By **Leif Azzopardi** and **David Maxwell**

Python web development with Django

Tango with Django

A beginner's guide to web development with **Django 1.9**.

Also compatible with **Django 1.10**

REVISED AND UPDATED FOR django version 1.9

Available from **www.tangowithdjango.com**

How to Tango with Django 1.9

A beginners guide to Python/Django

Leif Azzopardi and David Maxwell

This book is for sale at http://leanpub.com/tangowithdjango19

This version was published on 2017-02-28

Tweet This Book!

Please help Leif Azzopardi and David Maxwell by spreading the word about this book on Twitter!

The suggested tweet for this book is:

I'm now ready to Tango with Django @tangowithdjango

The suggested hashtag for this book is #tangowithdjango.

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

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

Contents

1. Overview

The aim of this book is to provide you with a practical guide to web development using *Django* and *Python*. The book is designed primarily for students, providing a walkthrough of the steps involved in getting a web application up and running with Django.

This book seeks to complement the official Django Tutorials and many of the other excellent tutorials available online. By putting everything together in one place, this book fills in many of the gaps in the official Django documentation providing an example-based design driven approach to learning the Django framework. Furthermore, this book provides an introduction to many of the aspects required to master web application development (e.g. HTML, CSS, JavaScript, etc.).

1.1 Why Work with this Book?

This book will save you time. On many occasions we've seen clever students get stuck, spending hours trying to fight with Django and other aspects of web development. More often than not, the problem was usually because a key piece of information was not provided, or something was not made clear. While the occasional blip might set you back 10-15 minutes, sometimes they can take hours to resolve. We've tried to remove as many of these hurdles as possible. This will mean you can get on with developing your application instead of stumbling along.

This book will lower the learning curve. Web application frameworks can save you a lot of hassle and lot of time. Well, that is if you know how to use them in the first place! Often the learning curve is steep. This book tries to get you going - and going fast by explaining how all the pieces fit together.

This book will improve your workflow. Using web application frameworks requires you to pick up and run with a particular design pattern - so you only have to fill in certain pieces in certain places. After working with many students, we heard lots of complaints about using web application frameworks - specifically about how they take control away from them (i.e. inversion of control). To help you, we've created a number of *workflows* to focus your development process so that you can regain that sense of control and build your web application in a disciplined manner.

This book is not designed to be read. Whatever you do, *do not read this book!* It is a hands-on guide to building web applications in Django. Reading is not doing. To increase the value you gain from this experience, go through and develop the application. When you code up the application, *do not just cut and paste the code.* Type it in, think about what it does, then read the explanations we have provided to describe what is going on. If you still do not understand, then check out the Django documentation, go to Stack Overflow or other helpful websites and fill in this gap in your knowledge. If you are really stuck, get in touch with us, so that we can improve this resource - we've already had contributions from numerous other readers!

1.2 What you will Learn

In this book, we will be taking an exampled-based approach. The book will show you how to design a web application called *Rango* (see the Design Brief below). Along the way, we'll show you how to perform the following key tasks.

- **How to setup your development environment** - including how to use the terminal, your virtual environment, the `pip` installer, how to work with Git, and more.
- **Setup a Django project** and create a basic Django application.
- **Configure the Django project** to serve static media and other media files.
- Work with Django's *Model-View-Template* design pattern.
- **Create database models** and use the *object relational mapping (ORM)* functionality provided by Django.
- **Create forms** that can utilise your database models to create dynamically generated web-pages.
- Use the **user authentication** services provided by Django.
- Incorporate **external services** into your Django application.
- Include *Cascading Styling Sheets (CSS)* and *JavaScript* within a web application.
- **Apply CSS** to give your application a professional look and feel.
- Work with **cookies and sessions** with Django.
- Include more advanced functionality like *AJAX* into your application.
- **Deploy your application** to a web server using *PythonAnywhere.*

At the end of each chapter, we have included a number of exercises designed to push you harder and to see if you can apply what you have learned. The later chapters of the book provide a number of open development exercises along with coded solutions and explanations.

Exercises will be clearly delineated like this!

In each chapter we have added a number of exercises to test your knowledge and skill.

You will need to complete these exercises as the subsequent chapters are dependent on them.

Don't worry if you get stuck, though, as you can always check out our solutions to all the exercises on our *GitHub* repository.

1.3 Technologies and Services

Through the course of this book, we will used various technologies and external services including:

- the Python programming language;
- the Pip package manager;
- Django;
- the Git version control system;
- GitHub;
- HTML;
- CSS;
- the JavaScript programming language;
- the JQuery library;
- the Twitter Bootstrap framework;
- the Webhose API (referred to as the *search API*); and
- the PythonAnywhere hosting service;

We've selected these technologies and services as they are either fundamental to web development, and/or enable us to provide examples on how to integrate your web application with CSS toolkits like *Twitter Bootstrap*, external services like those provided by the *Webhose API* and deploy your application quickly and easily with *PythonAnywhere*.

1.4 Rango: Initial Design and Specification

The focus of this book will be to develop an application called *Rango*. As we develop this application, it will cover the core components that need to be developed when building any web application. To see a fully functional version of the application, you can visit the How to Tango with Django website.

Design Brief

Your client would like you to create a website called *Rango* that lets users browse through user-defined categories to access various web pages. In Spanish, the word rango is used to mean *"a league ranked by quality"* or *"a position in a social hierarchy"*.

- For the **main page** of the Rango website, your client would like visitors to be able to see:
  - the *five most viewed pages*;
  - the *five most viewed (or rango'ed) categories*; and
  - *some way for visitors to browse or search* through categories.
- When a user views a **category page**, your client would like Rango to display:
  - the *category name, the number of visits, the number of likes*, along with the list of associated pages in that category (showing the page's title, and linking to its URL); and
  - *some search functionality (via the search API)* to find other pages that can be linked to this category.
- For a particular category, the client would like: the *name of the category to be recorded*; the *number of times each category page has been visited*; and how many users have *clicked a "like" button* (i.e. the page gets rango'ed, and voted up the social hierarchy).
- *Each category should be accessible via a readable URL* - for example, `/rango/books-about-django/`.
- Only *registered users will be able to search and add pages to categories*. Visitors to the site should therefore be able to register for an account.

At first glance, the specified application to develop seems reasonably straightforward. In essence, it is just a list of categories that link to pages. However, there are a number of complexities and challenges that need to be addressed. First, let's try and build up a better picture of what needs to be developed by laying down some high-level designs.

Exercises

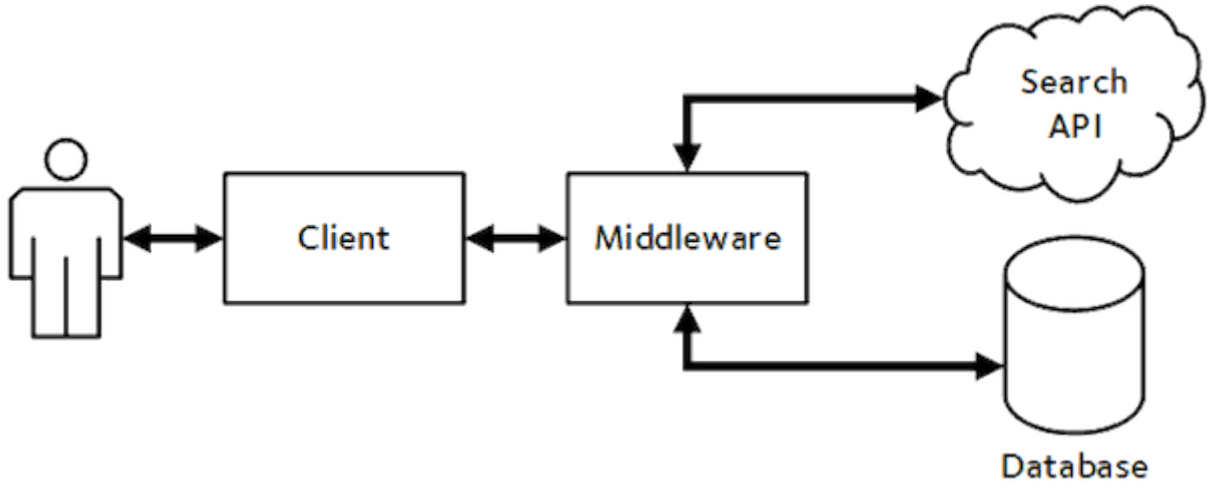
Before going any further, think about these specifications and draw up the following design artefacts.

- An **N-Tier or System Architecture** diagram.
- **Wireframes** of the main and category pages.
- A series of **URL mappings** for the application.
- An ***Entity-Relationship (ER)*** diagram to describe the data model that we'll be implementing.

Try these exercises out before moving on - even if you aren't familiar with system architecture diagrams, wireframes or ER diagrams, how would you explain and describe what you are going to build.

N-Tier Architecture

The high-level architecture for most web applications is a *3-Tier architecture.* Rango will be a variant on this architecture as it interfaces with an external service.



Overview of the 3-tier system architecture for our Rango application.

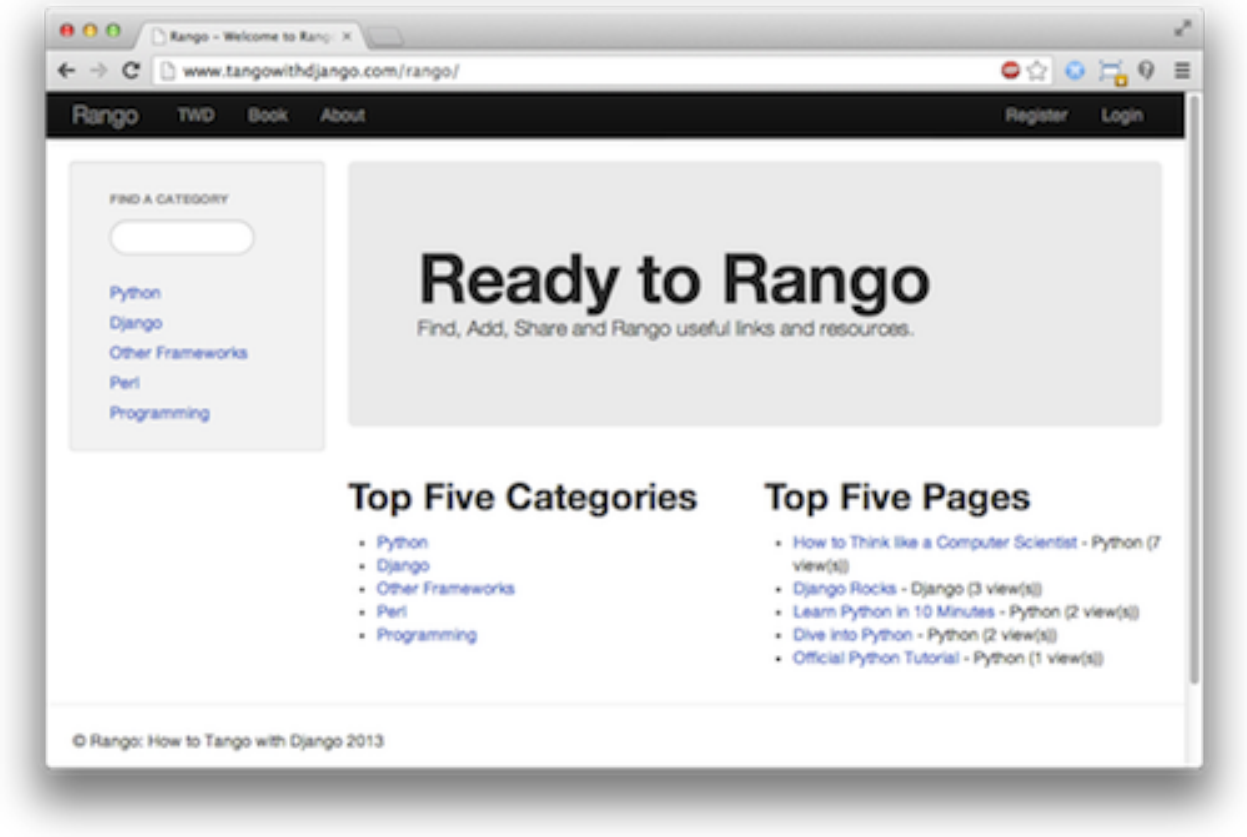
Since we are building a web application with Django, we will use the following technologies for the following tiers.

- The **client** will be a Web browser (such as *Chrome*, *Firefox*, and *Safari*) which will render HTML/CSS pages.
- The **middleware** will be a *Django* application, and will be dispatched through Django's built-in development Web server while we develop.
- The **database** will be the Python-based *SQLite3* Database engine.
- The **search API** will be the search API.

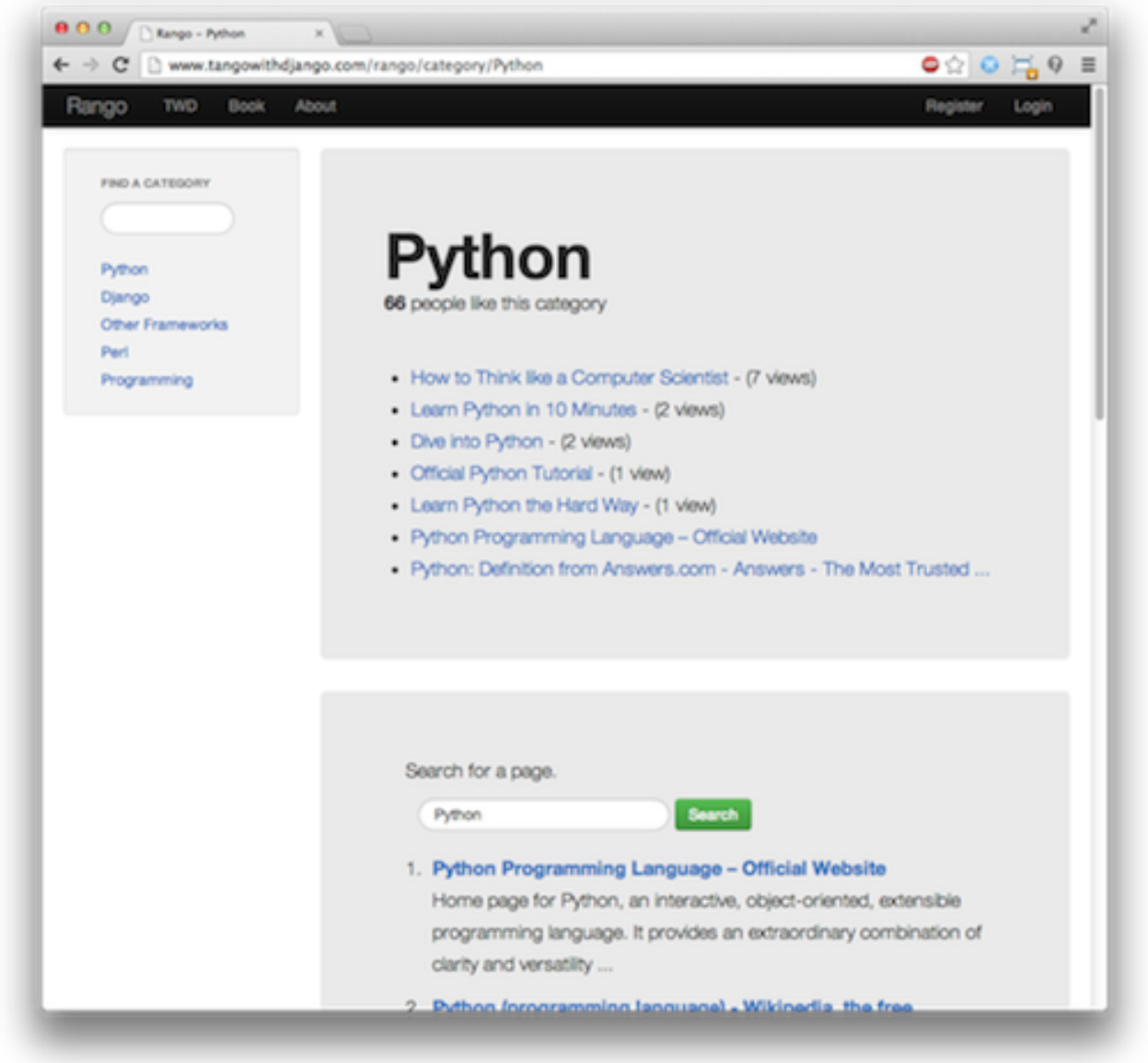
For the most part, this book will focus on developing the middleware. It should however be quite evident from the system architecture diagram that we will have to interface with all the other components.

Wireframes

Wireframes are great way to provide clients with some idea of what the application should look like when complete. They save a lot of time, and can vary from hand drawn sketches to exact mockups depending on the tools that you have at your disposal. For our Rango application, we'd like to make the index page of the site look like the screenshot below. Our category page is also shown below.



The index page with a categories search bar on the left, also showing the top five pages and top five categories.



The category page showing the pages in the category (along with the number of views). Below, a search for *Python* has been conducted, with the results shown underneath.

Pages and URL Mappings

From the specification, we have already identified two pages that our application will present to the user at different points in time. To access each page we will need to describe URL mappings. Think of a URL mapping as the text a user will have to enter into a browser's address bar to reach the given page. The basic URL mappings for Rango are shown below.

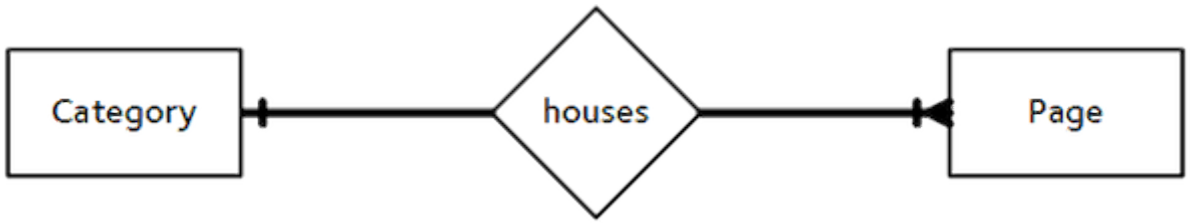
- `/` or `/rango/` will point to the main / index page.

- `/rango/about/` will point to the about page.
- `/rango/category/<category_name>/` will point to the category page for `<category_name>`, where the category might be:
  - games;
  - python-recipes; or
  - code-and-compilers.

As we build our application, we will probably need to create other URL mappings. However, the ones listed above will get us started and give us an idea of the different pages. Also, as we progress through the book, we will flesh out how to construct these pages using the Django framework and use its Model-View-Template design pattern. However, now that we have a gist of the URL mappings and what the pages are going to look like, we need to define the data model that will house the data for our Web application.

Entity-Relationship Diagram

Given the specification, it should be clear that we have at least two entities: a *category* and a *page*. It should also be clear that a *category* can house many *pages*. We can formulate the following ER Diagram to describe this simple data model.



The Entity Relationship Diagram of Rango's two main entities.

Note that this specification is rather vague. A single page could in theory exist in one or more categories. Working with this assumption, we could model the relationship between categories and pages as a many-to-many relationship. This approach however introduces a number of complexities, so we will make the simplifying assumption that *one category contains many pages, but one page is assigned to one category.* This does not preclude that the same page can be assigned to different categories - but the page would have to be entered twice, which is not ideal.

Take Note!

Get into the habit of noting down any working assumptions that you make, just like the one-to-many relationship assumption that we assume above. You never know when they may come back to bite you later on! By noting them down, this means you can communicate it with your development team and make sure that the assumption is sensible and that they are happy to proceed under such an assumption.

With this assumption, we then produce a series of tables that describe each entity in more detail. The tables contain information on what fields are contained within each entity. We use Django `ModelField` types to define the type of each field (i.e. `IntegerField`, `CharField`, `URLField` or `ForeignKey`). Note that in Django *primary keys* are implicit such that Django adds an `id` to each Model, but we will talk more about that later in the Models and Database chapter.

Category Model

Field	Type
`name`	`CharField`
`views`	`IntegerField`
`likes`	`IntegerField`

Page Model

Field	Type
`category`	`ForeignKey`
`title`	`CharField`
`url`	`URLField`
`views`	`IntegerField`

We will also have a model for the `User` so that they can register and login. We have not shown it here, but shall introduce it later in the book when we discuss User Authentication. In the following chapters, will we see how to instantiate these models in Django and how to use the built-in ORM to connect to the database.

1.5 Summary

These high level design and specifications will serve as a useful reference point when building our Web application. While we will be focusing on using specific technologies, these steps are common to most database driven websites. It's a good idea to become familiar with reading and producing such specifications and designs so that you can communicate your designs and ideas with others. Here we will be focusing on using Django and the related technologies to implement this specification.

Cut and Paste Coding

As you progress through the tutorial, you'll most likely be tempted to cut and paste the code from the book to your code editor. **However, it is better to type in the code**. We know that this is a hassle, but it will help you to remember the process better and the commands that you will be using later on.

Furthermore, cutting and pasting Python code is asking for trouble. Whitespace can end up being interpreted as spaces, tabs or a mixture of spaces and tabs. This will lead to all sorts of weird errors, and not necessarily indent errors. If you do cut and paste code be wary of this. Pay particular attention to this if you're using Python 3 - inconsistent use of tabs and spaces in your code's indentation will lead to a `TabError`.

Most code editors will show the whitespace and whether it is tabs or spaces. If so, turn it on and save yourself a lot of confusion.

2. Getting Ready to Tango

Before we get down to coding, it's really important that we get our development environment setup so that you can *Tango with Django!* You'll need to ensure that you have all the necessary components installed on your computer. This chapter outlines the five key components that you need to be aware of, setup and use. These are listed below.

- Working with the terminal or Command Prompt.
- *Python* and your *Python* installation.
- The Python Package Manager *pip* and *virtual environments*.
- Your *Integrated Development Environment (IDE)*, if you choose to use one.
- A *Version Control System (VCS), Git*.

If you already have Python 2.7/3.4/3.5 and Django 1.9/1.10 installed on your computer, and are familiar with the technologies mentioned, then you can skip straight to the Django Basics chapter. Otherwise, below we provide an overview of the different components and why they are important. We also provide a series of pointers on how to setup the various components.

Your Development Environment

Setting up your development environment is pretty tedious and often frustrating. It's not something that you'd do everyday. Below, we have put together the list of core technologies you need to get started and pointers on how to install them.

From experience, we can also say that it's a good idea when setting your development environment up to note down the steps you took. You'll need them again one day - whether because you have purchased a new computer, or you have been asked to help someone else set their computer up! Taking a note of everything you do will save you time and effort in the future. Don't just think short term!

2.1 Python

To work with Tango with Django, we require you to have installed on your computer a copy of the Python programming language. Any version from the `2.7` family - with a minimum of `2.7.5` - or version `3.4+` will work fine. If you're not sure how to install Python and would like some assistance, have a look at the chapter dealing with installing Python.

Not sure how to use Python?

If you haven't used Python before - or you simply wish to brush up on your skills - then we highly recommend that you check out and work through one or more of the following guides:

- **Learn Python in 10 Minutes** by Stavros;
- **The Official Python Tutorial**;
- **Think Python: How to Think like a Computer Scientist** by Allen B. Downey; or
- **Learn to Program** by Jennifer Campbell and Paul Gries.

These will get you familiar with the basics of Python so you can start developing using Django. Note you don't need to be an expert in Python to work with Django. Python is awesome and you can pick it up as you go, if you already know another programming language.

2.2 The Python Package Manager

Pip is the python package manager. The package manager allows you install various libraries for the Python programming language to enhance its functionality.

A package manager, whether for Python, your operating system or some other environment, is a software tool that automates the process of installing, upgrading, configuring and removing *packages* - that is, a package of software which you can use on your computer. This is opposed to downloading, installing and maintaining software manually. Maintaining Python packages is pretty painful. Most packages often have *dependencies* so these need to be installed too. Then these packages may conflict or require particular versions which need to be resolved. Also, the system path to these packages needs to be specified and maintained. Luckily *pip* handles all this for you - so you can sit back and relax.

Try and run pip with the command `$ pip`. If the command is not found, you'll need to install pip itself - check out the system setup chapter for more information. You should also ensure that the following packages are installed on your system. Run the following commands to install Django and pillow (an image manipulation library for Python).

```
$ pip install -U django==1.9.10
$ pip install pillow
```

Problems Installing `pillow`?

When installing Pillow, you may receive an error stating that the installation failed due to a lack of JPEG support. This error is shown as the following:

```
ValueError: jpeg is required unless explicitly disabled using
            --disable-jpeg, aborting
```

If you receive this error, try installing Pillow *without* JPEG support enabled, with the following command.

```
pip install pillow --global-option="build_ext"
                   --global-option="--disable-jpeg"
```

While you obviously will have a lack of support for handling JPEG images, Pillow should then install without problem. Getting Pillow installed is enough for you to get started with this tutorial. For further information, check out the Pillow documentation.

2.3 Virtual Environments

We're almost all set to go! However, before we continue, it's worth pointing out that while this setup is fine to begin with, there are some drawbacks. What if you had another Python application that requires a different version to run, or you wanted to switch to the new version of Django, but still wanted to maintain your Django 1.9 project?

The solution to this is to use virtual environments. Virtual environments allow multiple installations of Python and their relevant packages to exist in harmony. This is the generally accepted approach to configuring a Python setup nowadays.

Setting up a virtual environment is not necessarily but it is highly recommended. The virtual environment chapter details how to setup, create and use virtual environments.

2.4 Integrated Development Environment

While not absolutely necessary, a good Python-based IDE can be very helpful to you during the development process. Several exist, with perhaps *PyCharm* by JetBrains and *PyDev* (a plugin of the Eclipse IDE) standing out as popular choices. The Python Wiki provides an up-to-date list of Python IDEs.

Research which one is right for you, and be aware that some may require you to purchase a licence. Ideally, you'll want to select an IDE that supports integration with Django.

We use PyCharm as it supports virtual environments and Django integration - though you will have to configure the IDE accordingly. We don't cover that here - although JetBrains do provide a guide on setting PyCharm up.

2.5 Code Repository

We should also point out that when you develop code, you should always house your code within a version-controlled repository such as SVN or Git. We won't be explaining this right now, so that we can get stuck into developing an application in Django. We have however written a chapter providing a crash course on Git for your reference that you can refer to later on. **We highly recommend that you set up a Git repository for your own projects**.

Exercises

To get comfortable with your environment, try out the following exercises.

