
# By Nigel George

# 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 (**David used to work with Adrian and Simon at the Lawrence Journal-World newspaper. He was there was Django was born!**).

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.

## Before We Install Django

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.

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 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 set up Django when you are first starting out.

# Installing Python

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

## Python Versions

Django 1.11 LTS works with Python version 2.7, 3.4, 3.5 and 3.6. For each version of Python, only the latest micro release (A.B.C) is officially supported, although from experience, any recent release works fine.

---

### All of the code samples in this book are written for Python 3.

I provide no code examples written in Python 2. Not only does it add confusion for beginners, but Django 1.11 LTS will be the last version of Django to support Python 2.

You might hear a differing opinion from a few rusted-on individuals, but unless you have a very good reason to use Python 2 (e.g. legacy libraries), Python 3 is the way to go.

## Installation

If you're on Linux or Mac OS X, you probably have Python already installed. Type `python` into your terminal and if you see something like the following, then Python is installed:

```
Python 2.7.10 (default, Feb 6 2017, 23:53:20)
[GCC 4.2.1 Compatible Apple LLVM 8.0.0 (clang-800.0.34)]
on darwin
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.

**If Python3 is is not installed, install it!**

Assuming Python is not installed in your system, 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 3.6.1, but it may have been updated by the time you read this, so the numbers may be slightly different.

As I stated before, **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 there 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 – when the Python installer screen opens, make sure "Add Python 3.6 to PATH" is checked before installing (Figure 1-1).

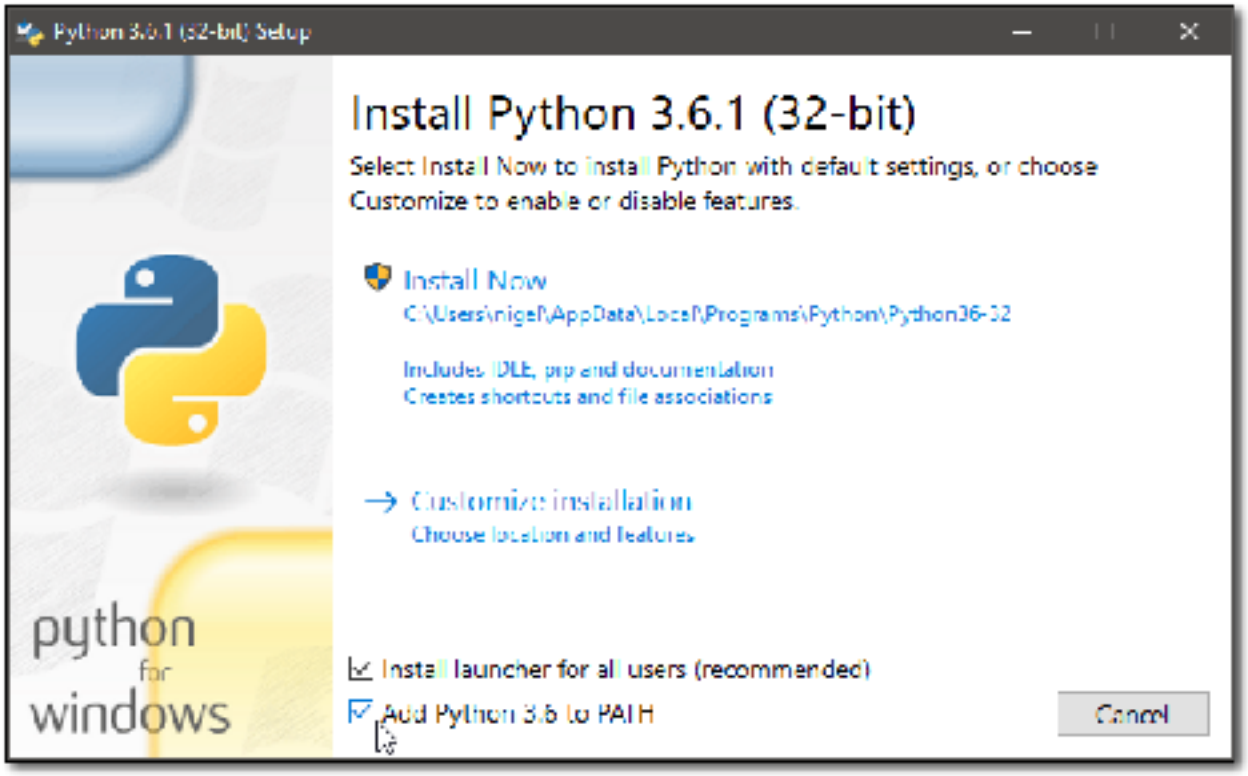


**Figure 1-1:** Check the 'Add Python 3.6 to PATH' box before installing.

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 3.6.0 (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 may have to restart Windows to get this to work.

# Installing a PIP

While you are at it, there is one more important thing to do. Exit out of Python, then enter the following at the command prompt:

