Tango With Django 2

A beginner's guide to web development with Django 2.

Leif Azzopardi and David Maxwell

This book is for sale at http://leanpub.com/tangowithdjango2

This version was published on 2019-10-21

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 ready to @tangowithdjango too! Check out https://www.tangowithdjango.com

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:

#tangowithdjango

Also By These Authors

Books by Leif Azzopardi

How to Tango with Django 1.9/1.10/1.11

Books by David Maxwell

How to Tango with Django 1.9/1.10/1.11

Contents

CONTENTS

1. Overview

This book aims to provide you with a practical guide to web development using *Django 2* and *Python 3*. The book is designed primarily for students, providing a walkthrough of the steps involved in getting a web application up and running with Django. However, anyone who's starting off with web development will find this book to be beneficial.

This book seeks to complement the official Django Tutorials[1] 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 by 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 (such as HTML, CSS and JavaScript).

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 getting stuck.

This book will lower the learning curve. Web application frameworks can save you a lot of hassle and a lot of time. But that is only true 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 and how to build your web app logically.

---

[1] https://docs.djangoproject.com/en/2.1/intro/tutorial01/

This book will improve your workflow. Using web application frameworks requires you to pick up and run with particular design patterns – 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 the software engineer (i.e. inversion of control[2]). To help you, we've created several *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. If you still do not understand, then check out the Django documentation, go to Stack Overflow[3] or other helpful websites and fill in this gap in your knowledge. If you are stuck, get in touch with us, so that we can improve the book – we've already had contributions from numerous other readers!

1.2 What you will Learn

In this book, we will be taking an example-based approach to web application development. In the process, we will show you how to design a web application called *Rango* (see the Design Brief below), and take a step by step in setting up, developing and deploying the application. Along the way, we'll show you how to perform the following key tasks which are common to most software engineering and web-based projects.

- How to **configure your development environment** – including how to use the terminal, your virtual environment, the `pip` installer, and how to work with Git.
- How to **set up a Django project** and **create a basic Django application**.
- How to **configure the Django project** to serve static media and user-uploaded media files (such as profile images).

---

[2]https://en.wikipedia.org/wiki/Inversion_of_control
[3]http://stackoverflow.com/questions/tagged/django

- How to **work with Django's Model-View-Template design pattern**.
- How to **work with database models** and use the *object-relational mapping (ORM)*[4] functionality provided by Django.
- How to **create forms** that can utilise your database models to create **dynamically-generated webpages**.
- How to use the **user authentication** services provided by Django.
- How to incorporate **external services** into your Django application.
- How to include **Cascading Styling Sheets (CSS)** and **JavaScript** within a web application to aid in styling and providing it with additional functionality.
- How to **apply CSS** to give your application a professional look and feel.
- How to work with **cookies and sessions** with Django.
- How to include more advanced functionality like **AJAX** into your application.
- How to **write class-based views** with Django.
- How to **Deploy your application** to a web server using *PythonAnywhere.*

At the end of each chapter, we have also included several exercises designed to push you to apply what you have learnt during the chapter. To push you harder, we've also included several open development challenges, which require you to use many of the lessons from the previous chapters – but don't worry, as we've also included solutions and explanations on these, too!

Exercises

In each chapter, we have added several exercises to test your knowledge and skill. Such exercises are denoted like this.

You will need to complete all of these exercises as subsequent chapters will assume that you have fully completed them.

Hints and Tips

For each set of exercises, we will provide a series of hints and tips that will assist you if you need a push. If you get stuck however, you can always check out our solutions to all the exercises on our *GitHub* repository[5].

---

[4]https://en.wikipedia.org/wiki/Object-relational_mapping
[5]https://github.com/maxwelld90/tango_with_django_2_code

1.3 Technologies and Services

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

- the Python[6] programming language;
- the Pip package manager[7];
- Django[8];
- unit testing[9];
- the Git[10] version control system;
- GitHub[11];
- HTML[12];
- CSS[13];
- the JavaScript[14] programming language;
- the JQuery[15] library;
- the Twitter Bootstrap[16] framework;
- the Bing Search API[17]; and
- the PythonAnywhere[18] hosting service.

We've selected all of 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 *Microsoft Bing Search API* and deploy your application quickly and easily with *PythonAnywhere*. Let's get started!

---

[6]https://www.python.org
[7]https://pip.pypa.io/en/stable/
[8]https://www.djangoproject.com
[9]https://en.wikipedia.org/wiki/Unit_testing
[10]https://git-scm.com
[11]https://github.com
[12]https://www.w3.org/html/
[13]https://www.w3.org/Style/CSS/
[14]https://www.javascript.com/
[15]https://jquery.com
[16]https://getbootstrap.com/
[17]https://docs.microsoft.com/en-gb/rest/api/cognitiveservices/bing-web-api-v7-reference
[18]https://www.pythonanywhere.com

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 our How to Tango with Django website[19].

Design Brief

Let's imagine that we would like to create a website called *Rango* that lets users browse through user-defined categories to access various web pages. In Spanish, the word rango[20] is used to mean *"a league ranked by quality"* or *"a position in a social hierarchy"* – so we can imagine that at some point, we will want to rank the web pages in Rango.

- 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 and/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/`.

---

[19]http://www.tangowithdjango.com/
[20]https://www.vocabulary.com/dictionary/es/rango

- Only *registered users will be able to search and add pages to categories*. Therefore, visitors to the site should 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 several 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.

- What is the high-level architecture going be? Draw up a **N-Tier or System Architecture** diagram to represent the high-level system components.
- What is the interface going to look like? Draw up some **Wireframes** of the main and category pages.
- What are the URLs that users visit going to look like? Write down a series of **URL mappings** for the application.
- What data are we going to have to store or represent? Construct an **Entity-Relationship (ER)**[21] 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, formally, what you are going to build so that someone else can understand it.

---

[21]https://en.wikipedia.org/wiki/Entityâ€"relationship_model

N-Tier Architecture

The high-level architecture for most web applications is based around a *3-Tier architecture.* Rango will be a variant on this architecture as also interfaces with an external service.
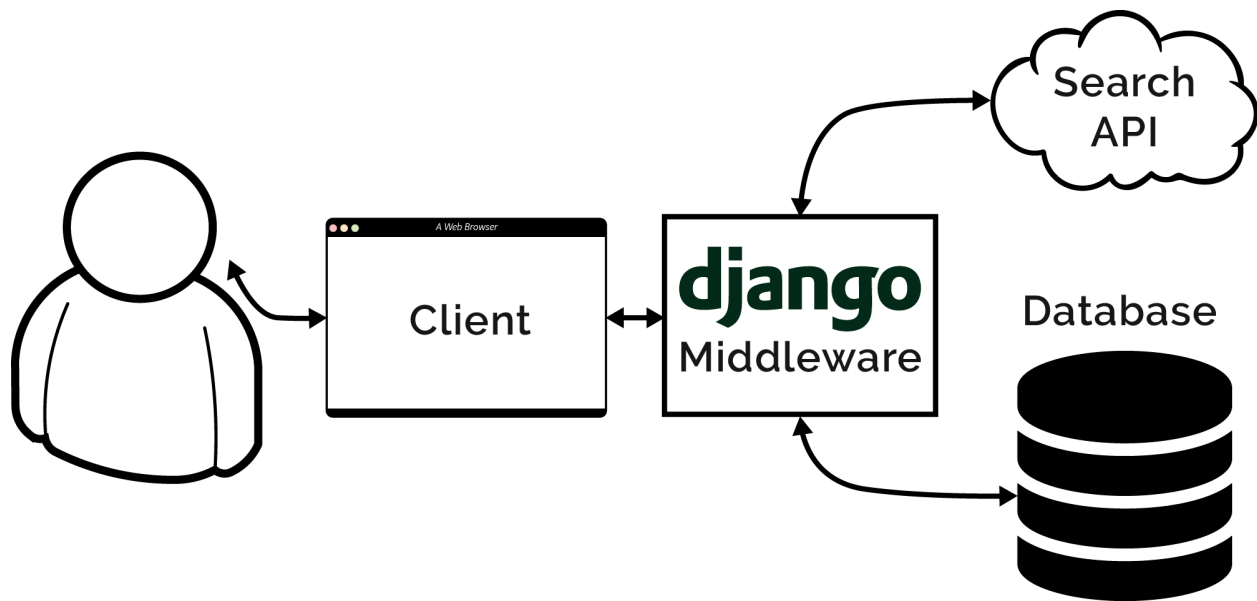
Overview of the 3-tier system architecture for our Rango application.

Given the different boxes within the high-level architecture, we need to start making some decisions about the technologies that will be going into each box. 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, and any interpret JavaScript code.
- The **middleware** will be a *Django* application and will be dispatched through Django's built-in development web server while we develop (and then later a web server like *Nginx* or *Apache web server*).
- The **database** will be the Python-based *SQLite3* Database engine.
- The **search API** will be the *Bing Search API*.

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

Wireframes

Wireframes are a great way to provide clients with some idea of what the application is going to look like, and what features it will provide. They 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.
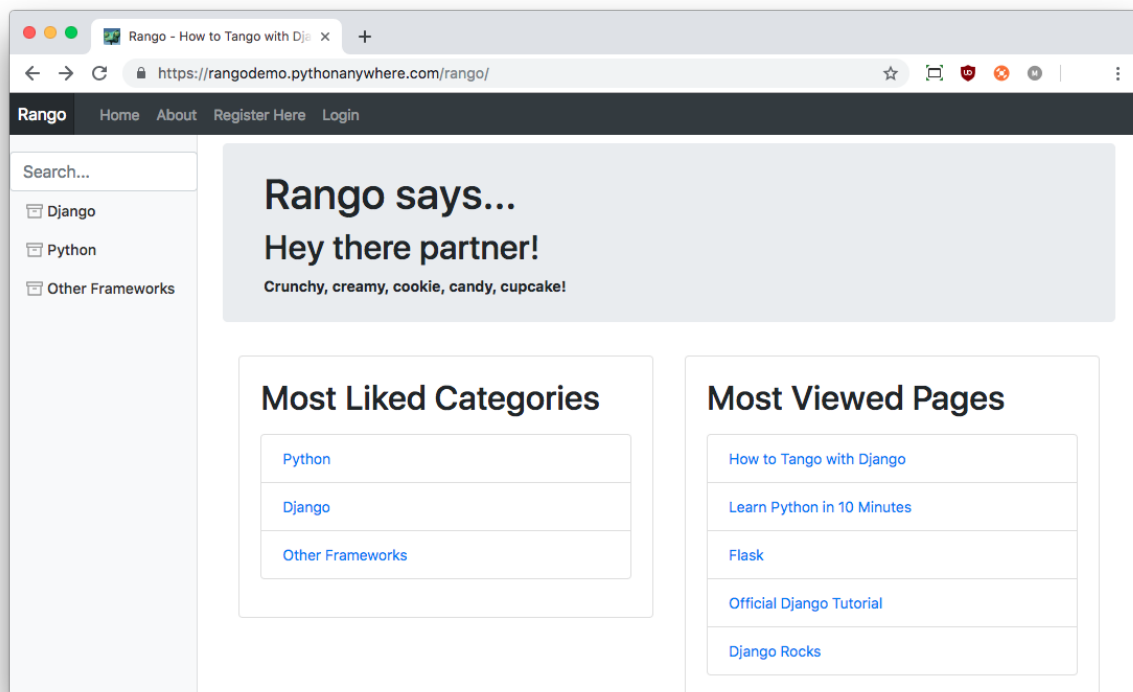
The index page with a categories search bar on the left, also showing the top five pages and top five categories.
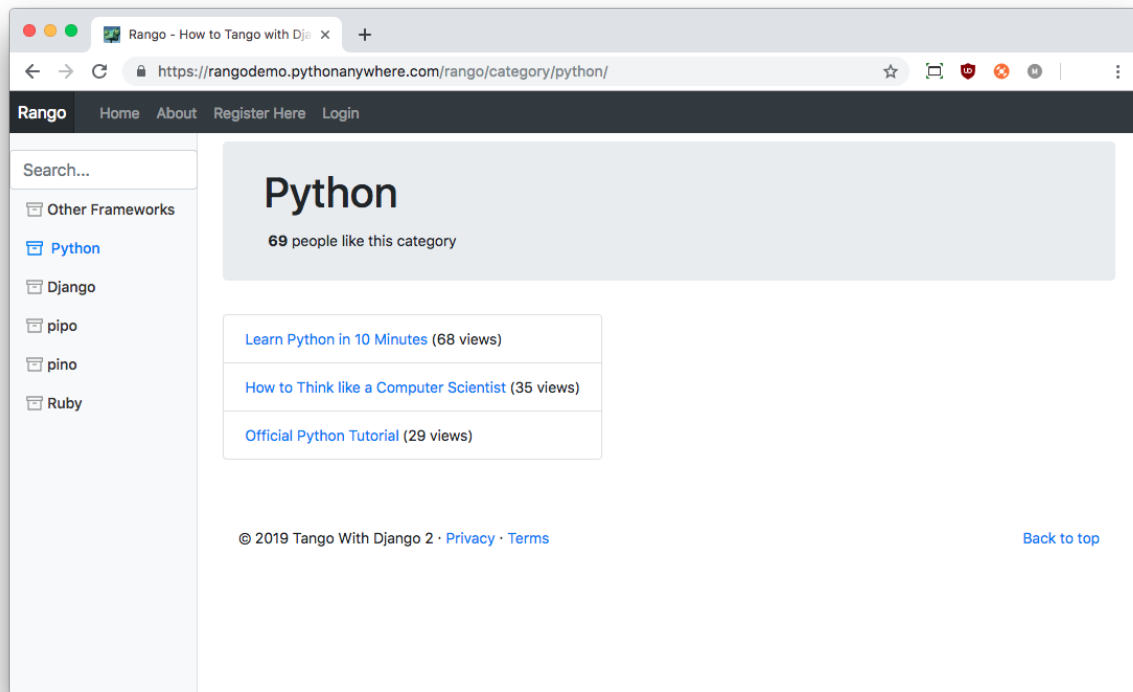
The category page showing the pages in the category (along with the number of views for the category and each page).

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 would have to enter into a browser's address bar to access a 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 mappings listed above are enough for us to get started. As we progress through the book, we will flesh out how to construct all of these pages using the Django framework and use its Model-View-Template[22] design pattern.

Entity-Relationship Diagram

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. 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 be associated with many *pages*. We can formulate the following ER Diagram to describe this simple data model.
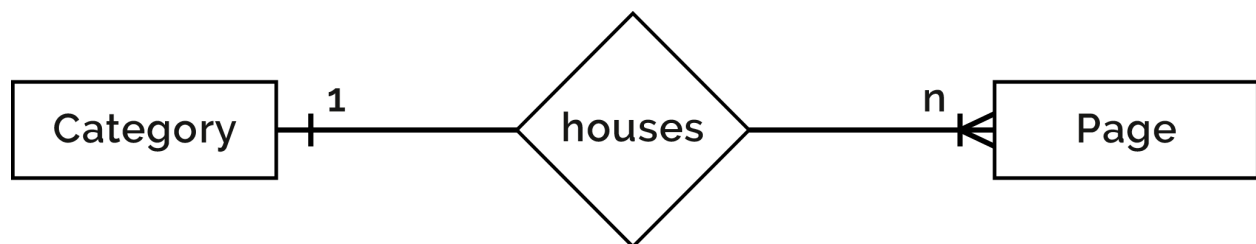
The Entity Relationship Diagram of Rango's two main entities.

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[23]. However, this approach introduces several complexities.

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. While this is not ideal, it does reduce the complexity of the models.

---

[22]https://docs.djangoproject.com/en/2.1/
[23]https://en.wikipedia.org/wiki/Many-to-many_(data_model)

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 can 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 Databases chapter.

Category Model Fields and Data Types

Field	Data Type
name	CharField
views	IntegerField
likes	IntegerField

Page Model Fields and Data Types

Field	Data 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 subsequent chapters, we will see how to instantiate these models in Django, and how we can use the built-in ORM to interact with 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 and get a feel for the commands that you will be using again and again.

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 with regards to tabs and spaces – mixing these up will likely lead to a `TabError`.

Most code editors will show the 'hidden characters', which in turn will show whether whitespace is either a tab or a space. If you have this option, turn it on. You will likely save yourself a lot of confusion.

🔑 Representing Commands

As you work through this book, you'll encounter lots of text that will be entered into your computer's terminal or Command Prompt. Snippets starting with a dollar sign ($) denotes a command that must be entered – the remainder of the line is the command. In a UNIX terminal, the dollar represents a separator between the *prompt* and the command that you enter.

```
david@seram:~ $ exit
```

In the example above, the prompt `david@seram:`~ tells us our username (`david`), computer name (`seram`) and our current directory (~, or our home directory). After the `$`, we have entered the command `exit`, which, when executed, will close the terminal. As such, the command that you would actually type here would simply be `exit`. Refer to the UNIX chapter for more information.

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? Once inside the Python interpreter, you can exit it by typing `quit()` or `exit()`.

2. Getting Ready to Tango

Before we start coding, it's really important that we set your development environment up correctly so that you can *Tango with Django* with ease. You'll need to make sure that you have all of the necessary components installed on your computer, and that they are configured correctly. This chapter outlines the six key components you'll need to be aware of, setup and use. These are:

- the terminal[1] (on macOS or UNIX/Linux systems), or the Command Prompt[2] (on Windows);
- *Python 3*, including how to code and run Python scripts;
- the Python Package Manager *pip*;
- *Virtual Environments*;
- your *Integrated Development Environment (IDE)*, if you choose to use one; and
- a *Version Control System (VCS)* called *Git*.

We also touch on the merits of *unit testing*, discussing how being able to test your implementation at different points can help keep you on track.

If you already have Python 3 and Django 2 installed on your computer and are familiar with the technologies listed above, you can skip straight ahead to the Django Basics chapter. If you are not familiar with some or all of the technologies listed, we provide an overview of each below. These go hand in hand with later supplementary chapter that provides a series of pointers on how to set the different components up, if you need help doing so.

---

[1]https://en.wikipedia.org/wiki/Terminal_emulator
[2]https://en.wikipedia.org/wiki/Cmd.exe

You Development Environment is Important!

Setting up your development environment can be a tedious and frustrating process. It's not something that you would do every day. The pointers we provide in this chapter (and the additional supplementary chapter) will help you in getting everything to a working state. The effort you expend now in making sure everything works will ensure that development can proceed unhindered, without you needing to go back and patch things up.

From experience, we can also say with confidence that as you set your environment up, it's a good idea to note down the steps that you took. You will probably need that workflow again one day – maybe you will purchase a new computer, or be asked to help a friend set their environment up, too. *Don't think short-term, think long-term!*

2.1 Python 3

To work with Tango with Django, we require you to have installed on your computer a copy of the *Python 3* programming language. A Python version of 3.5 or greater should work fine with Django 2.0, 2.1 and 2.2 – although the official Django website recommends that you have the most recent version of Python installed. As such, we recommend you install *Python 3.7*. At the time of writing, the most recent release is *Python 3.7.2.* If you're not sure how to install Python and would like some assistance, have a look at our quick guide on how to install Python.

Running macOS, Linux or UNIX?

On installations of macOS, Linux or UNIX, you will find that Python is already installed on your computer – albeit a much older version, typically 2.x. This version is required by your operating system to perform essential tasks such as downloading and installing updates. While you can use this version, it won't be compatible with Django 2, and you'll need to install a newer version of Python to run *side-by-side* with the old installation. *Do not uninstall or hack away at deleting Python 2.x* if it is already present on your system; you may break your operating system!

Django 2.0, 2.1 or 2.2?

In this book, we explicitly use Django version `2.1.5`. However, we have also tested the instructions provided with versions `2.0.13`, `2.1.10`, and `2.2.3`. Therefore, you will be able to use version 2.2 if you wish! If you do use a different version, substitute `2.1.5` with the version you are using.