- Install Python 2.7.5+/3.4+ and Pip.
- Play around with your *command line interface (CLI)* and create a directory called `code`, which we use to create our projects in.
- Setup your Virtual Environment (optional)
- Install the Django and Pillow packages
- Setup an account on a Git Repository site like: GitHub, BitBucket, etc if you haven't already done so.
- Download and setup an Integrated Development Environment like PyCharm

As previously stated, we've made the code for the book and application available on our GitHub repository.

- If you spot any errors or problem, please let us know by making a change request on GitHub.
- If you have any problems with the exercises, you can check out the repository to see how we completed them.

What is a Directory?

In the text above, we refer to creating a *directory*. But what exactly is a *directory*? If you have used a Windows computer up until now, you'll know a directory as a *folder*. The concept of a folder is analogous to a directory - it is a cataloguing structure that contains references to other files and directories.

3. Django Basics

Let's get started with Django! In this chapter, we'll be giving you an overview of the creation process. You'll be setting up a new project and a new Web application. By the end of this chapter, you will have a simple Django powered website up and running!

3.1 Testing Your Setup

Let's start by checking that your Python and Django installations are correct for this tutorial. To do this, open a new terminal window and issue the following command, which tells you what Python version you have.

```
$ python --version
```

The response should be something like `2.7.11` or `3.5.1`, but any 2.7.5+ or 3.4+ versions of Python should work fine. If you need to upgrade or install Python go to the chapter on setting up your system.

If you are using a virtual environment, then ensure that you have activated it - if you don't remember how go back to our chapter on virtual environments.

After verifying your Python installation, check your Django installation. In your terminal window, run the Python interpreter by issuing the following command.

```
$ python
Python 2.7.10 (default, Jul 14 2015, 19:46:27)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.39)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

At the prompt, enter the following commands:

```
>>> import django
>>> django.get_version()
'1.9.10'
>>> exit()
```

All going well you should see the correct version of Django, and then can use `exit()` to leave the Python interpreter. If `import django` fails to import, then check that you are in your virtual environment, and check what packages are installed with `pip list` at the terminal window.

If you have problems with installing the packages or have a different version installed, go to System Setup chapter or consult the Django Documentation on Installing Django.

Prompts

In this book, there's two things you should look out for when we include code snippets.

Snippets beginning with a dollar sign (`$`) indicates that the remainder of the following line is a terminal or Command Prompt command.

Whenever you see `>>>`, the following is a command that should be entered into the interactive Python interpreter. This is launched by issuing `$ python`. See what we did there? You can also exit the Python interpreter by entering `quit()`.

3.2 Creating Your Django Project

To create a new Django Project, go to your `workspace` directory, and issue the following command:

```
$ django-admin.py startproject tango_with_django_project
```

If you don't have a `workspace` directory, then create one, so that you can house your Django projects and other code projects within this directory. We will refer to your `workspace` directory in the code as `<workspace>`. You will have to substitute in the path to your `workspace` directory, for example: `/Users/leifos/Code/` or `/Users/maxwelld90/Workspace/`.

Can't find `django-admin.py`?

Try entering `django-admin` instead. Depending on your setup, some systems may not recognise `django-admin.py`.

While, on Windows, you may have to use the full path to the `django-admin.py` script, for example:

```
python c:\python27\scripts\django-admin.py
       startproject tango_with_django_project
```

as suggested on StackOverflow.

This command will invoke the `django-admin.py` script, which will set up a new Django project called `tango_with_django_project` for you. Typically, we append `_project` to the end of our Django

project directories so we know exactly what they contain - but the naming convention is entirely up to you.

You'll now notice within your workspace is a directory set to the name of your new project, `tango_with_django_project`. Within this newly created directory, you should see two items:

- another directory with the same name as your project, `tango_with_django_project`; and
- a Python script called `manage.py`.

For the purposes of this tutorial, we call this nested directory called `tango_with_django_project` the *project configuration directory*. Within this directory, you will find four Python scripts. We will discuss these scripts in detail later on, but for now you should see:

- `__init__.py`, a blank Python script whose presence indicates to the Python interpreter that the directory is a Python package;
- `settings.py`, the place to store all of your Django project's settings;
- `urls.py`, a Python script to store URL patterns for your project; and
- `wsgi.py`, a Python script used to help run your development server and deploy your project to a production environment.

In the project directory, you will see there is a file called `manage.py`. We will be calling this script time and time again as we develop our project. It provides you with a series of commands you can run to maintain your Django project. For example, `manage.py` allows you to run the built-in Django development server, test your application and run various database commands. We will be using the script for virtually every Django command we want to run.

The Django Admin and Manage Scripts

For Further Information on Django admin script, see the Django documentation for more details about the Admin and Manage scripts.

Note that if you run `python manage.py help` you can see the list of commands available.

You can try using the `manage.py` script now, by issuing the following command.

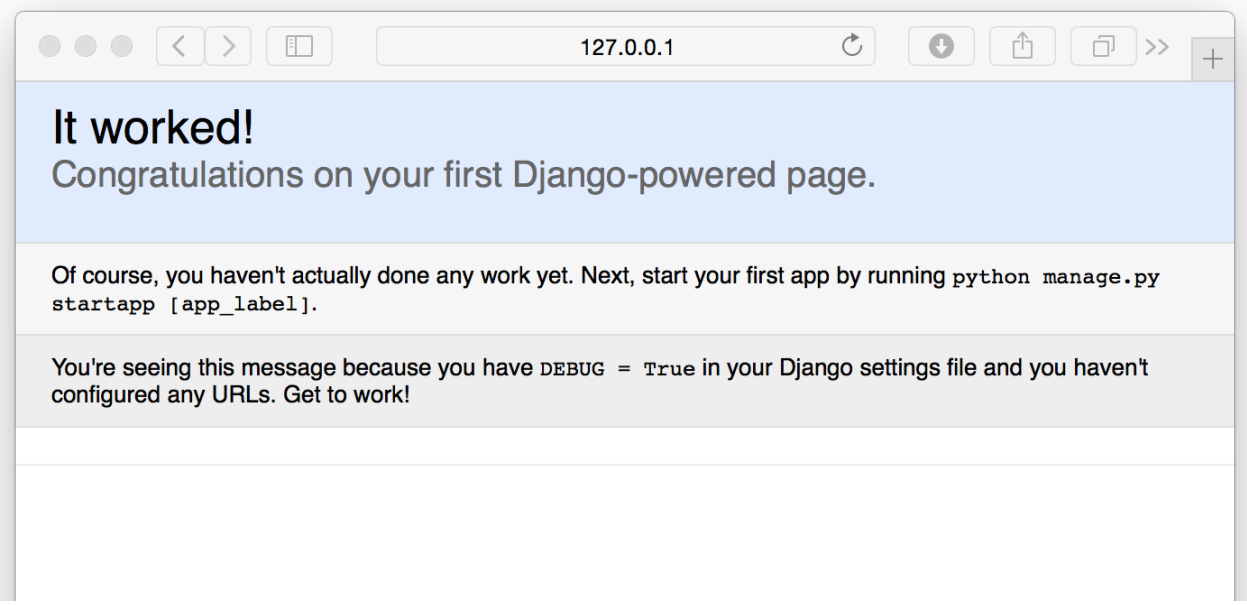
```
$ python manage.py runserver
```

Executing this command will launch Python, and instruct Django to initiate its lightweight development server. You should see the output in your terminal window similar to the example shown below:

```
$ python manage.py runserver

Performing system checks...

System check identified no issues (0 silenced).

You have unapplied migrations; your app may
not work properly until they are applied.

Run 'python manage.py migrate' to apply them.

October 2, 2016 - 21:45:32
Django version 1.9.10, using settings 'tango_with_django_project.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

In the output you can see a number of things. First, there are no issues that stop the application from running. Second, however, you will notice that a warning is raised, i.e. unapplied migrations. We will talk about this in more detail when we setup our database, but for now we can ignore it. Third, and most importantly, you can see that a URL has been specified: `http://127.0.0.1:8000/`, which is the address of the Django development webserver.

Now open up your Web browser and enter the URL `http://127.0.0.1:8000/`. You should see a webpage similar to the one shown in below.



A screenshot of the initial Django page you will see when running the development server for the first time.

You can stop the development server at anytime by pushing `CTRL + C` in your terminal or Command Prompt window. If you wish to run the development server on a different port, or allow users from other machines to access it, you can do so by supplying optional arguments. Consider the following command:

```
$ python manage.py runserver <your_machines_ip_address>:5555
```

Executing this command will force the development server to respond to incoming requests on TCP port 5555. You will need to replace `<your_machines_ip_address>` with your computer's IP address or `127.0.0.1`.

Don't know your IP Address?

If you use `0.0.0.0`, Django figures out what your IP address is. Go ahead and try:

```
python manage.py runserver 0.0.0.0:5555
```

When setting ports, it is unlikely that you will be able to use TCP port 80 or 8080 as these are traditionally reserved for HTTP traffic. Also, any port below 1024 is considered to be privileged by your operating system.

While you won't be using the lightweight development server to deploy your application, it's nice to be able to demo your application on another machine in your network. Running the server with your machine's IP address will enable others to enter in `http://<your_machines_ip_address>:<port>/` and view your Web application. Of course, this will depend on how your network is configured. There may be proxy servers or firewalls in the way that would need to be configured before this would work. Check with the administrator of the network you are using if you can't view the development server remotely.

3.3 Creating a Django App

A Django project is a collection of *configurations* and *apps* that together make up a given Web application or website. One of the intended outcomes of using this approach is to promote good software engineering practices. By developing a series of small applications, the idea is that you can theoretically drop an existing application into a different Django project and have it working with minimal effort.

A Django application exists to perform a particular task. You need to create specific apps that are responsible for providing your site with particular kinds of functionality. For example, we could imagine that a project might consist of several apps including a polling app, a registration app, and a specific content related app. In another project, we may wish to re-use the polling and registration apps, and so can include them in other projects. We will talk about this later. For now we are going to create the app for the *Rango* app.

To do this, from within your Django project directory (e.g. `<workspace>/tango_with_django_project`), run the following command.

```
$ python manage.py startapp rango
```

The `startapp` command creates a new directory within your project's root. Unsurprisingly, this directory is called `rango` - and contained within it are a number of Python scripts:

- another `__init__.py`, serving the exact same purpose as discussed previously;
- `admin.py`, where you can register your models so that you can benefit from some Django machinery which creates an admin interface for you;
- `apps.py`, that provides a place for any app specific configuration;
- `models.py`, a place to store your app's data models - where you specify the entities and relationships between data;
- `tests.py`, where you can store a series of functions to test your app's code;
- `views.py`, where you can store a series of functions that handle requests and return responses; and
- `migrations` directory, which stores database specific information related to your models.

`views.py` and `models.py` are the two files you will use for any given app, and form part of the main architectural design pattern employed by Django, i.e. the *Model-View-Template* pattern. You can check out the official Django documentation to see how models, views and templates relate to each other in more detail.

Before you can get started with creating your own models and views, you must first tell your Django project about your new app's existence. To do this, you need to modify the `settings.py` file, contained within your project's configuration directory. Open the file and find the `INSTALLED_APPS` tuple. Add the `rango` app to the end of the tuple, which should then look like the following example.

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

Verify that Django picked up your new app by running the development server again. If you can start the server without errors, your app was picked up and you will be ready to proceed to the next step.

`startapp` Magic

When creating a new app with the `python manage.py startapp` command, Django may add the new app's name to your `settings.py` `INSTALLED_APPS` list automatically for you. It's nevertheless good practice to check everything is setup correctly before you proceed.

3.4 Creating a View

With our Rango app created, let's now create a simple view. For our first view, let's just send some text back to the client - we won't concern ourselves about using models or templates just yet.

In your favourite IDE, open the file `views.py`, located within your newly created `rango` app directory. Remove the comment `# Create your views here.` so that you now have a blank file.

You can now add in the following code.

```
from django.http import HttpResponse

def index(request):
    return HttpResponse("Rango says hey there partner!")
```

Breaking down the three lines of code, we observe the following points about creating this simple view.

- We first import the `HttpResponse` object from the `django.http` module.
- Each view exists within the `views.py` file as a series of individual functions. In this instance, we only created one view - called `index`.
- Each view takes in at least one argument - a `HttpRequest` object, which also lives in the `django.http` module. Convention dictates that this is named `request`, but you can rename this to whatever you want if you so desire.
- Each view must return a `HttpResponse` object. A simple `HttpResponse` object takes a string parameter representing the content of the page we wish to send to the client requesting the view.

With the view created, you're only part of the way to allowing a user to access it. For a user to see your view, you must map a Uniform Resource Locator (URL) to the view.

To create an initial mapping, open `urls.py` located in your project directory and add the following lines of code to the `urlpatterns`:

```
from rango import views

urlpatterns = [
    url(r'^$', views.index, name='index'),
    url(r'^admin/', admin.site.urls),
]
```

This maps the basic URL to the `index` view in the `rango` app. Run the development server (e.g. `python manage.py runserver`) and visit `http://127.0.0.1:8000` or whatever address your development server is running on. You'll then see the rendered output of the `index` view.

3.5 Mapping URLs

Rather than directly mapping URLs from the project to the app, we can make our app more modular (and thus re-usable) by changing how we route the incoming URL to a view. To do this, we first need to modify the project's `urls.py` and have it point to the app to handle any specific Rango app requests. We then need to specify how Rango deals with such requests.

First, open the project's `urls.py` file which is located inside your project configuration directory. As a relative path from your workspace directory, this would be the file `<workspace>/tango_with_django_project/tango_with_django_project/urls.py`. Update the `urlpatterns` list as shown in the example below.

```
from django.conf.urls import url
from django.contrib import admin
from django.conf.urls import include
from rango import views

urlpatterns = [
    url(r'^$', views.index, name='index'),
    url(r'^rango/', include('rango.urls')),
    # above maps any URLs starting
    # with rango/ to be handled by
    # the rango application
    url(r'^admin/', admin.site.urls),
]
```

You will see that the `urlpatterns` is a Python list, which is expected by the Django framework. The added mapping looks for URL strings that match the patterns `^rango/`. When a match is made the remainder of the URL string is then passed onto and handled by `rango.urls` through the use of the `include()` function from within `django.conf.urls`.

Think of this as a chain that processes the URL string - as illustrated in the URL chain figure. In this chain, the domain is stripped out and the remainder of the URL string (`rango/`) is passed on to `tango_with_django` project, where it finds a match and strips away `rango/`, leaving an empty string to be passed on to the app `rango` for it to handle.

Consequently, we need to create a new file called `urls.py` in the `rango` app directory, to handle the remaining URL string (and map the empty string to the `index` view):

```
from django.conf.urls import url
from rango import views

urlpatterns = [
    url(r'^$', views.index, name='index'),
]
```

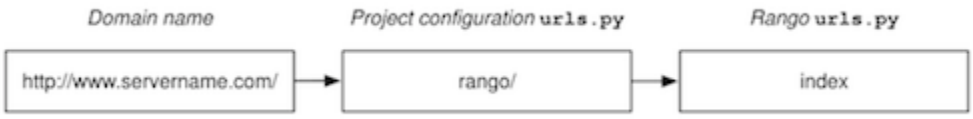
This code imports the relevant Django machinery for URL mappings and the `views` module from `rango`. This allows us to call the function `url` and point to the `index` view for the mapping in `urlpatterns`.

When we talk about URL strings, we assume that the host portion of a given URL has *already been stripped away*. The host portion of a URL denotes the host address or domain name that maps to the webserver, such as `http://127.0.0.1:8000` or `http://www.tangowithdjango.com`. Stripping the host portion away means that the Django machinery needs to only handle the remainder of the URL string. For example, given the URL `http://127.0.0.1:8000/rango/about/`, Django would have a URL string of `/rango/about/`.

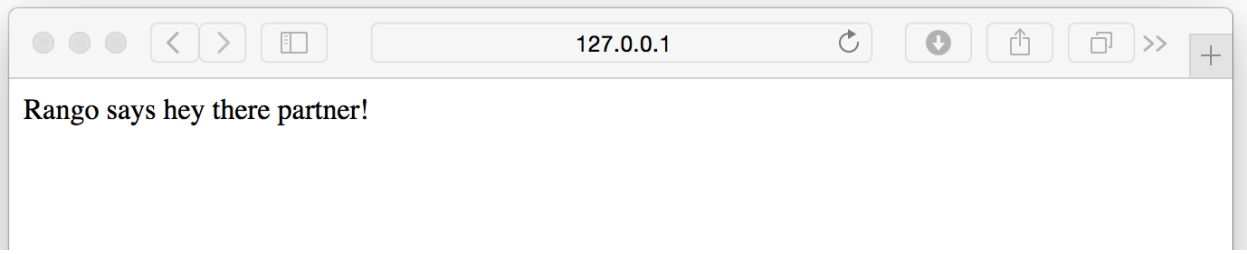
The URL mapping we have created above calls Django's `url()` function, where the first parameter is the regular expression `^$`, which matches to an empty string because `^` denotes starts with, while `$` denotes ends with. As there is nothing in between these characters then it only matches an empty string. Any URL string supplied by the user that matches this pattern means that the view `views.index()` would be invoked by Django. You might be thinking that matching a blank URL is pretty pointless - what use would it serve? Remember that when the URL pattern matching takes place, only a portion of the original URL string is considered. This is because Django will first process the URL patterns in the project processing the original URL string (i.e. `rango/`) and strip away the `rango/` part. Django will then pass on an empty string to the Rango app to handle via the URL patterns in `rango/urls.py`.

The next parameter passed to the `url()` function is the `index` view, which will handle the incoming requests, followed by the optional parameter, `name` that is set to a string `'index'`. By naming our URL mappings we can employ *reverse URL matching* later on. That is we can reference the URL mapping by name rather than by the URL. Later we will explain how to use this when creating templates. But do check out the Official Django documentation on this topic for more information.

Now, restart the Django development server and visit `http://127.0.0.1:8000/rango/`. If all went well, you should see the text `Rango says hey there partner!`. It should look just like the screenshot shown below.



An illustration of a URL, represented as a chain, showing how different parts of the URL following the domain are the responsibility of different `url.py` files.



A screenshot of a Web browser displaying our first Django powered webpage. Hello, Rango!

Within each app, you will create a number of URL mappings. The initial mapping is quite simple, but as we progress through the book we will create more sophisticated, parameterised URL mappings.

It's also important to have a good understanding of how URLs are handled in Django. It may seem a bit confusing right now, but as we progress through the book, we will be creating more and more URL mappings, so you'll soon be a pro. To find out more about them, check out the official Django documentation on URLs for further details and further examples.

Note on Regular Expressions

Django URL patterns use regular expressions to perform the matching. It is worthwhile familiarising yourself on how to use regular expressions in Python. The official Python documentation contains a useful guide on regular expressions, while `regexcheatsheet.com` provides a neat summary of regular expressions.

If you are using version control, now is a good time to commit the changes you have made to your workspace. Refer to the chapter providing a crash course on Git if you can't remember the commands and steps involved in doing this.

3.6 Basic Workflows

What you've just learnt in this chapter can be succinctly summarised into a list of actions. Here, we provide these lists for the two distinct tasks you have performed. You can use this section for a quick reference if you need to remind yourself about particular actions later on.

Creating a new Django Project

1. To create the project run, `python django-admin.py startproject <name>`, where `<name>` is the name of the project you wish to create.

Creating a new Django App

1. To create a new app, run `$ python manage.py startapp <appname>`, where `<appname>` is the name of the app you wish to create.
2. Tell your Django project about the new app by adding it to the `INSTALLED_APPS` tuple in your project's `settings.py` file.
3. In your project `urls.py` file, add a mapping to the app.
4. In your app's directory, create a `urls.py` file to direct incoming URL strings to views.
5. In your app's `view.py`, create the required views ensuring that they return a `HttpResponse` object.

Exercises

Now that you have got Django and your new app up and running, give the following exercises a go to reinforce what you've learnt. Getting to this stage is a significant landmark in working with Django. Creating views and mapping URLs to views is the first step towards developing more complex and usable Web applications.

- Revise the procedure and make sure you follow how the URLs are mapped to views.
- Create a new view method called `about` which returns the following `HttpResponse`: `'Rango says here is the about page.'`
- Map this view to `/rango/about/`. For this step, you'll only need to edit the `urls.py` of the Rango app. Remember the `/rango/` part is handled by the projects `urls.py`.
- Revise the `HttpResponse` in the `index` view to include a link to the about page.
- In the `HttpResponse` in the `about` view include a link back to the main page.
- Now that you have started the book, follow us on Twitter @tangowithdjango, and let us know how you are getting on!

Hints

If you're struggling to get the exercises done, the following hints will hopefully provide you with some inspiration on how to progress.

- In your `views.py`, create a function called: `def about(request):`, and have the function return a HttpResponse(), insert your HTML inside this response.
- The regular expression to match `about/` is `r'^about/'` - so in `rango/urls.py` add in a new mapping to the `about()` view.
- Update your `index()` view to include a link to the `about` view. Keep it simple for now - something like `Rango says hey there partner! <br/> <a href='/rango/about/'>About</a>`.
- Also add the HTML to link back to the index page is into your response from the `about()` view `<a href="/rango/">Index</a>`.
- If you haven't done so already, now's a good time to head off and complete part one of the official Django Tutorial.

4. Templates and Media Files

In this chapter, we'll be introducing the Django template engine, as well as showing how to serve both *static* files and *media* files, both of which can be integrated within your app's webpages.

4.1 Using Templates

Up until this point, we have only connected a URL mapping to a view. The Django framework, however, is based around the *Model-View-Template* architecture. In this section, we will go through the mechanics of how *Templates* work with *Views*, then in the next couple of chapters we will put these together with *Models*.

Why templates? The layout from page to page within a website is often the same. Whether you see a common header or footer on a website's pages, the repetition of page layouts aids users with navigation, promotes organisation of the website and reinforces a sense of continuity. Django provides templates to make it easier for developers to achieve this design goal, as well as separating application logic (code within your views) from presentational concerns (look and feel of your app). In this chapter, you'll create a basic template that will be used to create a HTML page. This template will then be dispatched via a Django view. In the chapter concerning databases and models, we will take this a step further by using templates in conjunction with models to dispatch dynamically generated data.

Summary: What is a Template?

In the world of Django, think of a *template* as the scaffolding that is required to build a complete HTML webpage. A template contains the *static parts* of a webpage (that is, parts that never change), complete with special syntax (or *template tags*) which can be overridden and replaced with *dynamic content* that your Django app's views can replace to produce a final HTML response.

Configuring the Templates Directory

To get templates up and running with your Django app, you'll need to create a directory in which template files are stored.

In your Django project's directory (e.g. `<workspace>/tango_with_django_project/`), create a new directory called `templates`. Within the new templates directory, create another directory called `rango`. This means that the path `<workspace>/tango_with_django_project/templates/rango/` will be the location in which we will store templates associated with our `rango` application.

Keep your Templates Organised

It's good practice to separate out your templates into subdirectories for each app you have. This is why we've created a `rango` directory within our `templates` directory. If you package your app up to distribute to other developers, it'll be much easier to know which templates belong to which app!

To tell the Django project where templates will be stored, open your project's `settings.py` file. Next, locate the `TEMPLATES` data structure. By default, when you create a new Django 1.9 project, it will look like the following.

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
            ],
        },
    },
]
```

What we need to do to is tell Django where our templates are stored by modifying the `DIRS` list, which is set to an empty list by default. Change the dictionary key/value pair to look like the following.

```
'DIRS': ['<workspace>/tango_with_django_project/templates']
```

Note that you are *required to use absolute paths* to locate the `templates` directory. If you are collaborating with team members or working on different computers, then this will become a problem. You'll have different usernames and different drive structures, meaning the paths to the `<workspace>` directory will be different. One solution would be to add the path for each different configuration. For example:

```
'DIRS': [ '/Users/leifos/templates',
          '/Users/maxwelld90/templates',
          '/Users/clueless_noob/templates', ]
```

However, there are a number of problems with this. First you have to add in the path for each setting, each time. Second, if you are running the app on different operating systems the backslashes have to be constructed differently.

Don't hard code Paths!

The road to hell is paved with hard coded paths. Hard-coding paths is a software engineering anti-pattern, and will make your project less portable - meaning that when you run it on another computer, it probably won't work!

Dynamic Paths

A better solution is to make use of built-in Python functions to work out the path of your `templates` directory automatically. This way, an absolute path can be obtained regardless of where you place your Django project's code. This in turn means that your project becomes more *portable.*

At the top of your `settings.py` file, there is a variable called `BASE_DIR`. This variable stores the path to the directory in which your project's `settings.py` module is contained. This is obtained by using the special Python `__file__` attribute, which is set to the absolute path of your settings module. The call to `os.path.dirname()` then provides the reference to the absolute path of the directory containing the `settings.py` module. Calling `os.path.dirname()` again removes another layer, so that `BASE_DIR` contains `<workspace>/tango_with_django_project/`. You can see this process in action, if you are curious, by adding the following lines to your `settings.py` file.

```
print(__file__)
print(os.path.dirname(__file__))
print(os.path.dirname(os.path.dirname(__file__)))
```

Having access to the value of `BASE_DIR` makes it easy for you to reference other aspects of your Django project. As such, we can now create a new variable called `TEMPLATE_DIR` that will reference your new `templates` directory. We can make use of the `os.path.join()` function to join up multiple paths, leading to a variable definition like the example below.

```
TEMPLATE_DIR = os.path.join(BASE_DIR, 'templates')
```

Here we make use of `os.path.join()` to mash together the `BASE_DIR` variable and `'templates'`, which would yield `<workspace>/tango_with_django_project/templates/`. This means we can then use our new `TEMPLATE_DIR` variable to replace the hard coded path we defined earlier in `TEMPLATES`. Update the `DIRS` key/value pairing to look like the following.

```
'DIRS': [TEMPLATE_DIR, ]
```

Why `TEMPLATE_DIR`?

You've created a new variable called `TEMPLATE_DIR` at the top of your `settings.py` file because it's easier to access should you ever need to change it. For more complex Django projects, the `DIRS` list allows you to specify more than one template directory - but for this book, one location is sufficient to get everything working.

Concatenating Paths

When concatenating system paths together, always use `os.path.join()`. Using this built-in function ensures that the correct path separators are used. On a UNIX operating system (or derivative of), forward slashes (`/`) would be used to separate directories, whereas a Windows operating system would use backward slashes (`\`). If you manually append slashes to paths, you may end up with path errors when attempting to run your code on a different operating system, thus reducing your project's portability.

Adding a Template

With your template directory and path now set up, create a file called `index.html` and place it in the `templates/rango/` directory. Within this new file, add the following HTML code.