```
python –m pip install –U pip
```
This command will either print a message saying `pip` is up to date, or give an output 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`, 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's 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 online 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 (your version numbers may be different):

```
C:\Users\nigel>pip install virtualenv
Collecting virtualenv
Downloading virtualenv-15.1.0-py2.py3-none-any.whl (1.8MB)
    100% |################################| 1.8MB 323kB/s
Installing collected packages: virtualenv
Successfully installed virtualenv-15.1.0
```

## Creating a Project Directory

Before we create our Django project, we first need to create a project directory. The project directory can go anywhere on your computer, although it's highly recommended that it be created somewhere in your user directory so you don't get any permission issues later on. A good place for your folder in Windows is your `My Documents` folder.

Create a new folder on your system, I have called the folder `mysite_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 OS X). The easiest way to do this on 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 1-2).

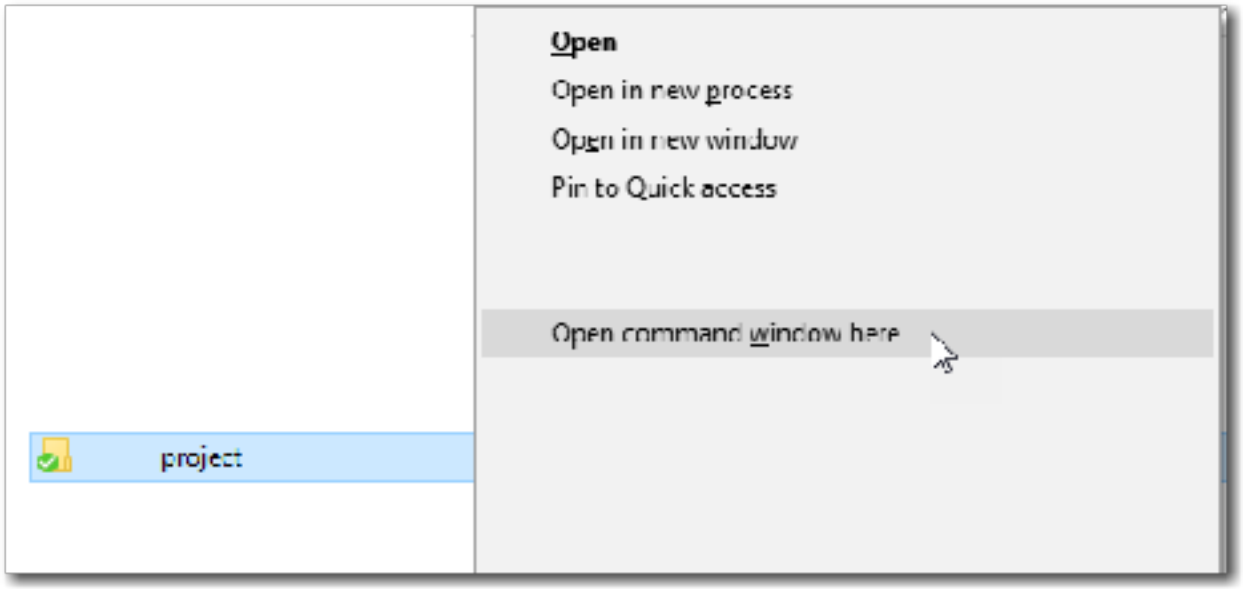


**Figure 1-2:** Hold the shift key and right-click a folder to open a command window.

If you are running newer versions of Windows 10, the old command prompt has been replaced by PowerShell. For the examples in this book, the old terminal and PowerShell are functionally the same and all commands will run in PowerShell unmodified.
Once you have created your project folder, you need to create a virtual environment for your project by typing `virtualenv env_mysite` at the command prompt you just opened:

```
 C:\Users\nigel\OneDrive\Documents\mysite_project> virtualenv
env_mysite
```

Again, the name of the virtual environment is not important, you can change the name to suit. On my system, the output from this command looks something like this:

```
Using base prefix 'c:\\users\\nigel\\appdata\\local\\programs\
\python\\python36-32'
New python executable in C:
\Users\nigel\OneDrive\Documents\mysite_project\env_mysite\Script
s\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 project 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
pip-selfcheck.json
```

If you look inside the `\Lib` folder, you will see `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.

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 3.6. Environments named "`env\_someapp\_python27`" and "`env\_someapp\_python36`" are going to be a lot easier to distinguish than if you had named them "`env`" and "`env1`".
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\OneDrive\Documents\mysite_project>
```

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==1.11.2
```

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

```
(env_mysite) C:\Users\nigel\OneDrive\Documents\mysite_project>
pip install django==1.11.2

Collecting django==1.11.2
Using cached Django-1.11.2-py2.py3-none-any.whl
Collecting pytz (from django==1.11)
Using cached pytz-2017.2-py2.py3-none-any.whl
Installing collected packages: pytz, django
Successfully installed django-1.11.2 pytz-2017.2
```

In this case, we are explicitly telling `pip` to install Django 1.11.2, which is the latest version of Django 1.11 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 1.11 LTS.

Also note my computer didn't need to download anything as I have another virtual environment I installed today and Windows used the cached version.

And finally, note that Django 1.11 requires the Python Timezone package (`pytz`), so `pip` installs that in your virtual environment as well.

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\OneDrive\Documents\mysite_project>
python

Python 3.6.1 (v3.6.1:69c0db5, Mar 21 2017, 17:54:52) [MSC v.1900
32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more
information.
>>> import django
>>> django.get_version()
'1.11'
>>>
```

## Starting a Project

Once you've installed Python, 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. The auto-generated code contains 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\`. Also make sure you are in the `mysite_project` directory.

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

```
django-admin startproject mysite
```

This command will automatically create a `mysite` directory in your project directory as well as all the necessary files for a basic, but fully functioning Django website.

**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.
- 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 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.

## Setting Up a Database

Django includes a number of 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 a database before we can use them. To do that, change into the `mysite` folder created in the last step (type `cd mysite` 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 in the book). If all goes to plan, you'll see a message for each migration it applies:

```
(env_mysite) C:
\Users\Nigel\OneDrive\Documents\mysite_project\mysite>python
manage.py migrate

Operations to perform:
Apply all migrations: admin, auth, contenttypes, sessions
Running migrations:
Applying contenttypes.0001_initial... OK
```

```
Applying auth.0001_initial... OK
Applying admin.0001_initial... OK
### several more migrations (not shown)
```
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 be using SQLite throughout this book. If you would like to work with a "large" database engine like PostgreSQL, MySQL, or Oracle, see Chapter 21.

## 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...

System check identified no issues (0 silenced).
May 16, 2017 - 16:48:29
Django version 1.11, 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. Django's creators 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!

