of the book.

As Django is written in Python, I also assume you have a basic understanding of Python. Although, since Python is so easy to learn and there are such great resources available online for Python, I haven't heard from a learner yet who said not knowing Python was a barrier. You will learn a lot about Python just by learning Django, and all the extras you need to know are easy to find online.

Software Versions

As the book only covers core Django, you don't require any special functions or libraries, so the latest versions of Python 3 and Django 2 or 3 are OK. At the time of writing, this is Python 3.8.3 and Django 2.2.12 (for Django 2) and Django 3.0.6 (for Django 3).

Django 2 or 3??

There is very little difference between Django 2.2 and Django 3.0, which can be confusing for programmers new to Django. What small differences exist, I have noted in the book.

Django 2.2 is a Long Term Support version, so is older, more stable and better supported than Django 3.0. In saying this, the differences between the two versions are sufficiently small that 99% of the code in this book will work on either version.

Code that only works on Django 3 is clearly marked in the text, so you shouldn't have any problems identifying the parts specific to Django 3.

I provide installation instructions for Windows and Mac users. Linux users can refer to the 90 million Linux installation tutorials online (OK, so that's maybe an exaggeration, but there are *lots*).

All the code in this book will run on Windows, macOS or Linux. While the code and screenshots are all from a Windows machine, the fundamentals remain the same—all three have a terminal or command window and management commands like `runserver` work the same on all three platforms.

Coding style is also identical across platforms with one exception—I use Windows-style backslashes in file paths. This is to assist Windows users to differentiate between Windows' native use of backslashes and Django's implementation of forward slashes in path names. Linux and macOS users, simply need to substitute forward slashes in these cases.

All browser screenshots are taken from the latest version of the Chrome browser. If you are using FireFox, Safari or some other browser, your screen may look different than the screenshots.

The images are in full-color in the PDF and eBook versions, but are grayscale in the printed book (color is *way* too expensive to print). Paperback users are encouraged to run the code to see the full-color effect in your browser.

Source Code

You can download the source code and resources in this book from `https://djangobook.com/mastering-django-source/`.

The included source has been written in Django 3.0. All source code has been tested and will run unmodified on Django 2.2, except for the MariaDB configuration files as MariaDB support was not added until Django 3.0.

The source has been tested against the version of SQLite that comes with Django. While the data structures used in the book are simple and shouldn't cause problems if you decide to use another database, there are no guarantees. If you find any database-related quirks, refer to the database engine documentation.

The source is broken up into folders, one for each chapter of the book. Rather than delete code that changes from chapter to chapter, I have commented out lines of code in the source so you can see where the code has changed.

Code line numbering in the book is provided so you can easily cross-reference my explanations to individual lines of code in the book. In most cases, line numbering in the book does not match line numbers in the source files.

The source is not designed to be executed as-is. The SQLite database file and migrations for each chapter have been removed from the source. While copying the source code from a chapter and running it inside a virtual environment will work in most cases (after running the migrations), there is no guarantee it will. The source code is for your reference and to assist your learning, it's not fully functioning code that you can just copy and use in your projects.

Found a Bug or Typo?

With a book project the size of Mastering Django the odd bug or typo will slip through. This is even after countless edits from me and plenty of feedback from my awesome beta testers (you know who you are :)).

In a perfect world, I'd be able to pay the megabucks for professional editors, but there's a reason why those of us who support Django all have day jobs; it ain't exactly a meal ticket.

So, if you find a bug or a typo, get in touch with me via the Djangobook website[1] and let me know so I can fix the error. The major plus of self-publishing is that I can make changes quickly and send updates to everyone as soon as I have a batch of edits completed. I will even make sure you get a free electronic copy of the updated version if one of your edits makes it through to a published version of the book.

Errata and Django 3 Updates

I will be publishing errata (bugs and typos) information and updates for future versions of Django 3 on the website here:

```
https://djangobook.com/mastering-django-errata-and-updates/
```

Getting Help

For any other questions regarding the book, you can contact me via the help page on the Djangobook website. Note that, due to working full-time, it's very difficult for me to find time to answer Django questions that don't directly relate to my books and courses.

1 https://djangobook.com/django-help/

Your first points of contact for general information on Django and answers to "how do I…" questions should be Stack Overflow[2], your favorite search engine, and any one of the dozens of social media groups relating to Django.

2 https://stackoverflow.com/questions/tagged/django

Part 1

Django Fundamentals

2

Installing Python and Django

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

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

I've written this chapter mostly for those of you running Windows, as most new users are on Windows. I have also included a section on installing Python 3 and Django on macOS.

If you are using Linux, there are many resources on the Internet—the best place to start is Django's own installation instructions[1].

For Windows users, your computer can be running any recent version of Windows (7, 8.1 or 10).

This chapter also assumes you're installing Django on a desktop or laptop computer and will use the development server and SQLite to run all the code in this book. This is by far the easiest and best way to set up Django when you are first starting.

1 https://docs.djangoproject.com/en/dev/intro/install/

Installing Python

A lot of Windows applications use Python, so it may be already installed on your system. You can check this out by opening a command prompt, or running PowerShell, and typing `python` at the prompt.

If Python isn't installed you'll get a message saying that Windows can't find Python. If Python is installed, the python command will open the Python interactive interpreter:

```
C:\Users\Nigel>python
Python 3.6.0 (v3.6.0:41df79263a11, Dec 23 2016, 07:18:10) [MSC
v.1900 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

You can see in the above example that my PC is running Python 3.6.0. Django 2.2 is compatible with Python version 3.5 and later. Django 3.0 is compatible with Python version 3.6 and later. If you have an older version of Python, you must install Python 3.7 or 3.8 for the code in this book to work. If you have Python 3.5 or 3.6, I still recommend you install Python 3.8 to ensure you have the latest version installed on your machine.

Assuming Python 3 is not installed on your system, you first need to get the installer. Go to `https://www.python.org/downloads/` and click the big yellow button that says "Download Python 3.8.x".

At the time of writing, the latest version of Python is 3.8.3, but it may have been updated by the time you read this, so the numbers may be slightly different. 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, there is one essential customization you must make.

By default, the Python executable is not added to the Windows PATH. For Django to work correctly, Python must be listed in the PATH statement. Fortunately, this is easy to rectify—when the Python installer screen opens, make sure "Add Python 3.8 to PATH" is checked before installing (Figure 2-1).

Do Not Forget This Step!

It will solve most problems that arise from the incorrect mapping of pythonpath (an important variable for Python installations) in Windows.

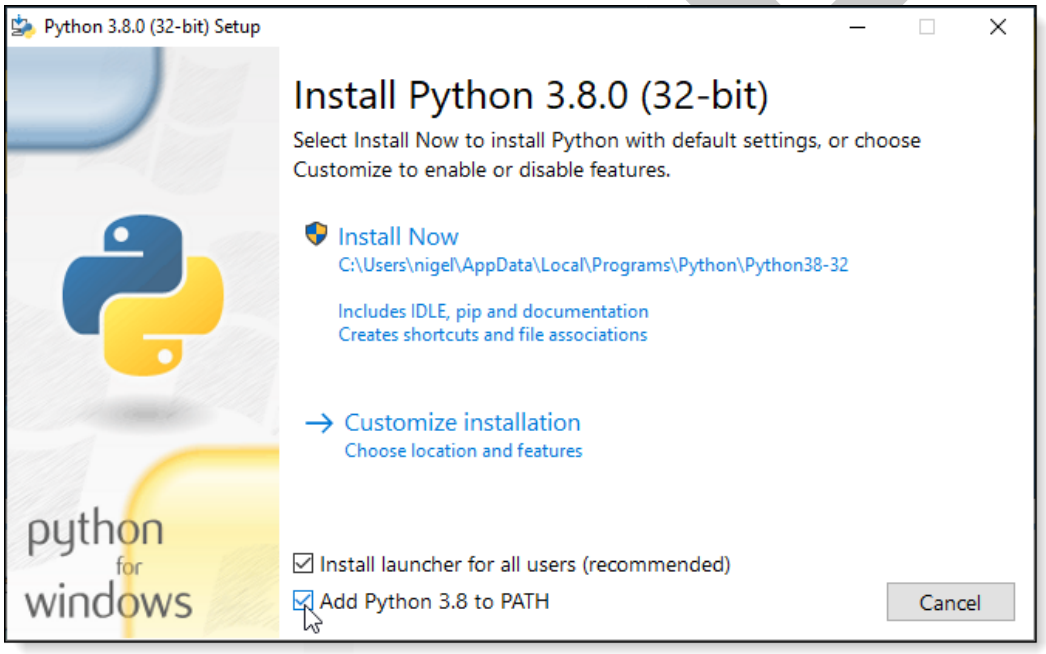

Figure 2-1. *Check the "Add Python 3.8 to PATH" box before installing.*

Once Python is installed, restart Windows and then type `python` at the command prompt. You should see something like this:

```
C:\Users\nigel> python
Python 3.8.3 (tags/v3.8.3:6f8c832, May 13 2020, 22:20:19) [MSC
v.1925 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

Installing Python on macOS

If you open a terminal and type `python` at the prompt, you will see that the system version is Python 2 (Figure 2-2). Django is not compatible with Python 2, so we need to install the latest version of Python 3.