```
<!DOCTYPE html>
<html>

    <head>
        <title>Rango</title>
    </head>

    <body>
        <h1>Rango says...</h1>
        <div>
            hey there partner! <br />
            <strong>{{ boldmessage }}</strong><br />
        </div>
        <div>
            <a href="/rango/about/">About</a><br />
        </div>
    </body>

</html>
```

From this HTML code, it should be clear that a simple HTML page is going to be generated that greets a user with a *hello world* message. You might also notice some non-HTML in the form of `{{ boldmessage }}`. This is a *Django template variable*. We can set values to these variables so they are replaced with whatever we want when the template is rendered. We'll get to that in a moment.

To use this template, we need to reconfigure the `index()` view that we created earlier. Instead of dispatching a simple response, we will change the view to dispatch our template.

In `rango/views.py`, check to see if the following `import` statement exists at the top of the file. If it is not present, add it.

```
from django.shortcuts import render
```

You can then update the `index()` view function as follows. Check out the inline commentary to see what each line does.

```
def index(request):
    # Construct a dictionary to pass to the template engine as its context.
    # Note the key boldmessage is the same as {{ boldmessage }} in the template!
    context_dict = {'boldmessage': "Crunchy, creamy, cookie, candy, cupcake!"}

    # Return a rendered response to send to the client.
    # We make use of the shortcut function to make our lives easier.
    # Note that the first parameter is the template we wish to use.
    return render(request, 'rango/index.html', context=context_dict)
```

First, we construct a dictionary of key/value pairs that we want to use within the template. Then, we call the `render()` helper function. This function takes as input the user's `request`, the template filename, and the context dictionary. The `render()` function will take this data and mash it together with the template to produce a complete HTML page that is returned with a *HttpResponse*. This response is then returned and dispatched to the user's web browser.

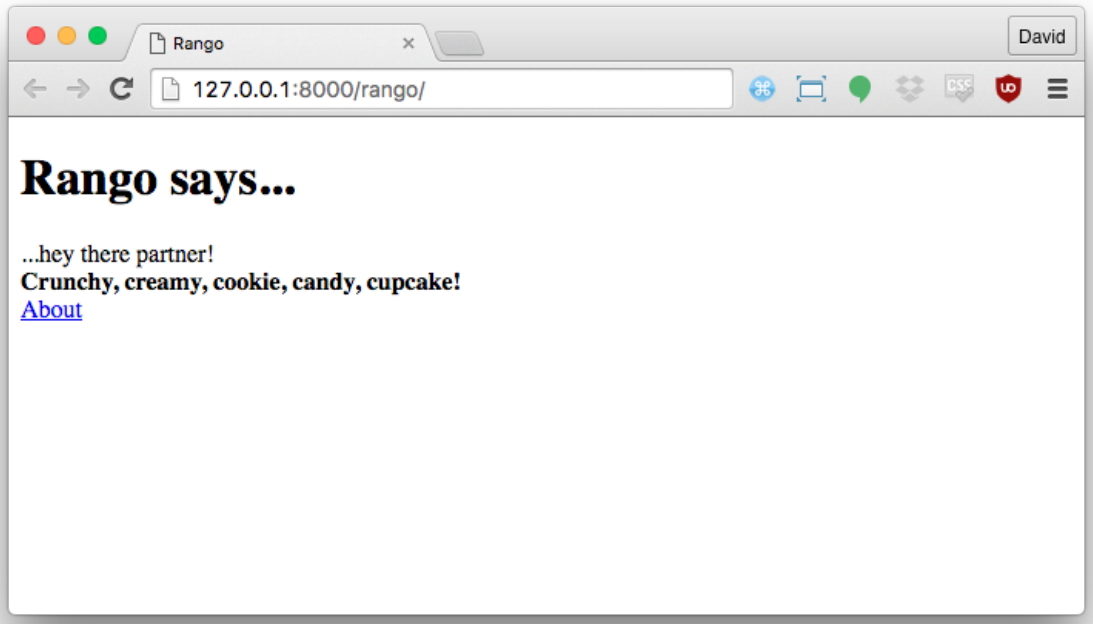
What is the Template Context?

When a template file is loaded with the Django templating system, a *template context* is created. In simple terms, a template context is a Python dictionary that maps template variable names with Python variables. In the template we created above, we included a template variable name called `boldmessage`. In our updated `index(request)` view example, the string `Crunchy, creamy, cookie, candy, cupcake!` is mapped to template variable `boldmessage`. The string `Crunchy, creamy, cookie, candy, cupcake!` therefore replaces *any* instance of `{{ boldmessage }}` within the template.

Now that you have updated the view to employ the use of your template, start the Django development server and visit `http://127.0.0.1:8000/rango/`. You should see your simple HTML template rendered, just like the example screenshot shown below.

If you don't, read the error message presented to see what the problem is, and then double check all the changes that you have made. One of the most common issues people have with templates is that the path is set incorrectly in `settings.py`. Sometimes it's worth adding a `print` statement to `settings.py` to report the `BASE_DIR` and `TEMPLATE_DIR` to make sure everything is correct.

This example demonstrates how to use templates within your views. However, we have only touched upon a fraction of the functionality provided by the Django templating engine. We will use templates in more sophisticated ways as you progress through this book. In the meantime, you can find out more about templates from the official Django documentation.



What you should see when your first template is working correctly. Note the bold text - `Crunchy, creamy, cookie, candy, cupcake!` - which originates from the view, but is rendered in the template.

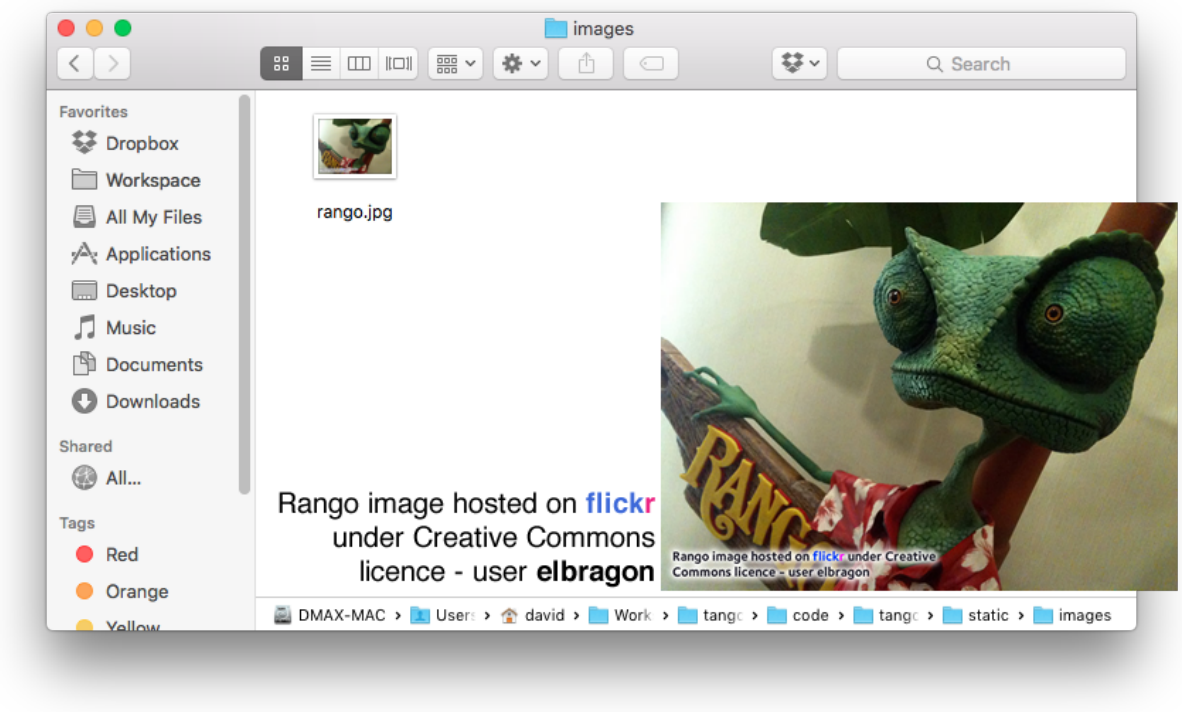
4.2 Serving Static Media Files

While you've got templates working, your Rango app is admittedly looking a bit plain right now - there's no styling or imagery. We can add references to other files in our HTML template such as *Cascading Style Sheets (CSS)*, *JavaScript* and images to improve the presentation. These are called *static files*, because they are not generated dynamically by a Web server; they are simply sent as is to a client's Web browser. This section shows you how to set Django up to serve static files, and shows you how to include an image within your simple template.

Configuring the Static Media Directory

To start, you will need to set up a directory in which static media files are stored. In your project directory (e.g. `<workspace>/tango_with_django_project/`), create a new directory called `static` and a new directory called `images` inside `static`. Check that the new `static` directory is at the same level as the `templates` directory you created earlier in this chapter.

Next, place an image inside the `images` directory. As shown in below, we chose a picture of the chameleon Rango - a fitting mascot, if ever there was one.



Rango the chameleon within our `static/images` media directory.

Just like the `templates` directory we created earlier, we need to tell Django about our new `static` directory. To do this, we once again need to edit our project's `settings.py` module. Within this file, we need to add a new variable pointing to our `static` directory, and a data structure that Django can parse to work out where our new directory is.

First of all, create a variable called `STATIC_DIR` at the top of `settings.py`, preferably underneath `BASE_DIR` and `TEMPLATES_DIR` to keep your paths all in the same place. `STATIC_DIR` should make use of the same `os.path.join` trick - but point to `static` this time around, just as shown below.

```
STATIC_DIR = os.path.join(BASE_DIR, 'static')
```

This will provide an absolute path to the location `<workspace>/tango_with_django_project/static/`. Once this variable has been created, we then need to create a new data structure called

`STATICFILES_DIRS`. This is essentially a list of paths with which Django can expect to find static files that can be served. By default, this list does not exist - **check** it doesn't before you create it. If you define it twice, you can start to confuse Django - and yourself.

For this book, we're only going to be using one location to store our project's static files - the path defined in `STATIC_DIR`. As such, we can simply set up `STATICFILES_DIRS` with the following.

```
STATICFILES_DIRS = [STATIC_DIR, ]
```

Keep `settings.py` Tidy!

It's in your best interests to keep your `settings.py` module tidy and in good order. Don't just put things in random places; keep it organised. Keep your `DIRS` variables at the top of the module so they are easy to find, and place `STATICFILES_DIRS` in the portion of the module responsible for static media (close to the bottom). When you come back to edit the file later, it'll be easier for you or other collaborators to find the necessary variables.

Finally, check that the `STATIC_URL` variable is defined within your `settings.py` module. If it is not, then define it as shown below. Note that this variable by default in Django 1.9 appears close to the end of the module, so you may have to scroll down to find it.

```
STATIC_URL = '/static/'
```

With everything required now entered, what does it all mean? Put simply, the first two variables `STATIC_DIR` and `STATICFILES_DIRS` refers to the locations on your computer where static files are stored. The final variable `STATIC_URL` then allows us to specify the URL with which static files can be accessed when we run our Django development server. For example, with `STATIC_URL` set to `/static/`, we would be able to access static content at `http://127.0.0.1:8000/static/`. *Think of the first two variables as server-side locations, and the third variable as the location with which clients can access static content.*

Test your Configuration

As a small exercise, test to see if everything is working correctly. Try and view the `rango.jpg` image in your browser when the Django development server is running. If your `STATIC_URL` is set to `/static/` and `rango.jpg` can be found at `images/rango.jpg`, what is the URL you enter into your Web browser's window?

Try to figure this out before you move on! The answer is coming up if you get stuck.

Don't Forget the Slashes!

When setting `STATIC_URL`, check that you end the URL you specify with a forward slash (e.g. `/static/`, not `/static`). As per the official Django documentation, not doing so can open you up to a world of pain. The extra slash at the end ensures that the root of the URL (e.g. `/static/`) is separated from the static content you want to serve (e.g. `images/rango.jpg`).

Serving Static Content

While using the Django development server to serve your static media files is fine for a development environment, it's highly unsuitable for a production environment. The official Django documentation on deployment provides further information about deploying static files in a production environment. We'll look at this issue in more detail however when we deploy Rango.

If you haven't managed to figure out where the image should be accessible from, point your web browser to `http://127.0.0.1:8000/static/images/rango.jpg`.

Static Media Files and Templates

Now that you have your Django project set up to handle static files, you can now make use of these files within your templates to improve their appearance and add additional functionality.

To demonstrate how to include static files, open up the `index.html` templates you created earlier, located in the `<workspace>/templates/rango/` directory. Modify the HTML source code as follows. The two lines that we add are shown with a HTML comment next to them for easy identification.

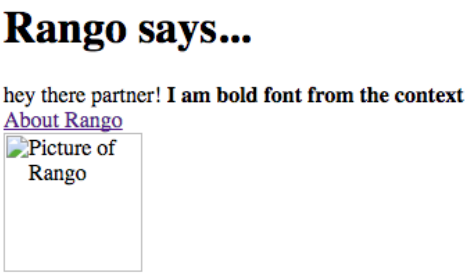
```
<!DOCTYPE html>

{% load staticfiles %} <!-- New line -->

<html>
    <head>
        <title>Rango</title>
    </head>

    <body>
        <h1>Rango says...</h1>

        <div>
            hey there partner! <br />
```

```
            <strong>{{ boldmessage }}</strong><br />
        </div>

        <div>
            <a href="/rango/about/">About</a><br />
            <img src="{% static "images/rango.jpg" %}"
                 alt="Picture of Rango" /> <!-- New line -->
        </div>
    </body>

</html>
```

The first new line added (`{% load staticfiles %}`) informs Django's template engine that we will be using static files within the template. This then enables us to access the media in the static directories via the use of the `static` template tag. This indicates to Django that we wish to show the image located in the static media directory called `images/rango.jpg`. Template tags are denoted by curly brackets (e.g. `{% %}`), and calling `static` will combine the URL specified in `STATIC_URL` with `images/rango.jpg` to yield `/static/images/rango.jpg`. The HTML generated by the Django template engine would be:

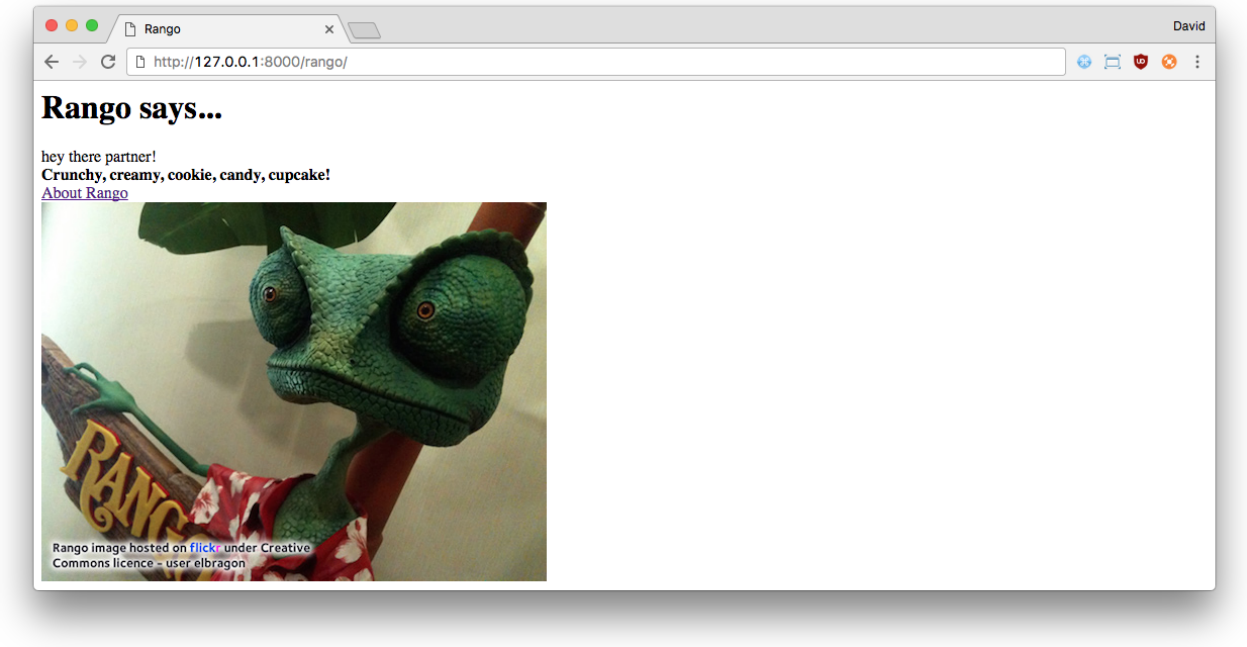
```
<img src="/static/images/rango.jpg" alt="Picture of Rango" />
```

If for some reason the image cannot be loaded, it is always a good idea to specify an alternative text tagline. This is what the `alt` attribute provides inside the `img` tag. You can see what happens in the image below.



The image of Rango couldn't be found, and is instead replaced with a placeholder containing the text from the `img` `alt` attribute.

With these minor changes in place, start the Django development server once more and navigate to `http://127.0.0.1:8000/rango`. If everything has been done correctly, you will see a Webpage that looks similar to the screenshot shown below.



Our first Rango template, complete with a picture of Rango the chameleon.

Templates and `<!DOCTYPE>`

When creating the HTML templates, always ensure that the `DOCTYPE` declaration appears on the **first line**. If you put the `{% load staticfiles %}` template command first, then whitespace will be added to the rendered template before the `DOCTYPE` declaration. This whitespace will lead to your HTML markup failing validation.

Loading other Static Files

The `{% static %}` template tag can be used whenever you wish to reference static files within a template. The code example below demonstrates how you could include JavaScript, CSS and images into your templates with correct HTML markup.

```
<!DOCTYPE html>
{% load staticfiles %}

<html>

    <head>
        <title>Rango</title>
        <!-- CSS -->
        <link rel="stylesheet" href="{% static "css/base.css" %}" />
        <!-- JavaScript -->
        <script src="{% static "js/jquery.js" %}"></script>
    </head>

    <body>
        <!-- Image -->
        <img src="{% static "images/rango.jpg" %}" alt="Picture of Rango" />
    </body>

</html>
```

Static files you reference will obviously need to be present within your `static` directory. If a requested file is not present or you have referenced it incorrectly, the console output provided by Django's development server will show a `HTTP 404` error. Try referencing a non-existent file and see what happens. Looking at the output snippet below, notice how the last entry's HTTP status code is `404`.

```
[10/Apr/2016 15:12:48] "GET /rango/ HTTP/1.1" 200 374
[10/Apr/2016 15:12:48] "GET /static/images/rango.jpg HTTP/1.1" 304 0
[10/Apr/2016 15:12:52] "GET /static/images/not-here.jpg HTTP/1.1" 404 0
```

For further information about including static media you can read through the official Django documentation on working with static files in templates.

4.3 Serving Media

Static media files can be considered files that don't change and are essential to your application. However, often you will have to store *media files* which are dynamic in nature. These files can be

uploaded by your users or administrators, and so they may change. As an example, a media file would be a user's profile picture. If you run an e-commerce website, a series of media files would be used as images for the different products that your online shop has.

In order to serve media files successfully, we need to update Django project's settings. This section details what you need to add - but we won't be fully testing it out until later where we implement the functionality for users to upload profile pictures.

Serving Media Files

Like serving static content, Django provides the ability to serve media files in your development environment - to make sure everything is working. The methods that Django uses to serve this content are highly unsuitable for a production environment, so you should be looking to host your app's media files by some other means. The deployment chapter will discuss this in more detail.

Modifying `settings.py`

First open your Django project's `settings.py` module. In here, we'll be adding a couple more things. Like static files, media files are uploaded to a specified directory on your filesystem. We need to tell Django where to store these files.

At the top of your `settings.py` module, locate your existing `BASE_DIR`, `TEMPLATE_DIR` and `STATIC_DIR` variables - they should be close to the top. Underneath, add a further variable, `MEDIA_DIR`.

```
MEDIA_DIR = os.path.join(BASE_DIR, 'media')
```

This line instructs Django that media files will be uploaded to your Django project's root, plus '/media' - or `<workspace>/tango_with_django_project/media/`. As we previously mentioned, keeping these path variables at the top of your `settings.py` module makes it easy to change paths later on if necessary.

Now find a blank spot in `settings.py`, and add two more variables. The variables `MEDIA_ROOT` and `MEDIA_URL` will be picked up and used by Django to set up media file hosting.

```
MEDIA_ROOT = MEDIA_DIR
MEDIA_URL = '/media/'
```

Once again, don't Forget the Slashes!

Like the `STATIC_URL` variable, ensure that `MEDIA_URL` ends with a forward slash (i.e. `/media/`, not `/media`). The extra slash at the end ensures that the root of the URL (e.g. `/media/`) is separated from the content uploaded by your app's users.

The two variables tell Django where to look in your filesystem for media files (`MEDIA_ROOT`) that have been uploaded/stored, and what URL to serve them from (`MEDIA_URL`). With the configuration defined above, the uploaded file `cat.jpg` will for example be available on your Django development server at `http://localhost:8000/media/cat.jpg`.

When we come to working with templates later on in this book, it'll be handy for us to obtain a reference to the `MEDIA_URL` path when we need to reference uploaded content. Django provides a *template context processor* that'll make it easy for us to do. While we don't strictly need this set up now, it's a good time to add it in.

To do this, find the `TEMPLATES` list in `settings.py`. Within that list, look for the nested `context_processors` list, and within that list, add a new processor, `django.template.context_processors.media`. Your `context_processors` list should then look similar to the example below.

```
'context_processors': [
    'django.template.context_processors.debug',
    'django.template.context_processors.request',
    'django.contrib.auth.context_processors.auth',
    'django.contrib.messages.context_processors.messages',
    'django.template.context_processors.media'
],
```

Tweaking your URLs

The final step for setting up the serving of media in a development environment is to tell Django to serve static content from `MEDIA_URL`. This can be achieved by opening your project's `urls.py` module, and modifying it by appending a call to the `static()` function to your project's `urlpatterns` list.

```
urlpatterns = [
    ...
    ...
] + static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

You'll also need to add the following `import` statements at the top of the `urls.py` module.

```
from django.conf import settings
from django.conf.urls.static import static
```

Once this is complete, you should be able to serve content from the `media` directory of your project from the `/media/` URL.

4.4 Basic Workflow

With the chapter complete, you should now know how to setup and create templates, use templates within your views, setup and use the Django development server to serve static media files, *and* include images within your templates. We've covered quite a lot!

Creating a template and integrating it within a Django view is a key concept for you to understand. It takes several steps, but will become second nature to you after a few attempts.

1. First, create the template you wish to use and save it within the `templates` directory you specified in your project's `settings.py` module. You may wish to use Django template variables (e.g. `{{ variable_name }}`) or template tags within your template. You'll be able to replace these with whatever you like within the corresponding view.
2. Find or create a new view within an application's `views.py` file.
3. Add your view specific logic (if you have any) to the view. For example, this may involve extracting data from a database and storing it within a list.
4. Within the view, construct a dictionary object which you can pass to the template engine as part of the template's *context*.
5. Make use of the `render()` helper function to generate the rendered response. Ensure you reference the request, then the template file, followed by the context dictionary.
6. If you haven't already done so, map the view to a URL by modifying your project's `urls.py` file and the application specific `urls.py` file if you have one.

The steps involved for getting a static media file onto one of your pages are part of another important process that you should be familiar with. Check out the steps below on how to do this.

1. Take the static media file you wish to use and place it within your project's `static` directory. This is the directory you specify in your project's `STATICFILES_DIRS` list within `settings.py`.
2. Add a reference to the static media file to a template. For example, an image would be inserted into an HTML page through the use of the `<img />` tag.
3. Remember to use the `{% load staticfiles %}` and `{% static "<filename>" %}` commands within the template to access the static files. Replace `<filename>` with the path to the image or resource you wish to reference. **Whenever you wish to refer to a static file, use the `static` template tag!**

The steps for serving media files are similar to those for serving static media.

1. Place a file within your project's `media` directory. The `media` directory is specified by your project's `MEDIA_ROOT` variable.
2. Link to the media file in a template through the use of the `{{ MEDIA_URL }}` context variable. For example, referencing an uploaded image `cat.jpg` would have an `<img />` tag like `<img src="{{ MEDIA_URL}}cat.jpg" />`.

Exercises

Give the following exercises a go to reinforce what you've learnt from this chapter.

- Convert the about page to use a template as well, using a template called `about.html`.
- Within the new `about.html` template, add a picture stored within your project's static files.
- On the about page, include a line that says, `This tutorial has been put together by <your-name>.`
- In your Django project directory, create a new directory called `media`, download a picture of a cat and save it to the `media` directory as `cat.jpg`.
- In your **about page**, add in the `<img>` tag to display the picture of the cat, to ensure that your media is being served correctly. Keep the static image of Rango in your index page so that your app has working examples of both static and media files.

Static and Media Files

Remember: static files, as the name implies, do not change. These files form the core components of your website. Media files are user defined; and as such, they may change often!

An example of a static file could be a stylesheet file, which determines the appearance of your app's webpages. An example of a media file could be a user profile image, which is uploaded by the user when they create an account on your app.

5. Models and Databases

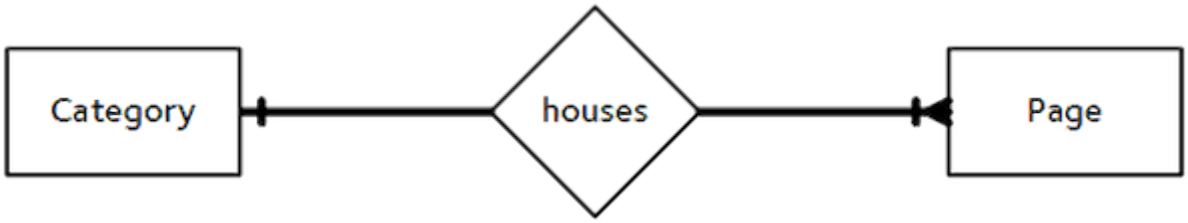
When you think of databases, you will usually think of the *Structured Query Language (SQL)*, the common means with which we query the database for the data we require. With Django, querying an underlying database - which can store all sorts of data, such as your website's user details - is taken care of by the *Object Relational Mapper (ORM)*. In essence, data stored within a database table can be encapsulated within a *model*. A model is a Python object that describes your database table's data. Instead of directly working on the database via SQL, you only need to manipulate the corresponding Python model object.

