



Quick answers to common problems

QGIS Python Programming Cookbook

Over 140 recipes to help you turn QGIS from a desktop GIS tool
into a powerful automated geospatial framework

Joel Lawhead

[PACKT] open source community experience distilled
PUBLISHING

QGIS Python Programming Cookbook

Over 140 recipes to help you turn QGIS from a desktop GIS tool into a powerful automated geospatial framework

Joel Lawhead



BIRMINGHAM - MUMBAI

QGIS Python Programming 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: March 2015

Production reference: 1240315

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78398-498-5

www.packtpub.com

Credits

Author

Joel Lawhead

Copy Editor

Dipti Kapadia

Reviewers

Joshua Arnott

Giuseppe De Marco

Jonathan Gross

Luigi Pirelli

Hiroaki Sengoku

Project Coordinator

Shipra Chawhan

Proofreaders

Safis Editing

Maria Gould

Commissioning Editor

Pramila Balan

Indexer

Hemangini Bari

Acquisition Editor

Sonali Vernekar

Production Coordinator

Nitesh Thakur

Content Development Editor

Prachi Bisht

Cover Work

Nitesh Thakur

Technical Editor

Deepti Tuscano

About the Author

Joel Lawhead is a PMI-certified Project Management Professional (PMP) and the Chief Information Officer (CIO) of NVisionSolutions Inc., an award-winning firm that specializes in geospatial technology integration and sensor engineering.

Joel began using Python in 1997 and began combining it with geospatial software development in 2000. He is the author of *Learning Geospatial Analysis with Python*, Packt Publishing. His Python cookbook recipes were featured in two editions of *Python Cookbook*, O'Reilly Media. He is also the developer of the widely used, open source Python Shapefile Library (PyShp) and maintains the geospatial technical blog GeospatialPython.com and the Twitter feed @SpatialPython, which discuss the use of the Python programming language within the geospatial industry.

In 2011, Joel reverse engineered and published the undocumented shapefile spatial indexing format and assisted fellow geospatial Python developer, Marc Pfister, in reversing the algorithm used, allowing developers around the world to create better-integrated and more robust geospatial applications involving shapefiles.

Joel served as the lead architect, project manager, and co-developer for geospatial applications used by US government agencies, including NASA, FEMA, NOAA, the US Navy, and many other commercial and non-profit organizations. In 2002, he received the international Esri Special Achievement in GIS award for his work on the Real-Time Emergency Action Coordination Tool (REACT), for emergency management using geospatial analysis.

I would like to acknowledge my beautiful family, including my wife, Julie, and four children, Lauren, Will, Lillie, and Lainie, who allowed me to write yet another book in our limited collective free time. I would also like to acknowledge my employers and coworkers at NVisionSolutions.com, a bright team of people dedicated to working together at the exciting bleeding edge of geospatial technology.

About the Reviewers

Joshua Arnott is an environmental scientist with four years of academic and consultancy experience. His expertise lies in environmental modeling, with a focus on hydrology and geoinformatics. He has contributed to a number of GIS-related open source projects, including QGIS and Shapely. He maintains a blog about programming and GIS at snorfalorpagus.net, and he likes cats just as much as everyone else on the Internet.

Giuseppe De Marco was born in 1973 in Ferentino, Italy. He has a high school certificate in humanities and attained a bachelor's degree in agriculture from the University of Pisa. When he was a small boy, he began to use computers and learn programming languages (BASIC, Pascal, Fortran, and so on). At the university, he began to encounter open source software and the Linux OS, and he developed a deep interest in geography and GIS and other programming languages, such as C++ and Python, by first getting in touch with Esri commercial products and later with GRASS and QGIS. Since the QGIS 1.7.4 release, he's been developing plugins for this software, sometimes purely to seek knowledge and at other times for work. In 2008, he began a professional partnership with two colleagues called Pienocampo (open field), and his plugins are hosted on Pienocampo's website and on the QGIS official repository. At the moment, he lives in his hometown Ferentino and works as a freelance agriculture engineer. His work activities include studying geography, surveying, tree risk assessment, landscaping, bioengineering, and farm consulting. In 2014, he also began to teach other colleagues how to use QGIS and other open source software.