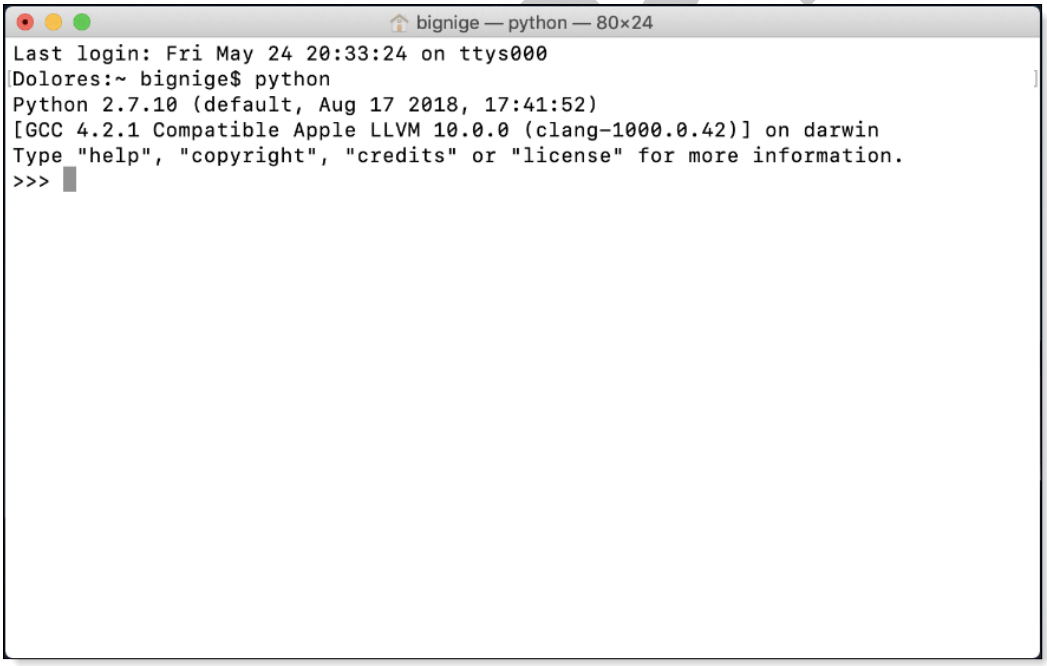

Figure 2-2. *macOS uses Python 2, which is incompatible with Django.*

Downloading a copy of Python 3 follows the same process as Windows—Go to `https://www.python.org/downloads/` and click the big yellow button that says "Download Python 3.x.x". Your browser should automatically detect that you are using macOS and take you to the correct download page. If it doesn't, select the correct operating system from the links below the button.

The Mac installer is in `.pkg` format, so once it's downloaded, double-click the file to run the package installer (Figure 2-3). The screenshot is for Python 3.7, but the process is identical for Python 3.8.

Figure 2-3. *Follow the prompts to install Python 3 on macOS.*

Follow the installations steps and, when Python 3 has been installed, open a new terminal window. If the installation was successful, typing `python3` at the prompt will open the Python 3 interactive shell (Figure 2-4). Note that macOS will happily run multiple versions of Python on the one machine, you just need to make sure you select the correct version when running the terminal.

```
bignige — Python — 80×24
Last login: Fri May 24 20:33:36 on ttys000
Dolores:~ bignige$ python
Python 2.7.10 (default, Aug 17 2018, 17:41:52)
[GCC 4.2.1 Compatible Apple LLVM 10.0.0 (clang-1000.0.42)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> ^D
Dolores:~ bignige$ python3
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 16:52:21)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Figure 2-4. *Once Python 3 is installed, run it from the terminal with the* `python3` *command.*

Creating a Python Virtual Environment

When you are writing new software programs, it's possible (and common!) to modify dependencies and environment variables that your other software depends on. This can cause many 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 filesystem separate from the rest of the software on your computer.

The virtual environment tool in Python is called `venv`, but before we set up `venv`, we need to create our club site project folder.

Create a Project Folder

Our project folder will house not only our virtual environment, but all the code and media for our Django club site.

The project folder can go anywhere on your computer, although it's highly recommended you create it somewhere in your user directory, so you don't get permission issues later on. A good place for your project in Windows is your *My Documents* folder. On a Mac your *Documents* folder is also a logical choice; however, it can go anywhere in your user directory.

Create a new folder on your system. I have named the folder `myclub_project`, but you can give the folder any name that makes sense to you.

For the next step, you need to be in a command window (terminal on Linux and macOS). The easiest way to do this in Windows is to open Windows Explorer, hold the SHIFT key and right-click the folder to get the context menu and click on **Open command window here** (Figure 2-5).

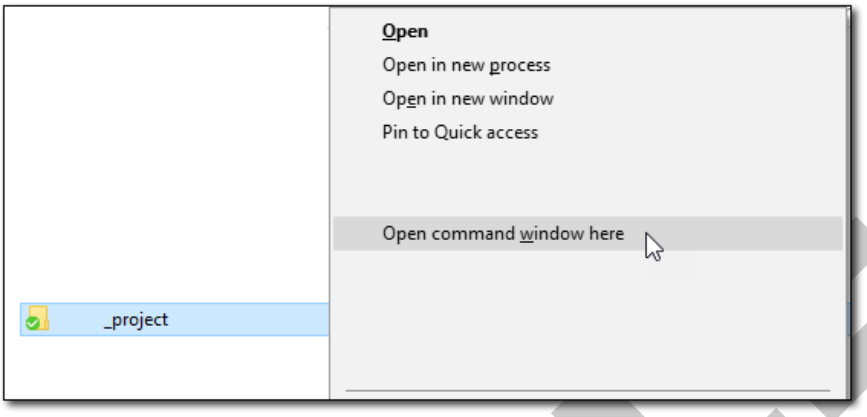

Figure 2-5. *Hold the shift key and right-click a folder to open a command window.*

Terminal in Windows 10

If you are running newer versions of Windows 10, the command prompt has been replaced by PowerShell. For the examples in this book, the command prompt and PowerShell are functionally the same, and all commands will run in PowerShell unmodified.

Create a Python Virtual Environment

Once you have created your project folder, you need to create a virtual environment for your project by typing the following at the command prompt you just opened:

On Windows

```
...\Documents\myclub_project> python -m venv env_myclub
```

On Mac

```
...$ python3 -m venv env_myclub
```

Remember, you must be inside the project folder!

The function of this command is straightforward—the `-m` option tells Python to run the `venv` module as a script. `venv` in turn requires one parameter: the name of the virtual environment to be created. So this command is saying "create a new Python virtual environment and call it *env_myclub*"

Once `venv` has finished setting up your new virtual environment, it will switch to an empty command prompt. When it's done, open Windows Explorer (Finder on a Mac) and have a look at what `venv` created for you. In your project folder, you will now see a folder called
`\env_myclub` (or whatever name you gave the virtual environment). If you open the folder on Windows, you will see the following:

```
\Include
\Lib
\Scripts
pyvenv.cfg
```

On a Mac, it's:

```
/bin
/Include
/Lib
pyvenv.cfg
```

On either platform, if you look inside the `\Lib` folder, you will see `venv` has created a complete Python installation for you, separate from your other software, so you can work on your project without affecting other software on your 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:

On Windows

```
env_myclub\scripts\activate
```

On Mac

```
source env_myclub/bin/activate
```

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

```
(env_myclub) ...\Documents\myclub_project>
```

On a Mac, the prompt looks like this:

```
(env_myclub) ... <yourusername>$
```

The `(env_myclub)` at the beginning of the command prompt lets you know that you are running in the virtual environment.

Oops! Script Error!

If you are using PowerShell and running this script for the first time, the `activate` command will throw a permission error.

If this happens to you, open PowerShell as an administrator and run the command:

```
Set-ExecutionPolicy remoteSigned
```

Once you have run this command, the activation script will run.

If you want to exit the virtual environment, you just type `deactivate` at the command prompt:

```
(env_myclub) ...\Documents\myclub_project> deactivate
...\Documents\myclub_project>
```

Installing Django

Mac Users Note

Once Python 3 and the virtual environment are installed, the installation steps for Django are identical on both Windows and macOS.

The critical thing to remember with macOS is that system Python is version 2 and Django requires Python 3, so you *must* be running the Python virtual environment on macOS to run any of the code in this book.

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

```
pip install "django>=2.2,<3"
```

For Django 3, the command is:

```
pip install "django>=3.0,<4"
```

If you are not familiar with the `pip` command, put briefly, it's the Python package manager and is used to install Python packages. To keep with Python programming tradition, `pip` is a recursive acronym for "Pip Installs Packages".

This will instruct `pip` to install the latest version of Django 2 or Django 3 into your virtual environment. Your command output should look like this (for Django 2.2):

```
(env_myclub) ...\myclub_project> pip install "django>=2.2,<3.0"
Collecting django<3.0,>=2.2
Downloading .../Django-2.2.12-py3-none-any.whl (7.5MB)
    |################################| 7.5MB 2.2MB/s
Collecting pytz (from django<3.0,>=2.2)
Downloading .../pytz-2020.1-py2.py3-none-any.whl (510kB)
    |################################| 512kB 3.3MB/s
Collecting sqlparse (from django<3.0,>=2.2)
Downloading .../sqlparse-0.3.1-py2.py3-none-any.whl (40kB)
    |################################| 40kB 2.6MB/s
Installing collected packages: pytz, sqlparse, django
Successfully installed django-2.2.12 pytz-2020.1 sqlparse-0.3.1
```

The Django 3 installation output is identical except for the version numbers.

To test the installation, go to 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_myclub) ...\myclub_project>python
Python 3.8.3 (tags/v3.8.3:6f8c832, May 13 2020, 22:20:19) [MSC
v.1925 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more
information.
>>> import django
>>> django.get_version()
'2.2.12' # Your version may be different.
>>> exit()
```

Don't forget to exit the Python interpreter when you are done (you can also use CTRL-Z).

You can also check if Django has been installed directly from the command prompt with:

```
(env_myclub) ...\myclub_project>python -m django --version
2.2.12 # Your version may be different.
```

Starting a Project

Once you've installed Python and Django, you can take the first step in developing a Django application by creating a Django *project*.

A Django project is a collection of settings and files for a single Django website. To create a new Django project, we'll be using a special command to auto-generate the folders, files and code that make up a Django project. This includes a collection of settings for an instance of Django, database configuration, Django-specific options and application-specific settings.

I am assuming you are still running the virtual environment from the previous installation step. If not, start it again with `env_myclub\scripts\activate\`.

From your virtual environment command line, run the following command:

```
(env_myclub) ...>django-admin startproject myclub_site
```

This command will automatically create a `myclub_site` folder in your project folder, and all the necessary files for a basic, but fully functioning Django website. Feel free to explore what `startproject` created now if you wish, however, we will go into greater detail on the structure of a Django project in the next chapter.

Creating a Database

Django includes several applications by default (e.g., the admin program and user management and authentication). Some of these applications make use of at least one database table, so we need to create tables in the project database before we can use them. To do this, change into the `myclub_site` folder created in the last step (type `cd myclub_site` at the command prompt) and run the following command:

```
python manage.py migrate
```

The migrate command creates a new SQLite database and any necessary database tables, according to the settings file created by the `startproject` command (more on the settings file later). If all goes to plan, you'll see a message for each migration it applies:

```
(env_myclub) ...\myclub_site>python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, sessions
Running migrations:
  Applying contenttypes.0001_initial... OK
  # several more migrations (not shown)
```

The Development Server

Let's verify your Django project works. Make sure you are in the outer `myclub_site` directory and run the following command:

```
python manage.py runserver
```

This will start the Django development server—a lightweight Web server written in Python. The development server was created so you can develop things rapidly, without having to deal with configuring a production server until you're ready for deployment.

When the server starts, Django will output a few messages before telling you that the development server is up and running at `http://127.0.0.1:8000/`. If you were wondering, 127.0.0.1 is the IP address for localhost, or your local computer. The 8000 on the end is telling you that Django is listening at port 8000 on your local host.

You can change the port number if you want to, but I have never found a good reason to change it, so best to keep it simple and leave it at the default.

Now that the server is running, visit `http://127.0.0.1:8000/` with your web browser. You'll see Django's default welcome page, complete with a cool animated rocket (Figure 2-6).

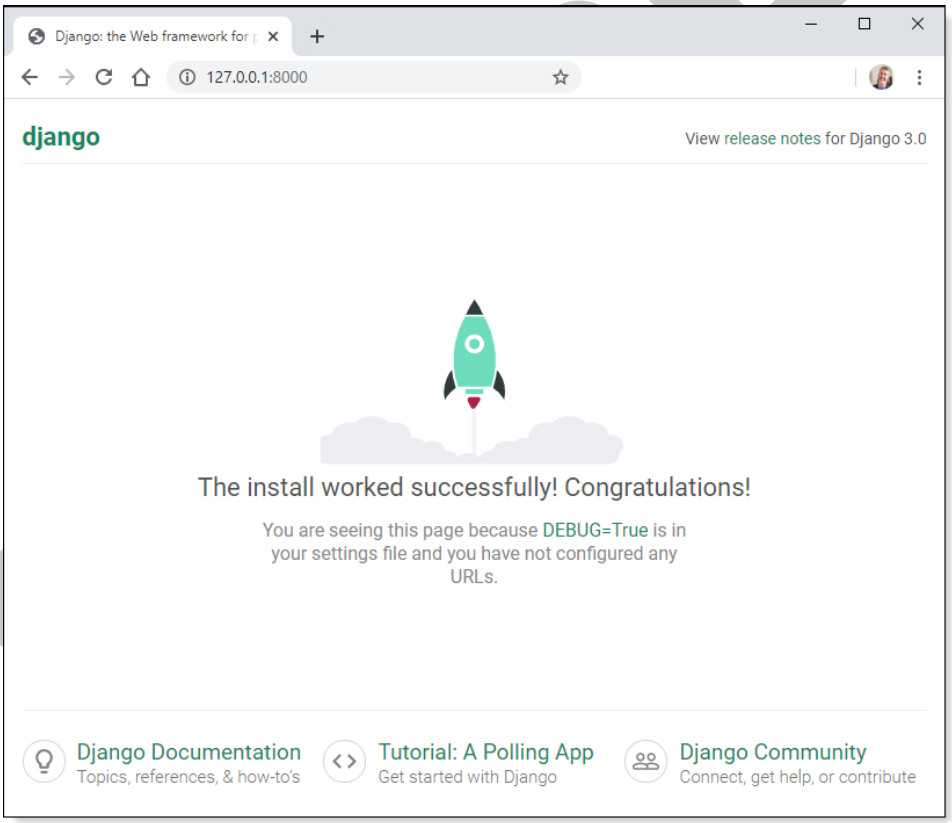

Figure 2-6. *Django's welcome page. The welcome page is the same for Django 2 and 3.*

TIP: Remember the Startup Sequence

It will help to make a note of this sequence, so you know how to start your Django project each time you return to the examples in this book:

On Windows:

1. Shift right-click your project folder to open a command window.
2. Type in `env_myclub\scripts\activate` to run your virtual environment.
3. Change into your site directory (`cd myclub_site`) to run `manage.py` commands (e.g., `runserver`).
4. Type `deactivate` to exit the virtual environment.