This chapter walks you through the basics of data management with Django and its ORM. You'll find it's incredibly easy to add, modify and delete data within your app's underlying database, and how straightforward it is to get data from the database to the Web browsers of your users.

5.1 Rango's Requirements

Before we get started, let's go over the data requirements for the Rango app that we are developing. Full requirements for the application are provided in detail earlier on, but to refresh your memory, let's quickly summarise our client's requirements.

- Rango is essentially a *web page directory* - a site containing links to other websites.
- There are a number of different *webpage categories* with each category housing a number of links. We assumed in the overview chapter that this is a one-to-many relationship. Check out the Entity Relationship diagram below.
- A category has a name, a number of visits, and a number of likes.
- A page refers to a category, has a title, URL and a number of views.



The Entity Relationship Diagram of Rango's two main entities.

5.2 Telling Django about Your Database

Before we can create any models, we need to set up our database with Django. In Django 1.9, a `DATABASES` variable is automatically created in your `settings.py` module when you set up a new project. It'll look similar to the following example.

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

We can pretty much leave this as is for our Rango app. You can see a `default` database that is powered by a lightweight database engine, SQLite (see the `ENGINE` option). The `NAME` entry for this database is the path to the database file, which is by default `db.sqlite3` in the root of your Django project.

Git Top Tip

If you are using Git, you might be tempted to add and commit the database file. This is not a good idea because if you are working on your app with other people, they are likely to change the database and this will cause endless conflicts.

Instead, add `db.sqlite3` to your `.gitignore` file so that it won't be added when you `git commit` and `git push`. You can also do this for other files like `*.pyc` and machine specific files.

Using other Database Engines

The Django database framework has been created to cater for a variety of different database backends, such as PostgresSQL, MySQL and Microsoft's SQL Server. For other database engines, other keys like `USER`, `PASSWORD`, `HOST` and `PORT` exist for you to configure the database with Django.

While we don't cover how to use other database engines in this book, there are guides online which show you how to do this. A good starting point is the official Django documentation.

Note that SQLite is sufficient for demonstrating the functionality of the Django ORM. When you find your app has become viral and has accumulated thousands of users, you may want to consider switching the database backend to something more robust.

5.3 Creating Models

With your database configured in `settings.py`, let's create the two initial data models for the Rango application. Models for a Django app are stored in the respective `models.py` module. This means that for Rango, models are stored within `rango/models.py`.

For the models themselves, we will create two classes - one class representing each model. Both must inherit from the `Model` base class, `django.db.models.Model`. The two Python classes will be the definitions for models representing *categories* and *pages*. Define the `Category` and `Page` model as follows.

```
class Category(models.Model):
    name = models.CharField(max_length=128, unique=True)

    def __str__(self):  # For Python 2, use __unicode__ too
        return self.name


class Page(models.Model):
    category = models.ForeignKey(Category)
    title = models.CharField(max_length=128)
    url = models.URLField()
    views = models.IntegerField(default=0)

    def __str__(self):  # For Python 2, use __unicode__ too
        return self.title
```

Check `import` Statements

At the top of the `models.py` module, you should see `from django.db import models`. If you don't see it, add it in.

`__str__()` or `__unicode__()`?

The `__str__()` and `__unicode__()` methods in Python generate a string representation of the class (similar to the `toString()` method in Java). In Python 2.x, strings are represented in ASCII format in the `__str__()` method. If you want Unicode support, then you need to also implement the `__unicode__()` method.

In Python 3.x, strings are Unicode by default - so you only need to implement the `__str__()` method.

When you define a model, you need to specify the list of fields and their associated types, along with any required or optional parameters. By default, all models have an auto-increment integer field called `id` which is automatically assigned and acts a primary key.

Django provides a comprehensive series of built-in field types. Some of the most commonly used are detailed below.

- `CharField`, a field for storing character data (e.g. strings). Specify `max_length` to provide a maximum number o characters the field can store.
- `URLField`, much like a `CharField`, but designed for storing resource URLs. You may also specify a `max_length` parameter.
- `IntegerField`, which stores integers.
- `DateField`, which stores a Python `datetime.date` object.

Other Field Types

Check out the Django documentation on model fields for a full listing of the Django field types you can use, along with details on the required and optional parameters that each has.

For each field, you can specify the `unique` attribute. If set to `True`, the given field's value must be unique throughout the underlying database table that is mapped to the associated model. For example, take a look at our `Category` model defined above. The field `name` has been set to unique, meaning that every category name must be unique. This means that you can use the field like a primary key.

You can also specify additional attributes for each field, such as stating a default value with the syntax `default='value'`, and whether the value for a field can be blank (or `NULL`) (`null=True`) or not (`null=False`).

Django provides three types of fields for forging relationships between models in your database. These are:

- `ForeignKey`, a field type that allows us to create a one-to-many relationship;
- `OneToOneField`, a field type that allows us to define a strict one-to-one relationship; and
- `ManyToManyField`, a field type which allows us to define a many-to-many relationship.

From our model examples above, the field `category` in model `Page` is of type `ForeignKey`. This allows us to create a one-to-many relationship with model/table `Category`, which is specified as an argument to the field's constructor.

Finally, it is good practice to implement the `__str__()` and/or `__unicode__()` methods. Without this method implemented when you go to `print` the object, it will show as `<Category: Category object>`. This isn't very useful when debugging or accessing the object - instead the code above will print, for example, `<Category: Python>` for the Python category. It is also helpful when we go to use the Admin Interface because Django will display the string representation of the object.

5.4 Creating and Migrating the Database

With our models defined in `models.py`, we can now let Django work its magic and create the tables in the underlying database. Django provides what is called a *migration tool* to help us set up and update the database to reflect any changes to your models. For example, if you were to add a new field then you can use the migration tools to update the database.

Setting up

First of all, the database must be *initialised*. This means creating the database and all the associated tables so that data can then be stored within it. To do this, you must open a terminal or command prompt, and navigate to your project's root directory - where `manage.py` is stored. Run the following command, *bearing in mind that the output may vary from what you see below.*

```
$ python manage.py migrate

Operations to perform:
  Apply all migrations: admin, contenttypes, auth, sessions
Running migrations:
  Rendering model states... DONE
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
  Applying auth.0004_alter_user_username_opts... OK
  Applying auth.0005_alter_user_last_login_null... OK
  Applying auth.0006_require_contenttypes_0002... OK
  Applying auth.0007_alter_validators_add_error_messages... OK
  Applying sessions.0001_initial... OK
```

All apps that are installed in your Django project (check `INSTALLED_APPS` in `settings.py`) will update their database representations with this command. After this command is issued, you should then see a `db.sqlite3` file in your Django project's root.

Next, create a superuser to manage the database. Run the following command.

```
$ python manage.py createsuperuser
```

The superuser account will be used to access the Django admin interface, used later on in this chapter. Enter a username for the account, e-mail address and provide a password when prompted. Once completed, the script should finish successfully. Make sure you take a note of the username and password for your superuser account.

Creating and Updating Models/Tables

Whenever you make changes to your app's models, you need to *register* the changes via the `makemigrations` command in `manage.py`. Specifying the `rango` app as our target, we then issue the following command from our Django project's root directory.

```
$ python manage.py makemigrations rango

Migrations for 'rango':
  0001_initial.py:
    - Create model Category
    - Create model Page
```

Upon the completion of this command, check the `rango/migrations` directory to see that a Python script has been created. It's called `0001_initial.py`, which contains all the necessary details to create your database schema for that particular migration.

Checking the Underlying SQL

If you want to check out the underlying SQL that the Django ORM issues to the database engine for a given migration, you can issue the following command.

```
$ python manage.py sqlmigrate rango 0001
```

In this example, `rango` is the name of your app, and `0001` is the migration you wish to view the SQL code for. Doing this allows you to get a better understanding of what exactly is going on at the database layer, such as what tables are created. You may find for complex database schemas including a many-to-many relationship that additional tables are created for you.

After you have created migrations for your app, you need to commit them to the database. Do so by once again issuing the `migrate` command.

```
$ python manage.py migrate

Operations to perform:
  Apply all migrations: admin, rango, contenttypes, auth, sessions
Running migrations:
  Rendering model states... DONE
  Applying rango.0001_initial... OK
```

This output confirms that the database tables have been created in your database, and you are good to go.

However, you may have noticed that our `Category` model is currently lacking some fields that were specified in Rango's requirements. **Don't worry about this, as these will be added in later, allowing you to go through the migration process again.**

5.5 Django Models and the Shell

Before we turn our attention to demonstrating the Django admin interface, it's worth noting that you can interact with Django models directly from the Django shell - a very useful tool for debugging purposes. We'll demonstrate how to create a `Category` instance using this method.

To access the shell, we need to call `manage.py` from within your Django project's root directory once more. Run the following command.

```
$ python manage.py shell
```

This will start an instance of the Python interpreter and load in your project's settings for you. You can then interact with the models, with the following terminal session demonstrating this functionality. Check out the inline commentary that we added to see what each command achieves. Note there are slight differences between what Django 1.9 and Django 1.10 return – these are both demonstrated below, complete with commentary.

```
# Import the Category model from the Rango application
>>> from rango.models import Category

# Show all the current categories
>>> print(Category.objects.all())
# The output examples below are for both Django 1.9 and Django 1.10.
# Both denote the same thing, that no categories have been defined.
[]  # Django 1.9 output -- an empty list.
<QuerySet []>  # Django 1.10 output -- an empty QuerySet object.

# Create a new category object, and save it to the database.
>>> c = Category(name="Test")
```

```
>>> c.save()

# Now list all the category objects stored once more.
>>> print(Category.objects.all())
# The output examples below are for both Django 1.9 and Django 1.10.
# You'll now see a 'test' category in both output examples.
[<Category: test>]  # Django 1.9
<QuerySet [<Category: test>]  # Django 1.10

# Quit the Django shell.
>>> quit()
```

In the example, we first import the model that we want to manipulate. We then print out all the existing categories. As our underlying `Category` table is empty, an empty list is returned. Then we create and save a `Category`, before printing out all the categories again. This second `print` then shows the new `Category` just added. Note the name, `Test` appears in the second `print` - this is your `__str__()` or `__unicode__()` method at work!

Complete the Official Tutorial

The example above is only a very basic taster on database related activities you can perform in the Django shell. If you have not done so already, it's now a good time to complete part two of the official Django Tutorial to learn more about interacting with models. Also check out the official Django documentation on the list of available commands for working with models.

5.6 Configuring the Admin Interface

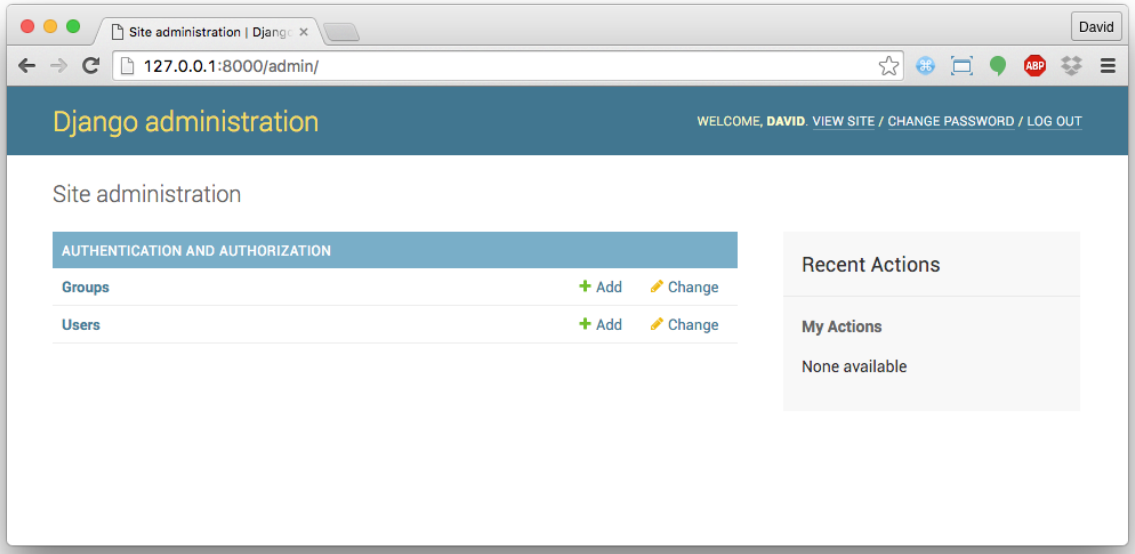
One of the standout features of Django is the built-in, Web-based administrative (or *admin*) interface that allows you to browse, edit and delete data represented as model instances (from the corresponding database tables). In this section, we'll be setting the admin interface up so you can see the two Rango models you have created so far.

Setting everything up is relatively straightforward. In your project's `settings.py` module, you will notice that one of the preinstalled apps (within the `INSTALLED_APPS` list) is `django.contrib.admin`. Furthermore, there is a `urlpattern` that matches `admin/` within your project's `urls.py` module.

By default, things are pretty much ready to go. Start the Django development server in the usual way with the following command.

```
$ python manage.py runserver
```

Navigate your Web browser to `http://127.0.0.1:8000/admin/`. You are then presented with a login prompt. Login using the credentials you created previously with the `$ python manage.py createsuperuser` command. You are then presented with an interface looking similar to the one shown below.



The Django admin interface, sans Rango models.

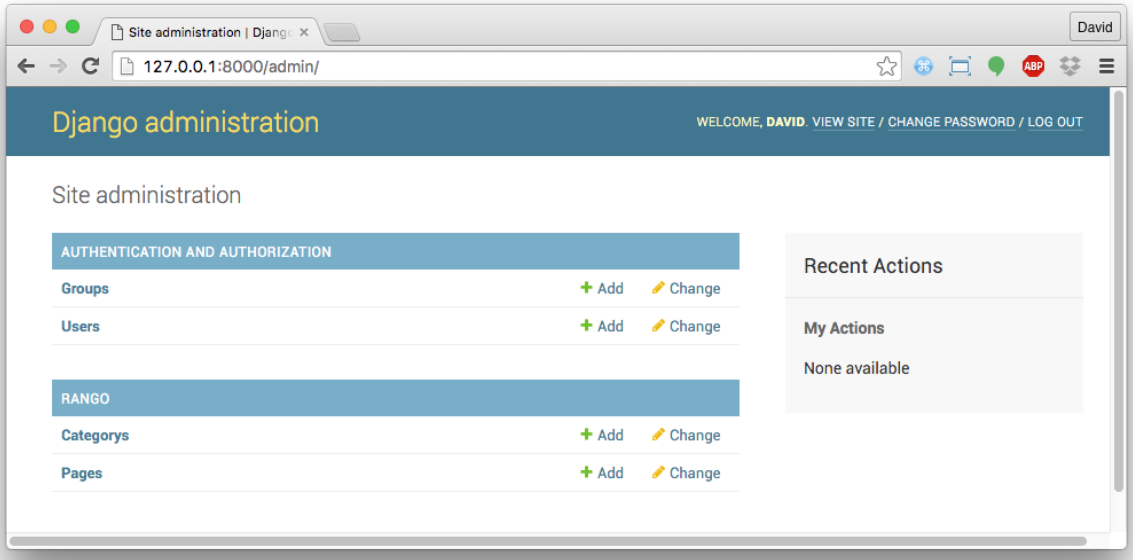
While this looks good, we are missing the `Category` and `Page` models that were defined for the Rango app. To include these models, we need to give Django some help.

To do this, open the file `rango/admin.py`. With an `include` statement already present, modify the module so that you `register` each class you want to include. The example below registers both the `Category` and `Page` class to the admin interface.

```
from django.contrib import admin
from rango.models import Category, Page

admin.site.register(Category)
admin.site.register(Page)
```

Adding further classes which may be created in the future is as simple as adding another call to the `admin.site.register()` method.

With these changes saved, restart the Django development server and revisit the admin interface at `http://127.0.0.1:8000/admin/`. You will now see the `Category` and `Page` models, as shown below.



The Django admin interface, complete with Rango models.

Try clicking the `Categorys` link within the `Rango` section. From here, you should see the `test` category that we created earlier via the Django shell.

Experiment with the Admin Interface

You'll be using the admin interface quite a bit to verify data is stored correctly as you develop the Rango app. Experiment with it, and see how it all works. The interface is self-explanatory and straightforward to understand.

Delete the `test` category that was previously created. We'll be populating the database shortly with more example data.

User Management

The Django admin interface is your port of call for user management, through the Authentication and Authorisation section. Here, you can create, modify and delete user accounts, and varying privilege levels.

Plural vs. Singular Spellings

Note the typo within the admin interface (`Categorys`, not `Categories`). This typo can be fixed by adding a nested `Meta` class into your model definitions with the `verbose_name_plural` attribute. Check out a modified version of the `Category` model below for an example, and Django's official documentation on models for more information about what can be stored within the `Meta` class.

```
class Category(models.Model):
    name = models.CharField(max_length=128, unique=True)

    class Meta:
        verbose_name_plural = 'Categories'

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

Expanding `admin.py`

It should be noted that the example `admin.py` module for your Rango app is the most simple, functional example available. However you can customise the Admin interface in a number of ways. Check out the official Django documentation on the admin interface for more information if you're interested.

5.7 Creating a Population Script

Entering test data into your database tends to be a hassle. Many developers will add in some bogus test data by randomly hitting keys, like `wTFzmN00bz7`. Rather than do this, it is better to write a script to automatically populate the database with realistic and credible data. This is because when you go to demo or test your app, you'll see some good examples in the database. Also, if you are deploying the app or sharing it with collaborators, then you/they won't have to go through the process of putting in sample data. It's therefore good practice to create what we call a *population script*.

To create a population script for Rango, start by creating a new Python module within your Django project's root directory (e.g. `<workspace>/tango_with_django_project/`). Create the `populate_rango.py` file and add the following code.

```
import os
os.environ.setdefault('DJANGO_SETTINGS_MODULE',
                      'tango_with_django_project.settings')

import django
django.setup()
from rango.models import Category, Page

def populate():
    # First, we will create lists of dictionaries containing the pages
    # we want to add into each category.
    # Then we will create a dictionary of dictionaries for our categories.
    # This might seem a little bit confusing, but it allows us to iterate
    # through each data structure, and add the data to our models.

    python_pages = [
        {"title": "Official Python Tutorial",
         "url":"http://docs.python.org/2/tutorial/"},
        {"title":"How to Think like a Computer Scientist",
         "url":"http://www.greenteapress.com/thinkpython/"},
        {"title":"Learn Python in 10 Minutes",
         "url":"http://www.korokithakis.net/tutorials/python/"} ]

    django_pages = [
        {"title":"Official Django Tutorial",
         "url":"https://docs.djangoproject.com/en/1.9/intro/tutorial01/"},
        {"title":"Django Rocks",
         "url":"http://www.djangorocks.com/"},
        {"title":"How to Tango with Django",
         "url":"http://www.tangowithdjango.com/"} ]

    other_pages = [
        {"title":"Bottle",
         "url":"http://bottlepy.org/docs/dev/"},
        {"title":"Flask",
         "url":"http://flask.pocoo.org"} ]

    cats = {"Python": {"pages": python_pages},
            "Django": {"pages": django_pages},
            "Other Frameworks": {"pages": other_pages} }

    # If you want to add more catergories or pages,
```

```
43      # add them to the dictionaries above.
44
45      # The code below goes through the cats dictionary, then adds each category,
46      # and then adds all the associated pages for that category.
47      # if you are using Python 2.x then use cats.iteritems() see
48      # http://docs.quantifiedcode.com/python-anti-patterns/readability/
49      # for more information about how to iterate over a dictionary properly.
50
51      for cat, cat_data in cats.items():
52          c = add_cat(cat)
53          for p in cat_data["pages"]:
54              add_page(c, p["title"], p["url"])
55
56      # Print out the categories we have added.
57      for c in Category.objects.all():
58          for p in Page.objects.filter(category=c):
59              print("- {0} - {1}".format(str(c), str(p)))
60
61  def add_page(cat, title, url, views=0):
62      p = Page.objects.get_or_create(category=cat, title=title)[0]
63      p.url=url
64      p.views=views
65      p.save()
66      return p
67
68  def add_cat(name):
69      c = Category.objects.get_or_create(name=name)[0]
70      c.save()
71      return c
72
73  # Start execution here!
74  if __name__ == '__main__':
75      print("Starting Rango population script...")
76      populate()
```

Understand this Code!

To reiterate, don't simply copy, paste and leave. Add the code to your new module, and then step through line by line to work out what is going on. It'll help with your understanding.

We've explanations below - hopefully you'll learn something new!

You should also note that when you see line numbers along side the code, it indicates that we have listed the entire file, rather than code fragments. It also makes things more difficult for you to copy and paste!

While this looks like a lot of code, what is going on is essentially a series of function calls to two small functions, `add_page()` and `add_cat()` defined towards the end of the module. Reading through the code, we find that execution starts at the *bottom* of the module - look at lines 75 and 76. This is because above this point, we define functions; these are not executed unless we call them. When the interpreter hits `if __name__ == '__main__'`, we call the `populate()` function.

What does `__name__ == '__main__'` Represent?

The `__name__ == '__main__'` trick is a useful one that allows a Python module to act as either a reusable module or a standalone Python script. Consider a reusable module as one that can be imported into other modules (e.g. through an `import` statement), while a standalone Python script would be executed from a terminal/Command Prompt by entering `python module.py`.

Code within a conditional `if __name__ == '__main__'` statement will therefore only be executed when the module is run as a standalone Python script. Importing the module will not run this code; any classes or functions will however be fully accessible to you.

Importing Models

When importing Django models, make sure you have imported your project's settings by importing `django` and setting the environment variable `DJANGO_SETTINGS_MODULE` to be your project's setting file, as demonstrated in lines 1 to 6 above. You then call `django.setup()` to import your Django project's settings.

If you don't perform this crucial step, you'll **get an exception when attempting to import your models**. **This is because the necessary Django infrastructure has not yet been initialised.** This is why we import `Category` and `Page` *after* the settings have been loaded on line 8.

The `for` loop occupying lines 51-54 is responsible for the calling the `add_cat()` and `add_page()` functions repeatedly. These functions are in turn responsible for the creation of new categories and pages. `populate()` keeps tabs on categories that are created. As an example, a reference to a new

category is stored in local variable `c` - check line 52 above. This is stored because a `Page` requires a `Category` reference. After `add_cat()` and `add_page()` are called in `populate()`, the function concludes by looping through all new `Category` and associated `Page` objects, displaying their names on the terminal.

Creating Model Instances

We make use of the convenience `get_or_create()` method for creating model instances in the population script above. As we don't want to create duplicates of the same entry, we can use `get_or_create()` to check if the entry exists in the database for us. If it doesn't exist, the method creates it. If it does, then a reference to the specific model instance is returned.

This helper method can remove a lot of repetitive code for us. Rather than doing this laborious check ourselves, we can make use of code that does exactly this for us.

The `get_or_create()` method returns a tuple of `(object, created)`. The first element `object` is a reference to the model instance that the `get_or_create()` method creates if the database entry was not found. The entry is created using the parameters you pass to the method - just like `category`, `title`, `url` and `views` in the example above. If the entry already exists in the database, the method simply returns the model instance corresponding to the entry. `created` is a boolean value; `True` is returned if `get_or_create()` had to create a model instance.

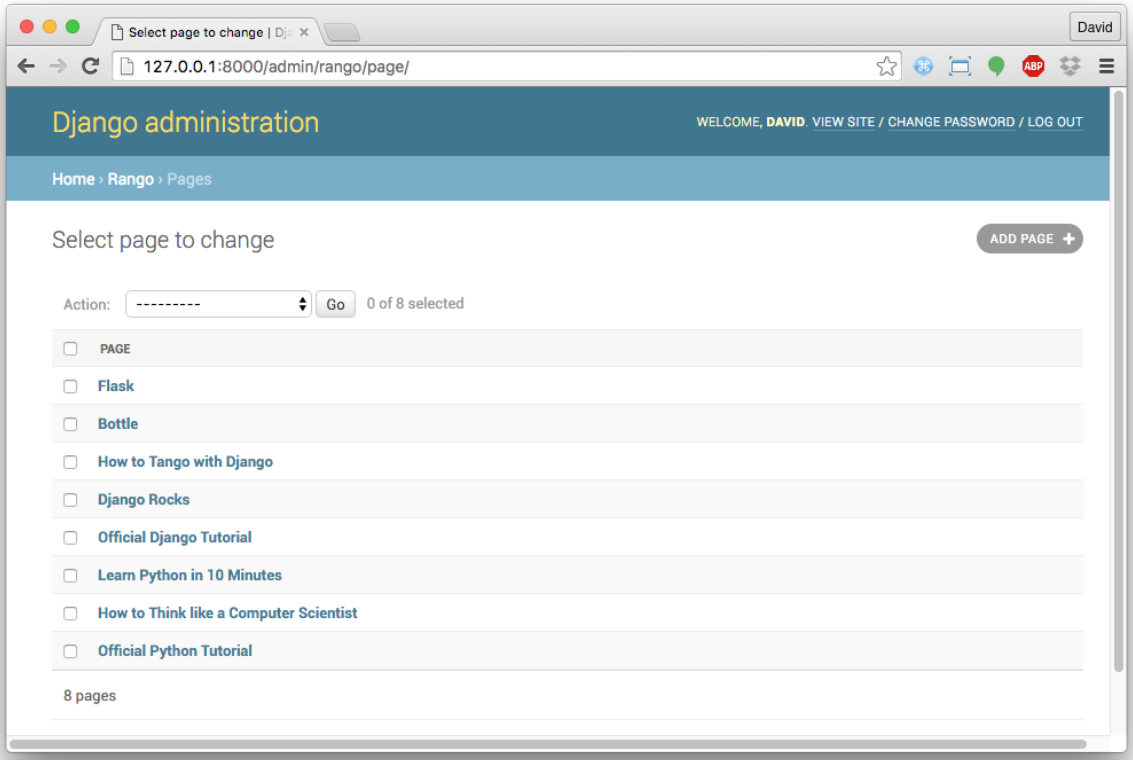
This explanation therefore means that the `[0]` at the end of our call to the `get_or_create()` returns the object reference only. Like most other programming language data structures, Python tuples use zero-based numbering.

You can check out the official Django documentation for more information on the handy `get_or_create()` method.

When saved, you can then run your new populations script by changing the present working directory in a terminal to the Django project's root. It's then a simple case of executing the command `$ python populate_rango.py`. You should then see output similar to that shown below – the order in which categories are added may vary depending upon how your computer is set up.