We'll be regularly checking the compatibility of the instructions provided with future Django releases. If you notice any issues with later versions, feel free to get in touch with us. You can send us a tweet[3], raise an issue on GitHub[4], or e-mail us – our addresses are available on the www.tangowithdjango.com[5] website.

You must however make sure you are using *at least* Python version 3.5. Version 3.4 and below are incompatible with these releases of Django.

Python Skills Rusty?

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

- **The Official Python Tutorial**[6];
- **Think Python: How to Think like a Computer Scientist**[7] by Allen B. Downey; or
- **Learn Python in 10 Minutes**[8] by Stavros;
- **Learn to Program**[9] by Jennifer Campbell and Paul Gries.

These guides will help you familiarise yourself with the basics of Python so you can start developing with Django. Note you don't need to be an expert in Python to work with Django – Python is straightforward to use, and you can pick it up as you go, especially if you already know the ins and outs of at least one other programming language.

---

[3] https://twitter.com/tangowithdjango
[4] https://github.com/leifos/tango_with_django_2/issues
[5] https://www.tangowithdjango.com
[6] https://docs.python.org/3/tutorial/
[7] https://greenteapress.com/wp/think-python-2e/
[8] https://www.stavros.io/tutorials/python/
[9] https://www.coursera.org/course/programming1

2.2 Virtual Environments

With a working installation of Python 3 (and the basic programming skills to go with it), we can now setup our environment for the Django project (called Rango) we'll be creating in this tutorial. One super useful tool we *strongly* encourage you to use is a virtual environment. Although not strictly necessary, it provides a useful separation between your computer's Python installation and the environment you'll be using to develop Rango with.

A virtual environment allows for multiple installations of Python packages to exist in harmony, within unique *Python environments*. Why is this useful? Say you have a project, `projectA` that you want to run in Django 1.11, and a further project, `projectB` written for Django 2.1. This presents a problem as you would normally only be able to install one version of the required software at a time. By creating virtual environments for each project, you can then install the respective versions of Django (and any other required Python software) within each unique environment. This ensures that the software installed in one environment does not tamper with the software installed on another.

You'll want to create a virtual environment using Python 3 for your Rango development environment. Call the environment `rangoenv`. If you are unsure as to how to do this, go to the supplementary chapter detailing how to set up virtual environments before continuing. If you do choose to use a virtual environment, remember to activate the virtual environment by issuing the following command.

```
$ workon rangoenv
```

From then on, all of your prompts with the terminal or Command Prompt will precede with the name of your virtual environment to remind you that it is switched on. Check out the following example to know what we are discussing.

```
$ workon rangoenv
(rangoenv) $ pip install django==2.1.5
...
(rangoenv) $ deactivate
$
```

The penultimate line of the example above demonstrates how to switch your virtual environment off after you have finished with it – note the lack of `(rangoenv)` before the prompt. Again, refer to the system setup chapter in the appendices of this book for more information on how to setup and use virtual environments.

2.3 The Python Package Manager

Going hand in hand with virtual environments, we'll also be making use of the Python package manager, *pip*, to install several different Python software packages – including Django – to our development environment. Specifically, we'll need to install two packages: Django 2 and *Pillow*. Pillow is a Python package providing support for handling image files (e.g. `.jpg` and `.png` files), something we'll be doing later in this tutorial.

A package manager, whether for Python, your operating system[10] or some other environment[11], 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 that provides some functionality. This is opposed to downloading, installing and maintaining software manually.

Maintaining Python packages is pretty painful. Most packages often have *dependencies* – additional packages that are required for your package to work! This can get very complex very quickly. A package manager handles all of this for you, along with issues such as conflicts regarding different versions of a package. Luckily, *pip* handles all this for you.

Try and run the command `$ pip` to execute the package manager. Make sure you do this with your virtual environment `acivated`. Globally, you may have to use the

---

[10]https://en.wikipedia.org/wiki/Advanced_Packaging_Tool
[11]https://docs.npmjs.com/cli/install

command `pip3`. If these don't work, you have a setup issue – refer to our `pip` setup guide for help.

With your virtual environment switched on, execute the following two commands to install Django and Pillow.

```
$ pip install django==2.1.5
$ pip install pillow==5.4.1
```

Installing these two packages will be sufficient to get you started. As you work through the tutorial, there will be a couple more packages that we will require. We'll tell you to install them as we require them. For now, you're good to go.

🛈 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==5.4.1 --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[12].

🔑 Working within in a Virtual Environment

Substitute `pip3` with `pip` when working within your virtual environment. The command `pip` is aliased to the correct one for your virtual environment.

---

[12]https://pillow.readthedocs.io/en/stable/installation.html

2.4 Integrated Development Environment

While not necessary, a good Python-based IDE can be very helpful to you during the development process. Several exist, with perhaps *PyCharm*[13] by JetBrains and *PyDev* (a plugin of the Eclipse IDE[14]) standing out as popular choices. The Python Wiki[15] 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. Of course, if you prefer not to use an IDE, using a simple text editor like Sublime Text[16], TextMate[17] or Atom[18] will do just fine. Many modern text editors support Python syntax highlighting, which makes things much easier!

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 does provide a guide on setting PyCharm up[19].

2.5 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[20] or Git[21]. 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 projects.**

---

[13]http://www.jetbrains.com/pycharm/
[14]http://www.eclipse.org/downloads/
[15]http://wiki.python.org/moin/IntegratedDevelopmentEnvironments
[16]https://www.sublimetext.com/
[17]https://macromates.com/
[18]https://atom.io/
[19]https://www.jetbrains.com/help/pycharm/2016.1/creating-and-running-your-first-django-project.html
[20]http://subversion.tigris.org/
[21]http://git-scm.com/

✏ Exercises

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

- Get up to speed with Python if you're new to the language. Try out one or more of the tutorials we listed earlier.
- Install Python 3.7. Make sure `pip3` (or `pip` within your virtual environment) is also installed and works on your computer.
- Play around with your *command line interface (CLI)*, whether it be the Command Prompt (Windows) or a terminal (macOS, Linux, UNIX, etc.).
- Create a new virtual environment using Python 3.7. This is optional, but we *strongly encourage you to use virtual environments.*
- Within your environment, install Django 2 and Pillow 5.4.1.
- Set up an account on a Git repository site like GitHub[22] or BitBucket[23] if you haven't already done so.
- Download and set up an IDE like PyCharm[24], or set up your favourite text editor for working with Python files.

As previously stated, we've made the code for the application available on our GitHub repository[25].

- If you spot any errors or problems in the book, please let us know by making an issue on GitHub[26].
- If you have any problems with the exercises, you can check out the repository to see how we completed them.
- If you spot any errors or problems with the unit tests and/or sample solutions we provide, you can also let us know by raising an issue in the relevant repository[27].

---

[22] https://github.com/
[23] https://bitbucket.org/
[24] https://www.jetbrains.com/pycharm/
[25] https://github.com/maxwelld90/tango_with_django_2_code
[26] https://github.com/leifos/tango_with_django_2/issues
[27] https://github.com/maxwelld90/tango_with_django_2_code/issues

2.6 Testing your Implementation

As you work through your implementation of the requirements for the Rango app, we want you to have the confidence to know that *what you are coding up is correct.* We can't physically sit next to you, so we've gone and done the next best thing – **we've implemented a series of different tests that you can run against your codebase to see what's correct, and what can be improved**.

These are available from our sample codebase repository, available on GitHub[28]. The `progress_tests` directory on this repository contains a number of different Python modules, each containing a series of different test modules you can run against your Rango implementation. Note that they are for individual chapters – for example, you should run the module `tests_chapter3.py` against your implementation *after* completion of Chapter 3, but before starting Chapter 4. Note that not every chapter will have tests at the end of it – some portions of the book are very difficult to write automated tests for without resorting to having to download sizable support libraries.

> ⚠️ **Complete the Exercises!**
>
> These tests assume that you complete all of the exercises for a chapter! If you don't do this, it's likely some tests will not pass.

We check the basic functionality that should be working up to the point you are testing at. We also check what is returned from the server when a particular URL is accessed – and if the response doesn't match *exactly* what we requested in the book, *the test will fail.* This might seem overly harsh, but we want to drill into your head that *you must satisfy requirements exactly as they are laid out – no deviation is acceptable.* This also drills into your head the idea of *test-driven development,* something that we outline at the start of the testing chapter.

Running the Unit Tests

How do you run the tests, though? This step-by-step process demonstrates the basic process on what you have to do. We will assume that you want to run the tests

---

for Chapter 3, Django Basics.

1. First, identify what chapter's tests you want to run.
2. Either make a clone of our sample code repository[29] on your computer, or access the individual test module that you want from the GitHub web interface[30].
   - To do the latter, click the module you require (i.e. `tests_chapter3.py`). When you see the code on the GitHub website, click the `Raw` button and save the page that then loads.
3. Move the `tests_chapter3.py` module to your project's `rango` directory. This step does not make sense right now; as you progress through the book and come back here to refresh your memory on what to do, this will make sense.
4. Run the command $ `python manage.py test rango.tests_chapter3`. This will start the tests.

You will also need to ensure that when these tests run, your `rangoenv` virtual environment is active.

Once the tests all complete, you should see `OK`. This means they all passed! If you don't see `OK`, something failed – look through the output of the tests to see what test failed, and why. Sometimes, you might have missed something which causes an exception to be raised before the test can be carried out. In instances like this, you'll need to look at what is expected, and go back and fill it in. You can tweak your code and re-run the tests to see if they then pass.

✏️ Test your Implementation

When you have completed enough of the book to reach another round of tests, we'll denote the prompt for you to do this like so. We'll tell you what module to run, and always point you back to here so you can refresh your memory if you forget how to run them.

---

[29]https://github.com/maxwelld90/tango_with_django_2_code
[30]https://github.com/maxwelld90/tango_with_django_2_code/tree/master/progress_tests

Delete when Complete!

When you have finished with the tests for a particular chapter, we **highly recommend** that you delete the module that you moved over to your `rango` directory. In the example above, we'd be looking to delete `tests_chapter3.py`. Once you have confirmed your solution passes the tests we provide, there's no need for the module anymore. Just delete it – don't clutter your repository up with these modules!

Test Updates

Over time, we may release updates to the unit test files if an issue is raised, or we decide to add extra tests. You should keep an eye on the repository for any changes – if you cloned it to your computer, you can simply `git pull` to retrieve updates.

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 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/Command Prompt window, and `activate` your `rangoenv` virtual environment.

Once activated, issue the following command. The output will tell what Python version you have.

```
$ python --version
```

The response should be something like `3.7.2`, but any 3.5+ 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 then have a look at 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 3.7.2 (default, Mar 30 2019, 05:40:15)
[Clang 9.0.0 (clang-900.0.39.2)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

At the prompt, enter the following commands:

```
>>> import django
>>> django.get_version()
'2.1.5'
>>> 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[1].

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, we recommend that you create one. This means that you can house your Django projects (and other code projects) within this directory. It keeps things organised, without you placing directories containing code in random places, such as your `Desktop` directory!

We will refer to your `workspace` directory throughout this book as `<workspace>`. You will have to substitute this with the path to your `workspace` directory. For example,

---

[1]https://docs.djangoproject.com/en/2.1/topics/install/

we recommend that you create a `workspace` directory in your home folder. The path `/Users/maxwelld90/Workspace/` would then constitute as a valid directory for the user `maxwelld90` on a Mac.

> ℹ **Can't find `django-admin.py`?**
>
> Try entering `django-admin` instead. Depending on your setup, some systems may not recognise `django-admin.py`. This is especially true on Windows computers – you may have to use the full path to the `django-admin.py` script, for example:
>
> ```
> python c:\Users\maxwelld90\.virtualenvs\rangoenv\bin\django-admin.py
>         startproject tango_with_django_project
> ```
>
> as suggested on StackOverflow[2]. Note that the path will likely vary on your own computer.

This command will invoke the `django-admin.py` script, which will set up a new Django project called `tango_with_django_project` for you. Naming conventions dictate that we would typically append `_project` to the end of our Django project directories so we know exactly what they contain – but naming this is really 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:

- `tango_with_django_project`, another *nested directory* with the same name as your main project directory (argh!); and
- a Python script called `manage.py`.

For the purposes of this tutorial, we call the `tango_with_django_project` nested directory 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:

---

[2]http://stackoverflow.com/questions/8112630/cant-create-django-project-using-command-prompt

- `__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 that 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[3].
>
> 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:

---

[3]https://docs.djangoproject.com/en/2.1/ref/django-admin/#django-admin-py-and-manage-py

```
$ python manage.py runserver

Performing system checks...

System check identified no issues (0 silenced).

You have 14 unapplied migration(s).
Your project may not work properly until you apply the migrations for app(s):
admin, auth, contenttypes, sessions.

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

July 23, 2019 - 17:12:34
Django version 2.1.5, 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 several things. First, there are no issues that stop the application from running. However, you will notice that a warning is raised – `unapplied migration(s)`. We will talk about this in more detail when we set up 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 that the Django development server is running at.

Now open up your web browser and enter the URL mentioned above into your browser's address bar – `http://127.0.0.1:8000/`[4]. You should see a webpage similar to the one shown below.

---

[4]http://127.0.0.1:8000/
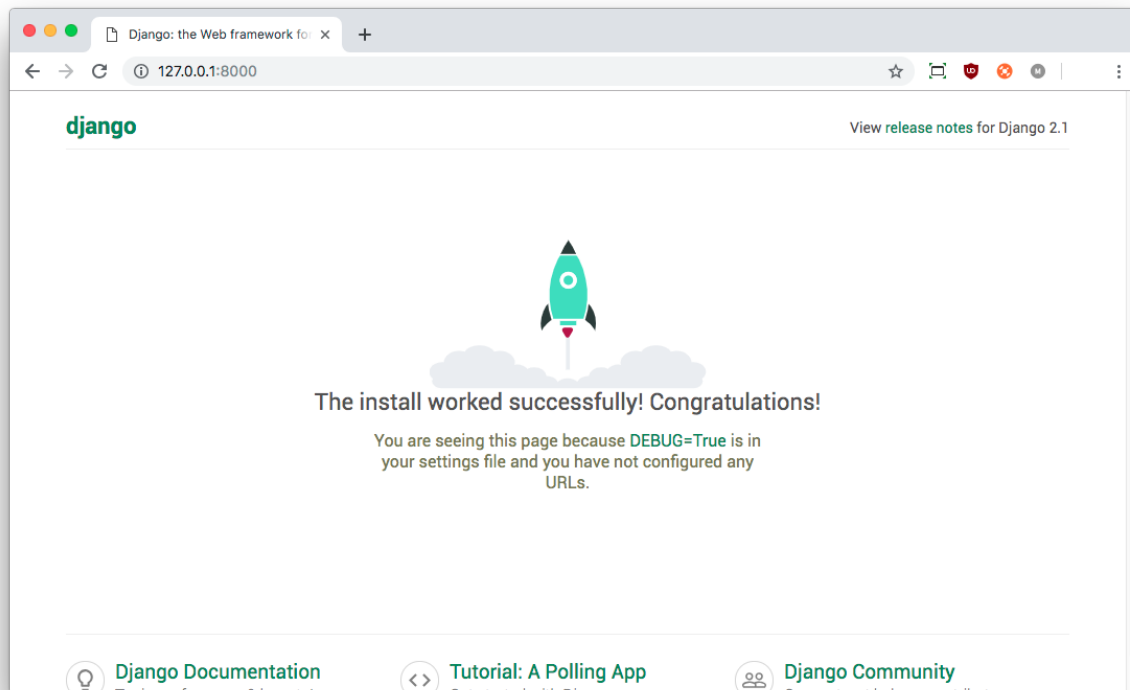
A screenshot of the initial Django page you will see when running the development server for the first time.

You can stop the development server at any time by pushing `CTRL + C` in your terminal or Command Prompt window. This applies to both Macs and PCs! 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[5] 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 achieve this, you need to make sure you're in your Django project's directory (e.g. `<workspace>/tango_with_django_project`). From there, run the following command.

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

---

[5]http://www.w3.org/Daemon/User/Installation/PrivilegedPorts.html

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

- another `__init__.py`, serving the 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 implementation;
- `views.py`, where you can store a series of functions that handle requests and return responses; and
- the `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[6] to see how models, views and templates relate to each other in more detail.

Before you can get started with creating your 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 list. Add the `rango` app to the end of the tuple, which should then look like the following example.

---

[6]https://docs.djangoproject.com/en/2.1/intro/overview/

```
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.

3.4 Creating a View

With our Rango app created, let's now create a simple view. Views handle a *request* that comes from the client, *executes some code*, and provides a *response* to the client. To fulfil the request, it may contact other services or query for data from other sources. The job of a view is to collate and package the data required to handle the request, as we outlined above. For our first view, given a request, the view will simply send some text back to the client. For the time being, we won't concern ourselves about using models (i.e. getting data from other sources) or templates (i.e. which help us package our responses nicely).

In your editor, 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.

```python
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`[7] 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`[8] 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)[9] to the view.

To create an initial mapping, open `urls.py` located in your project configuration directory (i.e. `<workspace>/tango_with_django_project/tango_with_django_project` – the second `tango_with_django_project` directory!) and add the following lines of code to the `urlpatterns` list:

```
from rango import views

urlpatterns = [
    path('', views.index, name='index'),
    path('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.

---

[7]https://docs.djangoproject.com/en/2.1/ref/request-response/#django.http.HttpResponse
[8]https://docs.djangoproject.com/en/2.1/ref/request-response/#django.http.HttpRequest
[9]http://en.wikipedia.org/wiki/Uniform_resource_locator

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.

```python
from django.contrib import admin
from django.urls import path
from django.urls import include

from rango import views

urlpatterns = [
    path('', views.index, name='index'),
    path('rango/', include('rango.urls')),
    # The above maps any URLs starting with rango/ to be handled by rango.
    path('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 the `django.urls` package.

`http://www.servername.com/rango/about/`

Protocol and Domain Name

`http://www.servername.com/`

Project Configuration's `urls.py`

`rango/`

Rango's (your app's) `urls.py`

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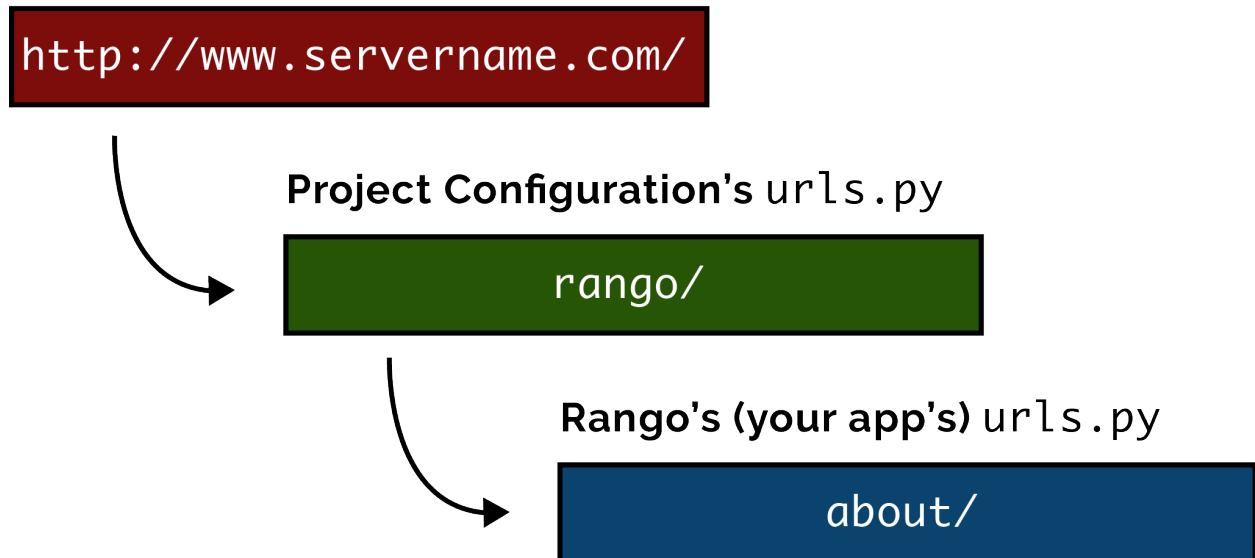
An illustration of a URL, represented as a chain, showing how different parts of the URL following the domain are the responsibility of different `url.py` files.