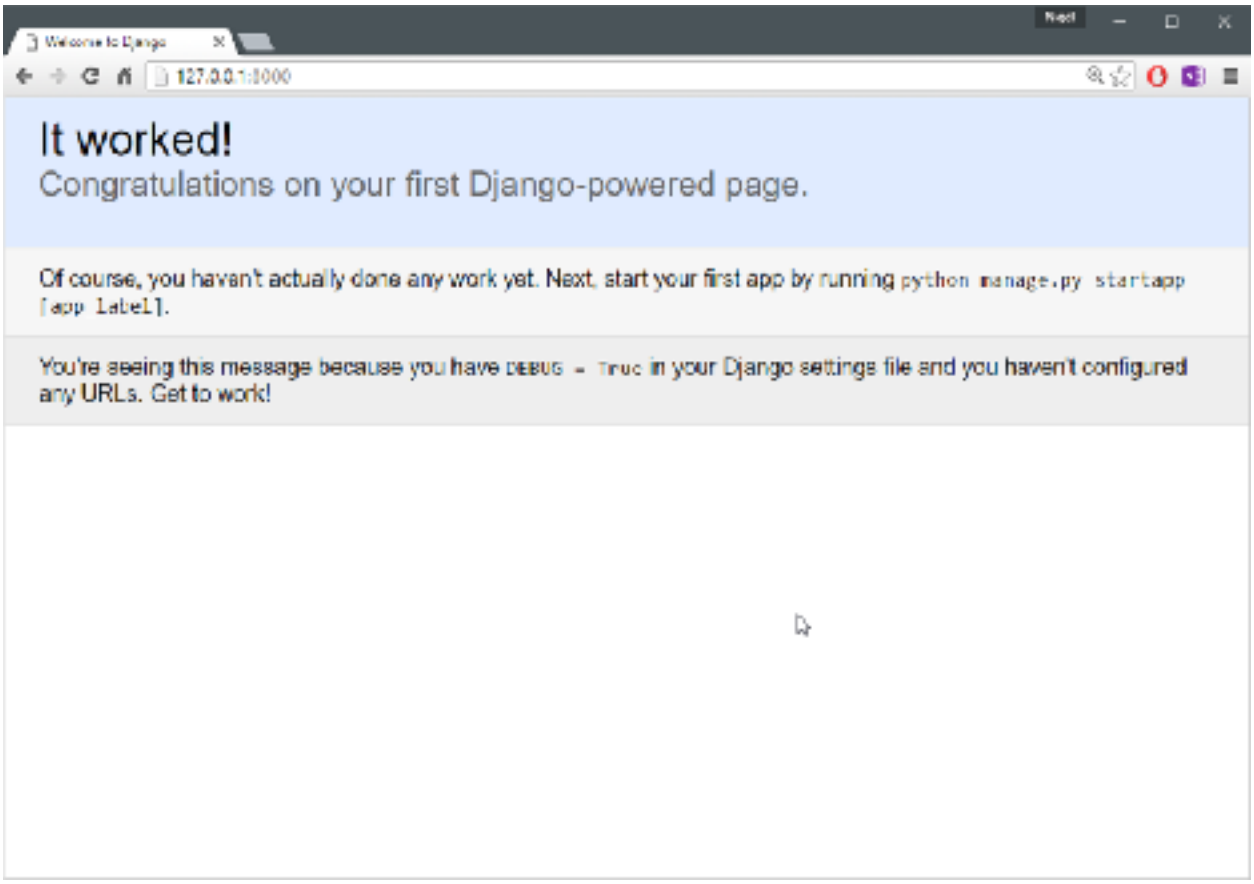


**Figure 1-3:** Django's welcome page

## 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 Concept (MVC)

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…

## The Model-View-Template Concept (MVP)

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.

# Django 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.

## Views

Within the inner `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.

Make sure you put the file in the inner `mysite` directory, i.e. the `\mysite\mysite\` directory, not the directory containing `manage.py`.

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.)