```
$ python populate_rango.py

Starting Rango population script...
- Python - Official Python Tutorial
- Python - How to Think like a Computer Scientist
- Python - Learn Python in 10 Minutes
- Django - Official Django Tutorial
- Django - Django Rocks
- Django - How to Tango with Django
- Other Frameworks - Bottle
- Other Frameworks - Flask
```

Next, verify that the population script actually populated the database. Restart the Django development server, navigate to the admin interface (at `http://127.0.0.1:8000/admin/`) and check that you have some new categories and pages. Do you see all the pages if you click `Pages`, like in the figure shown below?



The Django admin interface, showing the `Page` model populated with the new population script. Success!

While creating a population script may take time, you will save yourself time in the long run. When deploying your app elsewhere, running the population script after setting everything up means you can start demonstrating your app straight away. You'll also find it very handy when it comes to unit testing your code.

5.8 Workflow: Model Setup

Now that we've covered the core principles of dealing with Django's ORM, now is a good time to summarise the processes involved in setting everything up. We've split the core tasks into separate sections for you. Check this section out when you need to quickly refresh your mind of the different steps.

Setting up your Database

With a new Django project, you should first tell Django about the database you intend to use (i.e. configure `DATABASES` in `settings.py`). You can also register any models in the `admin.py` module of your app to make them accessible via the admin interface.

Adding a Model

The workflow for adding models can be broken down into five steps.

1. First, create your new model(s) in your Django application's `models.py` file.
2. Update `admin.py` to include and register your new model(s).
3. Perform the migration `$ python manage.py makemigrations <app_name>`.
4. Apply the changes `$ python manage.py migrate`. This will create the necessary infrastructure within the database for your new model(s).
5. Create/edit your population script for your new model(s).

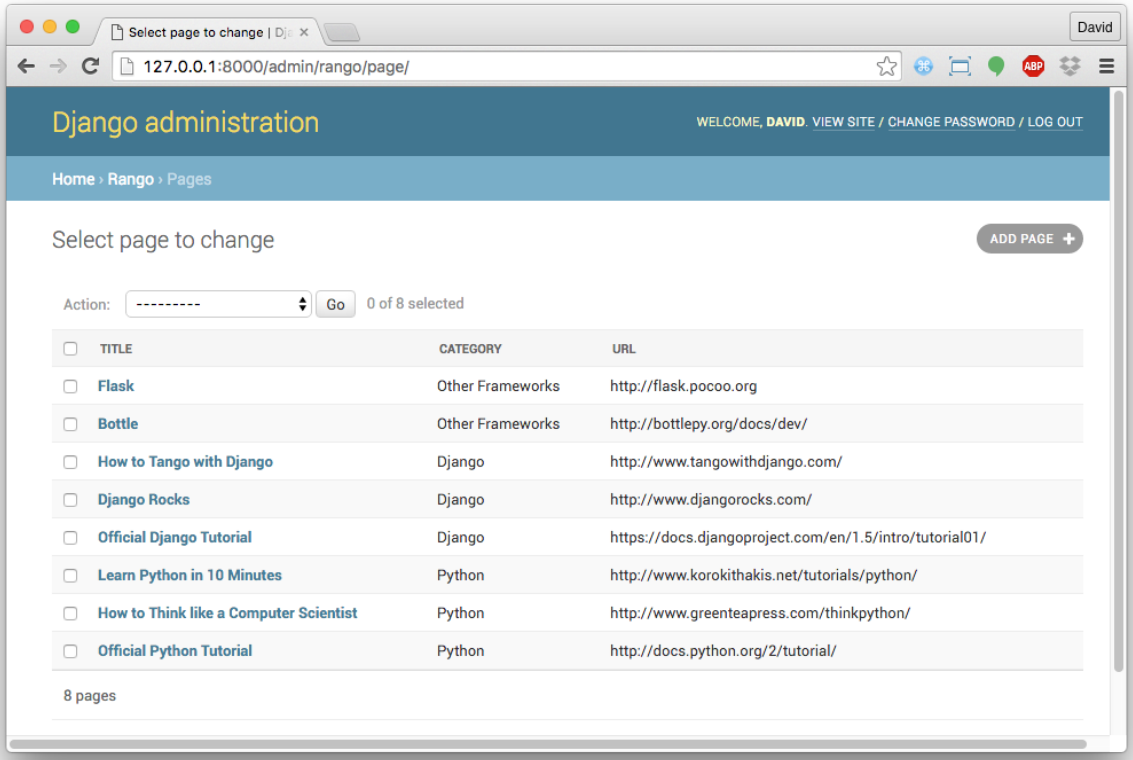
There will be times when you will have to delete your database – sometimes it's easier to just start afresh. When you want to do this, do the the following. Note that for this tutorial, you are using a SQLite database – Django does support a variety of other database engines.

1. If you're running it, stop your Django development server.
2. For an SQLite database, delete the `db.sqlite3` file in your Django project's directory. It'll be in the same directory as the `manage.py` file.
3. If you have changed your app's models, you'll want to run the `$ python manage.py makemigrations <app_name>` command, replacing `<app_name>` with the name of your Django app (i.e. `rango`). Skip this if your models have not changed.
4. Run the `$ python manage.py migrate` to create a new database file (if you are running SQLite), and migrate database tables to the database.
5. Create a new admin account with the `$ python manage.py createsuperuser` command.
6. Finally, run your population script again to insert credible test data into your new database.

Exercises

Now that you've completed this chapter, try out these exercises to reinforce and practice what you have learnt. **Once again, note that the following chapters will have expected you to have completed these exercises! If you're stuck, there are some hints to help you complete the exercises below**.

- Update the `Category` model to include the additional attributes `views` and `likes` where the `default` values for each are both zero (`0`).
- Make the migrations for your app and then migrate your database to commit the changes.
- Update your population script so that the `Python` category has `128` views and `64` likes, the `Django` category has `64` views and `32` likes, and the `Other Frameworks` category has `32` views and `16` likes.
- Delete and recreate your database, populating it with your updated population script.
- Complete parts two and seven of the official Django tutorial. These sections will reinforce what you've learnt on handling databases in Django, and show you additional techniques to customising the Django admin interface.
- Customise the admin interface. Change it in such a way so that when you view the `Page` model, the table displays the `category`, the `name` of the page and the `url` - just like in the screenshot shown below. You will need to complete the previous exercises or at least go through the official Django Tutorial to complete this exercise.



The updated admin interface `Page` view, complete with columns for category and URL.

Exercise Hints

If you require some help or inspiration to complete these exercises done, here are some hints.

- Modify the `Category` model by adding two `IntegerFields`: `views` and `likes`.
- In your population script, you can then modify the `add_cat()` function to manipulate the value of the new `views` and `likes` fields in the `Category` model.
  - You'll need to add two parameters to the definition of `add_cat()` so that `views` and `likes` values can be passed to the function, as well as a `name` for the category.
  - You can then use these parameters to set the `views` and `likes` fields within the new `Category` model instance you create within the `add_cat()` function. The model instance is assigned to variable `c` in the population script, as defined earlier in this chapter. As an example, you can access the `likes` field using the notation `c.likes`. Don't forget to `save()` the instance!
  - You then need to update the `cats` dictionary in the `populate()` function of your population script. Look at the dictionary. Each key/value pairing represents the *name* of the category as the key, and an additional dictionary containing additional information relating to the category as the *value*. You'll want to modify this dictionary to include `views` and `likes` for each category.
  - The final step involves you modifying how you call the `add_cat()` function. You now have three parameters to pass (`name`, `views` and `likes`); your code currently provides only the `name`. You need to add the additional two fields to the function call. If you aren't sure how the `for` loop works over dictionaries, check out this online Python tutorial. From here, you can figure out how to access the `views` and `likes` values from your dictionary.
- After your population script has been updated, you can move on to customising the admin interface. You will need to edit `rango/admin.py` and create a `PageAdmin` class that inherits from `admin.ModelAdmin`.
  - Within your new `PageAdmin` class, add `list_display = ('title', 'category', 'url')`.
  - Finally, register the `PageAdmin` class with Django's admin interface. You should modify the line `admin.site.register(Page)`. Change it to `admin.site.register(Page, PageAdmin)` in Rango's `admin.py` file.

Tests

We have written a few tests to check if you have completed the exercises. To check your work so far, download the `tests.py` script from our GitHub repository, and save it within your `rango` app directory.

To run the tests, issue the following command in the terminal or Command Prompt.

```
$ python manage.py test rango
```

If you are interested in learning about automated testing, now is a good time to check out the chapter on testing. The chapter runs through some of the basics on how you can write tests to automatically check the integrity of your code.

6. Models, Templates and Views

Now that we have the models set up and populated the database with some sample data, we can now start connecting the models, views and templates to serve up dynamic content. In this chapter, we will go through the process of showing categories on the main page, and then create dedicated category pages which will show the associated list of links.

6.1 Workflow: Data Driven Page

To do this there are five main steps that you must undertake to create a data driven webpage in Django.

1. In `views.py` file import the models you wish to use.
2. In the view function, query the model to get the data you want to present.
3. Then pass the results from your model into the template's context.
4. Create/modify the template so that it displays the data from the context.
5. If you have not done so already, map a URL to your view.

These steps highlight how we need to work within Django's framework to bind models, views and templates together.

6.2 Showing Categories on Rango's Homepage

One of the requirements regarding the main page was to show the top five rango'ed categories. To fulfil this requirement, we will go through each of the above steps.

Importing Required Models

First, we need to complete step one. Open `rango/views.py` and at the top of the file, after the other imports, import the `Category` model from Rango's `models.py` file.

```
# Import the Category model
from rango.models import Category
```

Modifying the Index View

Here we will complete step two and step three, where we need to modify the view `index()` function. Remember that the `index()` function is responsible for the main page view. Modify `index()` as follows:

```
def index(request):
    # Query the database for a list of ALL categories currently stored.
    # Order the categories by no. likes in descending order.
    # Retrieve the top 5 only - or all if less than 5.
    # Place the list in our context_dict dictionary
    # that will be passed to the template engine.

    category_list = Category.objects.order_by('-likes')[:5]
    context_dict = {'categories': category_list}

    # Render the response and send it back!
    return render(request, 'rango/index.html', context_dict)
```

Here, the expression `Category.objects.order_by('-likes')[:5]` queries the `Category` model to retrieve the top five categories. You can see that it uses the `order_by()` method to sort by the number of `likes` in descending order. The `-` in `-likes` denotes that we would like them in descending order (if we removed the `-` then the results would be returned in ascending order). Since a list of `Category` objects will be returned, we used Python's list operators to take the first five objects from the list (`[:5]`) to return a subset of `Category` objects.

With the query complete, we passed a reference to the list (stored as variable `category_list`) to the dictionary, `context_dict`. This dictionary is then passed as part of the context for the template engine in the `render()` call.

Warning

For this to work, you will have had to complete the exercises in the previous chapter where you need to add the field `likes` to the `Category` model.

Modifying the Index Template

With the view updated, we can complete the fourth step and update the template `rango/index.html`, located within your project's `templates` directory. Change the HTML so that it looks like the example shown below.

```
<!DOCTYPE html>
{% load staticfiles %}
<html>
<head>
    <title>Rango</title>
</head>

<body>
    <h1>Rango says...</h1>
    <div>hey there partner!</div>

    <div>
    {% if categories %}
    <ul>
        {% for category in categories %}
            <li>{{ category.name }}</li>
        {% endfor %}
    </ul>
    {% else %}
        <strong>There are no categories present.</strong>
    {% endif %}
    </div>

    <div>
        <a href="/rango/about/">About Rango</a><br />
        <img src="{% static "images/rango.jpg" %}" alt="Picture of Rango" />
    </div>
</body>
</html>
```

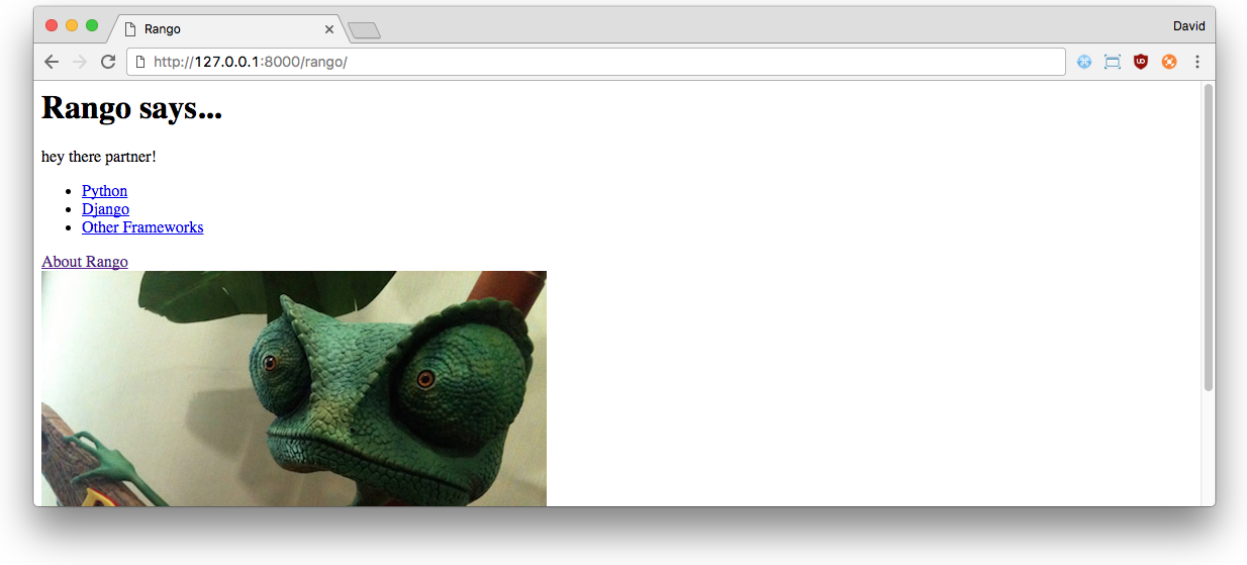
Here, we make use of Django's template language to present the data using `if` and `for` control statements. Within the `<body>` of the page, we test to see if `categories` - the name of the context variable containing our list - actually contains any categories (`{% if categories %}`).

If so, we proceed to construct an unordered HTML list (within the `<ul>` tags). The `for` loop (`{% for category in categories %}`) then iterates through the list of results, and outputs each category's name (`{{ category.name }}`) within a pair of `<li>` tags to indicate a list element.

If no categories exist, a message is displayed instead indicating that no categories are present.

As the example shows in Django's template language, all commands are enclosed within the tags `{%` and `%}`, while variables are referenced within `{{` and `}}` brackets.

If you now visit Rango's homepage at `<http://127.0.0.1:8000/rango/>`, you should see a list of categories underneath the page title just like in the figure below.



The Rango homepage - now dynamically generated - showing a list of categories.

6.3 Creating a Details Page

According to the specifications for Rango, we also need to show a list of pages that are associated with each category. We have a number of challenges here to overcome. A new view must be created, which should be parameterised. We also need to create URL patterns and URL strings that encode category names.

URL Design and Mapping

Let's start by considering the URL problem. One way we could handle this problem is to use the unique ID for each category within the URL. For example, we could create URLs like `/rango/category/1/` or `/rango/category/2/`, where the numbers correspond to the categories with unique IDs 1 and 2 respectively. However, it is not possible to infer what the category is about just from the ID.

Instead, we could use the category name as part of the URL. For example, we can imagine that the URL `/rango/category/python/` would lead us to a list of pages related to Python. This is a simple, readable and meaningful URL. If we go with this approach, we'll also have to handle categories that have multiple words, like 'Other Frameworks', etc.

Clean your URLs

Designing clean and readable URLs is an important aspect of web design. See Wikipedia's article on Clean URLs for more details.

To handle this problem we are going to make use of the `slugify` function provided by Django.

Update Category Table with a Slug Field

To make readable URLs, we need to include a slug field in the `Category` model. First we need to import the function `slugify` from Django that will replace whitespace with hyphens - for example, `"how do i create a slug in django"` turns into `"how-do-i-create-a-slug-in-django"`.

Unsafe URLs

While you can use spaces in URLs, it is considered to be unsafe to use them. Check out the Internet Engineering Task Force Memo on URLs to read more.

Next we need to override the `save()` method of the `Category` model, which we will call the `slugify` method and update the `slug` field with it. Note that every time the category name changes, the slug will also change. Update your model, as shown below, and add in the import.

```
from django.template.defaultfilters import slugify
...
class Category(models.Model):
    name = models.CharField(max_length=128, unique=True)
    views = models.IntegerField(default=0)
    likes = models.IntegerField(default=0)
    slug = models.SlugField()

    def save(self, *args, **kwargs):
        self.slug = slugify(self.name)
        super(Category, self).save(*args, **kwargs)

    class Meta:
        verbose_name_plural = 'categories'

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

Now that the model has been updated, the changes must now be propagated to the database. However, since data already exists within the database, we need to consider the implications of the change. Essentially, for all the existing category names, we want to turn them into slugs (which is performed when the record is initially saved). When we update the models via the migration tool, it will add the `slug` field and provide the option of populating the field with a default value. Of course, we want a specific value for each entry - so we will first need to perform the migration, and then re-run the population script. This is because the population script will explicitly call the `save()` method on each entry, triggering the 'save()' as implemented above, and thus update the slug accordingly for each entry.

To perform the migration, issue the following commands (as detailed in the Models and Databases Workflow).

```
$ python manage.py makemigrations rango
$ python manage.py migrate
```

Since we did not provide a default value for the slug and we already have existing data in the model, the `migrate` command will give you two options. Select the option to provide a default, and enter an empty string – denoted by two quote marks (i.e. `''`). Run the population script again, which will then update the slug fields.

```
$ python populate_rango.py
```

Now run the development server with the command `$ python manage.py runserver`, and inspect the data in the models with the admin interface at `http://127.0.0.1:8000/admin/`.

If you go to add in a new category via the admin interface you may encounter a problem, or two!

1. Let's say we added in the category, `Python User Groups`. If you do so, and try to save the record Django will not let you save it unless you also fill in the slug field too. While we could type in `python-user-groups` this is error prone. It would be better to have the slug automatically generated.
2. The next problem arises if we have one category called `Django` and one called `django`. Since the `slugify()` makes the slugs lower case it will not be possible to identify which category corresponds to the `django` slug.

To solve the first problem, we can either update our model so that the slug field allows blank entries, i.e.:

```
slug = models.SlugField(blank=True)
```

or we can customise the admin interface so that it automatically pre-populates the slug field as you type in the category name. To do this, update `rango/admin.py` with the following code.

```
from django.contrib import admin
from rango.models import Category, Page
...
# Add in this class to customise the Admin Interface
class CategoryAdmin(admin.ModelAdmin):
    prepopulated_fields = {'slug':('name',)}

# Update the registration to include this customised interface
admin.site.register(Category, CategoryAdmin)
...
```

Try out the admin interface and add in a new category.

Now that we have addressed the first problem, we can ensure that the slug field is also unique, by adding the constraint to the slug field.

```
slug = models.SlugField(unique=True)
```

Now that we have added in the slug field we can now use the slugs to uniquely identify each category. We could have added the unique constraint earlier, but if we performed the migration and set everything to be an empty string by default it would have raised an error. This is because the unique constraint would have been violated. We could have deleted the database and then recreated everything - but that is not always desirable.

Migration Woes

It's always best to plan out your database in advance and avoid changing it. Making a population script means that you easily recreate your database if you need to delete it.

Sometimes it is just better to just delete the database and recreate everything than try and work out where the conflict is coming from. A neat exercise is to write a script to output the data in the database so that any changes you make can be saved out into a file that can be read in later.

Category Page Workflow

To implement the category pages so that they can be accessed via `/rango/category/<category-name-slug>/` we need to make a number of changes and undertake the following steps:

1. Import the `Page` model into `rango/views.py`.
2. Create a new view in `rango/views.py` called `show_category()`. The `show_category()` view will take an additional parameter, `category_name_url` which will store the encoded category name.

    - We will need helper functions to encode and decode the `category_name_url`.
3. Create a new template, `templates/rango/category.html`.
4. Update Rango's `urlpatterns` to map the new `category` view to a URL pattern in `rango/urls.py`.

We'll also need to update the `index()` view and `index.html` template to provide links to the category page view.

Category View

In `rango/views.py`, we first need to import the `Page` model. This means we must add the following import statement at the top of the file.

```
from rango.models import Page
```

Next, we can add our new view, `show_category()`.

```
def show_category(request, category_name_slug):
    # Create a context dictionary which we can pass
    # to the template rendering engine.
    context_dict = {}

    try:
        # Can we find a category name slug with the given name?
        # If we can't, the .get() method raises a DoesNotExist exception.
        # So the .get() method returns one model instance or raises an exception.
        category = Category.objects.get(slug=category_name_slug)

        # Retrieve all of the associated pages.
        # Note that filter() will return a list of page objects or an empty list
        pages = Page.objects.filter(category=category)

        # Adds our results list to the template context under name pages.
        context_dict['pages'] = pages
        # We also add the category object from
        # the database to the context dictionary.
        # We'll use this in the template to verify that the category exists.
        context_dict['category'] = category
    except Category.DoesNotExist:
        # We get here if we didn't find the specified category.
        # Don't do anything -
```

```
        # the template will display the "no category" message for us.
        context_dict['category'] = None
        context_dict['pages'] = None

    # Go render the response and return it to the client.
    return render(request, 'rango/category.html', context_dict)
```

Our new view follows the same basic steps as our `index()` view. We first define a context dictionary and then attempt to extract the data from the models, and add the relevant data to the context dictionary. We determine which category by using the value passed as parameter `category_name_slug` to the `show_category()` view function. If the category slug is found in the `Category` model, we can then pull out the associated pages, and add this to the context dictionary, `context_dict`.

Category Template

Now let's create our template for the new view. In `<workspace>/tango_with_django_project/templates/rango/` directory, create `category.html`. In the new file, add the following code.

```
<!DOCTYPE html>
<html>
<head>
    <title>Rango</title>
</head>
<body>
    <div>
    {% if category %}
        <h1>{{ category.name }}</h1>
        {% if pages %}
            <ul>
            {% for page in pages %}
                <li><a href="{{ page.url }}">{{ page.title }}</a></li>
            {% endfor %}
            </ul>
        {% else %}
            <strong>No pages currently in category.</strong>
        {% endif %}
    {% else %}
        The specified category does not exist!
    {% endif %}
    </div>
</body>
</html>
```

The HTML code example again demonstrates how we utilise the data passed to the template via its context through the tags `{{ }}`. We access the `category` and `pages` objects, and their fields e.g. `category.name` and `page.url`.

If the `category` exists, then we check to see if there are any pages in the category. If so, we iterate through the pages using the `{% for page in pages %}` template tags. For each page in the `pages` list, we present their `title` and `url` attributes. This is displayed in an unordered HTML list (denoted by the `<ul>` tags). If you are not too familiar with HTML then check out the HTML Tutorial by W3Schools.com to learn more about the different tags.

Note on Conditional Template Tags

The Django template conditional tag - `{% if %}` - is a really neat way of determining the existence of an object within the template's context. Make sure you check the existence of an object to avoid errors.

Placing conditional checks in your templates - like `{% if category %}` in the example above - also makes sense semantically. The outcome of the conditional check directly affects the way in which the rendered page is presented to the user. Remember, presentational aspects of your Django appls should be encapsulated within templates.

Parameterised URL Mapping

Now let's have a look at how we actually pass the value of the `category_name_url` parameter to the `show_category()` function. To do so, we need to modify Rango's `urls.py` file and update the `urlpatterns` tuple as follows.

```
urlpatterns = [
    url(r'^$', views.index, name='index'),
    url(r'^about/$', views.about, name='about'),
    url(r'^category/(?P<category_name_slug>[\w\-]+)/$',
        views.show_category, name='show_category'),
]
```

We have added in a rather complex entry that will invoke `view.show_category()` when the URL pattern `r'^category/(?P<category_name_slug>[\w\-]+)/$'` is matched.

There are two things to note here. First we have added a parameter name with in the URL pattern, i.e. `<category_name_slug>`, which we will be able to access in our view later on. When you create a parameterised URL you need to ensure that the parameters that you include in the URL are declared in the corresponding view. The next thing to note is that the regular expression `[\w\-]+)` will look for any sequence of alphanumeric characters e.g. `a-z`, `A-Z`, or `0-9` denoted by `\w` and any hyphens (-) denoted by `\-`, and we can match as many of these as we like denoted by the `[ ]+` expression.

The URL pattern will match a sequence of alphanumeric characters and hyphens which are between the `rango/category/` and the trailing `/`. This sequence will be stored in the parameter `category_name_slug` and passed to `views.show_category()`. For example, the URL `rango/category/python-books/` would result in the `category_name_slug` having the value, `python-books`. However, if the URL was `rango/category/££££-$$$$$/` then the sequence of characters between `rango/category/` and the trailing `/` would not match the regular expression, and a `404 not found` error would result because there would be no matching URL pattern.

All view functions defined as part of a Django applications *must* take at least one parameter. This is typically called `request` - and provides access to information related to the given HTTP request made by the user. When parameterising URLs, you supply additional named parameters to the signature for the given view. That is why our `show_category()` view was defined as `def show_category(request, category_name_slug)`.

Regex Hell

"Some people, when confronted with a problem, think *'I know, I'll use regular expressions.'* Now they have two problems." Jamie Zawinski

Regular expressions may seem horrible and confusing at first, but there are tons of resources online to help you. This cheat sheet is an excellent resource for fixing problems with regular expressions.