I would like to thank my wife, Fabiola; my little daughter, Anna; my mother, Angela; and my colleagues, Marco De Castris, Ettore Arcangeletti, Luca Grande, and Ivan Solinas.

Jonathan Gross is the author of the Open Source GIS blog, <http://opensourcegisblog.blogspot.com/>. He has a master's of public health degree in epidemiology from the University of Michigan, Ann Arbor, and a graduate certificate in geographic information systems from Johns Hopkins Advanced Academic Programs. He has done graduate coursework in Python and uses Python for programming small tasks. He is currently an epidemiologist at the Baltimore City Health Department, Maryland, where he performs spatial analysis on health and crime data.

Luigi Pirelli is a freelance software analyst and developer with a honors degree in computer science from the University of Bari.

He has worked for 15 years in satellite ground segmentation and direct ingestion systems for the European Space Agency. Since 2006, he has been involved in the GFOSS world, contributing to QGIS, GRASS, and the MapServer core, and developing and maintaining many QGIS plugins. He actively participates in QGIS Hackmeetings.

He is the founder of the OSGEO Italian local chapter GFOSS.it and now lives in Spain, where he contributes to the GFOSS community. During the past few years, he started teaching PyQGIS by organizing trainings, from basic to advanced level, supporting companies to develop their specific QGIS plugins.

He has coauthored *Mastering QGIS*, Packt Publishing.

He is the founder of the local hackerspace group, Bricolabs.cc that is focused on all things related to open source hardware. He likes to cycle, repair everything, and train groups on conflict resolution.

Other than this book, he has also contributed to the guide, *Cycling Italy, Lonely Planet*.

A special thanks to the QGIS developer community and core developers because the project is managed in an open way, allowing contribution from everyone.

I want to thank everyone I have worked with. From each one of them, I learned something and without them, I wouldn't be here, contributing to free software and this book.

A special thanks to my friends and neighbors who helped me with my son during the review of the book.

I would like to dedicate this work to my partner and especially my son, for having the patience to see me sit in front of the computer for hours without playing with him.

Hiroaki Sengoku was born in 1987 in Gifu, Japan. He did his BA in environmental information from Keio University in 2009. He completed an MA in environmental studies from the University of Tokyo in 2011 and a PhD in environmental studies from the University of Tokyo in 2014. He is the founder and CEO of Microbase Inc., which he established when he was a PhD student. He is interested in the field of microgeographic simulation and has held many workshops on this. His dream is to create a real *SimCity*.

Microbase Inc. is the company that creates microdemographic data in Japan. This company has created simulated urban data, such as people flow or people's lifestyles, using open data. The members of Microbase Inc. aim to create microdemographic data all over the world and a simulation platform, such as *SimCity*, using this data.

You can watch a demo movie at <https://www.youtube.com/watch?v=kXKRU4CLJro> and <http://microgeodata.com/>.

I couldn't have reviewed this book without the help of the members of Microbase Inc. I'd like to thank them for their help in the reviewing process. Also, I would like to thank Shipra Chawhan and Paushali Desai, who gave me the chance to review this book. I had an exciting experience and appreciate their efforts.

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	vii
Chapter 1: Automating QGIS	1
Introduction	1
Installing QGIS for development	2
Using the QGIS Python console for interactive control	5
Using the Python ScriptRunner plugin	6
Setting up your QGIS IDE	8
Debugging QGIS Python scripts	13
Navigating the PyQGIS API	17
Creating a QGIS plugin	19
Distributing a plugin	22
Creating a standalone application	25
Storing and reading global preferences	27
Storing and reading project preferences	28
Accessing the script path from within your script	30
Chapter 2: Querying Vector Data	31
Introduction	31
Loading a vector layer from a file sample	32
Loading a vector layer from a spatial database	34
Examining vector layer features	36
Examining vector layer attributes	37
Filtering a layer by geometry	38
Filtering a layer by attributes	40
Buffering a feature intermediate	42
Measuring the distance between two points	44
Measuring the distance along a line sample	45