On macOS:

1. CTRL-click your project folder to open a terminal window.
2. Type in `source env_myclub/bin/activate` to run your virtual environment.
3. Change into your site directory (`cd myclub_site`) to run `manage.py` commands (e.g., `runserver`).
4. Type `deactivate` to exit the virtual environment.

Chapter Summary

In this chapter, I showed you how to install Python 3 and Django on both Windows and macOS. In the next chapter, we will step back a bit and have a big-picture look at Django's structure and how all the parts of Django work together to create powerful, scalable web applications.

3

The Big Picture

Django's Structure—A Heretic's Eye View

The most common complaints from those new to Django is "it's too hard to understand", or "it's too complex". You may even be thinking this yourself right now.

At the fundamental level, Django isn't complicated. Yes, it has its quirks and, yes, big Django projects can be very complex beasts, but bottom line: Django is a logically structured framework built on the easiest to learn programming language available (Python).

As an educator, I have spent countless hours trying to work out why people find Django complex and hard to learn. Thinking on this has led me to commit heresy #1: it's not your fault, we've just been teaching it wrong.

Remember all the books and tutorials that start with "Django is a Model-View-Controller (MVC) framework…"? (This is cut and paste from one of my books, so I am as guilty as anyone in this).

Stating up front that Django is an MVC framework gets one of two responses:

1. Beginners say "What the heck is MVC? **groan*. Guess that's one more fricking thing I have to learn!"

2. More experienced programmers say "Aha! It's just like Framework X."

In both cases, they're almost entirely wrong.

Django is a Loosely Coupled Framework

If you can indulge me a minute, and purge your brain of your favorite Three Letter Acronyms (TLAs), there's an easier way to understand this.

The first step is to understand that Django is not the result of an academic exercise, nor is it some guru's idea of cool—Django's creators designed Django to solve a particular set of problems in a busy and complex news organization. At the center of this set of problems were three very different needs:

1. The data guys (and gals) needed a common interface to work with disparate data sources, formats and database software.
2. The design teams needed to manage the user experience with the tools they already had (HTML, CSS, JavaScript etc.).
3. The hard-core coders required a framework that allowed them to deploy system changes rapidly and keep everyone happy.

Crucial to making this all work was ensuring each of these core components—data, design and business logic—could be managed independently, or to use the correct computer parlance—the framework had to employ *loose coupling*.

Now, it's important to understand that I'm not trying to say Django is doing anything magic or new here, nor were the problems Django's creators faced unique. The creators of Django are brilliant guys, and they certainly knew MVC was a well-established design pattern that would help solve their problems.

My point is it's highly unlikely any of them ever said, "Hang on boys, we need to change this code because Wikipedia says a controller should …".

You need not get hung up on semantics—you can safely forget about the confusing TLAs and whether Django is like Framework X and concentrate on what Django is.

Django's architecture comprises three major parts:

- **Part 1** is a set of tools that make working with data and databases much easier.
- **Part 2** is a plain-text template system suitable for non-programmers; and
- **Part 3** is a framework that handles communication between the user and the database and automates many of the painful parts of managing a complex website.

Parts 1 and 2 are instantly relatable:

- **Django Models** are the tools we use to work with data and databases; and
- **Django Templates** provide a designer-friendly plain-text template system.

But what about Part 3? I hear you ask, isn't that the controller, or a Django view?

Well, no. Which leads me to heresy #2:

A Django View is Not a Controller

Check out Figure 3-1 on the next page, does it look familiar?

This is one of my diagrams, but there are plenty of similar versions out there. A common way of explaining Django's architecture in terms of MVC is to describe it as a Model-Template-View (MTV) or Model-View-Template (MVT). There's no difference between MTV and MVT—they're two different ways to describe the same thing, which adds to the confusion.

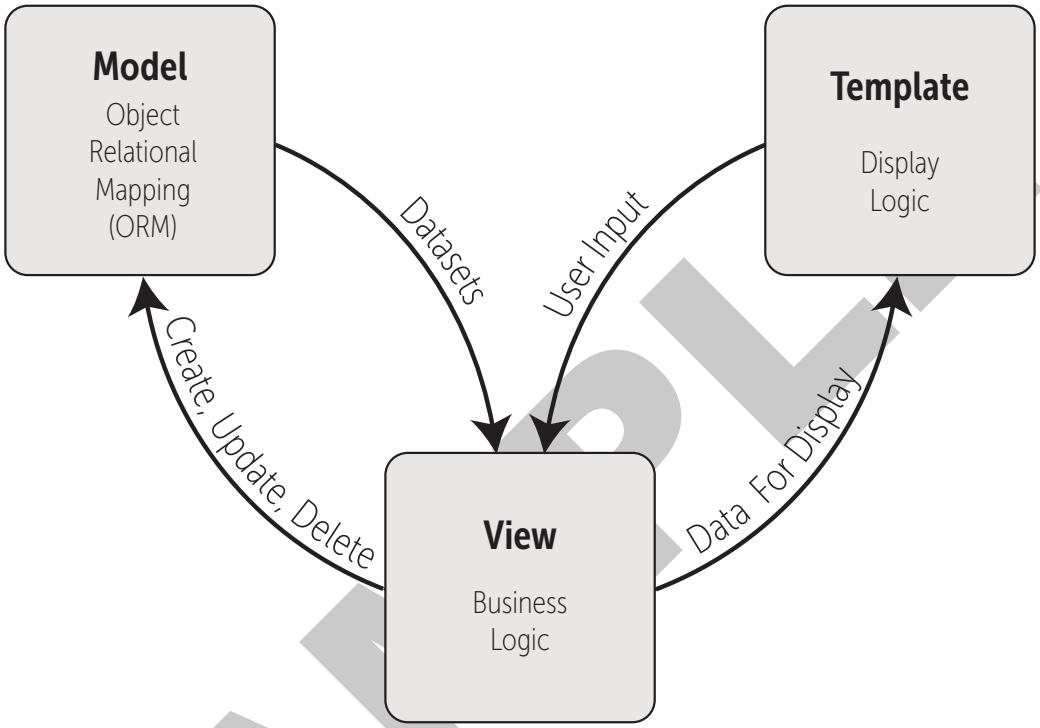

Figure 3-1. *The somewhat misleading Django MTV diagram.*

The misleading part of this diagram is the view. The view in Django is most often described as being equivalent to the controller in MVC, but it's not—it's still the view.

Figure 3-2 is a variation on Figure 3-1 to illustrate my point.