Modifying the Index Template

Our new view is set up and ready to go - but we need to do one more thing. Our index page template needs to be updated so that it links to the category pages that are listed. We can update the `index.html` template to now include a link to the category page via the slug.

```
<!DOCTYPE html>
{% load staticfiles %}
<html>
    <head>
        <title>Rango</title>
    </head>

    <body>
        <h1>Rango says...</h1>

        <div>
            hey there partner!
        </div>
```

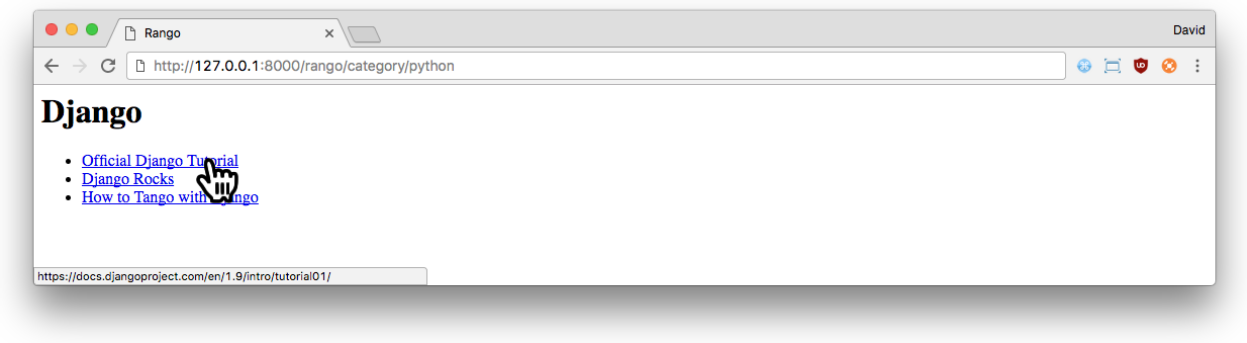
```
        <div>
        {% if categories %}
        <ul>
            {% for category in categories %}
            <!-- Following line changed to add an HTML hyperlink -->
            <li>
            <a href="/rango/category/{{ category.slug }}">{{ category.name }}</a>
            </li>
            {% endfor %}
        </ul>
        {% else %}
            <strong>There are no categories present.</strong>
        {% endif %}
        </div>

        <div>
            <a href="/rango/about/">About Rango</a><br />
            <img src="{% static "images/rango.jpg" %}" alt="Picture of Rango" />
        </div>
    </body>
</html>
```

Again, we used the HTML tag `<ul>` to define an unordered list. Within the list, we create a series of list elements (`<li>`), each of which in turn contains a HTML hyperlink (`<a>`). The hyperlink has an `href` attribute, which we use to specify the target URL defined by `/rango/category/{{ category.slug }}` which, for example, would turn into `/rango/category/python-books/` for the category `Python Books`.

Demo

Let's try everything out now by visiting the Rango homepage. You should see up to five categories on the index page. The categories should now be links. Clicking on `Django` should then take you to the `Django` category page, as shown in the figure below. If you see a list of links like `Official Django Tutorial`, then you've successfully set up the new page.

What happens when you visit a category that does not exist? Try navigating a category which doesn't exist, like `/rango/category/computers/`. Do this by typing the address manually into your browser's address bar. You should see a message telling you that the specified category does not exist.

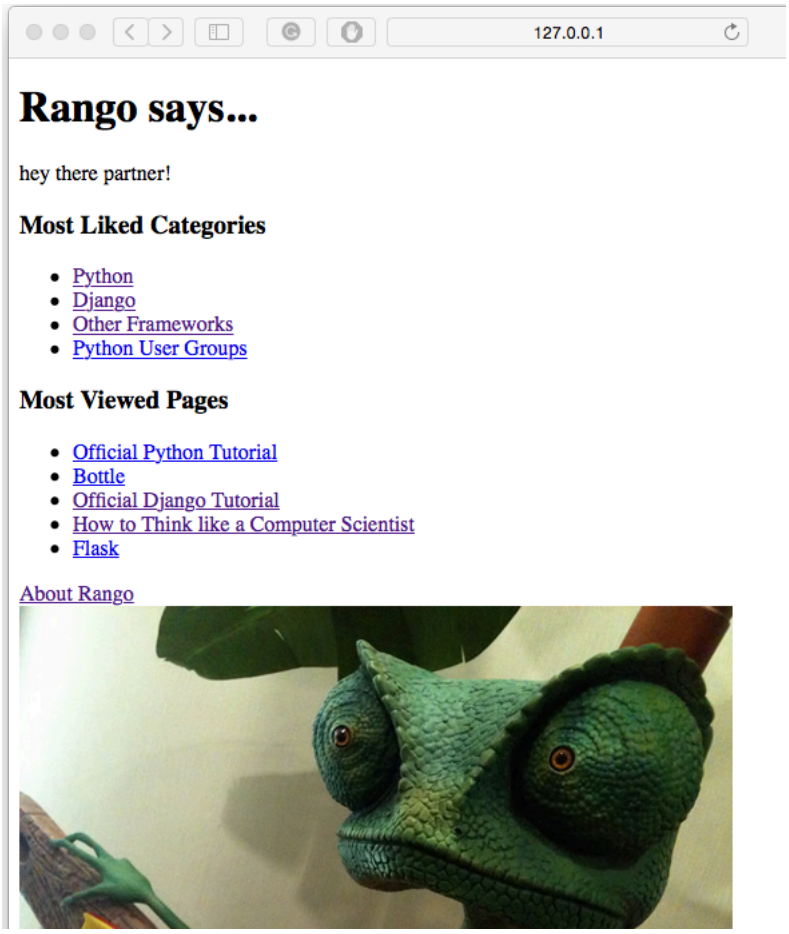


The links to Django pages. Note the mouse is hovering over the first link – you can see the corresponding URL for that link at the bottom left of the Google Chrome window.

Exercises

Reinforce what you've learnt in this chapter by trying out the following exercises.

- Update the population script to add some value to the `views` count for each page.
- Modify the index page to also include the top 5 most viewed pages.
- Include a heading for the "Most Liked Categories" and "Most Viewed Pages".
- Include a link back to the index page from the category page.
- Undertake part three of official Django tutorial if you have not done so already to reinforce what you've learnt here.



The index page after you complete the exercises, showing the most liked categories and most viewed pages.

Hints

- When updating the population script, you'll essentially follow the same process as you went through in the previous chapter's exercises. You will need to update the data structures for each page, and also update the code that makes use of them.
  - Update the three data structures containing pages for each category – `python_pages`, `django_pages` and `other_pages`. Each page has a `title` and `url` – they all now need a count of how many `views` they see, too.
  - Look at how the `add_page()` function is defined in your population script. Does it allow for you to pass in a `views` count? Do you need to change anything in this function?
  - Finally, update the line where the `add_page()` function is *called*. If you called the views count in the data structures `views`, and the dictionary that represents a page is called `p` in the context of where `add_page()` is called, how can you pass the views count into the function?
- Remember to re-run the population script so that the views are updated.
  - You will need to edit both the `index` view and the `index.html` template to put the most viewed (i.e. popular pages) on the index page.
  - Instead of accessing the `Category` model, you will have to ask the `Page` model for the most viewed pages.
  - Remember to pass the list of pages through to the context.
  - If you are not sure about the HTML template code to use, you can draw inspiration from the `category.html` template code as the markup is essentially the same.

Model Tips

For more tips on working with models you can take a look through the following blog posts:

1. Best Practices when working with models by Kostantin Moiseenko. In this post you will find a series of tips and tricks when working with models.
2. How to make your Django Models DRYer by Robert Roskam. In this post you can see how you can use the `property` method of a class to reduce the amount of code needed when accessing related models.

7. Forms

In this chapter, we will run through how to capture data through web forms. Django comes with some neat form handling functionality, making it a pretty straightforward process to collect information from users and save it to the database via the models. According to Django's documentation on forms, the form handling functionality allows you to:

1. display an HTML form with automatically generated *form widgets* (like a text field or date picker);
2. check submitted data against a set of validation rules;
3. redisplay a form in case of validation errors; and
4. convert submitted form data to the relevant Python data types.

One of the major advantages of using Django's forms functionality is that it can save you a lot of time and hassle creating the HTML forms.

7.1 Basic Workflow

The basic steps involved in creating a form and handling user input is as follows.

1. If you haven't already got one, create a `forms.py` file within your Django application's directory to store form-related classes.
2. Create a `ModelForm` class for each model that you wish to represent as a form.
3. Customise the forms as you desire.
4. Create or update a view to handle the form
    - including *displaying* the form,
    - *saving* the form data, and
    - *flagging up errors* which may occur when the user enters incorrect data (or no data at all) in the form.
5. Create or update a template to display the form.
6. Add a `urlpattern` to map to the new view (if you created a newone).

This workflow is a bit more complicated than previous workflows, and the views that we have to construct have a lot more complexity as well. However, once you undertake the process a few times it will be pretty clear how everything pieces together.

7.2 Page and Category Forms

Here, we will implement the necessary infrastructure that will allow users to add categories and pages to the database via forms.

First, create a file called `forms.py` within the `rango` application directory. While this step is not absolutely necessary (you could put the forms in the `models.py`), this makes your codebase tidier and easier to work with.

Creating `ModelForm` Classes

Within Rango's `forms.py` module, we will be creating a number of classes that inherit from Django's `ModelForm`. In essence, a `ModelForm` is a *helper class* that allows you to create a Django `Form` from a pre-existing model. As we've already got two models defined for Rango (`Category` and `Page`), we'll create `ModelForms` for both.

In `rango/forms.py` add the following code.

```
1  from django import forms
2  from rango.models import Page, Category
3
4  class CategoryForm(forms.ModelForm):
5      name = forms.CharField(max_length=128,
6                             help_text="Please enter the category name.")
7      views = forms.IntegerField(widget=forms.HiddenInput(), initial=0)
8      likes = forms.IntegerField(widget=forms.HiddenInput(), initial=0)
9      slug = forms.CharField(widget=forms.HiddenInput(), required=False)
10
11     # An inline class to provide additional information on the form.
12     class Meta:
13         # Provide an association between the ModelForm and a model
14         model = Category
15         fields = ('name',)
16
17 class PageForm(forms.ModelForm):
18     title = forms.CharField(max_length=128,
19                             help_text="Please enter the title of the page.")
20     url = forms.URLField(max_length=200,
21                          help_text="Please enter the URL of the page.")
22     views = forms.IntegerField(widget=forms.HiddenInput(), initial=0)
23
24     class Meta:
25         # Provide an association between the ModelForm and a model
```

```
26          model = Page
27
28          # What fields do we want to include in our form?
29          # This way we don't need every field in the model present.
30          # Some fields may allow NULL values, so we may not want to include them.
31          # Here, we are hiding the foreign key.
32          # we can either exclude the category field from the form,
33          exclude = ('category',)
34          # or specify the fields to include (i.e. not include the category field)
35          #fields = ('title', 'url', 'views')
```

We need to specify which fields are included on the form, via `fields`, or specify which fields are to be excluded, via `exclude`.

Django provides us with a number of ways to customise the forms that are created on our behalf. In the code sample above, we've specified the widgets that we wish to use for each field to be displayed. For example, in our `PageForm` class, we've defined `forms.CharField` for the `title` field, and `forms.URLField` for `url` field. Both fields provide text entry for users. Note the `max_length` parameters we supply to our fields - the lengths that we specify are identical to the maximum length of each field we specified in the underlying data models. Go back to the chapter on models to check for yourself, or have a look at Rango's `models.py` file.

You will also notice that we have included several `IntegerField` entries for the views and likes fields in each form. Note that we have set the widget to be hidden with the parameter setting `widget=forms.HiddenInput()`, and then set the value to zero with `initial=0`. This is one way to set the field to zero by default. And since the fields will be hidden the user won't be able to enter a value for these fields.

However, as you can see in the `PageForm`, despite the fact that we have a hidden field, we still need to include the field in the form. If in `fields` we excluded `views`, then the form would not contain that field (despite it being specified) and so the form would not return the value zero for that field. This may raise an error depending on how the model has been set up. If in the model we specified that the `default=0` for these fields then we can rely on the model to automatically populate field with the default value - and thus avoid a `not null` error. In this case, it would not be necessary to have these hidden fields. We have also included the field `slug` in the `CategoryForm`, and set it to use the `widget=forms.HiddenInput()`, but rather than specifying an initial or default value, we have said the field is not required by the form. This is because our model will be responsible on `save()` for populating this field. Essentially, you need to be careful when you define your models and forms to make sure that the form is going to contain and pass on all the data that is required to populate your model correctly.

Besides the `CharField` and `IntegerField` widgets, many more are available for use. As an example, Django provides `EmailField` (for e-mail address entry), `ChoiceField` (for radio input buttons), and `DateField` (for date/time entry). There are many other field types you can use, which perform error checking for you (e.g. *is the value provided a valid integer?*).

Perhaps the most important aspect of a class inheriting from `ModelForm` is the need to define *which model we're wanting to provide a form for.* We take care of this through our nested `Meta` class. Set the `model` attribute of the nested `Meta` class to the model you wish to use. For example, our `CategoryForm` class has a reference to the `Category` model. This is a crucial step enabling Django to take care of creating a form in the image of the specified model. It will also help in handling flagging up any errors along with saving and displaying the data in the form.

We also use the `Meta` class to specify which fields that we wish to include in our form through the `fields` tuple. Use a tuple of field names to specify the fields you wish to include.

More about Forms

Check out the official Django documentation on forms for further information about the different widgets and how to customise forms.

Creating an *Add Category* View

With our `CategoryForm` class now defined, we're now ready to create a new view to display the form and handle the posting of form data. To do this, add the following code to `rango/views.py`.

```
#Add this import at the top of the file
from rango.forms import CategoryForm
...
def add_category(request):
    form = CategoryForm()

    # A HTTP POST?
    if request.method == 'POST':
        form = CategoryForm(request.POST)

        # Have we been provided with a valid form?
        if form.is_valid():
            # Save the new category to the database.
            form.save(commit=True)
            # Now that the category is saved
            # We could give a confirmation message
            # But since the most recent category added is on the index page
            # Then we can direct the user back to the index page.
            return index(request)
        else:
            # The supplied form contained errors -
            # just print them to the terminal.
```

```
            print(form.errors)

    # Will handle the bad form, new form, or no form supplied cases.
    # Render the form with error messages (if any).
    return render(request, 'rango/add_category.html', {'form': form})
```

The new `add_category()` view adds several key pieces of functionality for handling forms. First, we create a CategoryForm(), then we check if the HTTP request was a `POST` i.e. if the user submitted data via the form. We can then handle the `POST` request through the same URL. The `add_category()` view function can handle three different scenarios:

- showing a new, blank form for adding a category;
- saving form data provided by the user to the associated model, and rendering the Rango homepage; and
- if there are errors, redisplay the form with error messages.

`GET` and `POST`

What do we mean by `GET` and `POST`? They are two different types of *HTTP requests*.

- A HTTP `GET` is used to *request a representation of the specified resource.* In other words, we use a HTTP `GET` to retrieve a particular resource, whether it is a webpage, image or other file.
- In contrast, a HTTP `POST` *submits data from the client's web browser to be processed.* This type of request is used for example when submitting the contents of a HTML form.
- Ultimately, a HTTP `POST` may end up being programmed to create a new resource (e.g. a new database entry) on the server. This can later be accessed through a HTTP `GET` request.
- Check out the w3schools page on `GET` vs. `POST` for more details.

Django's form handling machinery processes the data returned from a user's browser via a HTTP `POST` request. It not only handles the saving of form data into the chosen model, but will also automatically generate any error messages for each form field (if any are required). This means that Django will not store any submitted forms with missing information that could potentially cause problems for your database's referential integrity. For example, supplying no value in the `category` name field will return an error, as the field cannot be blank.

You'll notice from the line in which we call `render()` that we refer to a new template called `add_category.html`. This will contain the relevant Django template code and HTML for the form and page.

Creating the *Add Category* Template

Create the file `templates/rango/add_category.html`. Within the file, add the following HTML markup and Django template code.

```
<!DOCTYPE html>
<html>
    <head>
        <title>Rango</title>
    </head>

    <body>
        <h1>Add a Category</h1>
        <div>
            <form id="category_form" method="post" action="/rango/add_category/">
                {% csrf_token %}
                {% for hidden in form.hidden_fields %}
                    {{ hidden }}
                {% endfor %}
                {% for field in form.visible_fields %}
                    {{ field.errors }}
                    {{ field.help_text }}
                    {{ field }}
                {% endfor %}
                <input type="submit" name="submit" value="Create Category" />
            </form>
        </div>
    </body>
</html>
```

You can see that within the `<body>` of the HTML page we placed a `<form>` element. Looking at the attributes for the `<form>` element, you can see that all data captured within this form is sent to the URL `/rango/add_category/` as a HTTP `POST` request (the `method` attribute is case insensitive, so you can do `POST` or `post` - both provide the same functionality). Within the form, we have two for loops:

- one controlling *hidden* form fields, and
- the other *visible* form fields.

The visible fields i.e. those that will be displayed to the user, are controlled by the `fields` attribute within your `ModelForm` `Meta` class. These loops produce HTML markup for each form element. For visible form fields, we also add in any errors that may be present with a particular field and help text that can be used to explain to the user what he or she needs to enter.

Hidden Fields

The need for hidden as well as visible form fields is necessitated by the fact that HTTP is a stateless protocol. You can't persist state between different HTTP requests that can make certain parts of web applications difficult to implement. To overcome this limitation, hidden HTML form fields were created which allow web applications to pass important information to a client (which cannot be seen on the rendered page) in a HTML form, only to be sent back to the originating server when the user submits the form.

Cross Site Request Forgery Tokens

You should also take note of the code snippet `{% csrf_token %}`. This is a *Cross-Site Request Forgery (CSRF) token*, which helps to protect and secure the HTTP `POST` action that is initiated on the subsequent submission of a form. *The Django framework requires the CSRF token to be present. If you forget to include a CSRF token in your forms, a user may encounter errors when he or she submits the form.* Check out the official Django documentation on CSRF tokens (and this link for Django 1.10) for more information about this.

Mapping the *Add Category* View

Now we need to map the `add_category()` view to a URL. In the template we have used the URL `/rango/add_category/` in the form's action attribute. We now need to create a mapping from the URL to the View. In `rango/urls.py` modify the `urlpatterns`

```
urlpatterns = [
    url(r'^$', views.index, name='index'),
    url(r'about/$', views.about, name='about'),
    url(r'^add_category/$', views.add_category, name='add_category'),
    url(r'^category/(?P<category_name_slug>[\w\-]+)/$',
        views.show_category, name='show_category'),
]
```

Ordering doesn't necessarily matter in this instance. However, take a look at the official Django documentation on how Django process a request for more information. The URL for adding a category is `/rango/add_category/`.

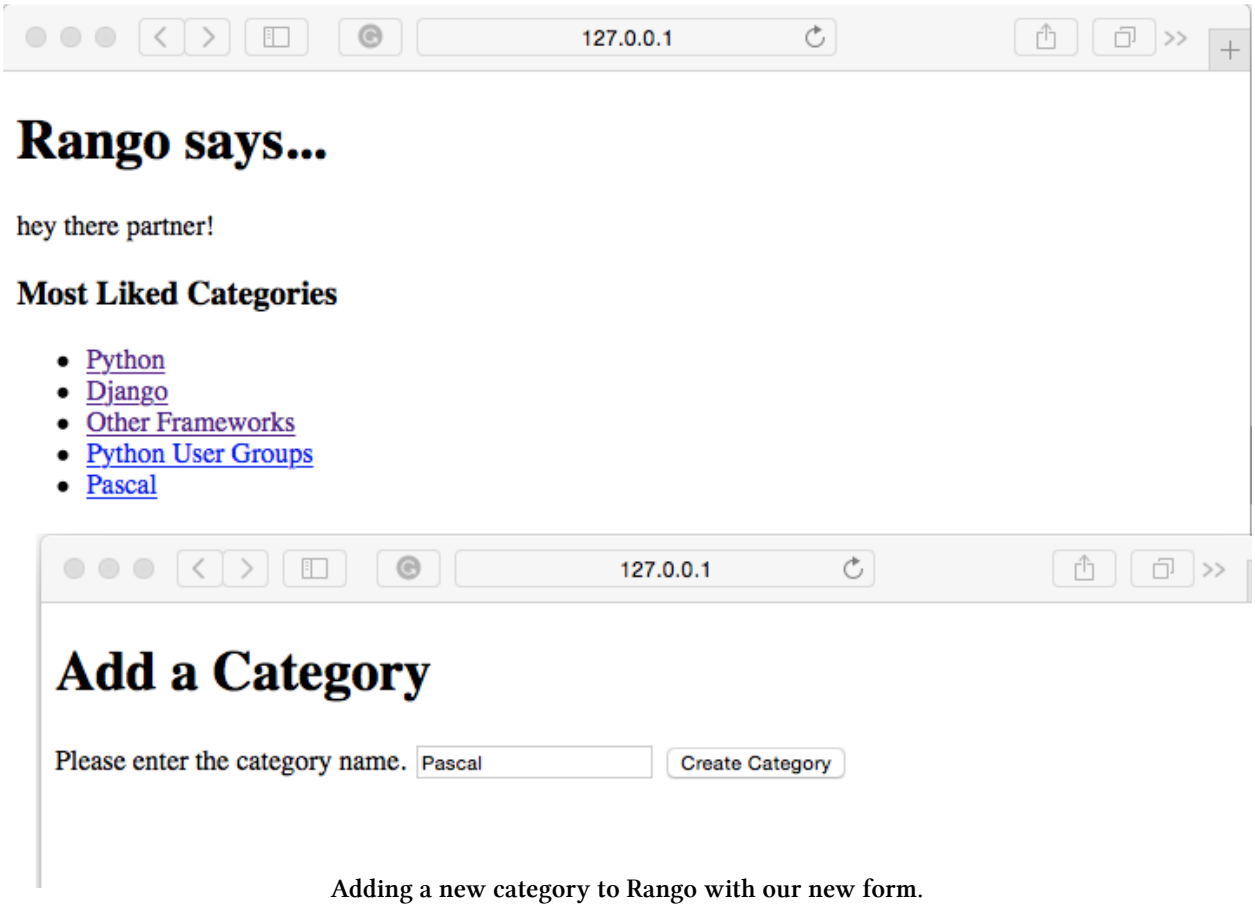
Modifying the Index Page View

As a final step let's put a link on the index page so that we can easily add categories. Edit the template `rango/index.html` and add the following HTML hyperlink in the `<div>` element with the about link.

```
<a href="/rango/add_category/">Add a New Category</a><br />
```

Demo

Now let's try it out! Start or restart your Django development server, and then point your web browser to Rango at `http://127.0.0.1:8000/rango/`. Use your new link to jump to the Add Category page, and try adding a category. The figure below shows screenshots of the Add Category and Index Pages.



Adding a new category to Rango with our new form.

Missing Categories

If you add a number of categories, they will not always appear on the index page. This is because we are only showing the top five categories on the index page. If you log into the Admin interface, you should be able to view all the categories that you have entered.

Another way to get some confirmation that the category is being added is to update the `add_category()` method in `rango/views.py` and change the line `form.save(commit=True)` to be `cat = form.save(commit=True)`. This will give you a reference to an instance of the category object created by the form. You can then print the category to console (e.g. `print(cat, cat.slug)`).

Cleaner Forms

Recall that our `Page` model has a `url` attribute set to an instance of the `URLField` type. In a corresponding HTML form, Django would reasonably expect any text entered into a `url` field to be a correctly formatted, complete URL. However, users can find entering something like `http://www.url.com` to be cumbersome - indeed, users may not even know what forms a correct URL!

URL Checking

Most modern browsers will now check to make sure that the URL is well-formed for you, so this example will only work on older browsers. However, it does show you how to clean the data before you try to save it to the database. If you don't have an old browser to try this example (in case you don't believe it), try changing the `URLField` to a `CharField`. The rendered HTML will then not instruct the browser to perform the checks on your behalf, and the code you implemented will be executed.

In scenarios where user input may not be entirely correct, we can *override* the `clean()` method implemented in `ModelForm`. This method is called upon before saving form data to a new model instance, and thus provides us with a logical place to insert code which can verify - and even fix - any form data the user inputs. We can check if the value of `url` field entered by the user starts with `http://` - and if it doesn't, we can prepend `http://` to the user's input.

```
class PageForm(forms.ModelForm):
    ...
    def clean(self):
        cleaned_data = self.cleaned_data
        url = cleaned_data.get('url')

        # If url is not empty and doesn't start with 'http://',
        # then prepend 'http://'.
        if url and not url.startswith('http://'):
            url = 'http://' + url
            cleaned_data['url'] = url

            return cleaned_data
```

Within the `clean()` method, a simple pattern is observed which you can replicate in your own Django form handling code.

1. Form data is obtained from the `ModelForm` dictionary attribute `cleaned_data`.

2. Form fields that you wish to check can then be taken from the `cleaned_data` dictionary. Use the `.get()` method provided by the dictionary object to obtain the form's values. If a user does not enter a value into a form field, its entry will not exist in the `cleaned_data` dictionary. In this instance, `.get()` would return `None` rather than raise a `KeyError` exception. This helps your code look that little bit cleaner!
3. For each form field that you wish to process, check that a value was retrieved. If something was entered, check what the value was. If it isn't what you expect, you can then add some logic to fix this issue before *reassigning* the value in the `cleaned_data` dictionary.
4. You *must* always end the `clean()` method by returning the reference to the `cleaned_data` dictionary. Otherwise the changes won't be applied.

This trivial example shows how we can clean the data being passed through the form before being stored. This is pretty handy, especially when particular fields need to have default values - or data within the form is missing, and we need to handle such data entry problems.

Clean Overrides

Overriding methods implemented as part of the Django framework can provide you with an elegant way to add that extra bit of functionality for your application. There are many methods which you can safely override for your benefit, just like the `clean()` method in `ModelForm` as shown above. Check out the Official Django Documentation on Models for more examples on how you can override default functionality to slot your own in.

Exercises

Now that you've worked through the chapter, consider the following questions, and how you could solve them.

- What would happen if you don't enter in a category name on the add category form?
- What happens when you try to add a category that already exists?
- What happens when you visit a category that does not exist? A hint for a potential solution to solving this problem can be found below.
- In the section above where we implemented our `ModelForm` classes, we repeated the `max_length` values for fields that we had previously defined in the models chapter. This is bad practice as we are *repeating ourselves!* How can you refactor your code so that you are *not* repeating the `max_length` values?
- If you have not done so already undertake part four of the official Django Tutorial to reinforce what you have learnt here.
- Now let users add pages to each category, see below for some example code and hints.

Creating an *Add Pages* View, Template and URL Mapping

A next logical step would be to allow users to add pages to a given category. To do this, repeat the same workflow above but for adding pages.

- create a new view, `add_page()`,
- create a new template, `rango/add_page.html`,
- add a URL mapping, and
- update the category page/view to provide a link from the category add page functionality.

To get you started, here is the code for the `add_page()` view function.

```
from rango.forms import PageForm

def add_page(request, category_name_slug):
    try:
        category = Category.objects.get(slug=category_name_slug)
    except Category.DoesNotExist:
        category = None

    form = PageForm()
    if request.method == 'POST':
        form = PageForm(request.POST)
        if form.is_valid():
            if category:
                page = form.save(commit=False)
                page.category = category
                page.views = 0
                page.save()
                return show_category(request, category_name_slug)
        else:
            print(form.errors)

    context_dict = {'form':form, 'category': category}
    return render(request, 'rango/add_page.html', context_dict)
```

Hints

To help you with the exercises above, the following hints may be of some use to you.