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):

```python
from django.urls import path
from rango import views

app_name = 'rango'

urlpatterns = [
    path('', 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 will handle the `/rango/about/` part of the URL string.

The URL mapping we have created above calls Django's `path()` function, where the first parameter is the string to match. As we have used an empty string `''`, Django will then only find a match if there is nothing after `http://127.0.0.1:8000/`. The second parameter tells Django what view to call if the pattern `''` is matched. In this case, `views.index()` will be called. The third and optional parameter is called `name`. It provides a convenient way to reference the view, and by naming our URL mappings we can employ *reverse URL matching*. That is we can reference the URL mapping by name rather than by the URL. Later, we will explain and show why this is incredibly useful. It can save you time and hassle as your application becomes more complex. This will go hand-in-hand with the `app_name` variable we've also placed in the new `urls.py` module.

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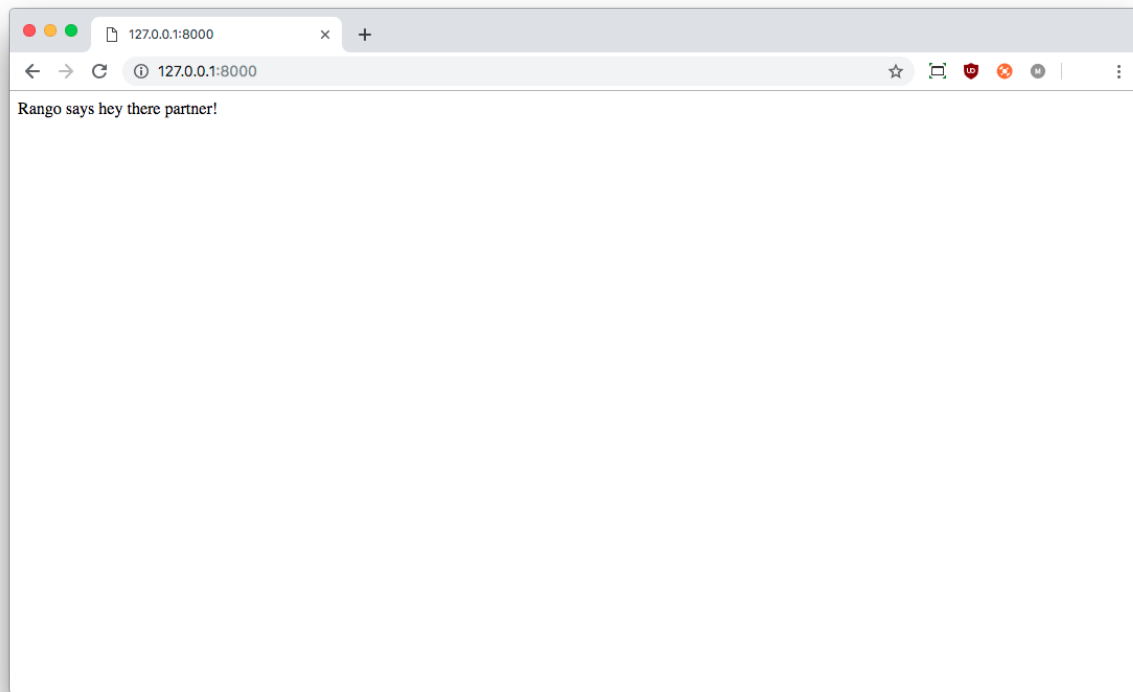
A screenshot of a web browser displaying our first Django powered webpage. Hello, Rango!

Within each app, you will create several URL mappings. The initial mapping is quite simple, but as we progress through the book we will create more sophisticated and 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[10] for further details and further examples.

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.

---

[10]https://docs.djangoproject.com/en/2.1/topics/http/urls/

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, try out the following exercises 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 hyperlink (or *anchor*) to the about page.
- Include a link back to the index page in the `about` view's response.
- Now that you have started the book, you can follow us on Twitter if you have it – our handle is @tangowithdjango[11]. Let us know how you are getting on!

---

[11]https://twitter.com/tangowithdjango

Hints

If you're struggling to get the exercises done, the following hints will 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()`. Within this `HttpResponse()`, insert the message that you want to return.
- The expression to use for matching the second view is `'about/'`. This means that in `rango/urls.py`, you would add in a new `path` mapping to the `about()` view.
- Within the `index()` view, you will want to include an HTML *hyperlink*[12] to provide a link to the about page. A `a` tag, complete with an `href` attribute (`<a href='/rango/about/'>About</a>`) will suffice.
- The same will also be added to the `about()` view, although this time it will point to `/rango/`, the homepage – not `/rango/about/`. An example would look like: `<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[13].

Test your Implementation

If you have completed everything in this chapter up to and including the exercises, you can test your implementation so far. Follow the guide we provided earlier, using the test module `tests_chapter3.py`. Do the tests pass when run against your implementation?

---

[12]https://en.wikipedia.org/wiki/Hyperlink
[13]https://docs.djangoproject.com/en/2.1/intro/tutorial01/

4. Templates and Media Files

In this chapter, we'll be introducing the Django template engine, as well as showing you how to serve both *static* files and *media* files. Rather than crafting each page by returning strings as our response, we can use *templates* to provide the skeleton structure of the page from a separate file. From the view that generates the response, we can provide the template with the necessary data to render that page in its entirety. To incorporate JavaScript and CSS (along with images and other media content) we will use the machinery provided by Django to include and dispatch such files to clients, which will in turn allow us to provide added functionality (in the case of JavaScript), or to provide styling to our pages.

4.1 Using Templates

Up until this point, we have only connected a URL mapping to a view. However, the Django framework is based around the *Model-View-Template* architecture. In this section, we will go through the mechanics of how *Templates* work with *Views*. In subsequent 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[1] aids users with navigation and reinforces a sense of continuity. Django provides templates[2] 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 generate an HTML page. This 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.

---

[1]http://www.techrepublic.com/blog/web-designer/effective-design-principles-for-web-designers-repetition/
[2]https://docs.djangoproject.com/en/2.1/ref/templates/

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 two directories 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`. To clarify, this is the directory that contains your project's `manage.py` script! Within the new templates directory, you will then want to create further directory called `rango`. This means that the path `<workspace>/tango_with_django_project/templates/rango/` is the location where we will store templates associated with our `rango` application.

Keep your Templates Organised

It's good practice to separate 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 project, it will look like the following.

```python
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 will be 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.

```python
'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:

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

However, there are several 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[3] is a software engineering anti-pattern[4], and will make your project less portable[5] – meaning that when you run it on another computer, it probably won't work! (Or it will work, just with a lot of effort!)

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 path of your settings module[6]. Using this as a parameter to `os.path.abspath()` guarantees the *absolute path* to the `settings.py` 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 directory layer, so that `BASE_DIR` then points to your project directory, or `<workspace>/tango_with_django_project/`. If you are curious, you can see how this works by adding the following lines to your `settings.py` file.

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

Having access to the value of `BASE_DIR` makes it easy for you to reference other aspects of your Django project. Using the `BASE_DIR` variable, 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

---

[3]http://en.wikipedia.org/wiki/Hard_coding
[4]http://sourcemaking.com/antipatterns
[5]http://en.wikipedia.org/wiki/Software_portability
[6]http://stackoverflow.com/a/9271479

to a variable definition like the example below. Make sure you put this underneath the definition of `BASE_DIR`!

> ### 🛈 Delete those Lines!
>
> If you included the three `print()` statements above to see what's going on, make sure you remove them once you understand. Don't just leave them lying there. They will clutter your settings module, and clutter the output of the Django development server!

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

Here, we make use of the `os.path.join()` function to join together (or *concatenate*) the value of the `BASE_DIR` variable, and the string `'templates'`. Upon completion, this concatenation yields `<workspace>/tango_with_django_project/templates/`. From here, 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 code snippet.

```
'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 to draw templates from. For this book however, 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 markup and Django template code.

```html
<!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. Django should have added it for you when you created the Rango app. 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.

```python
def index(request):
    # Construct a dictionary to pass to the template engine as its context.
    # Note the key boldmessage matches to {{ 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 in your browser – and it should look 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 on 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[7].
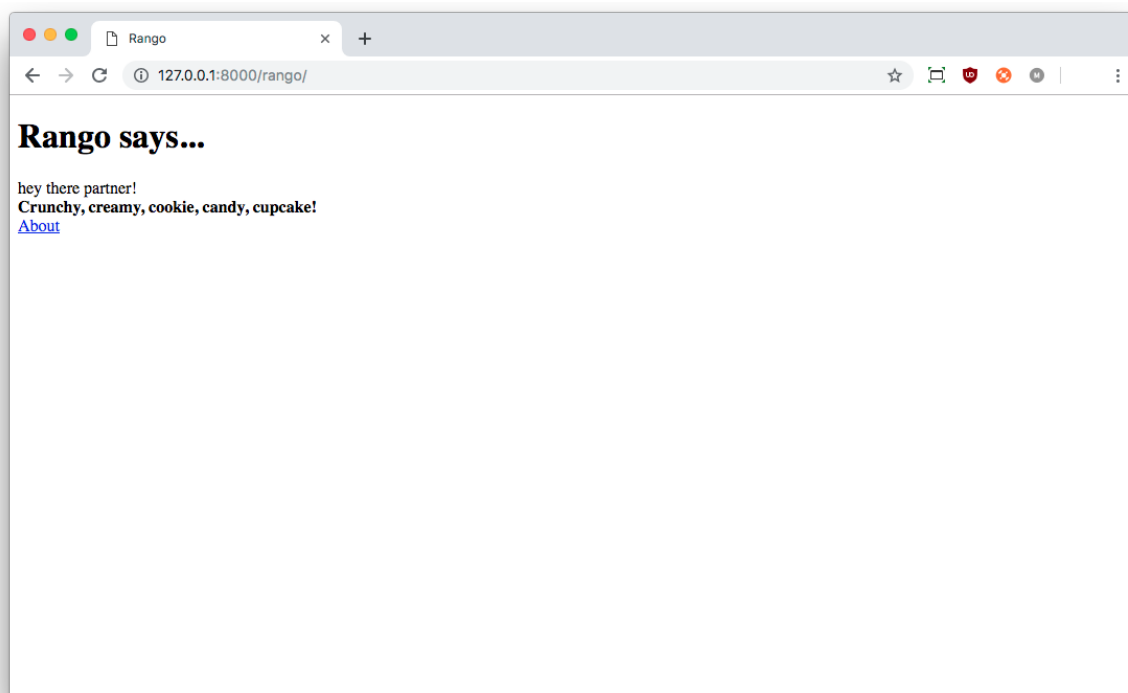
What you should see when your first template is working correctly. Note the bold text — `Crunchy, creamy, cookie, candy, cupcake!` — which originates from the view, but is rendered in the template.

---

[7]https://docs.djangoproject.com/en/2.1/ref/templates/

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)*[8], *JavaScript*[9] 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[10] – a fitting mascot, if ever there was one.

[8]http://en.wikipedia.org/wiki/Cascading_Style_Sheets
[9]https://en.wikipedia.org/wiki/JavaScript
[10]http://www.imdb.com/title/tt1192628/
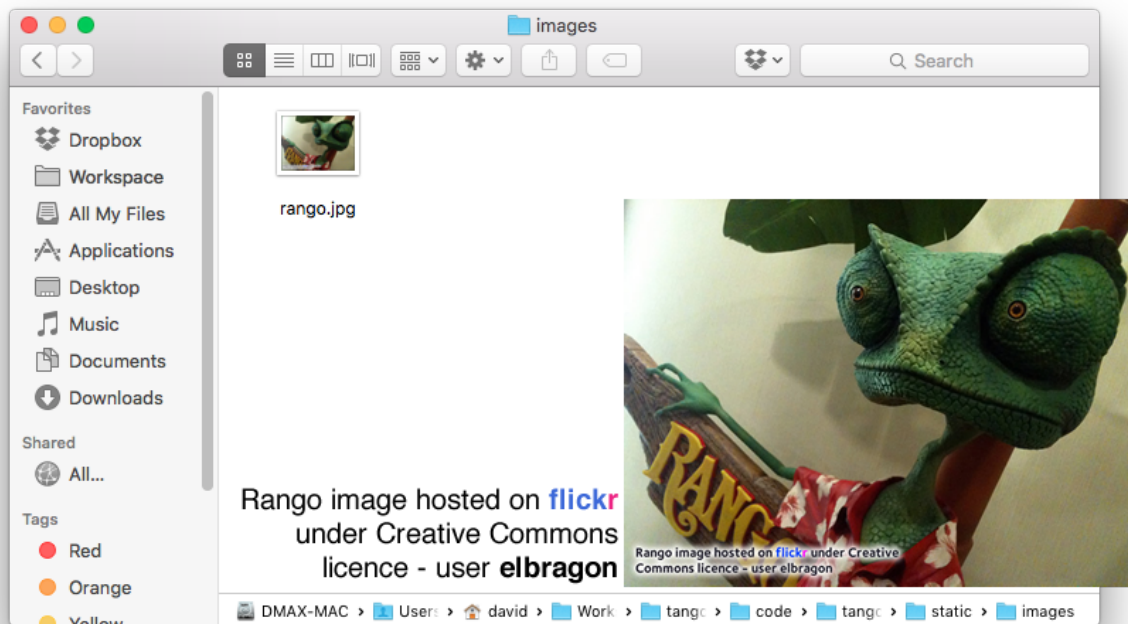
Rango the chameleon within our `static/images` media directory.

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 results in the path `<workspace>/tango_with_django_project/static/`. We then need to create a new data structure called `STATICFILES_DIRS`. This is essentially a list of paths with which Django will 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 a single location to store our project's static files – the path defined in `STATIC_DIR`. As such, we can simply set up the list `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 has not been defined, then add it in, as shown below. Note that this variable by default appears close to the end of the module, so you may have to scroll down to the bottom of `settings.py` to find it (if it's already there).

```
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, with 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 complete URL that you would enter into your web browser's window to access this resource?

Try to figure this out before you move on! It'll help you understand how to interpret static URLs. The answer is coming up if you are struggling to figure it out.

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[11], 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[12] 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.

---

[11]https://docs.djangoproject.com/en/2.1/ref/settings/#std:setting-STATIC_URL
[12]https://docs.djangoproject.com/en/2.1/howto/static-files/deployment/

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 an HTML comment next to them for easy identification.

```html
<!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[13]. 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:

---

[13]https://docs.djangoproject.com/en/2.1/ref/templates/builtins/

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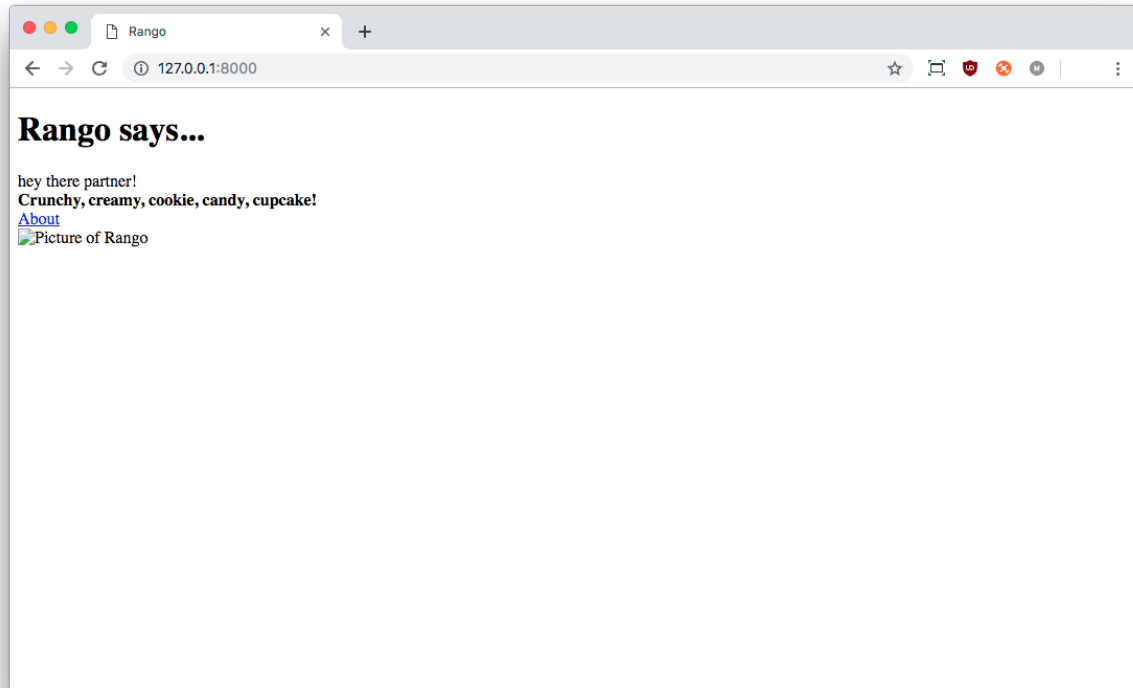
The image of Rango couldn't be found, and is instead replaced with a placeholder containing the text from the `img alt` attribute.

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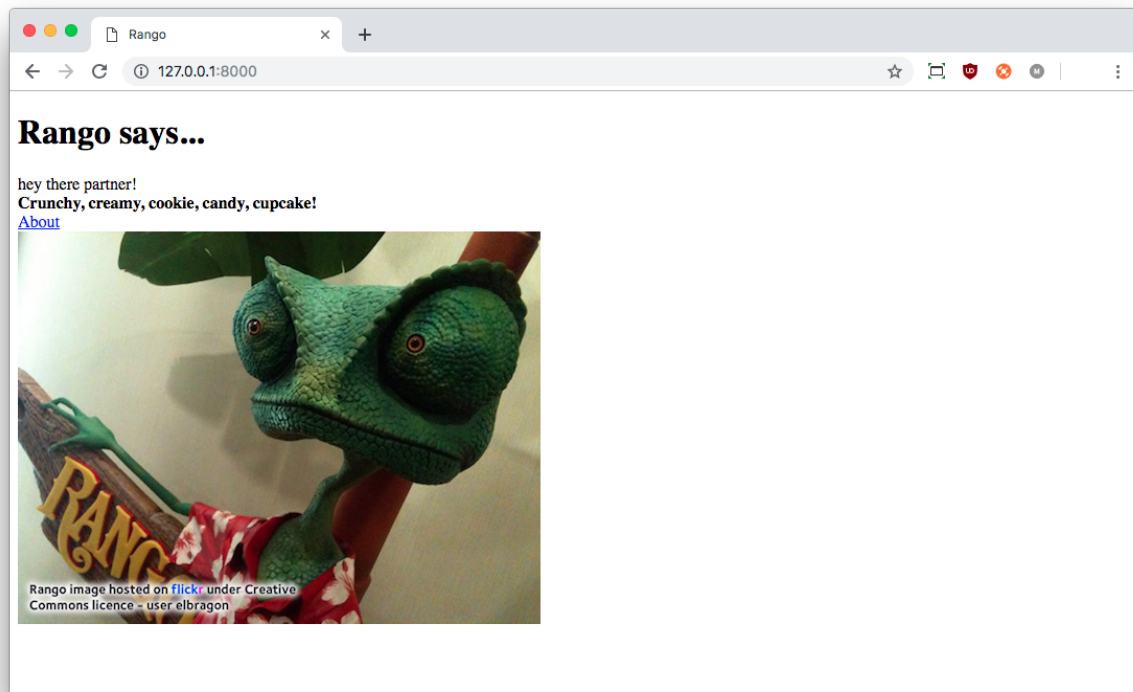
Our first Rango template, complete with a picture of Rango the chameleon.

🔑 Always put `<!DOCTYPE>` First!

When creating the HTML templates, always ensure that the `DOCTYPE` declaration[14] 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[15].