# URLS

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/1.11/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. Import the include() function: from django.conf.urls
import url, include
    2. Add a URL to urlpatterns:  url(r'^blog/',
include('blog.urls'))
"""
from django.conf.urls import url
from django.contrib import admin

urlpatterns = [
    url(r'^admin/', 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 url
from django.contrib import admin

urlpatterns = [
    url(r'^admin/', 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 url
from django.contrib import admin

from mysite.views import hello
```

```
urlpatterns = [
    url(r'^admin/', 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` from within your Python virtual environment. (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).

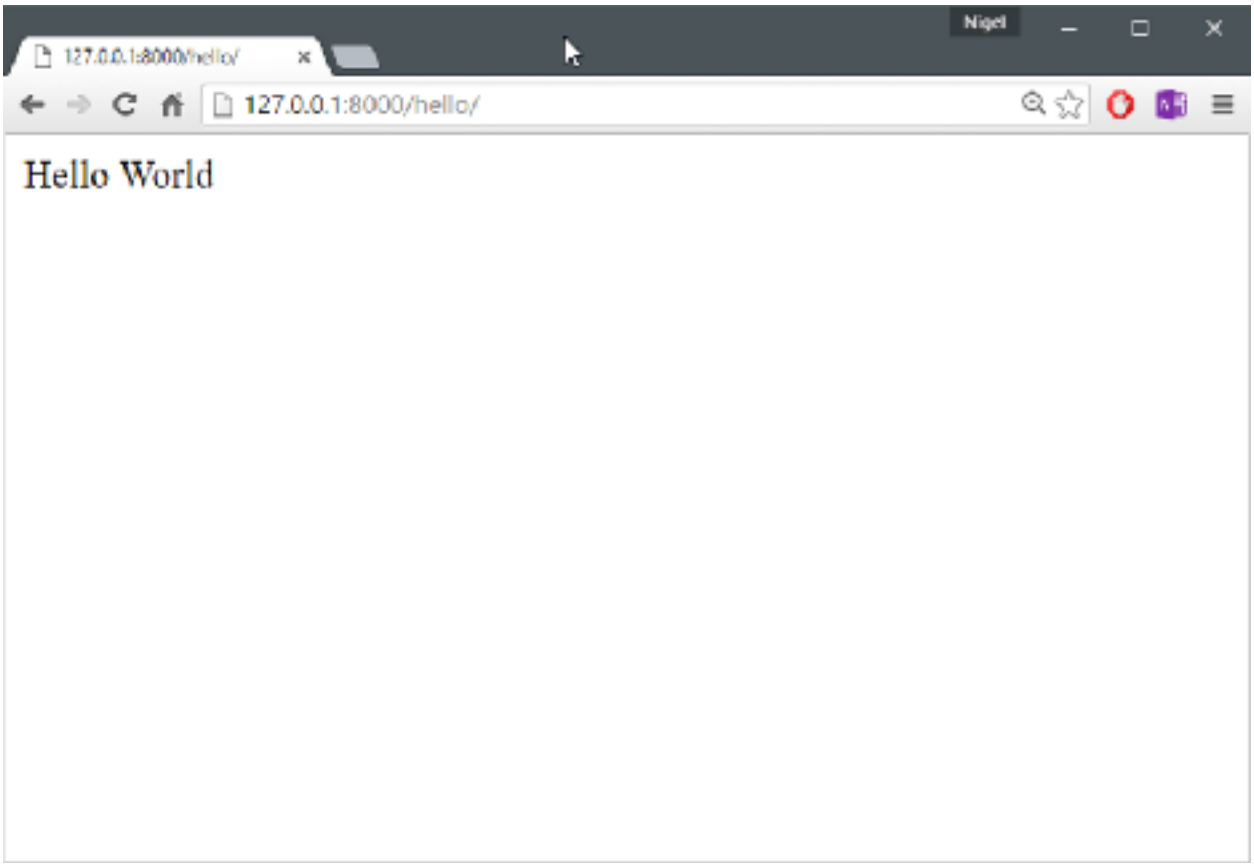


**Figure 2-1:** Hooray! Your first Django view.

## 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.

## 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.

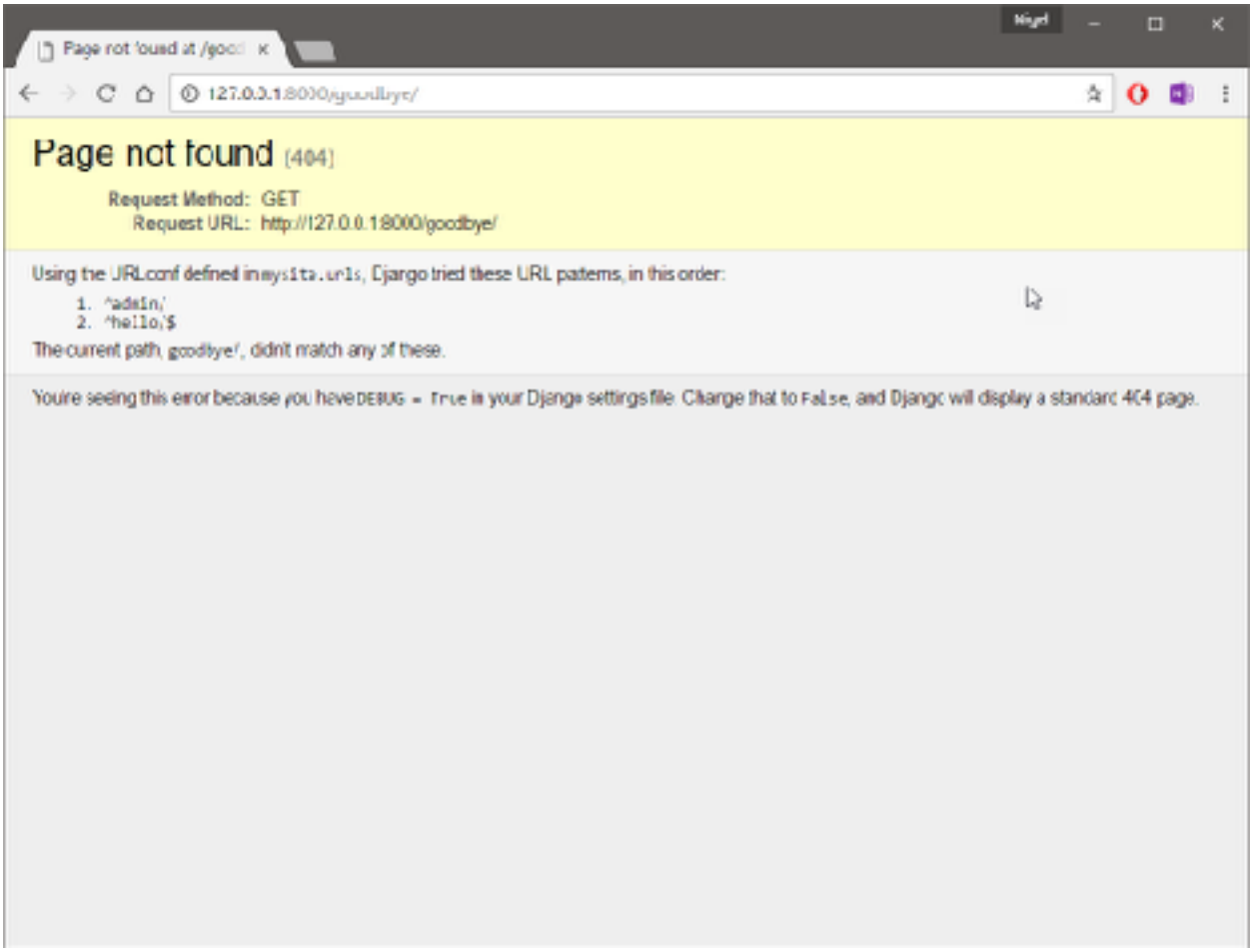


**Figure 2-2:** Django's 404 page

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

Django doesn't magically add anything to the site root; that URL is not special-cased in any way. This means, 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/`.

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: `TEMPLATES`, `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.