- In the `add_page.html` template you can access the slug with `{{ category.slug }}` because the view passes the `category` object through to the template via the context dictionary.
- Ensure that the link only appears when *the requested category exists* - with or without pages. i.e. in the template check with `{% if cat %} .... {% else %} A category by this name does not exist {% endif %}`.
- Update Rango's `category.html` template with a new hyperlink with a line break immediately following it: `<a href="/rango/category/{{category.slug}}/add_page/">Add Page</a> <br/>`
- Make sure that in your `add_page.html` template that the form posts to `/rango/category/{{ category.slug }}/add_page/`.
- Update `rango/urls.py` with a URL mapping (`/rango/category/<category_name_slug>/add_page/`) to handle the above link.
- You can avoid the repetition of `max_length` parameters through the use of an additional attribute in your `Category` class. This attribute could be used to store the value for `max_length`, and then be referenced where required.

If you get *really* stuck, you can always check out our code on GitHub.

8. Final Thoughts

In this book, we have gone through the process of web development from specification to deployment. Along the way we have shown how to use the Django framework to construct the models, views and templates associated with a Web application. We have also demonstrated how toolkits and services like Bootstrap, JQuery and PythonAnywhere. can be integrated within an application. However, the road doesn't stop here. While, as we have only painted the broad brush strokes of a web application - as you have probably noticed there are lots of improvements that could be made to Rango - and these finer details often take a lot more time to complete as you polish the application. By developing your application on a firm base and good setup you will be able to construct up to 80% of your site very rapidly and get a working demo online.

In future versions of this book we intend to provide some more details on various aspects of the framework, along with covering the basics of some of the other fundamental technologies associated with web development. If you have any suggestions or comments about how to improve the book please get in touch.

Please report any bugs, problems, etc., or submit change requests via GitHub. Thank you!

8.1 Acknowledgements

This book was written to help teach web application development to computing science students. In writing the book and the tutorial, we have had to rely upon the awesome Django community and the Django Documentation for the answers and solutions. This book is really the combination of that knowledge pieced together in the context of building Rango.

We would also like to thank all the people who have helped to improve this resource by sending us comments, suggestions, Git issues and pull requests. If you've sent in changes over the years, please do remind us if you are not on the list!

Adam Kikowski, Adam Mertz, **Alessio Oxilia**, Ally Weir, **bernieyangmh**, Breakerfall, **Brian**, Burak K., **Burak Karaboga**, Can Ibanoglu, **Charlotte** , Claus Conrad, **Codenius**, cspollar, **Dan C**, Darius, **David Manlove**, David Skillman, **Deep Sukhwani** Devin Fitzsimons, **Dhiraj Thakur**, Duncan Drizy, **Gerardo A-C**, Giles T., **Grigoriy M**, James Yeo, **Jan Felix Trettow**, Joe Maskell, **Jonathan Sundqvist**, Karen Little, **Kartik Singhal**, koviusesGitHub, **Krace Kumar**, ma-152478, **Manoel Maria**, Martin de G., **Matevz P.**, mHulb, **Michael Herman**, Michael Ho Chum, **Mickey P.**, Mike Gleen, **nCrazed**, Nitin Tulswani, **nolan-m**, Oleg Belausov, **pawonfire**, pdehaye, **Peter Mash**, Pierre-Yves Mathieu, **Praestgias**, pzkpfwVI, **Ramdog**, Rezha Julio, **rnevius**, Sadegh Kh, **Saex**, Saurabh Tandon, **Serede Sixty Six**, Svante Kvarnstrom, **Tanmay Kansara**, Thomas Murphy, **Thomas Whyyou**, William Vincent, and **Zhou**.

Thank you all very much!

9. Setting up your System

This chapter provides a brief overview of the different components that you need to have working in order to develop Django apps.

Choosing a Python Version

Django supports both the Python 2.7.x and 3 programming languages. While they both share the same name, they are fundamentally different programming languages. In this chapter, we assume you are setting up Python 2.7.5 - you can change the version number as you require.

9.1 Installing Python

So, how do you go about installing Python 2.7/3.4 on your computer? You may already have Python installed on your computer - and if you are using a Linux distribution or OS X, you will definitely have it installed. Some of your operating system's functionality is implemented in Python, hence the need for an interpreter!

Unfortunately, nearly all modern operating systems utilise a version of Python that is older than what we require for this tutorial. There's many different ways in which you can install Python, and many of them are sadly rather tricky to accomplish. We demonstrate the most commonly used approaches, and provide links to additional reading for more information.

Do not remove your default Python installation

This section will detail how to run Python 2.7.5 *alongside* your current Python installation. It is regarded as poor practice to remove your operating system's default Python installation and replace it with a newer version. Doing so could render aspects of your operating system's functionality broken!

Apple mac OS/OS X

The most simple way to get Python 2.7.5 installed on your Mac is to download and run the simple installer provided on the official Python website. You can download the installer by visiting the webpage at http://www.python.org/getit/releases/2.7.5/.

Make sure you have the correct version for your Mac

Ensure that you download the `.dmg` file that is relevant to your particular mac OS/OS X installation!

1. Once you have downloaded the `.dmg` file, double-click it in the Finder.
2. The file mounts as a separate disk and a new Finder window is presented to you.
3. Double-click the file `Python.mpkg`. This will start the Python installer.
4. Continue through the various screens to the point where you are ready to install the software. You may have to provide your password to confirm that you wish to install the software.
5. Upon completion, close the installer and eject the Python disk. You can now delete the downloaded `.dmg` file.

You should now have an updated version of Python installed, ready for Django! Easy, huh? You can also install Python 3.4+ in a similar version, if you prefer to use Python 3.

Linux Distributions

Unfortunately, there are many different ways in which you can download, install and run an updated version of Python on your Linux distribution. To make matters worse, methodologies vary from distribution to distribution. For example, the instructions for installing Python on Fedora may differ from those to install it on an Ubuntu installation.

However, not all hope is lost. An awesome tool (or a *Python environment manager*) called pythonbrew can help us address this difficulty. It provides an easy way to install and manage different versions of Python, meaning you can leave your operating system's default Python installation alone.

Taken from the instructions provided from the pythonbrew GitHub page and this Stack Overflow question and answer page, the following steps will install Python 2.7.5 on your Linux distribution.

1. Open a new terminal instance.
2. Run the command `curl -kL http://xrl.us/pythonbrewinstall | bash`. This will download the installer and run it within your terminal for you. This installs pythonbrew into the directory `~/.pythonbrew`. Remember, the tilde (~) represents your home directory!
3. You then need to edit the file `~/.bashrc`. In a text editor (such as `gedit`, `nano`, `vi` or `emacs`), add the following to a new line at the end of `~/.bashrc`: `[[ -s $HOME/.pythonbrew/etc/bashrc ]] && source $HOME/.pythonbrew/etc/bashrc`
4. Once you have saved the updated `~/.bashrc` file, close your terminal and open a new one. This allows the changes you make to take effect.
5. Run the command `pythonbrew install 2.7.5` to install Python 2.7.5.

6. You then have to *switch* Python 2.7.5 to the *active* Python installation. Do this by running the command `pythonbrew switch 2.7.5`.
7. Python 2.7.5 should now be installed and ready to go.

Hidden Directories and Files

Directories and files beginning with a period or dot can be considered the equivalent of *hidden files* in Windows. Dot files are not normally visible to directory-browsing tools, and are commonly used for configuration files. You can use the `ls` command to view hidden files by adding the `-a` switch to the end of the command, giving the command `ls -a`.

Windows

By default, Microsoft Windows comes with no installations of Python. This means that you do not have to worry about leaving existing versions be; installing from scratch should work just fine. You can download a 64-bit or 32-bit version of Python from the official Python website. If you aren't sure which one to download, you can determine if your computer is 32-bit or 64-bit by looking at the instructions provided on the Microsoft website.

1. When the installer is downloaded, open the file from the location to which you downloaded it.
2. Follow the on-screen prompts to install Python.
3. Close the installer once completed, and delete the downloaded file.

Once the installer is complete, you should have a working version of Python ready to go. By default, Python 2.7.5 is installed to the directory `C:\Python27`. We recommend that you leave the path as it is.

Upon the completion of the installation, open a Command Prompt and enter the command `python`. If you see the Python prompt, installation was successful. However, in certain circumstances, the installer may not set your Windows installation's `PATH` environment variable correctly. This will result in the `python` command not being found. Under Windows 7, you can rectify this by performing the following:

1. Click the *Start* button, right click *My Computer* and select *Properties*.
2. Click the *Advanced* tab.
3. Click the *Environment Variables* button.
4. In the *System variables* list, find the variable called *Path*, click it, then click the *Edit* button.
5. At the end of the line, enter `;C:\python27;C:\python27\scripts`. Don't forget the semicolon - and certainly *do not* add a space.

6. Click OK to save your changes in each window.
7. Close any Command Prompt instances, open a new instance, and try run the `python` command again.

This should get your Python installation fully working. Things might differ ever so slightly on Windows 10.

9.2 Setting Up the `PYTHONPATH`

With Python now installed, we now need to check that the installation was successful. To do this, we need to check that the `PYTHONPATH` environment variable is setup correctly. `PYTHONPATH` provides the Python interpreter with the location of additional Python packages and modules which add extra functionality to the base Python installation. Without a correctly set `PYTHONPATH`, we'll be unable to install and use Django!

First, let's verify that our `PYTHONPATH` variable exists. Depending on the installation technique that you chose, this may or may not have been done for you. To do this on your UNIX-based operating system, issue the following command in a terminal.

```
$ echo $PYTHONPATH
```

On a Windows-based machine, open a Command Prompt and issue the following.

```
$ echo %PYTHONPATH%
```

If all works, you should then see output that looks something similar to the example below. On a Windows-based machine, you will obviously see a Windows path, most likely originating from the C drive.

```
/opt/local/Library/Frameworks/Python.framework/
    Versions/2.7/lib/python2.7/site-packages:
```

This is the path to your Python installation's `site-packages` directory, where additional Python packages and modules are stored. If you see a path, you can continue to the next part of this tutorial. If you however do not see anything, you'll need to do a little bit of detective work to find out the path. On a Windows installation, this should be a trivial exercise: `site-packages` is located within the `lib` directory of your Python installation directory. For example, if you installed Python to `C:\Python27`, `site-packages` will be at `C:\Python27\Lib\site-packages\`.

UNIX-based operating systems however require a little bit of detective work to discover the path of your `site-packages` installation. To do this, launch the Python interpreter. The following terminal session demonstrates the commands you should issue.

```
$ python

Python 2.7.5 (v2.7.5:ab05e7dd2788, May 13 2013, 13:18:45)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>> import site
>>> print(site.getsitepackages()[0])

'/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/site-packages'

>>> quit()
```

Calling `site.getsitepackages()` returns a list of paths that point to additional Python package and module stores. The first typically returns the path to your `site-packages` directory - changing the list index position may be required depending on your installation. If you receive an error stating that `getsitepackages()` is not present within the `site` module, verify you're running the correct version of Python. Version 2.7.5 should include this function. Previous versions of the language do not include this function.

The string which is shown as a result of executing `print site.getsitepackages()[0]` is the path to your installation's `site-packages` directory. Taking the path, we now need to add it to your configuration. On a UNIX-based or UNIX-derived operating system, edit your `.bashrc` file once more, adding the following to the bottom of the file.

```
export PYTHONPATH=$PYTHONPATH:<PATH_TO_SITE-PACKAGES>
```

Replace `<PATH_TO_SITE-PACKAGES>` with the path to your `site-packages` directory. Save the file, and quit and reopen any instances of your terminal.

On a Windows-based computer, you must follow the instructions shown above to bring up the environment variables settings dialog. Add a `PYTHONPATH` variable with the value being set to your `site-packages` directory, which is typically `C:\Python27\Lib\site-packages\`.

9.3 Using `setuptools` and `pip`

Installing and setting up your development environment is a really important part of any project. While it is possible to install Python Packages such as Django separately, this can lead to numerous problems and hassles later on. For example, how would you share your setup with another developer? How would you set up the same environment on your new machine? How would you upgrade to the latest version of the package? Using a package manager removes much of the hassle involved in setting up and configuring your environment. It will also ensure that the package you

install is the correct for the version of Python you are using, along with installing any other packages that are dependent upon the one you want to install.

In this book, we use `pip`. `pip` is a user friendly wrapper over the `setuptools` Python package manager. Because `pip` depends on `setuptools`, we are required to ensure that both are installed on your computer.

To start, we should download `setuptools` from the official Python package website. You can download the package in a compressed `.tar.gz` file. Using your favourite file extracting program, extract the files. They should all appear in a directory called `setuptools-1.1.6` - where `1.1.6` represents the `setuptools` version number. From a terminal instance, you can then change into the directory and execute the script `ez_setup.py` as shown below.

```
$ cd setuptools-1.1.6
$ sudo python ez_setup.py
```

In the example above, we also use `sudo` to allow the changes to become system wide. The second command should install `setuptools` for you. To verify that the installation was successful, you should be able to see output similar to that shown below.

```
Finished processing dependencies for setuptools==1.1.6
```

Of course, `1.1.6` is substituted with the version of `setuptools` you are installing. If this line can be seen, you can move onto installing `pip`. This is a trivial process, and can be completed with one simple command. From your terminal instance, enter the following.

```
$ sudo easy_install pip
```

This command should download and install `pip`, again with system wide access. You should see the following output, verifying `pip` has been successfully installed.

```
Finished processing dependencies for pip
```

Upon seeing this output, you should be able to launch `pip` from your terminal. To do so, just type `pip`. Instead of an unrecognised command error, you should be presented with a list of commands and switches that `pip` accepts. If you see this, you're ready to move on!

No Sudo on Windows!

On Windows computers, follow the same basic process. You won't need to enter the `sudo` command, however.

9.4 Virtual Environments

We're almost all set to go! However, before we continue, it's worth pointing out that while this setup is fine to begin with, there are some drawbacks. What if you had another Python application that requires a different version to run? Or you wanted to switch to the new version of Django, but still wanted to maintain your Django 1.7 project?

The solution to this is to use virtual environments. Virtual environments allow multiple installations of Python and their relevant packages to exist in harmony. This is the generally accepted approach to configuring a Python setup nowadays. They are pretty easy to setup, once you have `pip` installed, and you know the right commands. You need to install a couple of additional packages.

```
$ pip install virtualenv
$ pip install virtualenvwrapper
```

The first package provides you with the infrastructure to create a virtual environment. See a non-magical introduction to `pip` and Virtualenv for Python Beginners by Jamie Matthews for details about using virtualenv. However, using just *virtualenv* alone is rather complex. The second package provides a wrapper to the functionality in the virtualenv package and makes life a lot easier.

If you are using a Linux/UNIX based OS, then to use the wrapper you need to call the following shell script from your command line: :

```
$ source virtualenvwrapper.sh
```

It is a good idea to add this to your bash/profile script. So you don't have to run it each and every time you want to use virtual environments. However, if you are using windows, then install the virtualenvwrapper-win package:

```
$ pip install virtualenvwrapper-win
```

Now you should be all set to create a virtual environment:

```
$ mkvirtualenv rango
```

You can list the virtual environments created with `lsvirtualenv`, and you can activate a virtual environment as follows:

```
$ workon rango
(rango)$
```

Your prompt with change and the current virtual environment will be displayed, i.e. `rango`. Now within this environment you will be able to install all the packages you like, without interfering with your standard or other environments. Try `pip list` to see you don't have Django or Pillow installed in your virtual environment. You can now install them with `pip` so that they exist in your virtual environment.

9.5 Version Control

We should also point out that when you develop code, you should always house your code within a version controlled repository such as SVN or Git. We have provided a chapter on using Git if you haven't used Git and GitHub before. We highly recommend that you set up a Git repository for your own projects. Doing so could save you from disaster.

Exercises

To get comfortable with your environment, try out the following exercises.

- Install Python 2.7.5+ or Python 3.4+ and `pip`.
- Play around with your CLI and create a directory called `code`, which we use to create our projects in.
- Install the Django and Pillow packages.
- Setup your Virtual Environment
- Setup your account on GitHub
- Download and setup a Integrated Development Environment like PyCharm Edu.
- We have made the code for the book and application that you build available on GitHub, see Tango With Django Book and Rango Application.
- If you spot any errors or problem with the book, you can make a change request!
- If you have any problems with the exercises, you can check out the repository and see how we completed them.

10. A Crash Course in UNIX-based Commands

Depending on your computing background, you may or may not have encountered a UNIX based system, or a derivative of. This small crash course focuses on getting you up to speed with the *terminal*, an application in which you issue commands for the computer to execute. This differs from a point-and-click *Graphical User Interface (GUI)*, the kind of interface that has made computing so much more accessible. A terminal based interface may be more complex to use, but the benefits of using such an interface include getting things done quicker, and more accurately, too.

Not for Windows!

Note that we're focusing on the Bash shell, a shell for UNIX-based operating systems and their derivatives, including OS X and Linux distributions. If you're a Windows user, you can use the Windows Command Prompt or Windows PowerShell. Users of Windows 10 with the 2016 Anniversary Update will also be able to issue Bash commands directly to the Command Prompt. You could also experiment by installing Cygwin to bring Bash commands to Windows.

10.1 Using the Terminal

UNIX based operating systems and derivatives - such as OS X and Linux distributions - all use a similar looking terminal application, typically using the Bash shell. All possess a core set of commands that allow you to navigate through your computer's filesystem and launch programs - all without the need for any graphical interface.

Upon launching a new terminal instance, you'll be typically presented with something resembling the following.

```
sibu:~ david$
```

What you see is the *prompt*, and indicates when the system is waiting to execute your every command. The prompt you see varies depending on the operating system you are using, but all look generally very similar. In the example above, there are three key pieces of information to observe:

- your username and computer name (username of `david` and computer name of `sibu`);

- your *present working directory* (the tilde, or ~); and
- the privilege of your user account (the dollar sign, or `$`).

What is a Directory?

In the text above, we refer to your present working directory. But what exactly is a *directory*? If you have used a Windows computer up until now, you'll probably know a directory as a *folder*. The concept of a folder is analogous to a directory - it is a cataloguing structure that contains references to other files and directories.

The dollar sign (`$`) typically indicates that the user is a standard user account. Conversely, a hash symbol (`#`) may be used to signify the user logged in has root privileges. Whatever symbol is present is used to signify that the computer is awaiting your input.

Prompts can Differ

The information presented by the prompt on your computer may differ from the example shown above. For example, some prompts may display the current date and time, or any other information. It all depends how your computer is set up.

When you are using the terminal, it is important to know where you are in the file system. To find out where you are, you can issue the command `pwd`. This will display your *Present Working Directory* (hence `pwd`). For example, check the example terminal interactions below.

```
Last login: Wed Mar 23 15:01:39 2016
sibu:~ david$ pwd
/users/grad/david
sibu:~ david$
```

You can see that the present working directory in this example is `/users/grad/david`.

You'll also note that the prompt indicates that the present working directory is a tilde ~. The tilde is used a special symbol which represents your *home directory*. The base directory in any UNIX based file system is the *root directory*. The path of the root directory is denoted by a single forward slash (`/`). As folders (or directories) are separated in UNIX paths with a `/`, a single `/` denotes the root!

If you are not in your home directory, you can *Change Directory* (`cd`) by issuing the following command:

```
sibu:/ david$ cd ~
sibu:~ david$
```

Note how the present working directory switches from `/` to ~ upon issuing the `cd ~` command.

Path Shortcuts

UNIX shells have a number of different shorthand ways for you to move around your computer's filesystem. You've already seen that a forward slash (`/`) represents the root directory, and the tilde (~) represents your home directory in which you store all your personal files. However, there are a few more special characters you can use to move around your filesystem in conjunction with the `cd` command.

- Issuing `cd ~` will always return you to your home directory. On some UNIX or UNIX derivatives, simply issuing `cd` will return you to your home directory, too.
- Issuing `cd ..` will move your present working directory **up one level** of the filesystem hierarchy. For example, if you are currently in `/users/grad/david/code/`, issuing `cd ..` will move you to `/users/grad/david/`.
- Issuing `cd -` will move you to the **previous directory you were working in**. Your shell remembers where you were, so if you were in `/var/tmp/` and moved to `/users/grad/david/`, issuing `cd -` will move you straight back to `/var/tmp/`. This command obviously only works if you've move around at least once in a given terminal session.

Now, let's create a directory within the home directory called `code`. To do this, you can use the *Make Directory* command, called `mkdir`.

```
sibu:~ david$ mkdir code
sibu:~ david$
```

There's no confirmation that the command succeeded. We can change the present working directory with the `cd` command to change to `code`. If this succeeds, we will know the directory has been successfully created.

```
sibu:~ david$ cd code
sibu:code david$
```

Issuing a subsequent `pwd` command to confirm our present working directory yields `/users/grad/david/code` - our home directory, with `code` appended to the end. You can also see from the prompt in the example above that the present working directory changes from ~ to `code`.

Change Back

Now issue the command to change back to your home directory. What command do you enter?

From your home directory, let's now try out another command to see what files and directories exist. This new command is called `ls`, shorthand for *list*. Issuing `ls` in your home directory will yield something similar to the following.

```
sibu:~ david$ ls
code
```

This shows us that there's something present our home directory called `code`, as we would expect. We can obtain more detailed information by adding a `l` switch to the end of the `ls` command - with `l` standing for *list*.

```
sibu:~ david$ ls -l
drwxr-xr-x  2 david  grad  68  2 Apr 11:07 code
```

This provides us with additional information, such as the modification date (`2 Apr 11:07`), whom the file belongs to (user `david` of group `grad`), the size of the entry (`68` bytes), and the file permissions (`drwxr-xr-x`). While we don't go into file permissions here, the key thing to note is the `d` at the start of the string that denotes the entry is a directory. If we then add some files to our home directory and reissue the `ls -l` command, we then can observe differences in the way files are displayed as opposed to directories.

```
sibu:~ david$ ls -l
drwxr-xr-x  2 david  grad      68  2 Apr 11:07 code
-rw-r--r--@ 1 david  grad  303844  1 Apr 16:16 document.pdf
-rw-r--r--  1 david  grad      14  2 Apr 11:14 readme.md
```

One final useful switch to the `ls` command is the `a` switch, which displays *all* files and directories. This is useful because some directories and files can be *hidden* by the operating system to keep things looking tidy. Issuing the command yields more files and directories!

```
sibu:~ david$ ls -la
-rw-r--r--   1  david  grad     463 20 Feb 19:58 .profile
drwxr-xr-x   16 david  grad     544 25 Mar 11:39 .virtualenvs
drwxr-xr-x   2  david  grad      68  2 Apr 11:07 code
-rw-r--r--@  1  david  grad  303844  1 Apr 16:16 document.pdf
-rw-r--r--   1  david  grad      14  2 Apr 11:14 readme.md
```

This command shows a hidden directory `.virtualenvs` and a hidden file `.profile`. Note that hidden files on a UNIX based computer (or derivative) start with a period (`.`). There's no special `hidden` file attribute you can apply, unlike on Windows computers.

Combining `ls` Switches

You may have noticed that we combined the `l` and `a` switches in the above `ls` example to force the command to output a list displaying all hidden files. This is a valid command - and there are even more switches you can use to customise the output of `ls`.

Creating files is also easy to do, straight from the terminal. The `touch` command creates a new, blank file. If we wish to create a file called `new.txt`, issue `touch new.txt`. If we then list our directory, we then see the file added.

```
sibu:~ david$ ls -l
drwxr-xr-x  2 david  grad      68  2 Apr 11:07 code
-rw-r--r--@ 1 david  grad  303844  1 Apr 16:16 document.pdf
-rw-r--r--  1 david  grad       0  2 Apr 11:35 new.txt
-rw-r--r--  1 david  grad      14  2 Apr 11:14 readme.md
```

Note the filesize of `new.txt` - it is zero bytes, indicating an empty file. We can start editing the file using one of the many available text editors that are available for use directly from a terminal, such as `nano` or `vi`. While we don't cover how to use these editors here, you can have a look online for a simple how-to tutorial. We suggest starting with `nano` - while there are not as many features available compared to other editors, using `nano` is much simpler.

10.2 Core Commands

In the short tutorial above, you've covered a few of the core commands such as `pwd`, `ls` and `cd`. There are however a few more standard UNIX commands that you should familiarise yourself with before you start working for real. These are listed below for your reference, with most of them focusing upon file management. The list comes with an explanation of each, and an example of how to use them.

- `pwd`: As explained previously, this command displays your *present working directory* to the terminal. The full path of where you are presently is displayed.
- `ls`: Displays a list of files in the current working directory to the terminal.
- `cd`: In conjunction with a path, `cd` allows you to change your present working directory. For example, the command `cd /users/grad/david/` changes the current working directory to `/users/grad/david/`. You can also move up a directory level without having to provide the absolute path by using two dots, e.g. `cd ..`.
- `cp`: Copies files and/or directories. You must provide the *source* and the *target*. For example, to make a copy of the file `input.py` in the same directory, you could issue the command `cp input.py input_backup.py`.

- `mv`: Moves files/directories. Like `cp`, you must provide the *source* and *target*. This command is also used to rename files. For example, to rename `numbers.txt` to `letters.txt`, issue the command `mv numbers.txt letters.txt`. To move a file to a different directory, you would supply either an absolute or relative path as part of the target - like `mv numbers.txt /home/david/numbers.txt`.
- `mkdir`: Creates a directory in your current working directory. You need to supply a name for the new directory after the `mkdir` command. For example, if your current working directory was `/home/david/` and you ran `mkdir music`, you would then have a directory `/home/david/music/`. You will need to then `cd` into the newly created directory to access it.
- `rm`: Shorthand for *remove*, this command removes or deletes files from your filesystem. You must supply the filename(s) you wish to remove. Upon issuing a `rm` command, you will be prompted if you wish to delete the file(s) selected. You can also remove directories using the recursive switch. Be careful with this command - recovering deleted files is very difficult, if not impossible!
- `rmdir`: An alternative command to remove directories from your filesystem. Provide a directory that you wish to remove. Again, be careful: you will not be prompted to confirm your intentions.
- `sudo`: A program which allows you to run commands with the security privileges of another user. Typically, the program is used to run other programs as `root` - the superuser of any UNIX-based or UNIX-derived operating system.

There's More!

This is only a brief list of commands. Check out Ubuntu's documentation on Using the Terminal for a more detailed overview, or the Cheat Sheet by FOSSwire for a quick, handy reference guide. Like anything else, the more you practice, the more comfortable you will feel working with the terminal.