---

[14]http://www.w3schools.com/tags/tag_doctype.asp
[15]https://validator.w3.org/

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>
```

Don't update the `index.html` template here – this is merely a demonstration to show you how the `{% static %}` template function works. You'll be adding CSS and JavaScript later on in this tutorial.

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[16]. 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`.

```
[24/Mar/2019 17:05:54] "GET /rango/ HTTP/1.1" 200 366
[24/Mar/2019 17:05:55] "GET /static/images/rango.jpg HTTP/1.1" 200 0
[24/Mar/2019 17:05:55] "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[17].

---

[16]https://en.wikipedia.org/wiki/HTTP_404

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. 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.

To serve media files successfully, we need to update the 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. This allows you to test and 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 (through a production-grade web server like Apache). 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`.

---

[17]https://docs.djangoproject.com/en/2.1/howto/static-files/#staticfiles-in-templates

```
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[18].

```
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 to access on your Django development server through the URL `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*[19] 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 that resides within your project's `settings.py` module. The list contains a dictionary; look for the `context_processors` list within the nested dictionary. Within the `context_processors` list, add a new string to include

---

[18]https://docs.djangoproject.com/en/2.1/howto/static-files/#serving-files-uploaded-by-a-user-during-development
[19]https://docs.djangoproject.com/en/2.1/ref/templates/api/#django-template-context-processors-media

an additional context processor: `'django.template.context_processors.media'`. Your `context_processors` list should then look like 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',  # Check/add this line!
],
```

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. Remember, your **project's** `urls.py` module is the one that lives within the `tango_with_django_project` directory!

```
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.

> ✏️ **Create the `media` Directory**
>
> Did you create the `media` directory within the `tango_with_django_project` directory? It should be a the same level as the `static` directory and the `manage.py` module.

4.4 Basic Workflow

With the chapter complete, you should now know how to set up 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[20] 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. Finally, map the view to a URL by modifying your project's `urls.py` file (or the application-specific `urls.py` file if you have one). This step is only required if you're creating a new view, or you are using an existing view that hasn't yet been mapped!

The steps involved in 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.

---

[20]https://docs.djangoproject.com/en/2.1/ref/templates/builtins/

1. Take the static media file you wish to use and place it within your project's `static` directory. This directory is defined in `STATICFILES_DIRS` – one of the variables that you set up in `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" alt="Picture of a Cat."/>`.

Exercises

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

- Convert the about page to use a template too. Use a template called `about.html` for this purpose. Base the contents of this file on `index.html`. In the new template's `<h1>` element, keep `Rango says...` – but on the line underneath, have the text `here is the about page..`
- Within the new `about.html` template, add a picture stored within your project's static files. You can just reuse the `rango.jpg` image you used in the index view! Make sure you keep the same `alt` text as the index page!
- On the about page, include a line that says `This tutorial has been put together by <your-name>`. If you copied over from `index.html`, replacing `{{ boldmessage }}` would be the perfect place for this.
- In your Django project directory, create a new directory called `media` (if you have not done so already). Download a JPEG image of a cat, and save it to the `media` directory as `cat.jpg`.
- In your `about.html` template, add in an `<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 about page has working examples of both static and media files. The cat image should have alternative text of `Picture of a Cat`. **This means you should have an image of both Rango (from `static`) and a cat (from `media`) in your rendered about page.**

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 (CSS), 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.

✏️ Test your Implementation

If you have completed everything in this chapter up to and including the exercises, you can test your implementation so far. Follow the guide we provided earlier, using the test module `tests_chapter4.py`. How does your implementation stack up against our tests?

5. Models and Databases

Typically, web applications require a backend to store the dynamic content that appears on the app's webpages. For Rango, we need to store pages and categories that are created, along with other details. The most convenient way to do this is by employing the services of a relational database. These will likely use the *Structured Query Language (SQL)* to allow you to query the data store.

However, Django provides a convenient way in which to access data stored in databases by using an *Object Relational Mapper (ORM)*[1]. In essence, data stored within a database table is encapsulated via Django *models*. A model is a Python object that describes the database table's data. Instead of directly working on the database via SQL, Django provides methods that let you manipulate the data via the corresponding Python model object. Any commands that you issue to the ORM are automatically converted to the corresponding SQL statement on your behalf.

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 see 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. Let's look at the database requirements again here.

- Rango is essentially a *web page directory* – it is a website that houses links to other, external websites.

---

[1]https://en.wikipedia.org/wiki/Object-relational_mapping

- There are several different *webpage categories* with each category housing none, one or many 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, several visits, and several likes.
- A page belongs to a particular category, has a title, a URL, and several views.
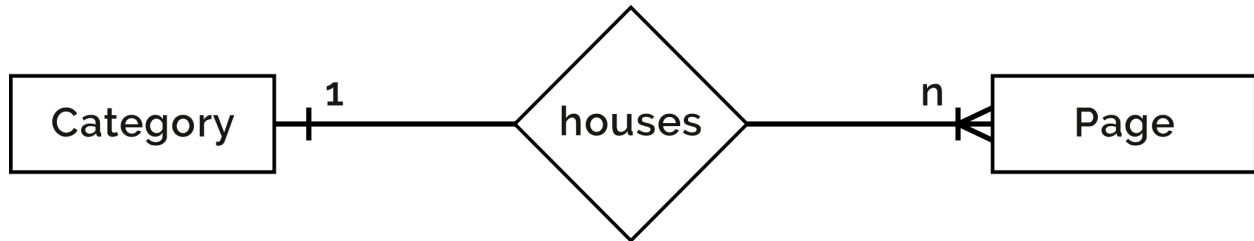
The Entity Relationship Diagram of Rango's two main entities.

5.2 Telling Django about Your Database

Before we can create any models, we need to set up our database to work with Django. In Django, a DATABASES variable is automatically created in your settings.py module when you set up a new project. Unless you changed it, it should look like 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[2] (see the ENGINE option). The NAME entry for this database is the path to the database file, which is by default <workspace>/tango_with_django_project/db.sqlite3 – or, in other words, the db.sqlite3 file that lives in the root of your Django project.

---

[2]https://www.sqlite.org/

Don't `git` `push` your Database!

If you are using Git, you might be tempted to add and commit the database file. However, we don't think that this is a particularly good idea. What if you are working on your app with other people? It would be highly likely that they will change the database as they test features. Different versions of database files (which ultimately do not matter in development) **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. For more information on how to set your `.gitignore` file up, you can refer to our Git familiarisation chapter in the appendices.

Using other Database Engines

The Django database framework has been created to cater for a variety of different database backends, such as PostgresSQL[3], MySQL[4] and Microsoft's SQL Server[5]. 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[6].

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[7].

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

---

[3] http://www.postgresql.org/
[4] https://www.mysql.com/
[5] https://en.wikipedia.org/wiki/Microsoft_SQL_Server
[6] https://docs.djangoproject.com/en/2.1/ref/databases/#storage-engines
[7] http://www.sqlite.org/whentouse.html

respective `models.py` module. This means that for Rango, models are stored within `<workspace>/tango_with_django_project/rango/models.py`.

For the models themselves, we will create two classes – one class representing each model. Both must inherit[8] 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.

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

    def __str__(self):
        return self.name

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

    def __str__(self):
        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.

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[9]. 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 of characters that a `CharField` field can store.

---

[8] https://en.wikipedia.org/wiki/Inheritance_(object-oriented_programming)
[9] https://docs.djangoproject.com/es/2.1/ref/models/fields/#model-field-types

- `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[10] 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 as 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`[11]) (`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[12];
- `OneToOneField`, a field type that allows us to define a strict one-to-one relationship[13]; and
- `ManyToManyField`, a field type which allows us to define a many-to-many relationship[14].

From our model examples above, the field `category` in model `Page` is a `ForeignKey`. This allows us to create a one-to-many relationship with model/table `Category`,

---

[10]https://docs.djangoproject.com/es/2.1/ref/models/fields/#model-field-types
[11]https://en.wikipedia.org/wiki/Nullable_type
[12]https://en.wikipedia.org/wiki/One-to-many_(data_model)
[13]https://en.wikipedia.org/wiki/One-to-one_(data_model)
[14]https://en.wikipedia.org/wiki/Many-to-many_(data_model)

which is specified as an argument to the field's constructor. When specifying the foreign key, we also need to include instructions to Django on how to handle the situation when the category that the page belongs to is deleted. `CASCADE` instructs Django to delete the pages associated with the category when the category is deleted. However, there are other settings which will provide Django with other instructions on how to handle this situation. See the Django documentation on Foreign Keys[15] for more details.

Finally, it is good practice to implement the `__str__()` method. Without this method implemented it will show as `<Category: Category object>` if you were to `print()` the object (perhaps in the Django shell, as we discuss later in this chapter). This isn't very useful when debugging or accessing the object. How do you know what category is being shown? When including `__str__()` as defined above, you will see `<Category: Python>` (as an example) for the `Python` category. It is also helpful when we go to use the admin interface later because Django will display the string representation of the object, derived from `__str__()`.

Always Implement `__str__()` in your Classes

Implementing the `__str()__` method in your classes will make debugging so much easier – and also permit you to take advantage of other built-in features of Django (such as the admin interface). If you've used a programming language like Java, `__str__()` is the Python equivalent of the `toString()` method!

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*[16] 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.

---

[15]https://docs.djangoproject.com/en/2.1/ref/models/fields/#django.db.models.ForeignKey
[16]https://en.wikipedia.org/wiki/Data_migration

Setting up

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

```
$ python manage.py migrate

Operations to perform:
        Apply all migrations: admin, auth, contenttypes, sessions
Running migrations:
        Applying contenttypes.0001_initial... OK
        Applying auth.0001_initial... OK
        Applying admin.0001_initial... OK
        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 auth.0008_alter_user_username_max_length... OK
        Applying auth.0009_alter_user_last_name_max_length... OK
        Applying sessions.0001_initial... OK
```

All apps installed in your Django project (check `INSTALLED_APPS` in `settings.py`) will be called to update their database representations when this command is issued. 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 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':
        rango/migrations/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.

> ### ℹ 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 will 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, auth, contenttypes, rango, sessions
Running migrations:
        Applying rango.0001_initial... OK
```

This output confirms that the database tables have been created in your database, and you are then ready to start using the new models and tables.

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 work through the migration process once more.**

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.

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

# Show all the current categories
>>> print(Category.objects.all())
# Since no categories have been defined we get an empty QuerySet object.
<QuerySet []>

# 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())
# You'll now see a 'Test' category.
<QuerySet [<Category: Test>]

# 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 the output of the `__str__()`, and neatly demonstrates why including these methods is important!

✏️ 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[17]. In addition, have a look at the official Django documentation on the list of available commands[18] for working with models.

---

[17]https://docs.djangoproject.com/en/2.1/intro/tutorial02/
[18]https://docs.djangoproject.com/en/2.1/ref/django-admin/#available-commands

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.
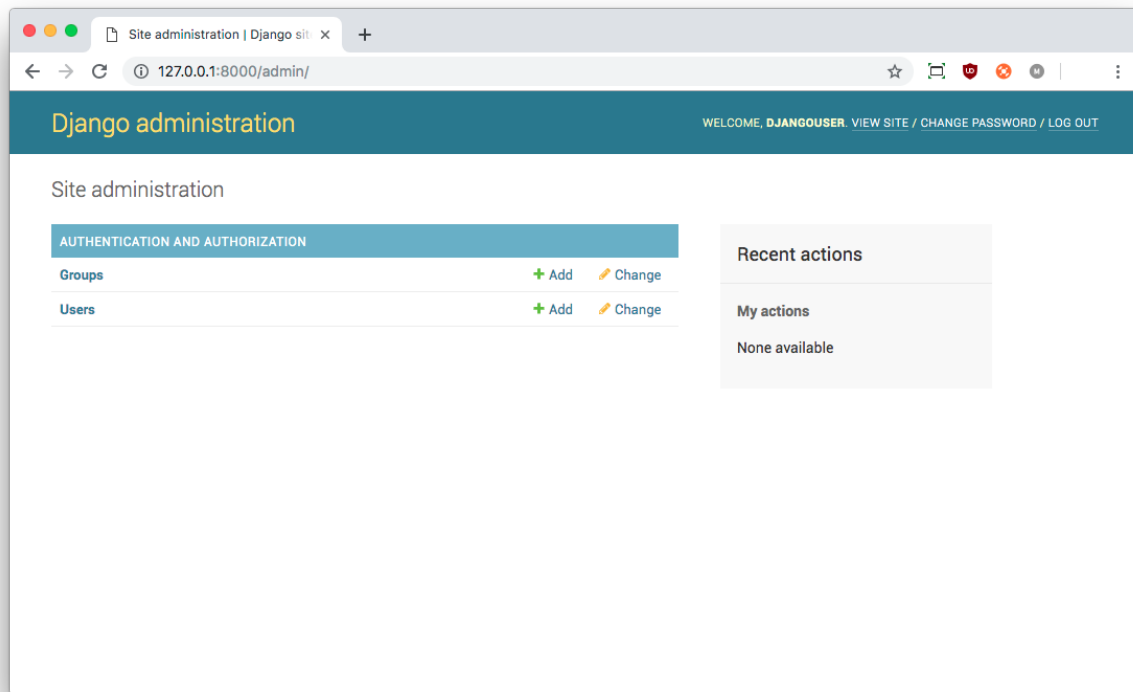
The Django admin interface, sans Rango models.

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 let Django know that we want to include them.

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.

```python
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, making sure that the model is `imported` at the top of the module.

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.
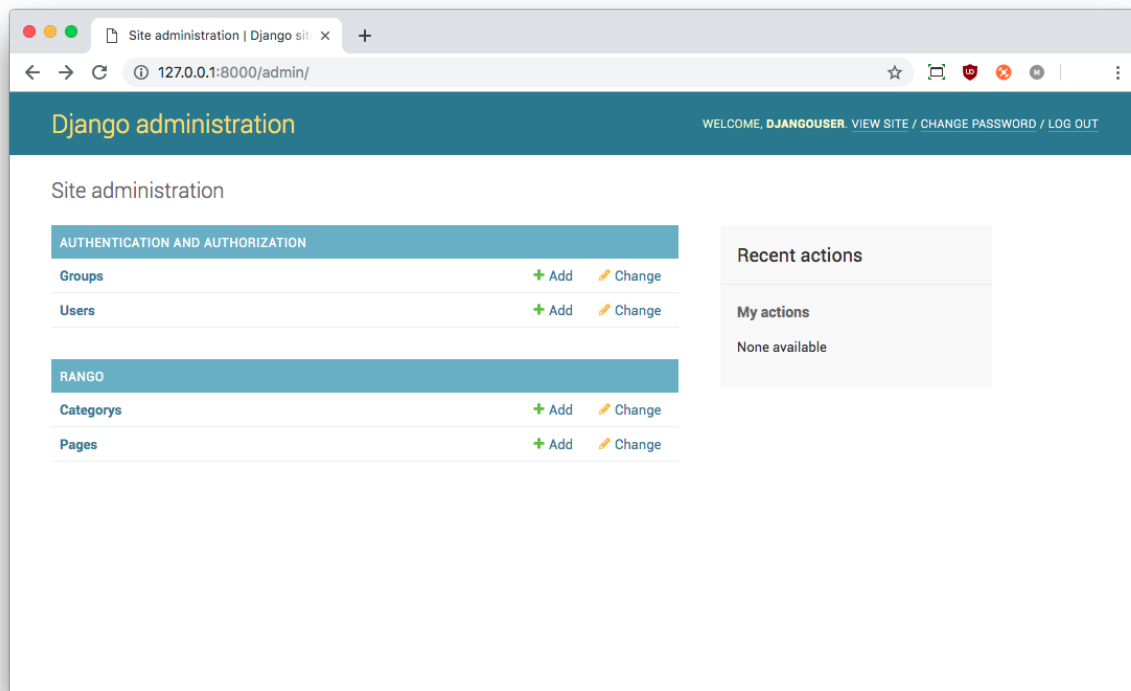
The Django admin interface, complete with Rango models.

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

As you move forward with Rango's development, you'll be using the admin interface extensively to verify data is stored correctly. 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 example data. You can delete the `Test` category from the admin interface by clicking the checkbox beside it, and selecting `Delete selected categorys` from the dropdown menu at the top of the page. Confirm your intentions by clicking the big red button that appears!

User Management

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

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[19] for more information about what can be stored within the `Meta` class.

```python
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 several ways. Check out the official Django documentation on the admin interface[20] for more information if you're interested. We'll be working towards manipulating the `admin.py` module later on in the tutorial.

---

[19]https://docs.djangoproject.com/en/2.1/topics/db/models/#meta-options
[20]https://docs.djangoproject.com/en/2.1/ref/contrib/admin/

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 need to be able to see some credible examples in the database. If you're working in a team, an automated script will mean each collaborator can simply run that script to initialise the database on their computer with the same sample data as you. It's therefore good practice to create what we call a *population script*.

To create a population script, create a new file called `populate_rango.py`. Create this file in `<workspace>/tango_with_django_project/`, or in other words, your Django project's root directory. When the file has been created, add the following code.

```python
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/3/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 = [
```

```
25              {'title':'Official Django Tutorial',
26               'url':'https://docs.djangoproject.com/en/2.1/intro/tutorial01/'},
27              {'title':'Django Rocks',
28               'url':'http://www.djangorocks.com/'},
29              {'title':'How to Tango with Django',
30               'url':'http://www.tangowithdjango.com/'} ]
31
32      other_pages = [
33              {'title':'Bottle',
34               'url':'http://bottlepy.org/docs/dev/'},
35              {'title':'Flask',
36               'url':'http://flask.pocoo.org'} ]
37
38      cats = {'Python': {'pages': python_pages},
39                  'Django': {'pages': django_pages},
40                  'Other Frameworks': {'pages': other_pages} }
41
42      # If you want to add more categories 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      for cat, cat_data in cats.items():
48          c = add_cat(cat)
49          for p in cat_data['pages']:
50              add_page(c, p['title'], p['url'])
51
52      # Print out the categories we have added.
53      for c in Category.objects.all():
54          for p in Page.objects.filter(category=c):
55              print('- {0} - {1}'.format(str(c), str(p)))
56
57  def add_page(cat, title, url, views=0):
58      p = Page.objects.get_or_create(category=cat, title=title)[0]
59      p.url=url
60      p.views=views
61      p.save()
62      return p
63
64  def add_cat(name):
65      c = Category.objects.get_or_create(name=name)[0]
66      c.save()
67      return c
```

```
68
69  # Start execution here!
70  if __name__ == '__main__':
71      print('Starting Rango population script...')
72      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 provided explanations below to help you learn from our code!

You should also note that when you see line numbers alongside the code. We've included these to make copying and pasting a laborious chore – why not just type it out yourself and think about each line instead?

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()`, both 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__'`[21], we call and begin execution of 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.

---

[21]http://stackoverflow.com/a/419185

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 the seventh line.

The `for` loop occupying lines 47-50 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 48 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 means that the `[0]` after `get_or_create()` returns the object reference only. Like most other programming language data structures, Python tuples use zero-based numbering[22].

You can check out the official Django documentation[23] for more information on the handy `get_or_create()` method. We'll be using this extensively throughout the rest of the tutorial.

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.