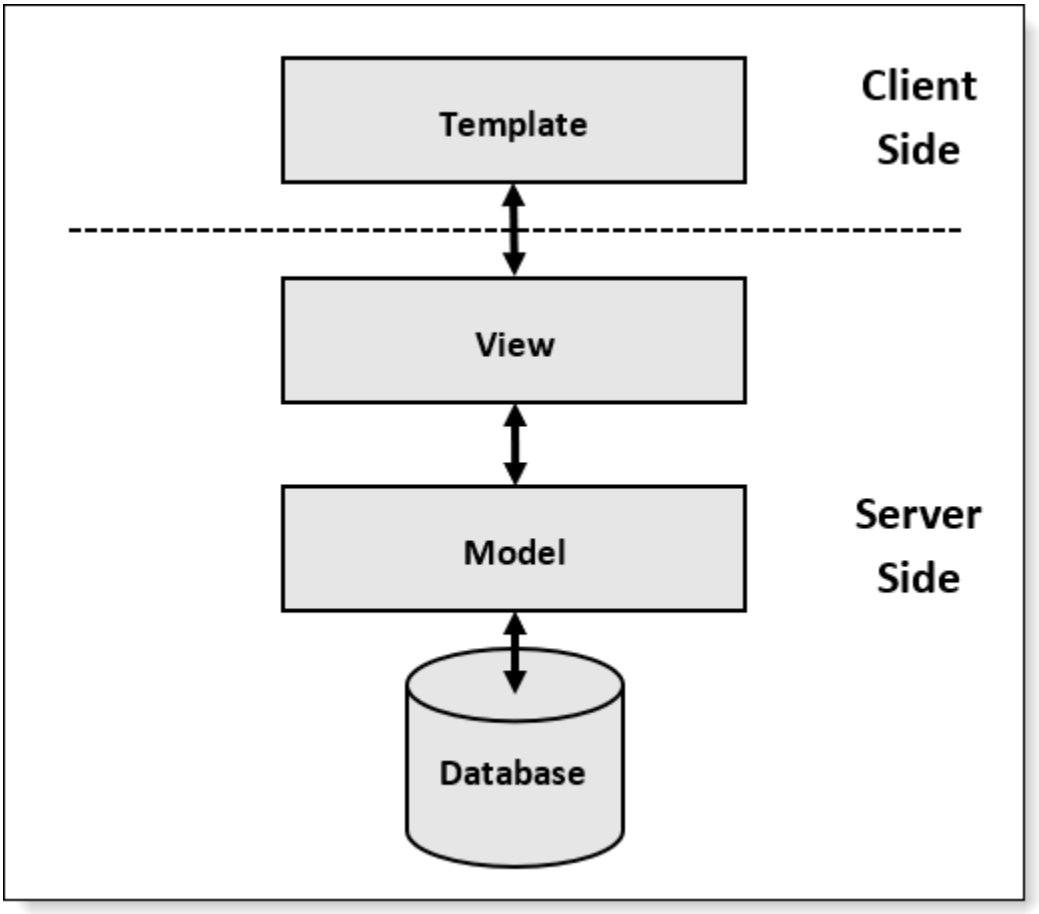

Figure 3-2. *A slightly different view of Django's MTV "stack".*

Note how I have drawn a line between the client- and server-side.
Like all client/server architectures, Django uses request and response objects to communicate between the client and the server. As Django is a web framework, we're talking about `HTTP` request and response objects.

So, in this simplified process, the view retrieves data from the database via the model, formats it, bundles it up in an HTTP response object and sends it to the client (browser).

In other words, the view presents the model to the client as an HTTP response. This is also the *exact* definition of the view in MVC, or to quote Wikipedia (not the most definitive source, I know, but close enough):

"The view means presentation of the model in a particular format"

Trying to bend the definition of a Django view to fit a particular viewpoint inevitably leads to one of two things:

1. Confused programmer puts everything in views module; or
2. Confused programmer says "Django is too hard!", and watches TV instead.

So getting away from our M's and T's and V's and C's, Figure 3-3 presents a more holistic view of Django's architecture.

The first point of confusion we can clear up is where to put a particular function or class:

Does the function/class return a response?

- **YES**—it's a view. Put it in the views module (`views.py`).
- **NO**—it's not a view, it's app logic. Put it somewhere else (`somewhere_else.py`).

We'll discuss the somewhere else part in the next section of this chapter.

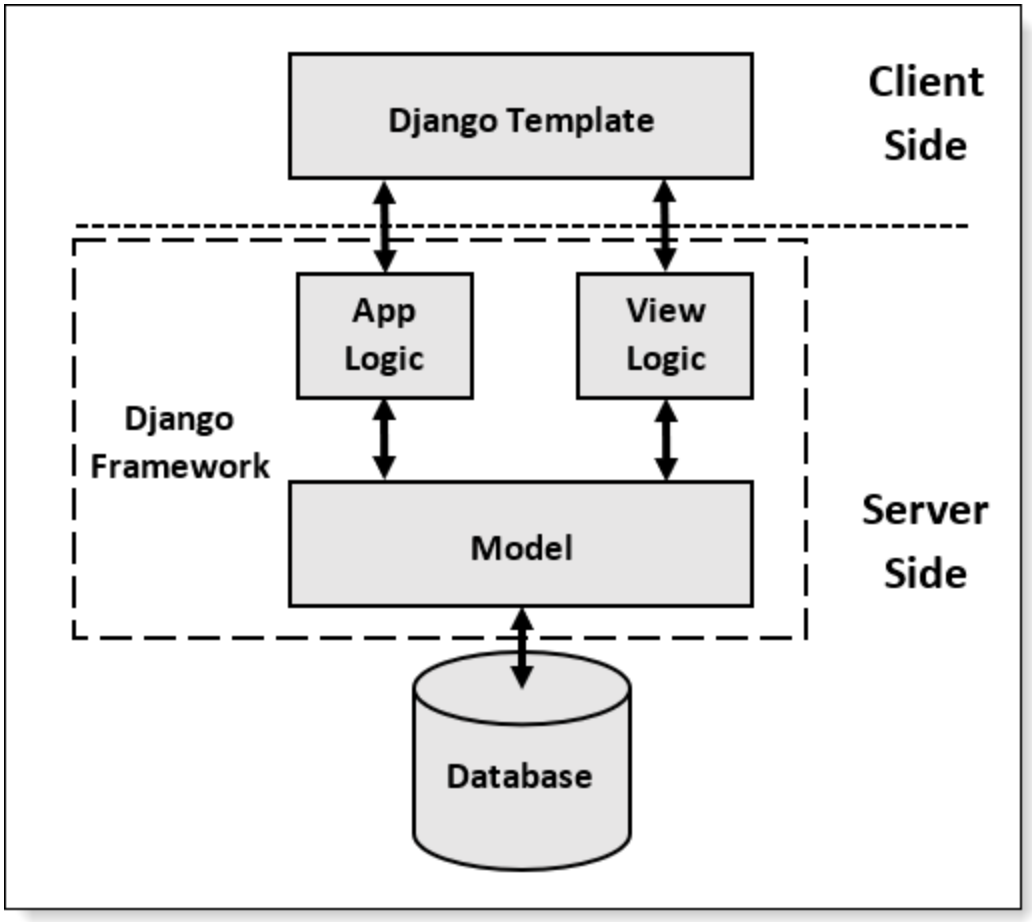

Figure 3-3. *A more holistic view of Django's architecture.*

The next point to note is that the Django framework encapsulates the model, view logic and business logic. In some tutorials, it's said that the Django framework is the controller, but that isn't true either—the Django framework can do much more than respond to user input and interact with data.

A perfect example of this extra power is Django's middleware, which sits between the view and the client-side. Django's middleware performs critical security and authentication checks before sending the response to the browser.

So, returning to the two confused responses from the beginning of the chapter:

1. **Beginners**—no, you don't have to learn about MVC because it's more than likely going to confuse you and lead to more questions than answers.
2. **Programmers**—no, Django is not like Framework X, and trying to think it is, is likely to confuse you and lead to more questions than answers.

Now we've got that out of the way, let's have a look at the structure of a Django project.

Django Project Structure

Django doesn't require you to build web applications in any particular way. In fact, billions of electrons have been sacrificed discussing the One Best Way to structure a Django project. We're all pragmatic programmers here, so we won't play that game.

Django does, however, have a default way of doing things, and there is a definite underlying logic to it you need to understand to become a professional Django programmer.

The fundamental unit of a Django web application is a Django project. A Django project comprises one or more Django apps (Figure 3-4)

A Django app is a self-contained package that should only do one thing. For example, a blog, a membership app or an event calendar. Notice at the bottom of Figure 3-4, there's an extra package called Django Apps.