11. Virtual Environments

Virtual environments allow multiple installations of Python and their relevant packages to exist in harmony. This is the generally accepted approach to configuring a Python setup nowadays.

They are pretty easy to setup. Assuming you have `pip` installed, you can install the following packages:

```
$ pip install virtualenv
$ pip install virtualenvwrapper
```

The first package provides you with the infrastructure to create a virtual environment. See a non-magical introduction to Pip and Virtualenv for Python Beginners by Jamie Matthews for details about using `virtualenv`. However, using just *virtualenv* alone is rather complex. The second package provides a wrapper to the functionality in the `virtualenv` package and makes life a lot easier. The wrapper provides a series of extensions by Doug Hellman to the original `virtualenv` tool, making it easier for us to create, delete and use virtual environments.

If you are using a linux/unix based OS, then to use the wrapper you need to call the following shell script from your command line:

```
$ source virtualenvwrapper.sh
```

It is a good idea to add this to your bash/profile script. You therefore don't have to run it each and every time you want to use a virtual environment. However, if you are using windows, then install the virtualenvwrapper-win package:

```
$ pip install virtualenvwrapper-win
```

Now you should be all set to create a virtual environment:

```
$ mkvirtualenv rango
```

You can list the virtual environments created with `lsvirtualenv`, and you can activate a virtual environment as follows:

```
$ workon rango
(rango)$
```

Your prompt with change and the current virtual environment will be displayed, i.e. rango. Now within this environment you will be able to install all the packages you like, without interfering with your standard or other environments. Try `pip list` to see you don't have `Django` or `Pillow` installed in your virtual environment. You can now install them with `pip` so that they exist in your virtual environment.

In our chapter on deployment, we will go through a similar process when deploying your application to PythonAnywhere.

12. A Git Crash Course

We strongly recommend that you spend some time familiarising yourself with a version control system for your application's codebase. This chapter provides you with a crash course in how to use Git, one of the many version control systems available. Originally developed by Linus Torvalds, Git is today one of the most popular version control systems in use, and is used by open-source and closed-source projects alike.

This tutorial demonstrates at a high level how Git works, explains the basic commands that you can use, and provides an explanation of Git's workflow. By the end of this chapter, you'll be able to make contributions to a Git repository, enabling you to work solo, or in a team.

12.1 Why Use Version Control?

As your software engineering skills develop, you will find that you are able to plan and implement solutions to ever more complex problems. As a rule of thumb, the larger the problem specification, the more code you have to write. The more code you write, the greater the emphasis you should put on software engineering practices. Such practices include the use of design patterns and the *DRY (Don't Repeat Yourself)* principle.

Think about your experiences with programming thus far. Have you ever found yourself in any of these scenarios?

- Made a mistake to code, realised it was a mistake and wanted to go back?
- Lost code (through a faulty drive), or had a backup that was too old?
- Had to maintain multiple versions of a product (perhaps for different organisations)?
- Wanted to see the difference between two (or more) versions of your codebase?
- Wanted to show that a particular change broke of fixed a piece of code?
- Wanted to submit a change (patch) to someone else's code?
- Wanted to see how much work is being done (where it was done, when it was done, or who did it)?

Using a version control system makes your life easier in *all* of the above cases. While using version control systems at the beginning may seem like a hassle it will pay off later - so it's good to get into the habit now!

We missed one final (and important) argument for using version control. With ever more complex problems to solve, your software projects will undoubtedly contain a large number of files containing source code. It'll also be likely that you *aren't working alone on the project; your project will probably have more than one contributor.* In this scenario, it can become difficult to avoid conflicts when working on files.

12.2 How Git Works

Essentially, Git comprises of four separate storage locations: your **workspace**, the **local index**, the **local repository** and the **remote repository**. As the name may suggest, the remote repository is stored on some remote server, and is the only location stored on a computer other than your own. This means that there are two copies of the repository - your local copy, and the remote copy. Having two copies is one of the main selling points of Git over other version control systems. You can make changes to your local repository when you may not have Internet access, and then apply any changes to the remote repository at a later stage. Only once changes are made to the remote repository can other contributors see your changes.

What is a Repository?

We keep repeating the word *repository*, but what do we actually mean by that? When considering version control, a repository is a data structure which contains metadata (a set of data that describes other data, hence *meta*) concerning the files which you are storing within the version control system. The kind of metadata that is stored can include aspects such as the historical changes that have taken place within a given file, so that you have a record of all changes that take place.

If you want to learn more about the metadata stored by Git, there is a technical tutorial available for you to read through.

For now though, let's provide an overview of each of the different aspects of the Git system. We'll recap some of the things we've already mentioned just to make sure it makes sense to you.

- As already explained, the **remote repository** is the copy of your project's repository stored on some remote server. This is particularly important for Git projects that have more than one contributor - you require a central place to store all the work that your team members produce. You could set up a Git server on a computer with Internet access and a properly configured firewall (check out this Server Fault question, for example), or simply use one of many services providing free Git repositories. One of the most widely used services available today is GitHub. In fact, this book has a Git repository on GitHub!
- The **local repository** is a copy of the remote repository stored on your computer (locally). This is the repository to which you make all your additions, changes and deletions. When you reach a particular milestone, you can then *push* all your local changes to the remote repository. From there, you can instruct your team members to retrieve your changes. This concept is known as *pulling* from the remote repository. We'll subsequently explain pushing and pulling in a bit more detail.
- The **local index** is technically part of the local repository. The local index stores a list of files that you want to be managed with version control. This is explained in more detail later in this chapter. You can have a look here to see a discussion on what exactly a Git index contains.

- The final aspect of Git is your **workspace**. Think of this folder or directory as the place on your computer where you make changes to your version controlled files. From within your workspace, you can add new files or modify or remove previously existing ones. From there, you then instruct Git to update the repositories to reflect the changes you make in your workspace. This is important - *don't modify code inside the local repository - you only ever edit files in your workspace.*

Next, we'll be looking at how to get your Git workspace set up and ready to go. We'll also discuss the basic workflow you should use when using Git.

12.3 Setting up Git

We assume that you've got Git installed with the software to go. One easy way to test the software out is to simply issue `git` to your terminal or Command Prompt. If you don't see a `command not found` error, you're good to go. Otherwise, have a look at how to install Git to your system.

Using Git on Windows

Like Python, Git doesn't come as part of a standard Windows installation. However, Windows implementations of the version control system can be downloaded and installed. You can download the official Windows Git client from the Git website. The installer provides the `git` command line program, which we use in this crash course. You can also download a program called *TortoiseGit*, a graphical extension to the Windows Explorer shell. The program provides a really nice right-click Git context menu for files. This makes version control really easy to use. You can download TortoiseGit for free. Although we do not cover how to use TortoiseGit in this crash course, many tutorials exist online for it. Check this tutorial if you are interested in using it.

We recommend however that you stick to the command line program. We'll be using the commands in this crash course. Furthermore, if you switch to a UNIX/Linux development environment at a later stage, you'll be glad you know the commands!

Setting up your Git workspace is a straightforward process. Once everything is set up, you will begin to make sense of the directory structure that Git uses. Assume that you have signed up for a new account on GitHub and created a new repository on the service for your project. With your remote repository setup, follow these steps to get your local repository and workspace setup on your computer. We'll assume you will be working from your `<workspace>` directory.

1. Open a terminal and navigate to your home directory (e.g. `$ cd ~`).
2. *Clone* the remote repository - or in other words, make a copy of it. Check out how to do this below.
3. Navigate into the newly created directory. That's your workspace in which you can add files to be version controlled!

How to Clone a Remote Repository

Cloning your repository is a straightforward process with the `git clone` command. Supplement this command with the URL of your remote repository - and if required, authentication details, too. The URL of your repository varies depending on the provider you use. If you are unsure of the URL to enter, it may be worth querying it with your search engine or asking someone in the know.

For GitHub, try the following command, replacing the parts below as appropriate:

```
$ git clone https://<USER>:<PASS>@github.com/<OWNER>/<REPO_NAME>.git <workspace>
```

where you replace

- `<USER>` with your GitHub username;
- `<PASS>` with your GitHub password;
- `<OWNER>` with the username of the person who owns the repository;
- `<REPO_NAME>` with the name of your project's repository; and
- `<workspace>` with the name for your workspace directory. This is optional; leaving this option out will simply create a directory with the same name as the repository.

If all is successful, you'll see some text like the example shown below.

```
$ git clone https://github.com/leifos/tango_with_django_19
Cloning into 'tango_with_django_19'...
remote: Counting objects: 18964, done.
remote: Total 18964 (delta 0), reused 0 (delta 0), pack-reused 18964
Receiving objects: 100% (18964/18964), 99.69 MiB | 3.51 MiB/s, done.
Resolving deltas: 100% (13400/13400), done.
Checking connectivity... done.
```

If the output lines end with `done`, everything should have worked. Check your filesystem to see if the directory has been created.

Not using GitHub?

There are many websites that provide Git repositories - some free, some paid. While this chapter uses GitHub, you are free to use whatever service you wish. Other providers include Atlassian Bitbucket and Unfuddle. You will of course have to change the URL from which you clone your repository if you use a service other than GitHub.

The Directory Structure

Once you have cloned your remote repository onto your local computer, navigate into the directory with your terminal, Command Prompt or GUI file browser. If you have cloned an empty repository the workspace directory should appear empty. This directory is therefore your blank workspace with which you can begin to add your project's files.

However, the directory isn't blank at all! On closer inspection, you will notice a hidden directory called `.git`. Stored within this directory are both the local repository and local index. **Do not alter the contents of the `.git` directory.** Doing so could damage your Git setup and break version control functionality. *Your newly created workspace therefore actually contains within it the local repository and index.*

Final Tweaks

With your workspace setup, now would be a good time to make some final tweaks. Here, we discuss two cool features you can try which could make your life (and your team members') a little bit easier.

When using your Git repository as part of a team, any changes you make will be associated with the username you use to access your remote Git repository. However, you can also specify your full name and e-mail address to be included with changes that are made by you on the remote repository. Simply open a Command Prompt or terminal and navigate to your workspace. From there, issue two commands: one to tell Git your full name, and the other to tell Git your e-mail address.

```
$ git config user.name "John Doe"
$ git config user.email "johndoe123@me.com"
```

Obviously, replace the example name and e-mail address with your own - unless your name actually is John Doe.

Git also provides you with the capability to stop - or ignore - particular files from being added to version control. For example, you may not wish a file containing unique keys to access web services from being added to version control. If the file were to be added to the remote repository, anyone could theoretically access the file by cloning the repository. With Git, files can be ignored by including them in the `.gitignore` file, which resides in the root of `<workspace>`. When adding files to version control, Git parses this file. If a file that is being added to version control is listed within `.gitignore`, the file is ignored. Each line of `.gitignore` should be a separate file entry.

Check out the following example of a `.gitignore` file:

```
`config/api_keys.py`
`*.pyc`
```

In this example file, there are two entries - one on each line. The first entry prompts Git to ignore the file `api_keys.py` residing within the `config` directory of your repository. The second entry then prompts Git to ignore *all* instance of files with a `.pyc` extension, or compiled Python files. This is a really nice feature: you can use *wildcards* to make generic entries if you need to!

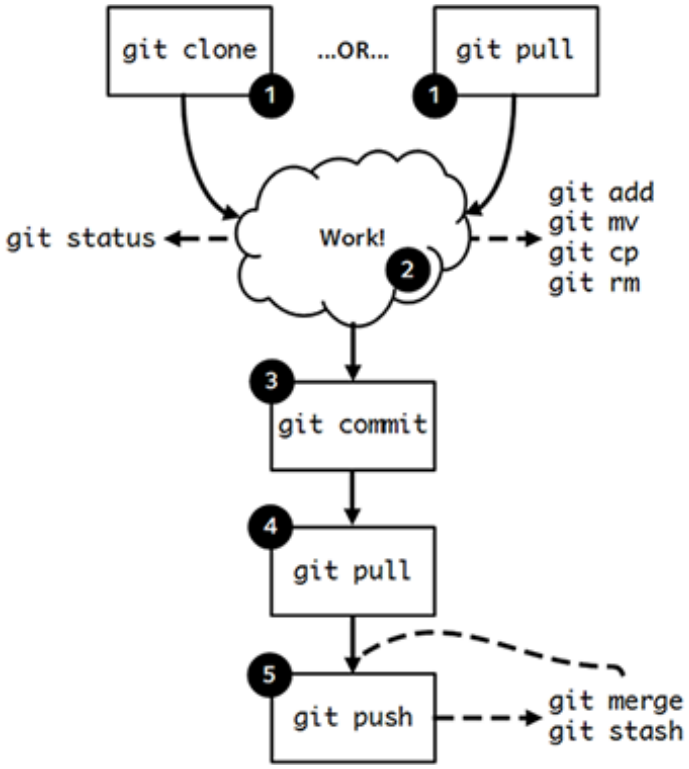
`.gitignore` - What else should I ignore?

There are many kinds of files you could safely ignore from being committed and pushed to your Git repositories. Examples include temporary files, databases (that can easily be recreated) and operating system-specific files. Operating system-specific files include configurations for the appearance of the directory when viewed in a given file browser. Windows computers create `thumbs.db` files, while OS X creates `.DS_Store` files.

When you create a new repository on GitHub, the service can offer to create a `.gitignore` file based upon the languages you will use in your project, which can save you some time setting everything up.

12.4 Basic Commands and Workflow

With your repository cloned and ready to go on your local computer, you're ready to get to grips with the Git workflow. This section shows you the basic Git workflow - and the associated Git commands you can issue.



A Figure of the Git Workflow

We have provided a pictorial representation of the basic Git workflow as shown above. Match each of the numbers in the black circles to the numbered descriptions below to read more about each stage. **Refer to this diagram whenever you're unsure about the next step you should take - it's very useful!**

1. Starting Off

Before you can start work on your project, you must prepare Git. If you haven't yet sorted out your project's Git workspace, you'll need to `clone` your repository to set it up.

If you've already cloned your repository, it's good practice to get into the habit of updating your local copy by using the `git pull` command. This *pulls* the latest changes from the remote repository onto your computer. By doing this, you'll be working from the same page as your team members. This will reduce the possibility of conflicting versions of files, which really does make your life a bit of a nightmare.

To perform a `git pull`, first navigate to your `<workspace>` directory within your Command Prompt or terminal, then issue `git pull`. Check out the snippet below from a Bash terminal to see exactly what you need to do, and what output you should expect to see.

```
$ cd <workspace>
$ git pull
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
From https://github.com/someuser/somerepository
    86a0b3b..a7cec3d  master     -> origin/master
Updating 86a0b3b..a7cec3d
Fast-forward
  README.md | 1 +
  1 file changed, 1 insertion(+)
  create mode 100644 README.md
```

This example shows that a `README.md` file has been updated or created from the latest pull.

Getting an Error?

If you receive `fatal: Not a git repository (or any of the parent directories): .git`, you're not in the correct directory. You need `cd` to your workspace directory - the one in which you cloned your repository to. A majority of Git commands only work when you're in a Git repository.

Pull before you Push!

Always `git pull` on your local copy of your repository before you begin to work. **Always!**

Before you are about to push, do another pull.

Remember to talk to your team to coordinate your activity so you are not working on the same files, or using branching.

2. Doing Some Work!

Once your workspace has been cloned or updated with the latest changes, it's time for you to get some work done! Within your workspace directory, you can take existing files and modify them. You can delete them too, or add new files to be version controlled.

When you modify your repository in any way, you need to keep Git up-to-date of any changes. Doing so allows Git to update your local index. The list of files stored within the local index are then used to perform your next *commit*, which we'll be discussing in the next step. To keep Git informed, there are several Git commands that let you update the local index. Three of the commands are near identical to those that were discussed in the Unix Crash Course (e.g. `cp`, `mv`), with the addition of a `git` prefix.

- The first command `git add` allows you to request Git to add a particular file to the next commit for you. A common newcomer mistake is to assume that `git add` is used for adding new files to your repository only - *this is not the case. You must tell Git what modified files you wish to commit, too.* The command is invoked by typing `git add <filename>`, where `<filename>` is the name of the file you wish to add to your next commit. Multiple files and directories can be added with the command `git add .` - **but be careful with this**.
- `git mv` performs the same function as the Unix `mv` command - it moves files. The only difference between the two is that `git mv` updates the local index for you before moving the file. Specify the filename with the syntax `git mv <current_filename> <new_filename>`. For example, with this command you can move files to a different directory within your repository. This will be reflected in your next commit. The command is also used to rename files - from the old filename to the new.
- `git cp` allows you to make a copy of a file or directory while adding references to the new files into the local index for you. The syntax is the same as `git mv` above where the filename or directory name is specified thus: `git cp <current_filename> <copied_filename>`.
- The command `git rm` adds a file or directory delete request into the local index. While the `git rm` command does not delete the file straight away, the requested file or directory is removed from your filesystem and the Git repository upon the next commit. The syntax is similar to the `git add` command, where a filename can be specified thus: `git rm <filename>`. Note that you can add a large number of requests to your local index in one go, rather than removing each file manually. For example, `git rm -rf media/` creates delete requests in your local index for the `media/` directory. The `r` switch enables Git to *recursively* remove each file within the `media/` directory, while `f` allows Git to *forcibly* remove the files. Check out the Wikipedia page on the `rm` command for more information.

Lots of changes between commits can make things pretty confusing. You may easily forget what files you've already instructed Git to remove, for example. Fortunately, you can run the `git status` command to see a list of files which have been modified from your current working directory, but haven't been added to the local index for processing. Check out typical output from the command below to get a taste of what you can see.

Working with `.gitignore`

If you have set up your `.gitignore` file correctly, you'll notice that files matching those specified within the `.gitignore` file are...ignored when you `git add` them. This is the intended behaviour - these files are not supposed to be committed to version control! If you however do need a file to be included that is in `.gitignore`, you can *force* Git to include it if necessary with the `git add -f <filename>` command.

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
    modified:   chapter-unix.md
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)
    modified:   chapter-git.md
```

From this example above, we can see that the file `chapter-unix.md` has been added to the latest commit, and will therefore be updated in the next `git push`. The file `chapter-git.md` has been updated, but `git add` hasn't been run on the file, so the changes won't be applied to the repository.

Checking Status

For further information on the `git status` command, check out the official Git documentation.

3. Committing your Changes

We've mentioned *committing* several times in the previous step - but what does it mean? Committing is when you save changes - which are listed in the local index - that you have made within your workspace. The more often you commit, the greater the number of opportunities you'll have to revert back to an older version of your code if things go wrong. Make sure you commit often, but don't commit an incomplete or broken version of a particular module or function. There's a lot of discussion as to when the ideal time to commit is. Have a look at this Stack Overflow page for the opinions of several developers. It does however make sense to commit only when everything is working. If you find you need to roll back to a previous commit only to find nothing works, you won't be too happy.

To commit, you issue the `git commit` command. Any changes to existing files that you have indexed will be saved to version control at this point. Additionally, any files that you've requested to be

copied, removed, moved or added to version control via the local index will be undertaken at this point. When you commit, you are updating the *HEAD* of your local repository.

Commit Requirements

In order to successfully commit, you need to modify at least one file in your repository and instruct Git to commit it, through the `git add` command. See the previous step for more information on how to do this.

As part of a commit, it's incredibly useful to your future self and others to explain why you committed when you did. You can supply an optional message with your commit if you wish to do so. Instead of simply issuing `git commit`, run the following amended command.

```
$ git commit -m "Updated helpers.py to include a Unicode conversion function."
```

From the example above, you can see that using the `-m` switch followed by a string provides you with the opportunity to append a message to your commit. Be as explicit as you can, but don't write too much. People want to see at a glance what you did, and do not want to be bored or confused with a long essay. At the same time, don't be too vague. Simply specifying `Updated helpers.py` may tell a developer what file you modified, but they will require further investigation to see exactly what you changed.

Sensible Commits

Although frequent commits may be a good thing, you will want to ensure that what you have written actually *works* before you commit. This may sound silly, but it's an incredibly easy thing to not think about. To reiterate, committing code which doesn't actually work can be infuriating to your team members if they then rollback to a version of your project's codebase which is broken!

4. Synchronising your Repository

Important when Collaborating

Synchronising your local repository before making changes is crucial to ensure you minimise the chance for conflicts occurring. Make sure you get into the habit of doing a `pull` before you `push`.

After you've committed your local repository and committed your changes, you're just about ready to send your commit(s) to the remote repository by *pushing* your changes. However, what if someone

within your group pushes their changes before you do? This means your local repository will be out of sync with the remote repository, meaning that any `git push` command that you issue will fail.

It's therefore always a good idea to check whether changes have been made on the remote repository before updating it. Running a `git pull` command will pull down any changes from the remote repository, and attempt to place them within your local repository. If no changes have been made, you're clear to push your changes. If changes have been made and cannot be easily rectified, you'll need to do a little bit more work.

In scenarios such as this, you have the option to *merge* changes from the remote repository. After running the `git pull` command, a text editor will appear in which you can add a comment explaining why the merge is necessary. Upon saving the text document, Git will merge the changes from the remote repository to your local repository.

Editing Merge Logs

If you do see a text editor on your Mac or Linux installation, it's probably the `vi` text editor. If you've never used vi before, check out this helpful page containing a list of basic commands on the Colorado State University Computer Science Department website. If you don't like vi, you can change the default text editor that Git calls upon. Windows installations most likely will bring up Notepad.

5. Pushing your Commit(s)

Pushing is the phrase used by Git to describe the sending of any changes in your local repository to the remote repository. This is the way in which your changes become available to your other team members, who can then retrieve them by running the `git pull` command in their respective local workspaces. The `git push` command isn't invoked as often as committing - *you require one or more commits to perform a push.* You could aim for one push per day, when a particular feature is completed, or at the request of a team member who is after your updated code.

To push your changes, the simplest command to run is:

```
$ git push origin master
```

As explained on this Stack Overflow question and answer page this command instructs the `git push` command to push your local master branch (where your changes are saved) to the *origin* (the remote server from which you originally cloned). If you are using a more complex setup involving branching and merging, alter `master` to the name of the branch you wish to push.

Important Push?

If your `git push` is particularly important, you can also alert other team members to the fact they should really update their local repositories by pulling your changes. You can do this through a *pull request.* Issue one after pushing your latest changes by invoking the command `git request-pull master`, where master is your branch name (this is the default value). If you are using a service such as GitHub, the web interface allows you to generate requests without the need to enter the command. Check out the official GitHub website's tutorial for more information.

12.5 Recovering from Mistakes

This section presents a solution to a coder's worst nightmare: what if you find that your code no longer works? Perhaps a refactoring went terribly wrong, or another team member without discussion changed something. Whatever the reason, using a form of version control always gives you a last resort: rolling back to a previous commit. This section details how to do just that. We follow the information given from this Stack Overflow question and answer page.

Changes may be Lost!

You should be aware that this guide will rollback your workspace to a previous iteration. Any uncommitted changes that you have made will be lost, with a very slim chance of recovery! Be wary. If you are having a problem with only one file, you could always view the different versions of the files for comparison. Have a look at this Stack Overflow page to see how to do that.

Rolling back your workspace to a previous commit involves two steps: determining which commit to roll back to, an performing the rollback. To determine what commit to rollback to, you can make use of the `git log` command. Issuing this command within your workspace directory will provide a list of recent commits that you made, your name and the date at which you made the commit. Additionally, the message that is stored with each commit is displayed. This is where it is highly beneficial to supply commit messages that provide enough information to explain what is going on. Check out the following output from a `git log` invocation below to see for yourself.

```
commit 88f41317640a2b62c2c63ca8d755feb9f17cf16e                <- Commit hash
Author: John Doe <someaddress@domain.com>                      <- Author
Date:   Mon Jul 8 19:56:21 2013 +0100                          <- Date/time
    Nearly finished initial version of the requirements chapter <- Message
commit f910b7d557bf09783b43647f02dd6519fa593b9f
Author: John Doe <someaddress@domain.com>
Date:   Wed Jul 3 11:35:01 2013 +0100
    Added in the Git figures to the requirements chapter.
commit c97bb329259ee392767b87cfe7750ce3712a8bdf
Author: John Doe <someaddress@domain.com>
Date:   Tue Jul 2 10:45:29 2013 +0100
    Added initial copy of Sphinx documentation and tutorial code.
commit 2952efa9a24dbf16a7f32679315473b66e3ae6ad
Author: John Doe <someaddress@domain.com>
Date:   Mon Jul 1 03:56:53 2013 -0700
    Initial commit
```

From this list, you can choose a commit to rollback to. For the selected commit, you must take the commit hash - the long string of letters and numbers. To demonstrate, the top (or `HEAD`) commit hash in the example output above is `88f41317640a2b62c2c63ca8d755feb9f17cf16e`. You can select this in your terminal and copy it to your computer's clipboard.

With your commit hash selected, you can now rollback your workspace to the previous revision. You can do this with the `git checkout` command. The following example command would rollback to the commit with hash `88f41317640a2b62c2c63ca8d755feb9f17cf16e`.

```
$ git checkout 88f41317640a2b62c2c63ca8d755feb9f17cf16e .
```

Make sure that you run this command from the root of your workspace, and do not forget to include the dot at the end of the command! The dot indicates that you want to apply the changes to the entire workspace directory tree. After this has completed, you should then immediately commit with a message indicating that you performed a rollback. Push your changes and alert your collaborators - perhaps with a pull request. From there, you can start to recover from the mistake by putting your head down and getting on with your project.

Exercises

If you haven't undertaken what we've been discussing in this chapter already, you should go through everything now to ensure your Git repository is ready to go. To try everything out, you can create a new file `README.md` in the root of your `<workspace>` directory. The file will be used by GitHub to provide information on your project's GitHub homepage.

- Create the file, and write some introductory text to your project.
- Add the file to the local index upon completion of writing, and commit your changes.
- Push the new file to the remote repository and observe the changes on the GitHub website.

Once you have completed these basic steps, you can then go back and edit the readme file some more. Add, commit and push - and then try to revert to the initial version to see if it all works as expected.

There's More!

There are other more advanced features of Git that we have not covered in this chapter. Examples include **branching** and **merging**, which are useful for projects with different release versions, for example. There are many fantastic tutorials available online if you are interested in taking you super-awesome version control skills a step further. For more details about such features take a look at this tutorial on getting started with Git, the Git Guide or Learning about Git Branching.

However, if you're only using this chapter as a simple guide to getting to grips with Git, everything that we've covered should be enough. Good luck!