---

[22]http://en.wikipedia.org/wiki/Zero-based_numbering
[23]https://docs.djangoproject.com/en/2.1/ref/models/querysets/#get-or-create

```
$ 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 worked as it should have and populated the database. To do this, restart the Django development server, and navigate to the admin interface (at `http://127.0.0.1:8000/admin/`). Once there, 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?
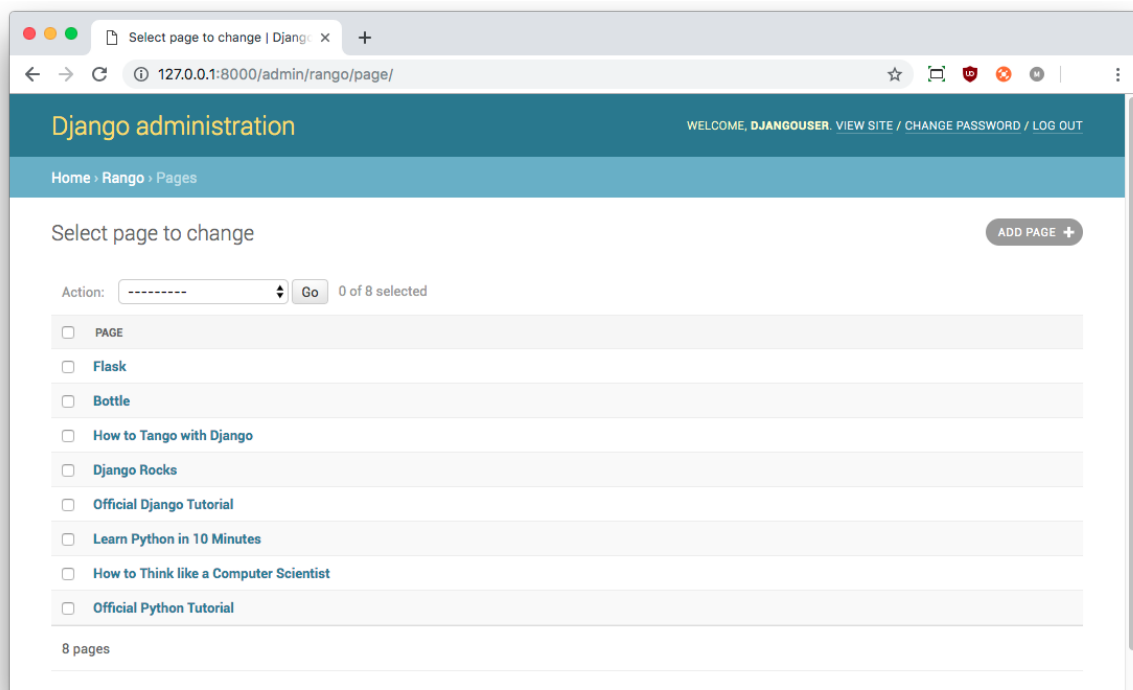
The Django admin interface, showing the `Page` model populated with the new population script. Success!

While creating a population script may take time initially, you will save yourself heaps of 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 then make them accessible via the web-based 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) if you want to make them accessible to the admin interface.
3. Perform the migration $ `python manage.py makemigrations <app_name>`.
4. Apply the changes $ `python manage.py migrate`. This will create the necessary infrastructure (tables) 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. Perhaps you might end up caught in a loop when trying to make further migrations, and something goes wrong.

When you encounter the need to refresh the database, you can go through the following steps. Note that for this tutorial, you are using an SQLite database – Django does support a variety of other database engines[24].

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` command 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.

---

[24]https://docs.djangoproject.com/en/2.1/ref/databases/

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!**

- Update the `Category` model to include the additional attributes `views` and `likes` where the `default` values for each are both zero (`0`).
- As you have changed your models, make the migrations for your Rango app. After making migrations, commit the changes to the database.
- Next 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[25] and seven[26] of the official Django tutorial. The two sections will reinforce what you've learnt on handling databases in Django, and will also provide you with additional techniques to customising the Django admin interface. This knowledge will help you complete the final exercise below.
- Customise the admin interface. Change it in such a way so that when you view the `Page` model, the table displays the `category`, the `title` of the page, and the `url` – just like in the screenshot shown below. **Make sure the three fields appear in that order, too.** Complete the official Django tutorial or look at the tip below to complete this particular exercise.

---

[25]https://docs.djangoproject.com/en/2.1/intro/tutorial02/
[26]https://docs.djangoproject.com/en/2.1/intro/tutorial07/
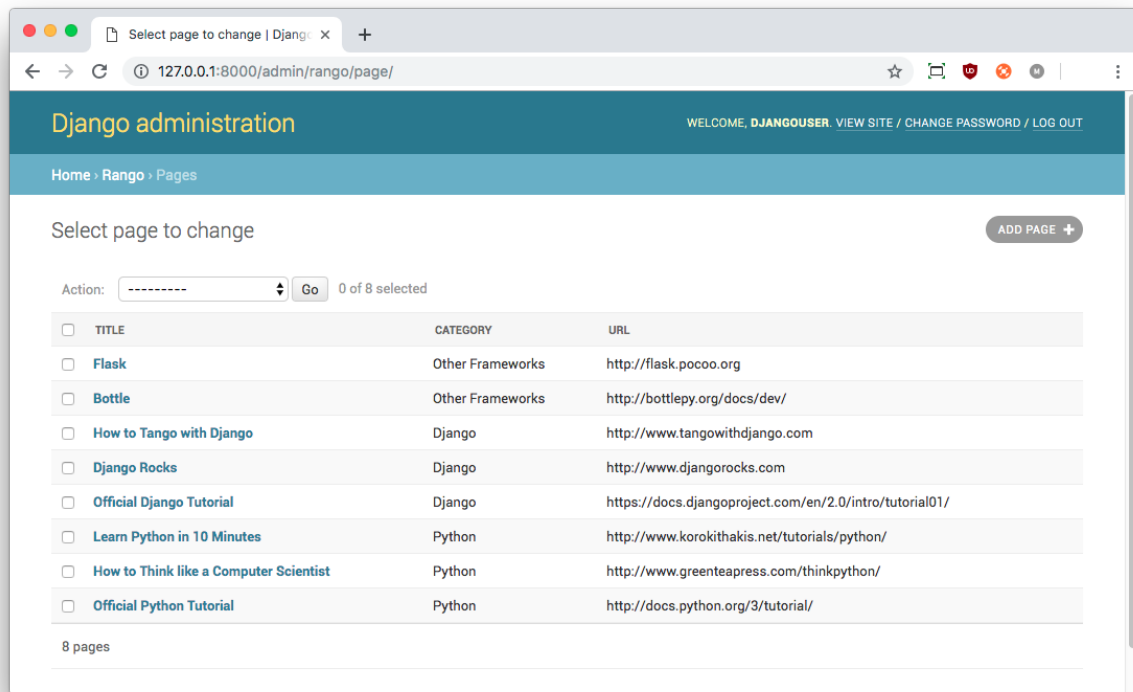
The updated admin interface Page view, complete with columns for category and URL.

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 `IntegerField`s: `views` and `likes`.
- In your population script, you can then modify the `add_cat()` function to take the values of the `views` and `likes`.
    - 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[27] 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[28]. 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.

---

[27]https://www.tutorialspoint.com/python/python_dictionary.htm

Test your Implementation

Like in the previous chapter, we've implemented a series of unit tests to allow you to check your implementation up until this point. Follow the guide we provided earlier, using the test module `tests_chapter5.py`. How does your implementation stack up against our tests?

---

[28]https://www.tutorialspoint.com/python/python_dictionary.htm

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: A 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 the `views.py` module, `import` the models you wish to use.
2. In the view function, query the model to get the data you want to present.
3. Also in the view, 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 categories present within your app's database. 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.

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

Modifying the Index View

Here we will complete steps 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:

```python
def index(request):
    # Query the database for a list of ALL categories currently stored.
    # Order the categories by the number of likes in descending order.
    # Retrieve the top 5 only -- or all if less than 5.
    # Place the list in our context_dict dictionary (with our boldmessage!)
    # that will be passed to the template engine.
    category_list = Category.objects.order_by('-likes')[:5]

    context_dict = {}
    context_dict['boldmessage'] = 'Crunchy, creamy, cookie, candy, cupcake!'
    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[1] to take the first five objects from the list (`[:5]`) to return a subset of `Category` objects.

---

[1]https://www.quackit.com/python/reference/python_3_list_methods.cfm

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. Note that above, we still include our `boldmessage` in the `context_dict` – this is still required for the existing template to work! This means our context dictionary now contains two key/value pairs: `boldmessage`, representing our `Crunchy, creamy, cookie, candy, cupcake!` message, and `categories`, representing our top five categories that have been extracted from the database.

> ⚠️ **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. Like we have said already, we assume you complete all exercises as you progress through this book.

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 and Django template code so that it looks like the example shown below. Note that the major changes start at line 15.

```html
1  <!DOCTYPE html>
2  {% load staticfiles %}
3  <html>
4  <head>
5      <title>Rango</title>
6  </head>
7
8  <body>
9      <h1>Rango says...</h1>
10     <div>
11         hey there partner! <br/>
12         <strong>{{ boldmessage }}</strong>
13     </div>
14
15     <div>
```

```
16        {% if categories %}
17            <ul>
18            {% for category in categories %}
19                <li>{{ category.name }}</li>
20            {% endfor %}
21            </ul>
22        {% else %}
23            <strong>There are no categories present.</strong>
24        {% endif %}
25        </div>
26
27        <div>
28            <a href="/rango/about/">About Rango</a><br />
29            <img src="{% static "images/rango.jpg" %}" alt="Picture of Rango" />
30        </div>
31    </body>
32    </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 of (a maximum of) five categories – 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 an `<li>` element to denote that it is a *list element.*

If no categories exist, a message is displayed instead indicating that no categories are present – `There are no categories present.`. As this is wrapped in a `<strong>` element, it will appear in **bold**.

As the example also demonstrates Django's template language, all template commands are enclosed within the tags `{%` and `%}`, while variables whose values are to be placed on the page are referenced within `{{` and `}}` brackets. *Everything* within these tags and brackets is interpreted by the Django templating engine before sending a completed response back to the client.

Now, save the template file and head over to your web browser. Refresh Rango's homepage at `http://127.0.0.1:8000/rango/`, and you show then see a list of cat-

egories underneath the page title and your bold message, just like in the figure below. Well done, this is your first *data-driven webpage!*
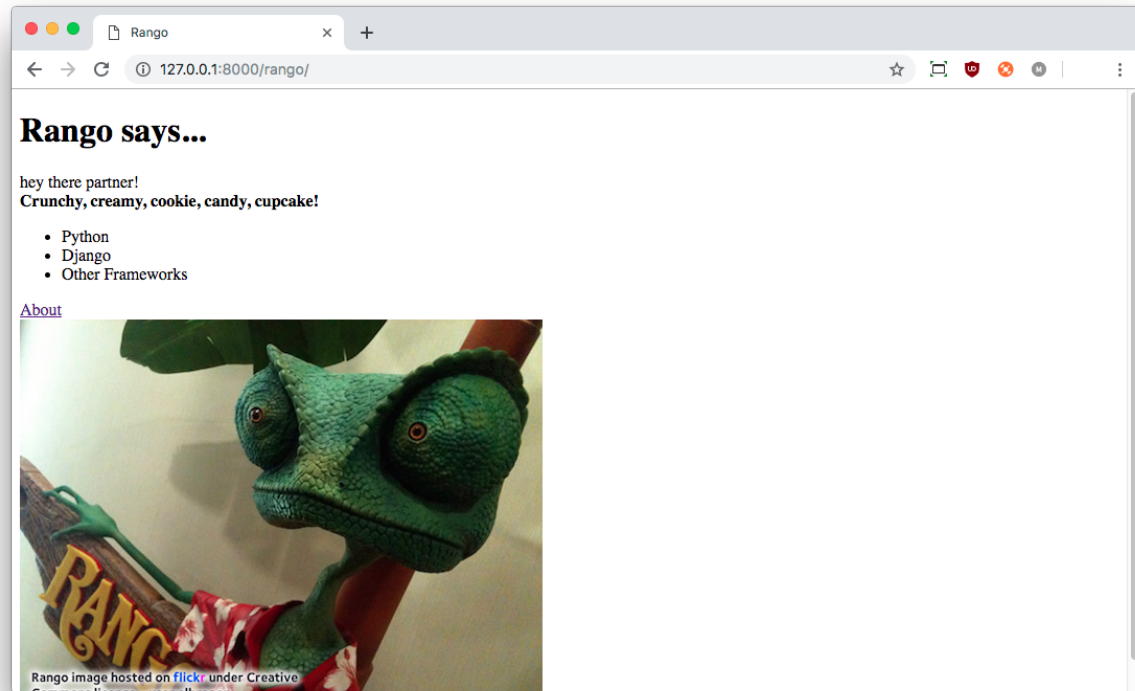
The Rango homepage – now dynamically generated – showing a list of categories.

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. To accomplish this, there are several different challenges that we need to overcome. First, a new view must be created. *This new view will have to 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`, for example. **Putting spaces in URLs is generally regarded as bad practice** (as we describe below). Spaces need to be *percent-encoded*[2]. For example, the percent-encoded string for `Other  Frameworks` would read `Other%20Frameworks`. This looks messy and confusing!

🔑 Clean your URLs

Designing clean and readable URLs is an important aspect of web design. See Wikipedia's article on Clean URLs[3] for more details.

To solve this problem, we will make use of the so-called `slugify()` function provided by Django.

Update the Category Table with a Slug Field

To make readable URLs, we need to include a slug[4] field in the `Category` model. First, we need to import the function `slugify` from Django that will replace whitespace with hyphens, circumnavigating the percent-encoded problem. 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[5] to read more.

Next, we need to override the `save()` method of the `Category` model. This overridden function will call the `slugify()` function and update the new `slug` field with it. Note

---

[2]https://www.w3schools.com/tags/ref_urlencode.asp
[3]http://en.wikipedia.org/wiki/Clean_URL
[4]https://prettylinks.com/2018/03/url-slugs/
[5]http://www.ietf.org/rfc/rfc1738.txt

that every time the category name changes, the slug will also change – the `save()` method is always called when creating or updating an instance of a Django model. Update your `Category` model as shown below.

```python
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
```

Don't forget to add in the following `import` at the top of the module, either.

```python
from django.template.defaultfilters import slugify
```

The overriden `save()` method is relatively straightforward to understand. When called, the `slug` field is set by using the output of the `slugify()` function as the new field's value. Once set, the overriden `save()` method then calls the parent (or `super`) `save()` method defined in the base `django.db.models.Model` class. It is this call that performs the necessary logic to take your changes and save the said changes to the correct database table.

Now that the model has been updated, the changes must now be propagated to the database. However, since data already exists within the database from previous chapters, 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
```

Since we did not provide a default value for the slug and we already have existing data in the model, the `makemigrations` command will give you two options. Select the option to provide a default (option 1), and enter an empty string – denoted by two quote marks (i.e. `''`).

You should then see output that confirms that the migrations have been created.

```
You are trying to add a non-nullable field 'slug' to category without a default;
    we can't do that (the database needs something to populate existing rows).
Please select a fix:
 1) Provide a one-off default now (will be set on all existing rows with a null
        value for this column)
 2) Quit, and let me add a default in models.py
Select an option: 1
Please enter the default value now, as valid Python
The datetime and django.utils.timezone modules are available, so you can do
    e.g. timezone.now
Type 'exit' to exit this prompt
>>> ''
Migrations for 'rango':
  rango/migrations/0003_category_slug.py
    - Add field slug to category
```

From there, you can then migrate the changes, and run the population script again to update the new slug fields.

```
$ python manage.py migrate
$ 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. Remember that the admin interface is reached by pointing your browser to `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 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`, relying on users to fill out fields will always be error-prone, and wouldn't provide a good user experience. 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, there are two possible solutions. The easiest solution would be to update our model so that the slug field allows blank entries, i.e.:

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

However, this is less than desirable. A blank slug would probably not make any sense, and at that, you could define multiple blank slugs! How could we then identify what category a user is referring to? **A much better solution** would be to 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)
...
```

Note that with the above `admin.site.register()` call, you should ensure that you replace the existing call to `admin.site.register(Category)` with the one provided

above. Once completed, try out the admin interface and add in a new category. Note that the `slug` field automatically populates as you type into the `name` field. Try adding a space and see what happens!

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. Update your `models.py` module to reflect this change with the inclusion of `unique=True`.

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

Now that we have added in the slug field, we can now use the slugs to guarantee a unique match to an associated category. We could have added the unique constraint earlier. However, if we had done that and then performed the migration (and thus setting everything to be an empty string by default), the migration would have failed. 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 find that you need to delete it and start again.

If you find yourself completely stuck, it is sometimes just more straightforward to delete the database and recreate everything from scratch, rather than trying to work out where the issue is coming from.

Category Page Workflow

Now we need to figure out how to create a page for individual categories. This will allow us to then be able to see the pages associated with a category. To implement the category pages so that they can be accessed using the URL pattern `/rango/category/<category-name-slug>/`, we need to 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_slug`, which will store the encoded category name.

- We will need helper functions to encode and decode `category_name_slug`.

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.

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

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

```python
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.
        # 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.
        # The 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. Then, we attempt to extract the data from the models and add the relevant data to the context dictionary. We determine which category has been requested by using the value passed `category_name_slug` to the `show_category()` view function (in addition to the `request` parameter).

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`. If the category requested was not found, we set the associated context dictionary values to `None`. Finally, we `render()` everything together, using a new `category.html` template.

Category Template

In your `<workspace>/tango_with_django_project/templates/rango/` directory, create a new template called `category.html`. In the new file, add the following code.

```html
1   <!DOCTYPE html>
2   <html>
3       <head>
4           <title>Rango</title>
5       </head>
6
7       <body>
8           <div>
9           {% if category %}
10              <h1>{{ category.name }}</h1>
11              {% if pages %}
12              <ul>
13                  {% for page in pages %}
14                  <li><a href="{{ page.url }}">{{ page.title }}</a></li>
15                  {% endfor %}
16              </ul>
17              {% else %}
18              <strong>No pages currently in category.</strong>
19              {% endif %}
20          {% else %}
21              The specified category does not exist!
22          {% endif %}
23          </div>
24      </body>
25  </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 – such as `category.name` and `page.url`, for example.

If the `category` exists, then we check to see if there are any pages in the category. If so, we iterate through the returned pages using the {% for page in pages %} template tags. For each `page` in the `pages` list, we present their `title` and `url` attributes as listed hyperlink (e.g. within `<li>` and `<a>` elements). These are displayed in an unordered HTML list (denoted by the `<ul>` tags). If you are not too familiar with HTML, then have a look at the HTML Tutorial by W3Schools.com[6] to learn more about the different tags.

---

[6]http://www.w3schools.com/html/

ℹ Note on Conditional Template Tags

The Django template conditional tag – represented with `{% if condition %}` – 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 when rendering your templates, especially if your associated view's logic doesn't populate the context dictionary in all possible scenarios.

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 how the rendered page is presented to the user. Remember, presentational aspects of your Django apps should be encapsulated within templates.

Parameterised URL Mapping

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

```python
urlpatterns = [
    path('', views.index, name='index'),
    path('about/', views.about, name='about'),
    path('category/<slug:category_name_slug>/',
        views.show_category, name='show_category'),
]
```

A parameter, represented by `<slug:category_name_slug>`, is added to a new mapping. This indicates to Django that we want to match a string which is a `slug`, and to assign it to variable `category_name_slug`. You will notice that this variable name is what we pass through to the view `show_category()`. If these two names do not match exactly, Django will get confused and raise an error. Instead of slugs, you can also extract out other variables like strings and integers. Refer to the Django documentation on URL paths[7] for more details. If you need to parse more complicated expressions, you can use `re_path()` instead of `path()`. This will allow

---

[7]https://docs.djangoproject.com/en/2.1/ref/urls/

you to match all sorts of regular (and irregular) expressions. Luckily for us, Django provides matches for the most common patterns.

All view functions defined as part of a Django app *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's why `show_category()` 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[8]

Django's `path()` method means you can generally avoid Regex Hell – but if you need to use a regular expression (with the `re_path()` function, for instance), this cheat sheet[9] is really useful.