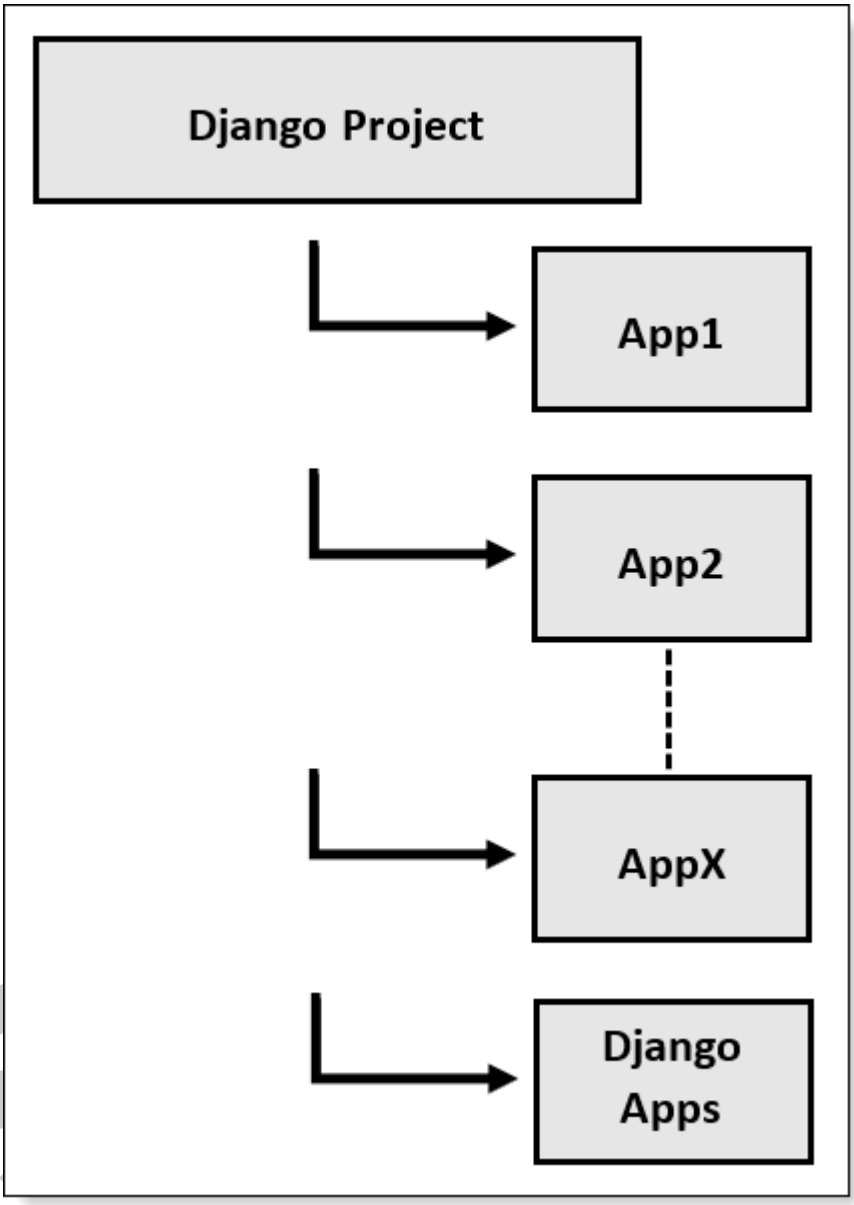

Figure 3-4. *Django's project structure.*

This is another case where Django's logic carries right through the framework—Django itself is a collection of apps, each designed to do one thing. With Django's built-in apps, they're all designed to make your life easier, which is a Good Thing.

While the built-in apps are invisible in your project tree, you can see them in your `settings.py` file:

```
# ...\myclub_project\myclub_site\myclub_site\settings.py

# partial listing

INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
]
```

You can see that Django has added several apps to your project automatically. There are also many other built-in Django apps you can add to the `INSTALLED_APPS` list. When you add your apps to a Django project, you also add a link to the app configuration class to this list.

You can see this logic in what Django has created for you so far. Open up your `\myclub_project` folder. The folder structure should look something like this:

```
# ...\my_clubproject

\env_myclub
\myclub_site        <= This is your Django project
    \myclub_site    <= This is a Django app
    db.sqlite3      <= Your project database
    manage.py       <= Django project management utility
```

Let's examine these files and folders in more detail:

- The **`env_myclub`** folder is where Django stores your virtual environment files. Generally, you should leave everything inside this folder alone.
- The outer **`myclub_site`** folder is your Django project. Django created this folder and its contents when you ran the `startproject` command in the last chapter. Django doesn't care about the folder name, so you can rename it to something meaningful to you.
- Inside the outer `myclub_site` folder are two files:
  - **`db.sqlite3`**. The database created when you ran the `migrate` command; and
  - **`manage.py`**. A command-line utility for executing Django commands from within your project.
- The inner **`myclub_site`** folder is your Django website application. This is the one application that Django creates automatically for you. Because Django is a web framework, it assumes you want a website app.

This should make more sense by now, but I bet there is one thing that's still a bit confusing—the two `myclub_site` folders.

A very common complaint from programmers new to Django is how confusing it is to know which folder they should work in when there are two folders named the same. Django is not alone with this convention—Integrated Development Environments (IDEs) like Visual Studio create a project folder and application folder with the same name. But just because it's common, that doesn't mean it isn't confusing.

As I said a moment ago, Django doesn't care what you name this folder—so let's commit heresy #3, breaking thirteen years of Django tutorial convention while we are at it, and rename the folder!

Here, we're renaming it to "myclub_root".

Once you have made the change, your folder structure should go from this:

```
\myclub_project
    \myclub_site
        \myclub_site
```

To this:

```
\myclub_project
    \myclub_root
        \myclub_site
```

Now we've taken care of that source of confusion, let's have a look inside the `myclub_site` website app Django created for us:

```
# \myclub_project\myclub_root\

\myclub_site
    __init.py__
    asgi.py # Django 3 only
    settings.py
    urls.py
    wsgi.py
```

Looking closer at these files:

- The **`__init__.py`** file tells Python that this folder (your Django app) is a Python package.
- **`asgi.py`** enables ASGI compatible web servers to serve your project (Django 3 only).
- **`settings.py`** contains the settings for your Django project. Every Django project must have a settings file. By convention, Django puts it in your website app, but it doesn't have to live there. There are proponents for other structures as I mentioned earlier, but here we're using the default.

- **`urls.py`** contains project-level URL configurations. By default, this contains a single URL pattern for the admin. We will cover more on URLs later in the chapter, and in great detail in Chapter 5.
- **`wsgi.py`** enables WSGI compatible web servers to serve your project.

Now that we've had a good look at the basic structure of a Django project, it's time to take the next step and add our own Django app.

Creating Your Own Django Apps

You might have noticed that there is no real program code in your project so far. There is a settings file with configuration information, an almost empty URLs file, and a command-line utility that launches a website that doesn't do much.

This is because to create a functioning Django website, you need to create *Django applications*. A Django application (or app for short) is where the work gets done. Apps are one of Django's killer features. Not only do they allow you to add functionality to a Django project without interfering with other parts of the project, but apps are designed to be portable so you can use one app in multiple projects.

So, let's create our first custom Django app. Our social club website needs an events calendar to show upcoming events for the club, so we're creating a Django app called `events`.

Fire up your Python virtual environment, switch into the `\myclub_root` folder and run the command:

```
python manage.py startapp events
```

This is what your command shell output should look like:

```
(env_myclub) ...> cd myclub_root
```

```
(env_myclub) ...\myclub_root> python manage.py startapp events
(env_myclub) ...\myclub_root>
```

Once you have created your app, you must tell Django to install it into your project. This is easy to do—inside your `settings.py` file is a list named `INSTALLED_APPS`. This list contains all the apps installed in your Django project. Django comes with a few apps pre-installed, we just have to add your new `events` app to the list (change in bold):

```
1 INSTALLED_APPS = [
2     'events.apps.EventsConfig',
3     'django.contrib.admin',
4     # more apps
5 ]
```

Inside every app, Django creates a file, `apps.py`, containing a configuration class named after your app. Here, the class is named `EventsConfig`. To register our app with Django, we need to point to the `EventsConfig` class—which is what we are doing in **line 2** of our modified `INSTALLED_APPS` list.

If you were wondering, `EventsConfig` contains a single configuration option by default—the name of the app ("events").

Line Numbering in Code Examples

Throughout the book, I use line numbering to make it easier for you to follow along with the explanations.

As I often use code snippets from your application files, the line numbering in the example is not the same as the line numbering in the actual source code file.