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.

```
1  <!DOCTYPE html>
2  {% load staticfiles %}
3  <html>
4      <head>
5          <title>Rango</title>
6      </head>
7
8      <body>
9              hey there partner! <br />
10             <strong>{{ boldmessage }}</strong>
11
```

---

[8]http://blog.codinghorror.com/regular-expressions-now-you-have-two-problems/
[9]http://cheatography.com/davechild/cheat-sheets/regular-expressions/

```
12          <div>
13              hey there partner!
14          </div>
15
16          <div>
17          {% if categories %}
18          <ul>
19              {% for category in categories %}
20              <!-- The following line is changed to add an HTML hyperlink -->
21              <li>
22              <a href="/rango/category/{{ category.slug }}/">{{ category.name }}</a>
23              </li>
24              {% endfor %}
25          </ul>
26          {% else %}
27              <strong>There are no categories present.</strong>
28          {% endif %}
29          </div>
30
31          <div>
32              <a href="/rango/about/">About Rango</a><br />
33              <img src="{% static "images/rango.jpg" %}" alt="Picture of Rango" />
34          </div>
35      </body>
36 </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 an 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/other-frameworks/` for the category `Other Frameworks`.

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. Look at your template's logic and work through it to understand what is going on.
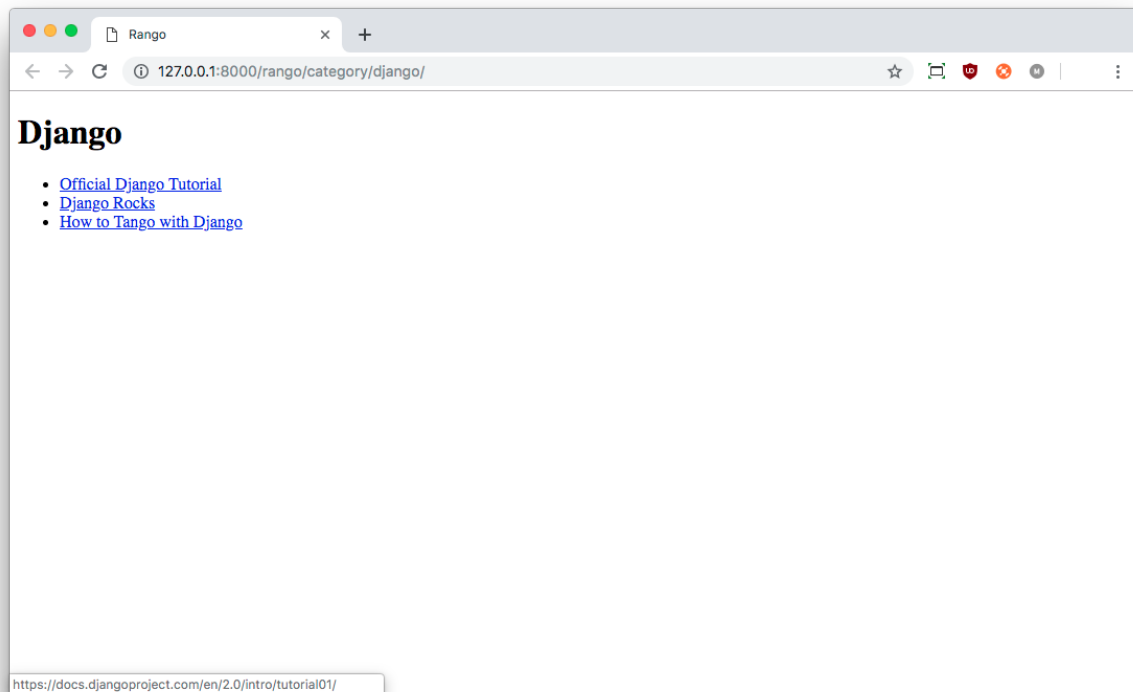
The links to Django pages. Note the mouse is hovering over the first link – you can see the corresponding URL for that link at the bottom left of the Google Chrome window.

✏ Exercises

Reinforce what you've learnt in this chapter!

- Update the population script to add some value to the `views` count for each **page**. Pick whatever integers you want – as long as each page receives a whole (integer) number greater than zero.
- Modify the index page to also include the top five most viewed pages. When no pages are present, you should include a friendly message in place of a list, stating: `There are no pages present`. This message should be bolded – wrap it around `<strong>...</strong>` tags.
- Leading on from the exercise above, include a heading for the `Most Liked Categories` and `Most Viewed Pages`. These must be placed as second-level headers, using the `<h2>` tag. Place each of the headers above their respective list.
- Include a link back to the index page from the category page.
- Undertake part three of official Django tutorial[10] if you have not done so already to reinforce what you've learnt here.

---

[10]https://docs.djangoproject.com/en/2.1/intro/tutorial03/
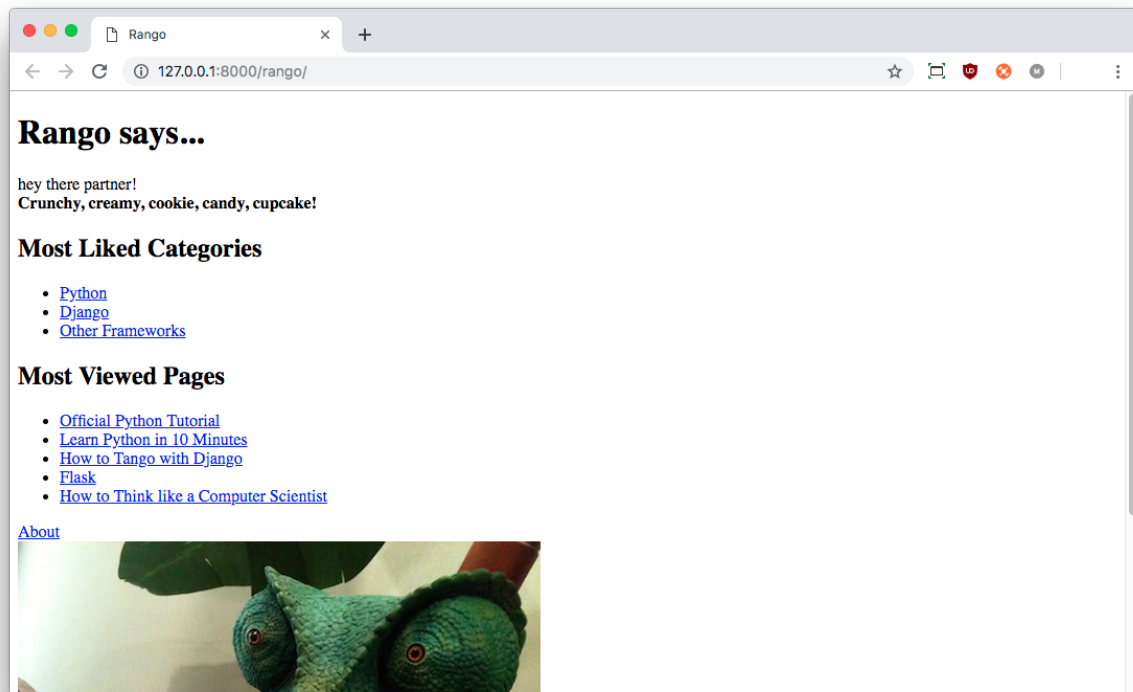
The index page after you complete the exercises. Your output may vary slightly.

🔑 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 `python_pages`, `django_pages` and `other_pages` data structures. 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 database is updated with your new counts.
  - 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 markup. The markup that you need to write 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[11] 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[12] 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.

Test your Implementation

Like in the previous chapter, we've implemented a series of unit tests to allow you to check your implementation up until this point. Follow the guide we provided earlier, using the test module `tests_chapter6.py`. How does your implementation stack up against our tests? Remember that your implementation should have fully completed the exercises listed above for the tests to pass.

Some of these tests may seem overly harsh – but remember, this book is a specification for the product we want you to develop. If you don't develop software *exactly* as specified, it can produce undesirable results when your bit of code is plugged into a larger framework. By following specifications to the letter, you'll be learning a valuable lesson that you can take forward in a future development career.

---

[11]http://steelkiwi.com/blog/best-practices-working-django-models-python/
[12]https://medium.com/@raiderrobert/make-your-django-models-dryer-4b8d0f3453dd#.ozrdt3rsm

7. Forms

As part of the Rango application, we will want to capture new categories and new pages from users. In this chapter, we will run through how to capture data through web forms. Django comes with some excellent 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](#)[1], 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` module within your Django app's directory (`rango`) 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

---

[1]https://docs.djangoproject.com/en/2.1/topics/forms/

- *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 new one).

This workflow is a bit more complicated than those we have previously seen, and the views that we have to construct are lot more complex, too. However, once you undertake the process a few times, it will become clearer how everything pieces together. Trust us.

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 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 several classes that inherit from Django's `ModelForm`. In essence, a `ModelForm`[2] 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.

---

[2]https://docs.djangoproject.com/en/2.1/topics/forms/modelforms/#modelform

```
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; 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 (don't 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 several 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` module.

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. Since the fields will be hidden, the user won't be able to enter a value for these fields.

However, even though we have a `hidden` field in the `PageForm`, 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). This would mean that 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 the 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()`.

However, 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 for populating the field when the form is eventually saved. 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 the flagging up of any errors, along with saving and displaying the data in the form.

We also use the `Meta` class to specify which fields 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.

> ℹ️ **More about Forms**
>
> Check out the official Django documentation on forms[3] 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`.

```python
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)
```

---

[3]https://docs.djangoproject.com/en/2.1/ref/forms/

```
    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})
```

You'll need to add the following `import` at the top of the module, too.

```
from rango.forms import CategoryForm
```

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` (did the user submit 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 *HTTP requests*.

- An 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 some other file.
- In contrast, an 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, an HTTP POST may end up being programmed to create a new resource (e.g. a new database entry) on the server. This could later be accessed through an HTTP GET request.
- Check out the w3schools page on GET vs. POST[4] for more details.

Django's form handling machinery processes the data returned from a user's browser via an 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[5]. For example, supplying no value in the category name field will return an error, as the field cannot be blank.

From the render() call, you'll see that we refer to add_category.html – a new template (we define this below!). 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.

---

[4]http://www.w3schools.com/tags/ref_httpmethods.asp
[5]https://en.wikipedia.org/wiki/Referential_integrity

```
1   <!DOCTYPE html>
2   <html>
3       <head>
4           <title>Rango</title>
5       </head>
6
7       <body>
8           <h1>Add a Category</h1>
9           <div>
10              <form id="category_form" method="post" action="/rango/add_category/">
11                  {% csrf_token %}
12                  {% for hidden in form.hidden_fields %}
13                      {{ hidden }}
14                  {% endfor %}
15                  {% for field in form.visible_fields %}
16                      {{ field.errors }}
17                      {{ field.help_text }}
18                      {{ field }}
19                  {% endfor %}
20                  <input type="submit" name="submit" value="Create Category" />
21              </form>
22          </div>
23      </body>
24  </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 an 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,

- with the first controlling *hidden* form fields, and
- the second controlling *visible* form fields.

The visible fields (those that will be displayed to the user) are controlled by the
`fields` attribute within your `ModelForm` `Meta` class. These template loops produce the
necessary 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.

ℹ Hidden Fields

The need for hidden as well as visible form fields are 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 an HTML form, only to be sent back to the originating server when the user submits the form.

ℹ 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` request 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[6] 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` list to make it look like the following.

---

[6]https://docs.djangoproject.com/en/2.1/ref/csrf/

```
urlpatterns = [
    path('', views.index, name='index'),
    path('about/', views.about, name='about'),
    path('category/<slug:category_name_slug>/', views.show_category,
        name='show_category'),
    path('add_category/', views.add_category, name='add_category'),
]
```

Ordering doesn't necessarily matter in this instance. However, take a look at the official Django documentation on how Django processes a request[7] for more information. The URL for adding a category is /rango/add_category/, with the name of add_category.

Modifying the Index Page View

As a final step, we can put a link on the index page so that users can then easily navigate to the page that allows them to 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 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. In the screenshots, we add the category *Assembly*.

---

[7]https://docs.djangoproject.com/en/2.1/topics/http/urls/#how-django-processes-a-request
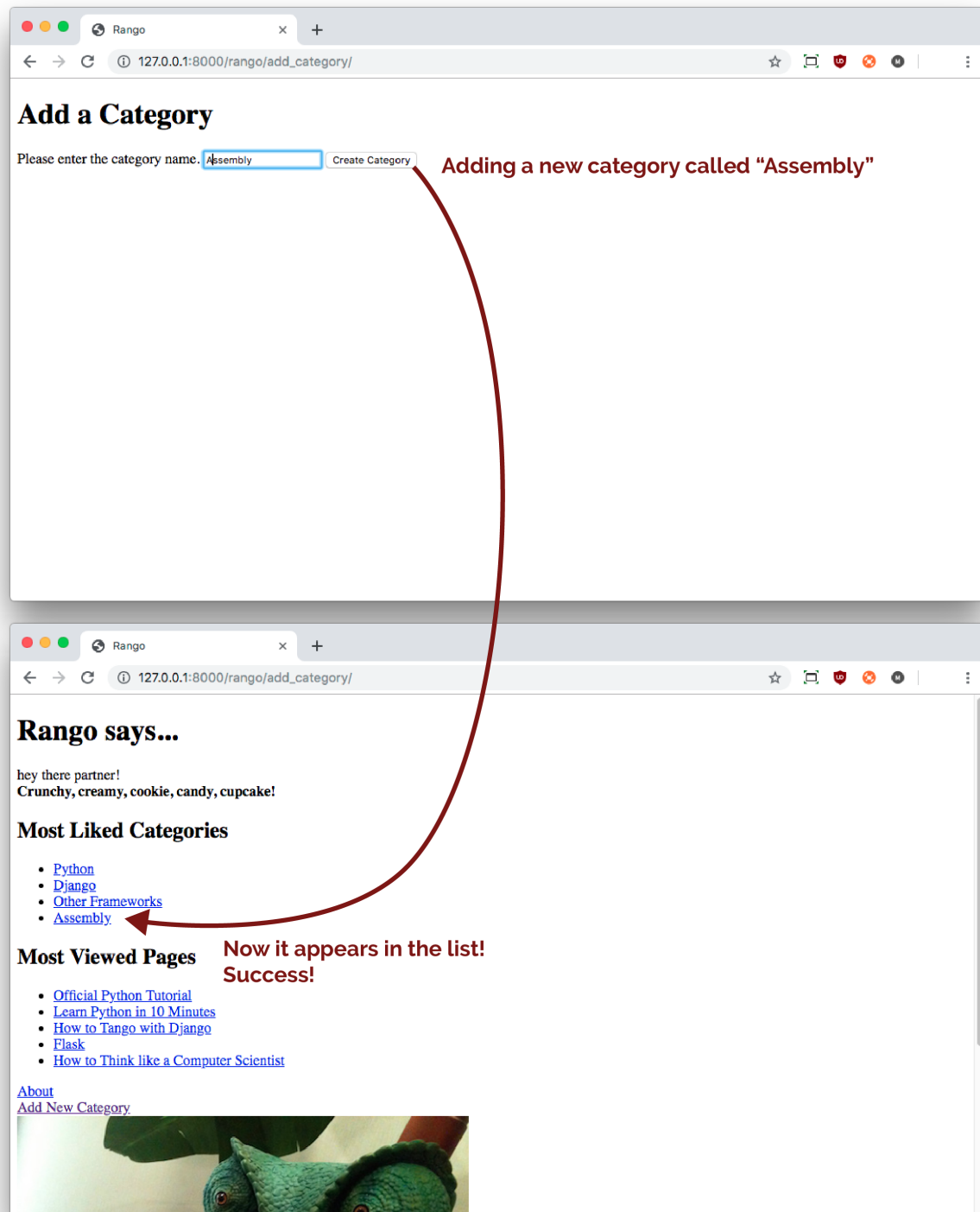
Adding a new category to Rango with our new form.

> ### ℹ Missing Categories?
>
> If you play around with this new functionality and add several different categories, remember that they will not always appear on the index page. This is because we coded up our index view to only show the *top five categories* in terms of the number of likes they have received. If you log into the admin interface, you should be able to view all the categories that you have entered.
>
> For 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 created `Category` object. You can then `print` the category(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[8]!

> ### ℹ 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

---

[8] https://support.google.com/webmasters/answer/76329?hl=en

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.

```python
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.

What about `https`?

The overridden `clean()` method we provide above only considers `http://` as a valid protocol/schema. As using secure HTTP is now commonplace in today's world, you should also really consider URLs starting with `https://`, too. The above however only serves as a simple example of how you can check and clean a form's fields before saving its data.

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[9] for more examples on how you can override default functionality to slot your own in.

[9]https://docs.djangoproject.com/en/2.1/topics/db/models/#overriding-predefined-model-methods

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[10] to reinforce what you have learnt here.
- Now, implement functionality to 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`.
- Create a mapping between `/rango/category/<category_name_slug>/add_page/` and the new view.
- 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.

---

[10]https://docs.djangoproject.com/en/2.1/intro/tutorial04/

```python
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 redirect(reverse('rango:show_category',
                                        kwargs={'category_name_slug':
                                                category_name_slug}))
        else:
            print(form.errors)

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

Note that in the example above, we need to *redirect* the user to the `show_category()` view once the page has been created. This involves the use of the `redirect()` and `reverse()` helper functions to redirect the user and to lookup the appropriate URL, respectively. The following imports at the top of Rango's `views.py` module will therefore be required for this code to work.

```python
from django.shortcuts import redirect
from django.urls import reverse
```

Here, the `redirect()` function is called which in turn calls the `reverse()` function. `reverse()` looks up URL names in your `urls.py` modules – in this instance, `rango:show_category`. If a match is found against the name provided, the complete URL is returned. The added complication here is that the `show_category()` view

takes an additional parameter `category_name_slug`. By providing this value in a dictionary as `kwargs` to the `reverse()` function, it has all of the information it needs to formulate a complete URL. This completed URL is then used as the parameter to the `redirect()` method, and the response is complete!

Hints

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

- In the `add_page.html` template, you can access the slug with `{{ category.slug }}`. This is 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. In terms of code, we mean that your template should have the following conditional: `{% if category %}` .... `{% else %}` `The specified category does not exist. {% endif %}`.
- Update Rango's `category.html` template with a new hyperlink, complete 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. Provide a name of `add_page` to the new mapping.
- You can avoid the repetition of `max_length` parameters through the use of an additional attribute in your `Category` model. 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[11].

---

8. Final Thoughts

In this tutorial, 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 necessary models, views and templates associated with a sample web application, *Rango.* 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.

We have only painted the broad brush strokes of a web application in this tutorial. 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 with solid foundations, 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 typos, bugs, or other issues online via Twitter[1], or submit change requests via GitHub[2]. 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.

---

[1] https://twitter.com/tangowithdjango
[2] https://github.com/leifos/tango_with_django_2

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[3], **Alessio Oxilia,** Ally Weir, **bernieyangmh**[4]**,** Breakerfall[5], **Brian**[6]**,** Burak K.[7], **Burak Karaboga,** Can Ibanoglu[8], **Charlotte,** Claus Conrad[9], **Codenius**[10]**,** cspollar[11], **Dan C,** Darius[12], **David Manlove,** David Skillman[13], **Deep Sukhwani**[14] Devin Fitzsimons[15], **Dhiraj Thakur**[16]**,** Duncan Drizy, **Gerardo A-C**[17]**,** Giles T.[18], **Grigoriy M**[19]**,** James Yeo, **Jan Felix Trettow**[20]**,** Joe Maskell, **Jonathan Sundqvist**[21]**,** Karen Little, **Kartik Singhal**[22]**,** koviusesGitHub[23], **Krace Kumar**[24]**,** ma-152478[25], **Manoel Maria,** marshwiggle[26] **Martin de G.**[27]**,** Matevz P.[28], **mHulb**[29]**,** Michael Herman[30], **Michael Ho Chum**[31]**,** Mickey P.[32], **Mike Gleen,** Muhammad Radifar, **nCrazed**[33]**,** Nitin Tulswani, **nolan-m**[34]**,** Oleg Belausov, **Olivia Foulds,** pawonfire[35], **pdehaye**[36]**,** Peter Mash, **Pierre-Yves Mathieu**[37]**,** Piotr Synowiec[38], **Praest-**

[3] https://github.com/Amertz08
[4] https://github.com/bernieyangmh
[5] https://github.com/breakerfall
[6] https://github.com/flycal6
[7] https://github.com/McMutton
[8] https://github.com/canibanoglu
[9] https://github.com/cconrad
[10] https://twitter.com/Codenius
[11] https://github.com/cspollar
[12] https://github.com/dariushazimi
[13] https://github.com/reggaedit
[14] https://github.com/ProProgrammer
[15] https://github.com/aisflat439
[16] https://github.com/dhirajt
[17] https://github.com/gerac83
[18] https://github.com/gpjt
[19] https://github.com/GriMel
[20] https://twitter.com/JanFelixTrettow
[21] https://github.com/jonathan-s
[22] https://github.com/k4rtik
[23] https://github.com/koviusesGitHub
[24] https://github.com/kracekumar
[25] https://github.com/ma-152478
[26] https://github.com/marshwiggle
[27] https://github.com/martindegroot
[28] https://github.com/matonsjojc
[29] https://github.com/mHulb
[30] https://github.com/mjhea0
[31] https://github.com/michaelchum
[32] https://github.com/mickeypash
[33] https://github.com/nCrazed
[34] https://github.com/nolan-m
[35] https://github.com/pawonfire
[36] https://github.com/pdehaye
[37] https://github.com/pywebdesign
[38] https://holysheep.co/

gias, pzkpfwVI[39], **Ramdog**[40]**,** Rezha Julio[41], **rnevius**[42]**,** Sadegh Kh, **Saex**[43]**,** Saurabh Tandon[44], **Serede Sixty Six,** Svante Kvarnstrom, **Tanmay Kansara,** Thomas Murphy, **Thomas Whyyou**[45]**,** William Vincent, and **Zhou**[46]**.**

Thank you all very much!

---

[39]https://github.com/pzkpfwVI
[40]https://github.com/ramdog
[41]https://github.com/kimiamania
[42]https://github.com/rnevius
[43]https://github.com/SaeX
[44]https://twitter.com/saurabhtand
[45]https://twitter.com/thomaswhyyou
[46]https://github.com/AugustLONG

9. Setting up your System

This supplementary chapter provides additional setup guides that complement the initial setup chapter. We provide setup guides for installing and configuring the various technologies that you will be using within Tango with Django tutorial. Refer to the section that is relevant to you; you do not need to work through all of this chapter if things are already working for you.

> ### Common Guides
>
> This chapter provides instructions on how to set up the various technologies that you'll be using throughout Tango with Django that we *believe* will work on the largest number of systems. However, every computer setup is different. Different versions of software exist, complete with subtle differences. These differences make providing universal setup guides a very difficult thing to do.
>
> If you are using this book as part of a course, you may be provided with setup instructions unique to your lab computers. Follow these instructions instead – a majority of the setup work will likely be taken care of for you already.
>
> However if you are working solo on your computer and you follow the instructions provided in this chapter without success, we recommend heading to your favourite search engine and entering the problem you're having. Typically, this will involve copying and pasting the error message you see at whatever step you're struggling at. By pasting in the message verbatim, chances are you'll find someone who suffered the same issue as you – and from that point, you'll hopefully find a solution to resolve your problem.

9.1 Installing Python 3 and `pip`

How do you go about installing Python 3.7 on your computer? This section answers that question. As we discussed previously, you may find that you already

have Python installed on your computer. If you are using a Linux distribution or macOS, you will have it installed. Some of your operating system's functionality is implemented in Python[1], hence the need for an interpreter! Unfortunately, most modern operating systems that come preloaded with Python use a version that is much older than what we require. Exceptions include Ubuntu, coming with version 3.6.5 which should be sufficient for your needs. If you do need to install 3.7, we must install this version of Python *side-by-side* with the old one.

There are many different ways in which you can install Python. We demonstrate here the most common approaches that you can use on Apple's macOS, various Linux distributions and Windows 10. Pick the section associated with your operating system to proceed. Note that we favour the use of package managers[2] where appropriate to ensure that you have the means of maintaining and updating the software easily (when required).

Apple macOS

The simplest way to acquire Python 3 for macOS is to download a `.dmg` image file from the official Python website[3]. This will provide you with a step-by-step installation interface that makes setting everything up straightforward. If your development environment will be kept lightweight (i.e. Python only), this option makes sense. Simply download the installer, and Python 3 should then be available on your Mac's terminal!

However, package managers make life easier[4] when development steps up and involves a greater number of software tools. Installing a package manager makes it easy to maintain and update the software on your computer – and even to install new software, too. macOS does not come preinstalled with a package manager, so you need to download and install one yourself. If you want to go down this route, we'll introduce you to *MacPorts*, a superb package manager offering a large host of tools[5] for you to download and use. We recommend that you follow this route. Although more complex, the result will be a complete development environment,

---

[1]http://en.wikipedia.org/wiki/Yellowdog_Updater,_Modified

[2]https://en.wikipedia.org/wiki/Package_manager

[3]https://www.python.org/downloads/mac-osx/

[4]https://softwareengineering.stackexchange.com/questions/372444/why-prefer-a-package-manager-over-a-library-folder

[5]https://www.macports.org/ports.php

ready for you to get coding and working on whatever project you (or your future self!) will be entrusted with.

A prerequisite for using MacPorts is that you have Apple's *Xcode* environment installed. This download is several gigabytes in size. The easiest way to acquire this is through the App Store on your macOS installation. You'll need your Apple account to download that software. Once XCode has been installed, follow the following steps to setup MacPorts.

1. Verify that XCode is installed by launching it. You should see a welcome screen. If you see this, everything is ready, so you can then quit the app.
2. Open a Terminal window. Install the XCode command line tools by entering the command

   ```
   $ xcode-select --install
   ```

   This will download additional software tools that will be required by XCode and additional development software that you later install.
3. Agree to the XCode license, if you have not already. You can do this by entering the command

   ```
   $ xcode-build license
   ```

   Read the terms to the bottom of the page, and type Y to complete – but only if you agree to the terms!
4. From the MacPorts installation page[6], download the MacPorts installer for your correct macOS version.
5. On your Mac's Finder, open the directory where you downloaded the installer to, and double-click the file to launch the installation process.
6. Follow the steps, and provide your password to install the software.
7. Once complete, delete the installer file – you no longer require it. Close down any Terminal windows that are still open.

Once the MacPorts installation has been completed, installing Python is straight-forward.

---

[6]https://www.macports.org/install.php

1. Open a new Terminal window. It is important that you launch a new window after MacPorts installation to ensure that all the changes the installer made come into effect!

2. Enter the following command.

   ```
   $ sudo port install python37
   ```

   After entering your password, this will take a few minutes. Several dependencies will be required – agree to these being installed by responding with `Y`.

3. Once installation completes, activate your new Python installation. Enter the command

   ```
   $ sudo port select --set python python37
   ```

4. Test that the command succeeds by issuing the command `$ python`. You should then see the interpreter for Python 3.7.2 (or whatever version you just installed – as long as it starts with `3.7`, this will be fine).

Once this has been completed, Python has been successfully installed and is ready to use. However, we still need to set up virtual environments to work with your installation.

1. Enter the following commands to install `virtualenv` and helper functions provided in the user-friendly wrapper.

   ```
   $ sudo port install py37-virtualenv
   $ sudo port install py37-virtualenvwrapper
   ```

2. Activate `py37-virtualenv` with the following command.

   ```
   $ sudo port select --set virtualenv virtualenv37
   ```

3. Edit your ∼/`.profile` file. This can be done with the *TextEdit* app in macOS by issuing the command `$ open ∼/.profile`. Add the following four lines at the end of the file.

```
export VIRTUALENVWRAPPER_PYTHON='/opt/local/bin/python3.7'
export VIRTUALENVWRAPPER_VIRTUALENV='/opt/local/bin/virtualenv-3.7'
export VIRTUALENVWRAPPER_VIRTUALENV_CLONE='/opt/local/bin/virtualenv-clone-3.7'
source /opt/local/bin/virtualenvwrapper.sh-3.7
```

4. Save the file and close all open Terminals. After doing this, open a new Terminal. Everything should now be working and ready for you to use. Test with the following command. You should *not* see the `command not found` error.

   ```
   $ mkvirtualenv
   ```

> ### ℹ️ Installing Additional Software with MacPorts
>
> MacPorts provides an extensive, preconfigured library of open-source software suited specifically for development environments. When you need something new, it's a cinch to install. **For example,** you want to install the *LaTeX* typesetting system, search the MacPorts ports list[7] – the resultant package name being `texlive-latex`. This could then be installed with the command `$ sudo port install texlive-latex`. All software that LaTeX is dependent upon is also installed. This saves you significant amounts of time trying to find all the right bits of software to make things work.
>
> To view the packages MacPorts has already installed on your system, issue the command `$ port list installed`. You will see `python37` listed!

Linux Distributions

There are many different ways in which you can download, install and run an updated version of Python on your Linux distribution. Unfortunately, methodologies vary from distribution to distribution. To compound this, almost all distributions of Linux don't have a precompiled version of Python 3.7 ready for you to download and start using (at the time of writing), although the latest release of Ubuntu uses Python 3.6 (which is sufficient).

If you do choose to install a new version of Python, we've put together a series of steps that you can follow. These will allow you to install Python 3.7.2 from scratch.

---

[7]https://www.macports.org/ports.php

The steps have been tested thoroughly in Ubuntu 18.04 LTS; other distributions should also work with minor tweaks, especially concerning the package manager being used in step 1. A cursory search on your favourite search engine should reveal the correct command to enter. For example, on a *Red Hat Enterprise Linux* installation, the system package manager is `yum` instead of `apt`.

⚠ Assumption of Knowledge

To complete these steps, we assume you know the basics for Bash interaction, including what the tilde ($\sim$) means, for example. Be careful with the `sudo` command, and do not execute it except for the steps we list requiring it below.

1. Install the required packages for Python to be built successfully. These are listed below, line by line. These can be entered into an Ubuntu Terminal as-is; slight modifications will be required for other Linux distributions.

   ```
   $ apt install wget
   $ apt install build-essential
   $ apt install libssl-dev
   $ apt install libffi-dev
   $ apt install python-dev
   $ apt install zlib1g-dev
   $ apt install libbz2-dev
   $ apt install libreadline-dev
   $ apt install libsqlite3-dev
   ```

2. Once the above packages are installed, download the source for Python 3.7.2. We create a `pytemp` directory to download the file to. We'll delete this once everything has completed.

   ```
   $ mkdir ~/pytemp
   $ cd ~/pytemp
   $ wget https://www.python.org/ftp/python/3.7.2/Python-3.7.2.tgz
   ```

3. Extract the `.tgz` file.

```
$ tar zxf Python-3.7.2.tgz
$ cd Python-3.7.2
```

4. Configure the Python source code for your computer, and build it. `altinstall` tells the installer to install the new version of Python to a different directory from the pre-existing version of Python on your computer. You'll need to enter your password for the system to make the necessary changes. This process will take a few minutes, and you'll see a lot of output. Don't panic. This is normal. If the build fails, you haven't installed all of the necessary prerequisites. Check you have installed everything correctly from step 1, and try again.

```
$ sudo ./configure --enable-optimizations
$ sudo make altinstall
```

5. Once complete, delete the source files. You don't need them anymore.

```
$ cd ~
$ rm -rf ~/pytemp
```

6. Attempt to run the new installation of Python. You should see the interpreter prompt for version 3.7.2, as expected.

```
$ python3.7
Python 3.7.2 (default, Jan. 8 2019, 20:05:08)
[GCC 7.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information
>>> quit()
$
```

7. By default, Python executables install to `/usr/local/bin`. Check this is correct for you. If not, run the `which` command to find out where it is installed. You'll need this path later.

```
$ which python3.7
/usr/local/bin/python3.7
```

8. Install `virtualenv` and `virtualenvwrapper` for your new Python installation.

```
$ pip3.7 install virtualenv
$ pip3.7 install virtualenvwrapper
```

9. Modify your ~/.bashrc file, and include the following lines at the very bottom of the file. Note that if you are not using Ubuntu, you might need to edit ~/.profile instead. Check the documentation of your distribution for more information. A simple text editor will allow you to do this, like nano.

```
EXPORT VIRTUALENVWRAPPER_PYTHON=/usr/local/bin/python3.7
source /usr/local/bin/virtualenvwrapper.sh
```

10. Restart your Terminal. Python 3.7.2 will now be set up and ready for you to use, along with the `pip` and `virtualenv` tools.

Windows

By default, Microsoft Windows comes with no installation of Python. This means that you do not have to worry about leaving existing installations alone; installing from scratch should work just fine. You can download a 64-bit of 32-bit version of Python from the official Python website[8]. If you aren't sure what 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[9].

1. Download the appropriate installer from the official Python website[10]. At the time of writing, the latest release was version 3.7.2.
2. Run the installer. You'll want to make sure that you check the box saying that Python 3.7 is added to `PATH`. You'll want to install for all users, too. Choose the `Customize` option.
3. Proceed with the currently selected checkboxes, and choose `Next`.
4. Make sure that the checkbox for installing Python for all users is checked. The installation location will change. Refer to the figure below for an example.
5. Click `Next` to install Python. You will need to give the installer elevated privileges to install the software.
6. Close the installer when completed, and delete the file you downloaded.

Once the installer is complete, you should have a working version of Python 3.7 installed and ready to go. Following the instructions above, Python 3.7 is installed to the directory `C:\Program Files\Python37`. If you checked all of the options correctly, the `PATH` environment variable used by Windows should also have been updated to incorporate the new installation of Python. To test this, launch a Command Prompt window and type $ `python`. Execute the command. You should see the

---

[8]http://www.python.org/download/
[9]https://support.microsoft.com/en-gb/help/13443/windows-which-operating-system
[10]http://www.python.org/download/

Python interpreter launch. If this fails, check your PATH environment variable is set correctly by following an online guide[11].

Once you are sure that Python is installed correctly, you need to install virtual environment support. Issue the following two commands, and then restart all open Command Prompt windows.

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

Once completed, everything should be set up and ready for you to use.
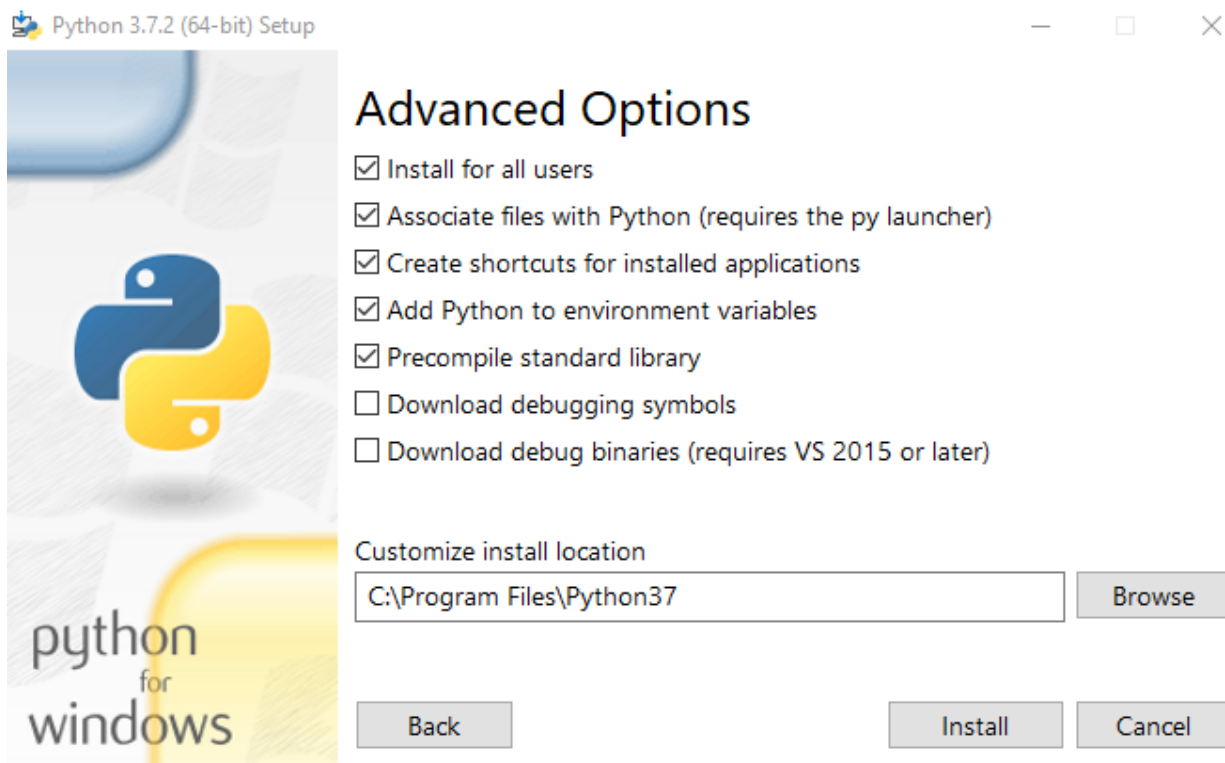
Configuring Python 3.7.2 on Windows 10 x64 – allowing the installation to be run by all users.

9.2 Virtual Environments

By default, when you install software for Python, it is installed *system-wide*. All Python applications can see the new software and make use of it. However, issues

---
[11]https://www.architectryan.com/2018/03/17/add-to-the-path-on-windows-10/

can occur with this setup. Earlier in the book, we discussed a scenario of two pieces of software requiring two different versions of the *same dependency*. This presents a headache; you cannot typically install two different versions of the same software into your Python environment!

The solution to this is to use a *virtual environment*. Using a virtual environment, each different piece of software that you wish to run can be given its environment, and by definition, its set of installed dependencies. If you have `ProjectA` requiring Django 1.11 and `ProjectB` requiring Django 2.1, you could create a virtual environment for each with their packages installed.

The five basic commands one would use to manipulate virtual environments are listed below.

- `mkvirtualenv <name>` creates and activates a new virtual environment of name `<name>`.
- `workon <name>` switches on a virtual environment of name `<name>`.
- `deactivate` switches off a virtual environment you are currently using.
- `rmvirtualenv <name>` deletes a virtual environment of name `<name>`.
- `lsvirtualenv` lists all user-created virtual environments.

Following the examples above, we can then create an environment for each, installing the required software in the relevant environment. For `ProjectA`, the environment is called `projAENV`, with `projBENV` used for `ProjectB`. Note that to install software to the respective environments we use `pip`. The commands used for `pip` are discussed below.

```
$ mkvirtualenv projAENV
(projAENV) $ pip install Django==1.11
(projAENV) $ pip freeze
Django==1.11

(projAENV) $ deactive
$ pip
<Command not found>

$ workon projAENV
(projAENV) $ pip freeze
Django=1.11

(projAENV) $ cd ProjectA/
(projAENV) $ python manage.py runserver
...

(projAENV) $ deactivate
$
```

The code blocks above create the new virtual environment, projAENV. We then install Django 1.11 to that virtual environment, before issuing pip freeze to list the installed packages – confirming that Django was installed. We then deactivate the virtual environment. pip then cannot be found as we are no longer in the virtual environment! By switching the virtual environment back on with workon, our Django package can once again be found. The final two commands launch the Django development server for ProjectA.

We can then create a secondary virtual environment, projBENV.

```
$ mkvirtualenv projBENV
(projBENV) $ pip install Django==2.1
(projBENV) $ pip freeze
Django==2.1

(projBENV) $ cd ProjectA/
(projBENV) $ python manage.py runserver
<INCORRECT PYTHON VERSION!>

(projBENV) $ workon projAENV
(projAENV) $ python manage.py runserver

(projAENV) $ workon projBENV
(projBENV) $ cd ProjectB/
$ python manage.py runserver
...
```

We create our new environment with the `mkvirtualenv` command. This creates and activates `projBENV`. However, when trying to launch the code for `ProjectA`, we get an error! We are using the wrong virtual environment. By switching to `projAENV` with `workon projAENV`, we can then launch the software correctly. This demonstrates the power of virtual environments, and the advantages that they can bring. Further tutorials can be found online.[12]

🔑 workon and deactivate

Start your session by switching on your virtual environment with the `workon` command. Finish your session by closing it with `deactivate`.

You can tell if a virtual environment is active by the brackets before your prompt, like `(envname)  $`. This means that virtual environment `envname` is currently switched on, with its settings loaded. Turn it off with `deactivate` and the brackets will disappear.

---

[12]https://realpython.com/python-virtual-environments-a-primer/

🔑 Multiple Python Versions

If you have multiple versions of Python installed, you can choose what version of Python to use when you create a virtual environment. If you have installations for `python` (which launches 2.7.15) and `python3` (which launches 3.7.2), you can issue the following command to create a Python 3 virtual environment.