Now let's look inside the `\myclub_root` folder to see what Django has created for us:

```
\events
    \migrations
    __init__.py
    admin.py
    apps.py
    models.py
    tests.py
    views.py
```

- The **`migrations`** folder is where Django stores migrations, or changes to your database. There's nothing in here you need to worry about right now.
- **`__init__.py`** tells Python that your `events` app is a package.
- **`admin.py`** is where you register your app's models with the Django admin application.
- **`apps.py`** is a configuration file common to all Django apps.
- **`models.py`** is the module containing the models for your app.
- **`tests.py`** contains test procedures that run when testing your app.
- **`views.py`** is the module containing the views for your app.

Now we have a complete picture of a Django project, we can also answer the question from earlier in the chapter: "well, if it's not a view, where does it go?"

When you have code that isn't a view, you create a new Python module (`.py` file) inside your app and put *related* functions and classes inside the file. Note the emphasis on related. If you have a bunch of functions that provide database management utilities, for example, put them all in one file. Functions and classes not related to database management should go in another file. You should also try to be descriptive in naming modules—after all, it's more sensible to put your database functions in a file called `db_utils.py` than a file called `monkeys.py`...

When creating new modules for your Django project, you should also consider scope. While adding custom modules to apps is far more common (and more portable), you

can have project-level modules (e.g., Django's `manage.py`) and site-level modules. In the latter case, your custom modules should go in the same folder as your `settings.py` file.

The last couple of points might seem blindingly obvious, but it's important to understand that, while Django has a default logic to its structure, nothing is cast in stone. Django is flexible and allows you to expand and change your project structure to suit the logic of your web application.

Now we have a thorough understanding of the structure of Django's projects and apps, the next obvious question, given we are building web applications is "how do we navigate a Django project?"

To answer this question, we need to check out the final piece of the Django big picture puzzle—URL configurations.

URLconfs—Django's Navigator

There's one last piece to the Django framework puzzle—the critical communication pathway that matches a request on the client-side with a project resource (the arrows between the view and the template in Figure 3-3). Like all web applications, Django uses Uniform Resource Locators (URLs) to match content with a request.

Django's `urls` package provides dozens of functions and classes for working with different URL formats, name resolution, exception handling and other navigational utilities. However, at its most basic, it allows you to map a URL to a function or class within your Django project.

A Django URL configuration (or URLconf for short) matches a unique URL with a project resource. You can think of it being like matching a person's name with their address. Except in Django, we're not matching a street address—we're matching a Python path using Python's *dot notation*.

Assuming you're not familiar with dot notation, it's a common idiom in object-oriented programming. I like to think of the dot like a point because the dot points to something. With Python, the dot operator points to the next object in the object chain.

In Django classes, the object chain is like this:

```
package.module.class.method
```

Or with functions:

```
package.module.function.attribute
```

Some real-life examples:

- `forms.Form` points to the `Form` class in the `forms` package.
- `events.apps.EventsConfig` points to the `EventsConfig` class in the `apps` sub-package of the events package (i.e., the `apps.py` file in your `events` app).
- `django.conf.urls` points to the `urls` package inside the `conf` package inside Django, which is also a Python package!

This can sometimes get a bit confusing, but if you remember to join the dots (sorry, a bad pun there), you can usually find out what the dot operator is referring to.

With a URLconf, the path points to a function or class inside a module (`.py` file). Let's look at our Django project diagram again (Figure 3-5).

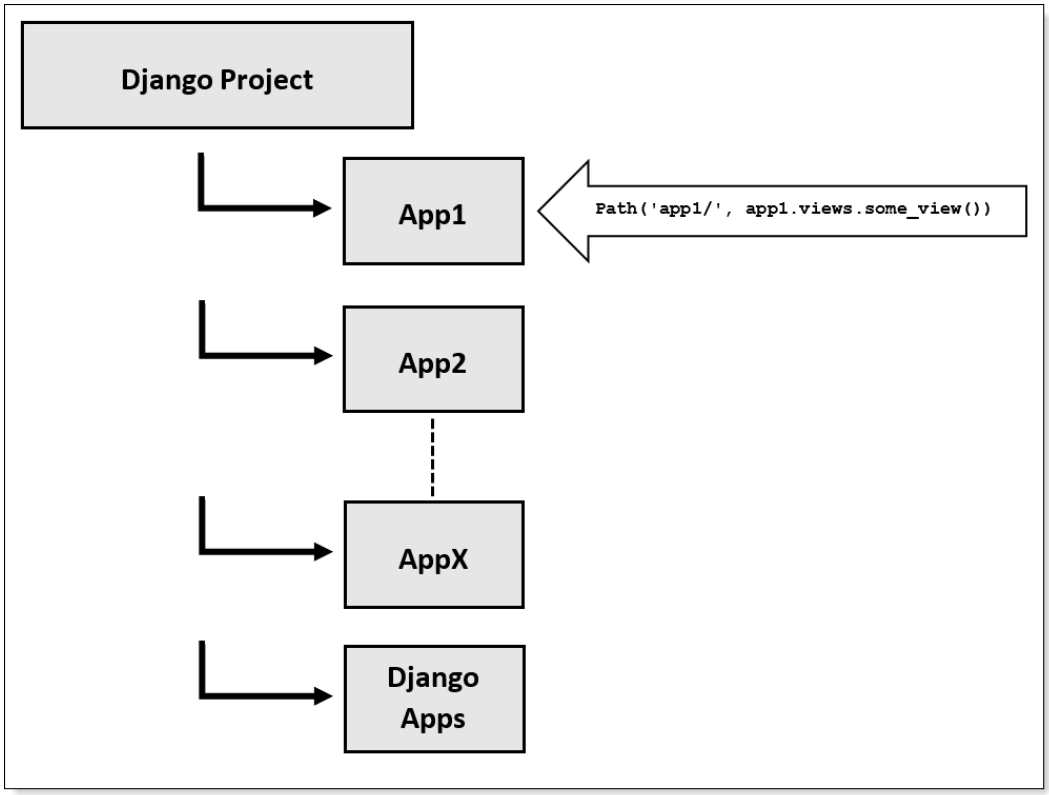

Figure 3-5. *Finding functions and classes with Django's URLconfs.*

To create a URLconf, we use the `path()` function. The first part of the function is the URL, so in Figure 3-5 the URL is `app1/`. The `path()` function then maps this URL to `app1.views.some_view()`.

Assuming your site address is `http://www.mycoolsite.com`, in plain English we're saying:

"When someone navigates to `http://www.mycoolsite.com/app1/`, run the `some_view()` function inside `app1`'s `views.py` file"

Note a URL doesn't have to map to a view—it can map to any module in your Django app. For example, you may have a set of wireless environmental sensors that post data back to the server. You could have a custom module called `sensors.py` that has a function or class to record the sensor data to your database, all without ever touching a view.

And that's all there is to it. Of course, URLconfs can do a lot more than map a static URL to a function or class, but if you can understand the basics—that Django's incredibly fast and powerful navigation system is based on the simple concept of matching a URL with a resource—then you have all you need to tie all your Django apps together into a navigable web project.

A Final Note on Writing Django Apps

A common and inevitable question arises once you get your head around Django's basic structure:

"Where do I start? Should I start with writing my models, the URL configurations, my views? Or what?"

Well, here's your final heresy for the chapter: *it doesn't matter*.

Some people like to start by building all the models so they can see how the data structure looks; others prefer to build the visual layout first, so they start with templates. Others might like to get the basic communication framework in place, so they start with views and URLconfs. Others will start at whatever point seems logical for the project.

Being pragmatic to the bone, I am usually in the last group. I try not to get fixated on what someone else thinks is the right or the wrong way to do things and try to find the simplest and quickest way to achieve the result I want. I also like to work incrementally starting small getting the flow right and building on it to create the

complete application. This approach means I inevitably end up jumping from one element to another as the application grows.

Your brain is wired differently to mine and every other programmer. This is a Good Thing. Just remember, an imperfect start to a project is *way* better than not starting at all. Do what works for you.

Chapter Summary

In this chapter, I gave you a high-level overview of how Django projects are structured, how each component works with other parts of Django to create a web application, and how to create a Django app.

In the next chapter, we will start diving into the inner working of Django's core modules by exploring the fundamentals of Django's models.