```
$ mkvirtualenv -p python3 someenv
$ python
$ Python 3.7.2
>>>
```

Note that when you enable your virtual environment, the command you enter to start Python is simply `python`. The same is applied for `pip` – if you launch `pip3` outside a virtual environment, `pip` will be the command you use inside the virtual environment.

9.3 Using `pip`

The Python package manager is very straightforward to use and allows you to keep track of the various Python packages (software) that you have installed. We highly recommend that you use `pip` alongside a virtual environment, as packages installed using `pip` appear only within the said virtual environment.

When you find a package that you want to install, the command required is `$ pip install <packagename>==<version>`. Note that the version component is optional; omitting the version of a particular package will mean that the latest available version is installed.

You can find the name of a package by examining the *PyPi package index*[13], from which `pip` downloads software. Simply search or browse the index to find what you are looking for. Once you know the package's name, you can issue the installation command in your Terminal or Command Prompt.

`pip` is also super useful for listing packages that are installed in your environment.

---

[13]https://pypi.org/

This can be achieved through the `pip freeze` command. Sample output is issued below.

```
(rangoenv) $ pip freeze
Django==2.1.5
Pillow==5.4.1
pytz==2018.9
```

This shows that three packages are installed in a given environment: `Django`, `Pillow` and `pytz`. The output from this command is typically saved to a file called `requirements.txt`, stored in the root directory of a Python project. If somebody wants to use your software, they can then download your software – complete with the `requirements.txt` file. Using this file, they can then create a new virtual environment set up with the required packages to make the software work.

The recommended way to create your own `requirements.txt` file is to pipe the output of the `pip freeze` command to the file. Navigate to the directory where you want to create the file, and issue the following command.

```
$ cd awesome_python_project
$ pip freeze > requirements.txt
```

This then creates your `requirements.txt` file. You can then add this to your project's version control. A Python developer who sees a `requirements.txt` file should know exactly what to do with it!

If you find yourself in a situation like this, you run `pip` with the `-r` switch. Given a `requirements.txt` in a directory `downloaded_project` with only `pytz==2018.9` listed, an example CLI session would involve something like the following.

```
$ cd downloaded_project
$ mkvirtualenv someenv
(someenv) $ pip install -r requirements.txt
...
(someenv) $ pip freeze
pytz==2018.9
```

`pip install` installs packages from `requirements.txt`, and `pip freeze`, once everything has been installed, demonstrates that the packages have been installed correctly.

9.4 Version Control System

When developing code, it's highly recommended that you house your codebase within a version-controlled repository such as SVN[14] or Git[15]. We have provided a chapter on how to use Git if you haven't used Git and GitHub before. We highly recommend that you set up a Git repository for your projects. Doing so could save you from disaster.

To use Git, we recommend that you use the command-line tool to interact with your repositories. This is done through the `git` command. On Windows, you'll need to download Git from the Git website[16]. If using macOS or Linux, the Git website also has downloadable installers for you to use[17]. However, why not get into the habit of using a package manager to install the software? This is generally the recommended way for downloading and using software developed on the UNIX design principles (including macOS).

For example, installing Git is as simple as typing $ `sudo apt install git`. Let the software download, and the `apt` package manager takes care of the rest. If you installed MacPorts on your macOS installation as described above, Git will already be present for you as it is part of the Apple XCode Command Line Developer Tools.

Once installed, typing `git` will show the commands you can use, as shown in the example below.

---

[14]http://subversion.tigris.org/
[15]http://git-scm.com/
[16]https://git-scm.com/download/win
[17]https://git-scm.com/downloads

```
$ git
usage: git [--version] [--help] [-C <path>] [-c name=value]
       [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
       [-p | --paginate | --no-pager] [--no-replace-objects] [--bare]
       [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
       <command> [<args>]
```

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 macOS and Linux distributions. If you're a Windows user, you can use the Windows Command Prompt[1] or Windows PowerShell[2]. Users of Windows 10 can now install the necessary infrastructure straight from Microsoft[3] to use Bash on your Windows installation! You could also experiment by installing Cygwin[4] to bring Bash commands to Windows.

10.1 Using the Terminal

UNIX based operating systems and derivatives – such as macOS and Linux distributions – all use a similar-looking terminal application, typically using the Bash shell[5]. All possess a core set of commands that allow you to navigate through

---

[1]http://www.ai.uga.edu/mc/winforunix.html
[2]https://msdn.microsoft.com/en-us/powershell/mt173057.aspx
[3]https://docs.microsoft.com/en-us/windows/wsl/install-win10
[4]https://www.cygwin.com/
[5]https://en.wikipedia.org/wiki/Bash_(Unix_shell)

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 $\sim$); 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[6]. Whatever symbol is present is used to signify that the computer is awaiting your input.

ℹ 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 on how your computer is set up.

---

[6]http://en.wikipedia.org/wiki/Superuser

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: Sun Jul 21 15:47:24 2019
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 as 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 several different shorthand ways for you to move around your computer's filesystem. You've already seen that a forward slash (/) represents the [root directory](https://en.wikipedia.org/wiki/Root_directory)[7], and the tilde (∼) represents your home directory in which you store all your 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 only works if you've moved 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.

---

[7]https://en.wikipedia.org/wiki/Root_directory

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

Issuing another `pwd` command to confirm our present working directory returns the output `/users/grad/david/code`. This is 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 is something present in 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 21 Jul 15:00 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 21 Jul 15:00 code
-rw-r--r--@ 1 david  grad  303844  2 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 21 Jul 15:00 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 1s 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[8] 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.

---

[8]http://man7.org/linux/man-pages/man1/ls.1.html

```
sibu:~ david$ ls -l
drwxr-xr-x  2 david  grad      68 21 Jul 15:00 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`[9] or `vi`[10]. While we don't cover how to use these editors here, you can have a look online for a simple how-to tutorial[11]. 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 present 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 present working directory to `/users/grad/david/`. You can also move up a directory level without having to provide the absolute path[12] 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`.

[9]http://www.nano-editor.org/
[10]http://en.wikipedia.org/wiki/Vi
[11]http://www.howtogeek.com/howto/42980/the-beginners-guide-to-nano-the-linux-command-line-text-editor/
[12]http://www.coffeecup.com/help/articles/absolute-vs-relative-pathslinks/

- `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 present working directory. You need to supply a name for the new directory after the `mkdir` command. For example, if your present 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[13]. 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[14] 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[15] for a more detailed overview, or the Cheat Sheet[16] 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.

---

[13]http://www.computerhope.com/issues/ch000798.htm
[14]http://en.wikipedia.org/wiki/Superuser
[15]https://help.ubuntu.com/community/UsingTheTerminal
[16]http://fosswire.com/post/2007/08/unixlinux-command-cheat-sheet/

11. A Git Crash Course

We strongly recommend that you spend some time familiarising yourself with a version control[1] system for your application's codebase. This chapter provides you with a crash course in how to use Git[2], one of the many version control systems available. Originally developed by Linus Torvalds[3], Git is today one of the most popular version control systems in use[4], 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 explains 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.

11.1 Why Use Version Control?

As your software engineering skills develop, you will find that you can 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)*[5] 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?

---

[1]https://en.wikipedia.org/wiki/Version_control
[2]http://en.wikipedia.org/wiki/Git_(software)
[3]http://en.wikipedia.org/wiki/Linus_Torvalds
[4]https://en.wikipedia.org/wiki/Comparison_of_source_code_hosting_facilities#Popularity
[5]https://en.wikipedia.org/wiki/Don't_repeat_yourself

- 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 (or 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.

11.2 How Git Works

Essentially, Git is comprised 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 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[6] 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[7], for example), or simply use one of many services providing free Git repositories. One of the most widely used services available today is GitHub[8]. In fact this book has a Git repository[9] 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.

---

[6]http://www.sbf5.com/~cduan/technical/git/git-1.shtml
[7]http://serverfault.com/questions/189070/what-firewall-ports-need-to-be-open-to-allow-access-to-external-git-repositories
[8]https://github.com/
[9]https://github.com/maxwelld90/tango_with_django_2_code/

- 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[10] 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 the 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.

11.3 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.

---

[10]http://stackoverflow.com/questions/4084921/what-does-the-git-index-exactly-contains

ℹ️ 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[11]. 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 nice right-click Git context menu for files. This makes version control easy to use. You can download TortoiseGit[12] for free. Although we do not cover how to use TortoiseGit in this crash course, many tutorials exist online for it. Check this tutorial[13] 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[14] and created a new repository on the service[15] for your project. With your remote repository setup, follow these steps to get your local repository and workspace set up 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!

---

[11]http://git-scm.com/download/win
[12]https://code.google.com/p/tortoisegit/
[13]http://robertgreiner.com/2010/02/getting-started-with-git-and-tortoisegit-on-windows/
[14]https://github.com/
[15]https://help.github.com/articles/create-a-repo

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://github.com/<OWNER>/<REPO_NAME> <workspace>
```

where you replace

- `<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/maxwelld90/tango_with_django_2_code/ twd2code
Cloning into 'twd2cc'...
remote: Enumerating objects: 354, done.
remote: Counting objects: 100% (354/354), done.
remote: Compressing objects: 100% (253/253), done.
remote: Total 354 (delta 217), reused 223 (delta 86), pack-reused 0
Receiving objects: 100% (354/354), 1.11 MiB | 1.78 MiB/s, done.
Resolving deltas: 100% (217/217), done.
```

If the output lines end with `done`, everything should have worked. Check your filesystem to see if the directory has been created by running $ `ls` – is the directory now listed?

Not using GitHub?

Many websites 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[16] and Unfuddle[17]. You will 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 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 useful 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.

---

[16]https://bitbucket.org/
[17]https://unfuddle.com/

```
$ git config user.name "Ivan Petrov"
$ git config user.email "ivanpetrov123@me.bg"
```

Replace the example name and e-mail address with your own – unless your name is John Doe!

The `.gitignore` File

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:

```
.pyc
.DS_Store
__pycache__/
db.sqlite3
*.key
thumbs.db
```

In this example `.gitignore` file, there are six entries – one on each line. We detail each of the entries below.

- The first entry prompts Git to ignore any file with an extension of `.pyc` – the wildcard `*` denoting *anything.* A `.pyc` file is a bytecode representation of a Python module[18], something that Python creates for modules to speed up execution. These can be safely ignored when committing to version control.

---

[18]https://stackoverflow.com/a/2998228

- The second entry, `.DS_Store`, is a hidden file created by macOS systems. These files contain custom attributes that you set when browsing through your filesystem using the Finder. Custom attributes may, for example, represent the position of icons, or the view that you use to show your files in a given directory.
- The third entry represents any directory of the name `__pycache__`. This directory is new to Python 3, and is where the bytecode representation[19] of your modules live. Again, these directories can be safely ignored.
- `db.sqlite3` is your database. Read the note below for a detailed explanation on why this should always be excluded from your repository.
- `*.key` is the fifth entry. This represents any file with the extension `key`, a convention we use where files with this extension contain sensitive information – like keys for API authentication. These should never be committed – doing so would be the equivalent of publicly sharing your password!
- The final entry is `thumbs.db`. Like the `.DS_Store` file created by macOS, this is the Windows equivalent.

Remember, wildcards are a neat feature to use here. If you ever have a type of file you do not wish to commit, then just use a `*.abc` to denote any filename, with the extension `.abc`.

.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 macOS creates `.DS_Store` files.

---

[19]https://stackoverflow.com/a/16869074

(Not) Committing your Database

You should also take steps to ignore other binary-based files that will frequently change within your repository. As an example in this book, you'll want to consider ignoring the database (as we demonstrate above). Why? When working in a team, everyone's copy of the database is likely going to be different. Different team members may add sample data that will differ from others. If all team members were to continually push up different versions of the database, the chance of creating conflicts increases. This issue is also the reason why we encourage the development of a population script to quickly fill a new database with sample data! The rule of thumb here is: don't commit a database to your project's repository. Do commit the necessary infrastructure to create the database from scratch, and as a bonus, provide the functionality to fill this blank database with sample data.

11.4 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.
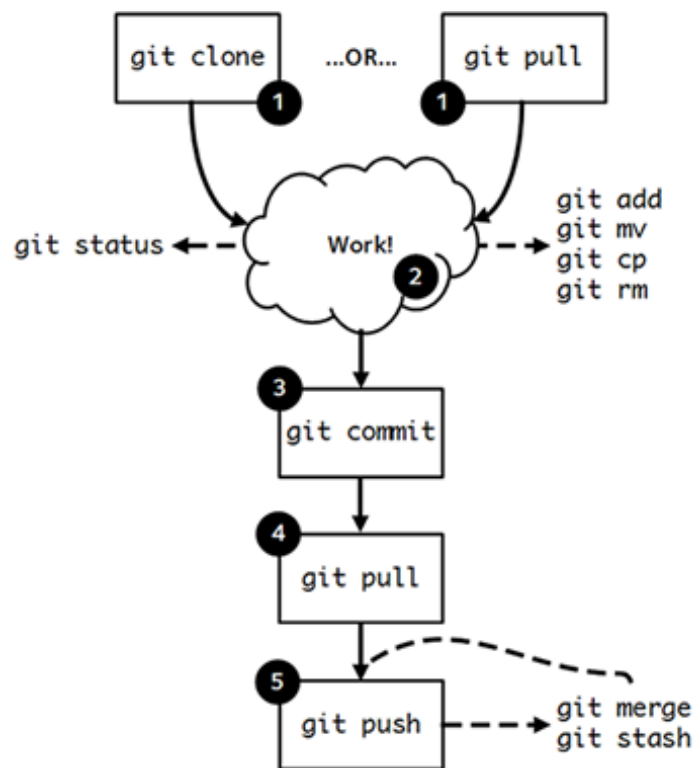
A Figure of the Git workflow.

We have provided a 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 with the most up-to-date version of the repository. This will reduce the possibility of conflicting versions of files, which really can make your life a bit of a nightmare.

To perform a `git pull`, first navigate to your repository directory within your Command Prompt or terminal, then issue the command `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 somerepository/
$ 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 local repository directory – the one in which you cloned your repository to. **The 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` to ensure you have the latest changes.

Remember to talk to your team to coordinate your activity so you are not working on the same files, or use *branching*[20] to keep things separate. Branching will then at some point involve merging back to a single branch.

---

[20]https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging

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 you make – this does not take place automatically. Telling Git what has changed allows you to keep your local index updated. The list of files stored within the local index is then used to perform your next *commit*, which we'll be discussing in the next step. To keep Git informed, several Git commands 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`), save for the addition of a `git` prefix.

- The first command `git add` allows you to request Git to add a particular file to *staging* for the next commit. For any file that you wish to include in version control – whether it is new or updated, you must `git add` it. 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**[21].
- `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 current 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

---

[21]http://stackoverflow.com/a/16969786

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[22] 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. Have a look at the 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 is 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 really 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.

---

[22]http://en.wikipedia.org/wiki/Rm_(Unix)

```
On branch master
Your branch is up-to-date with 'origin/master'.

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
  modified:   chapter-system-setup.md

Untracked files:
  (use "git add <file>..." to include in what will be committed)

  images/screenshot_template.ai

no changes added to commit (use "git add" and/or "git commit -a")
```

Files `chapter-git.md`, `chapter-system-setup.md` and `images/screenshot_template.ai` are all listed in the example output shown above. All paths are relative from your present working directory, hence the addition of `images/` for the last file – it lives within an `images` subdirectory.

Reading the output of the `git status` command carefully, we can see that the files `chapter-git.md` and `chapter-system-setup.md` are already known to your Git repository, but have been *modified* from the previous commit. The output of the command is telling you that for these changes to be reflected in the next commit, you must `git add` these files. Files that have been deleted or moved concerning the previous commit will also appear here.

There is also a section for *untracked files.* These files have never been committed to version control – and as such will not show up on any Git commits. Files that you have just created will appear in this section. For them to be committed, you simply `git add` each file in turn. If you then run `git status` again, you will see them move up to the first section.

Checking Status

For further information on the `git status` command, check out the official Git documentation[23].

---
[23]https://git-scm.com/docs/git-status

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 to an older version of your code if things go wrong. It's like a checkpoint in a game – reach the checkpoint and you can save your status, perhaps giving you the ability to roll back in the future if you need to.

Make sure you commit often, **but don't commit an incomplete or broken version of a particular module or function. Your colleagues will hate you for it.** There's a lot of discussion as to when the ideal time to commit is. Have a look at this Stack Overflow page[24] for the opinions of several developers. However, it does make sense to commit only when everything is working. To reiterate, put yourself in the shoes of a developer who has found their colleague has committed something that is broken. How would you feel?

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[25].

ℹ️ **Commit Requirements**

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.

---

[24]http://stackoverflow.com/questions/1480723/dvcs-how-often-and-when-to-commit-changes
[25]http://stackoverflow.com/questions/2304087/what-is-git-head-exactly

```
$ git commit -m "Chapter 5 exercises completed"
```

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 models.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 *works* before you commit. This may sound silly, but it's an incredibly easy thing to not think about. To reiterate (for the third time), **committing code which doesn'troll back 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. The simplest kind of merge is for a file that someone else has changed, but you haven't touched. This is to be expected – and a simple merge of this kind simply merges the other person's edits with your edits to produce a merged version of the repository.

The horrible kind of merge conflict happens when you and someone else work on the same file at the same time. To fix this, you'll need to carefully examine the file and figure out what changes stay, and what changes go. This can be very time consuming, so communication between team members is the key to avoid this scenario.

Editing Merge Logs

If you do see a text editor on your Mac or Linux installation, it's probably the `vi`[26] text editor. If you've never used vi before, check out this helpful page containing a list of basic commands[27] on the Colorado State University Computer Science Department website. If you don't like vi, you can change the default text editor[28] 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 how your changes become available to your other team members, who can then retrieve them by running the `git`

---

[26]http://en.wikipedia.org/wiki/Vi
[27]http://www.cs.colostate.edu/helpdocs/vi.html
[28]http://git-scm.com/book/en/Customizing-Git-Git-Configuration#Basic-Client-Configuration

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 in this Stack Overflow question and answer page[29] 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[30], 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 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[31] for more information.

11.5 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 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

---

[29]http://stackoverflow.com/questions/7311995/what-is-git-push-origin-master-help-with-gits-refs-heads-and-remotes
[30]http://git-scm.com/book/en/Git-Branching-Basic-Branching-and-Merging
[31]https://help.github.com/articles/using-pull-requests

commit. This section details how to do just that. We follow the information given from this Stack Overflow[32] 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[33] to see how to do that.

Rolling back your workspace to a previous commit involves two steps: determining which commit to rollback to, and 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 why it is highly beneficial to supply commit messages that provide enough information to explain what you do at each commit!** 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 2019 +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 2019 +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 2019 +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 2019 -0700
    Initial commit
```

---

[32]http://stackoverflow.com/questions/2007662/rollback-to-an-old-commit-using-git
[33]http://stackoverflow.com/a/3338145

From this list, you can choose a commit to rollback to. For the commit you want to rollback to, 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 the aforementioned hash.

```
$ 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[34] 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.

---

[34]https://help.github.com/articles/github-flavored-markdown

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[35], the Git Guide[36] or Learning about Git Branching[37].

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!

---

[35]https://veerasundar.com/blog/2011/06/git-tutorial-getting-started/
[36]https://rogerdudler.github.io/git-guide/
[37]https://pcottle.github.io/learnGitBranching/