4

Django's Models

Unless you are creating a simple website, there is little chance of avoiding the need to interact with some form of database when building modern web applications.

Unfortunately, this usually means you have to get your hands dirty with Structured Query Language (SQL)—which is just about nobody's idea of fun. In Django, the messy issues with SQL is a solved problem: you don't have to use SQL at all unless you want to. Instead, you use a Django *model* to access the database.

Django's models provide an *Object-relational Mapping* (ORM) to the underlying database. ORM is a powerful programming technique that makes working with data and relational databases much easier.

Most common databases are programmed with some form of SQL, but each database implements SQL in its own way. SQL can also be complicated and difficult to learn. An ORM tool simplifies database programming by providing a simple mapping between an object (the 'O' in ORM) and the underlying database. This means the programmer need not know the database structure, nor does it require complex SQL to manipulate and retrieve data (Figure 4-1).

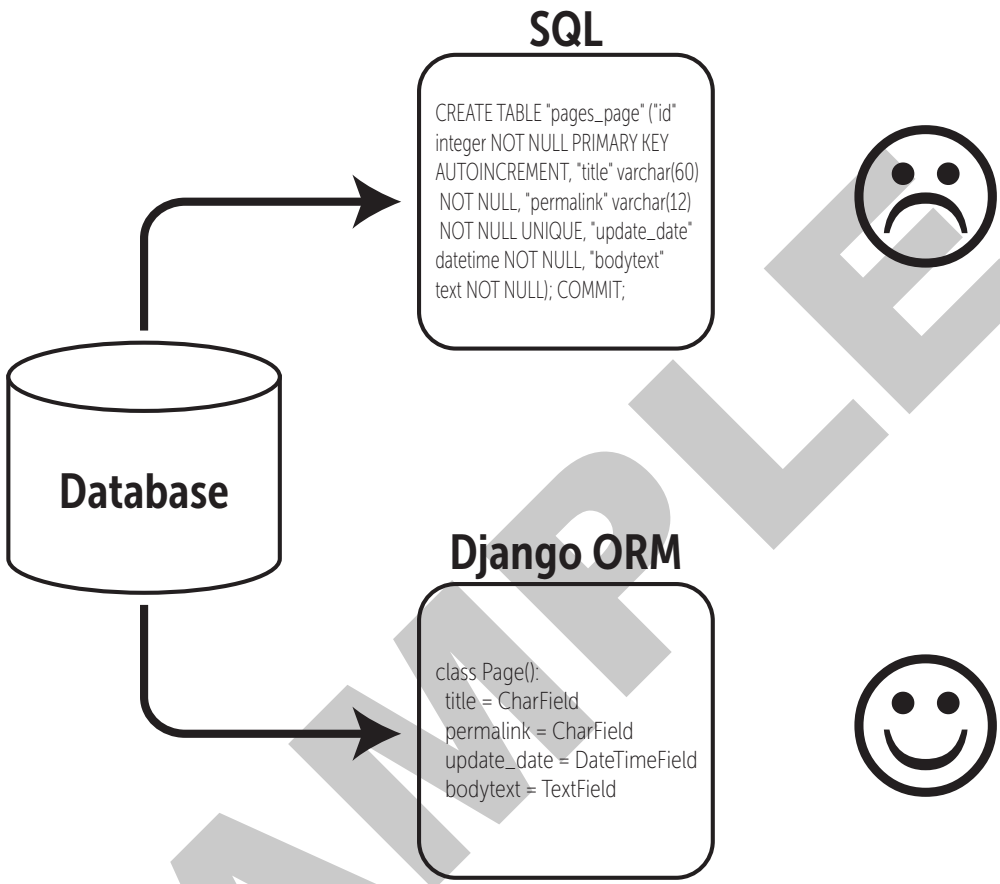

Figure 4-1. *An ORM allows for simple manipulation of data without having to write complex SQL.*

In Django, the model is the object mapped to the database. When you create a model, Django executes SQL to create a corresponding table in the database (Figure 4-2) without you having to write a single line of SQL. Django prefixes the table name with the name of your Django application. The model also links related information in the database.

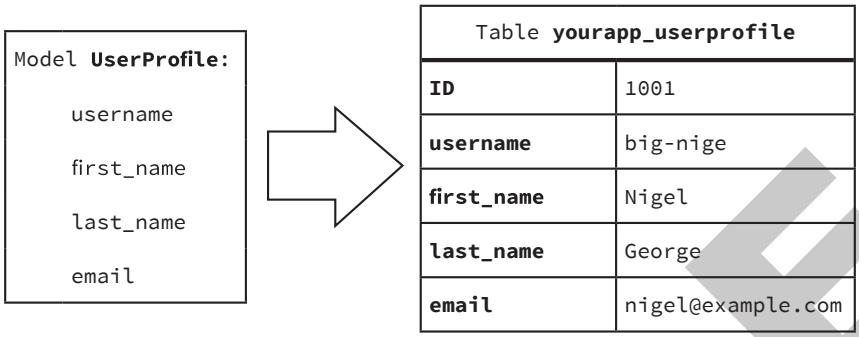

Figure 4-2. *Creating a Django model creates a corresponding table in the database.*

In Figure 4-3, a second model keeps track of a user's course enrollments. Repeating all the user's information in the `yourapp_Course` table would be against sound design principles, so we instead create a relationship (the 'R' in ORM) between the `yourapp_Course` table and the `yourapp_UserProfile` table.

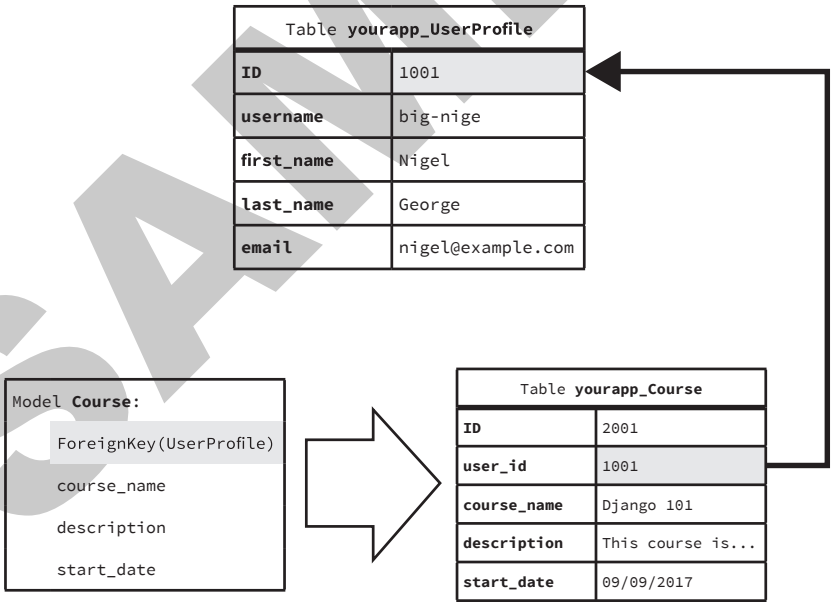

Figure 4-3. *Foreign key links in Django models create relationships between tables.*

This relationship is created by linking the models with a foreign key—i.e., the `user_id` field in the `yourapp_Course` table is a key field linked to the `id` field in the foreign table `yourapp_UserProfile`.

This is a simplification, but is a handy overview of how Django's ORM uses the model data to create database tables. We will dig much deeper into models shortly, so don't worry if you don't 100% understand what is going on right now. Things become clearer once you have had the chance to build actual models.

Supported Databases

Django officially supports five databases:

- PostgreSQL
- MySQL
- SQLite
- Oracle
- MariaDB (Django 3 only)

There are also several third-party applications available if you need to connect to an unofficially supported database.

The preference for most Django developers, myself included, is PostgreSQL. MySQL is also a common database back end for Django. Installing and configuring a database is not a task for a beginner. Luckily, Django installs and configures SQLite automatically, with no input from you, so we will use SQLite in this book.

I cover running your project with PostgreSQL, MySQL and MariaDB in Chapter 16.

End of sample content.

To purchase the complete book, go to https://djangobook.com/mastering-django-2-book/.