Table of Contents

Calculating the area of a polygon	47
Creating a spatial index	48
Calculating the bearing of a line	49
Loading data from a spreadsheet	51
Chapter 3: Editing Vector Data	55
Introduction	56
Creating a vector layer in memory	56
Adding a point feature to a vector layer	57
Adding a line feature to a vector layer	59
Adding a polygon feature to a vector layer	60
Adding a set of attributes to a vector layer	62
Adding a field to a vector layer	63
Joining a shapefile attribute table to a CSV file	65
Moving vector layer geometry	67
Changing a vector layer feature's attribute	68
Deleting a vector layer feature	70
Deleting a vector layer attribute	71
Reprojecting a vector layer	72
Converting a shapefile to KML	73
Merging shapefiles	74
Splitting a shapefile	75
Generalizing a vector layer	76
Dissolving vector shapes	78
Performing a union on vector shapes	80
Rasterizing a vector layer	82
Chapter 4: Using Raster Data	85
Introduction	86
Loading a raster layer	87
Getting the cell size of a raster layer	89
Obtaining the width and height of a raster	90
Counting raster bands	91
Swapping raster bands	92
Querying the value of a raster at a specified point	93
Reprojecting a raster	94
Creating an elevation hillshade	96
Creating vector contours from elevation data	98
Sampling a raster dataset using a regular grid	100
Adding elevation data to line vertices using a digital elevation model	104

Creating a common extent for rasters	106
Resampling raster resolution	108
Counting the unique values in a raster	110
Mosaicing rasters	111
Converting a TIFF image to a JPEG image	112
Creating pyramids for a raster	113
Converting a pixel location to a map coordinate	114
Converting a map coordinate to a pixel location	116
Creating a KML image overlay for a raster	117
Classifying a raster	121
Converting a raster to a vector	122
Georeferencing a raster from control points	124
Clipping a raster using a shapefile	126
Chapter 5: Creating Dynamic Maps	129
Introduction	130
Accessing the map canvas	130
Changing the map units	131
Iterating over layers	132
Symbolizing a vector layer	133
Rendering a single band raster using a color ramp algorithm	135
Creating a complex vector layer symbol	137
Using icons as vector layer symbols	139
Creating a graduated vector layer symbol renderer	141
Creating a categorized vector layer symbol	142
Creating a map bookmark	144
Navigating to a map bookmark	146
Setting scale-based visibility for a layer	147
Using SVG for layer symbols	148
Using pie charts for symbols	150
Using the OpenStreetMap service	154
Using the Bing aerial image service	155
Adding real-time weather data from OpenWeatherMap	157
Labeling features	158
Changing map layer transparency	159
Adding standard map tools to the canvas	160
Using a map tool to draw points on the canvas	163
Using a map tool to draw polygons or lines on the canvas	165
Building a custom selection tool	168
Creating a mouse coordinate tracking tool	171

Chapter 6: Composing Static Maps	173
Introduction	173
Creating the simplest map renderer	174
Using the map composer	176
Adding labels to a map for printing	179
Adding a scale bar to the map	181
Adding a north arrow to the map	183
Adding a logo to the map	186
Adding a legend to the map	188
Adding a custom shape to the map	189
Adding a grid to the map	193
Adding a table to the map	195
Adding a world file to a map image	197
Saving a map to a project	199
Loading a map from a project	200
Chapter 7: Interacting with the User	201
Introduction	202
Using log files	202
Creating a simple message dialog	203
Creating a warning dialog	204
Creating an error dialog	205
Displaying a progress bar	206
Creating a simple text input dialog	208
Creating a file input dialog	209
Creating a combobox	211
Creating radio buttons	212
Creating checkboxes	214
Creating tabs	216
Stepping the user through a wizard	218
Keeping dialogs on top	221
Chapter 8: QGIS Workflows	223
Introduction	224
Creating an NDVI	224
Geocoding addresses	227
Creating raster footprints	229
Performing network analysis	233
Routing along streets	236
Tracking a GPS	238
Creating a mapbook	242

Table of Contents

Finding the least cost path	245
Performing nearest neighbor analysis	247
Creating a heat map	249
Creating a dot density map	253
Collecting field data	255
Computing road slope using elevation data	258
Geolocating photos on the map	262
Image change detection	266
Chapter 9: Other Tips and Tricks	269
Introduction	270
Creating tiles from a QGIS map	270
Adding a layer to geojson.io	274
Rendering map layers based on rules	276
Creating a layer style file	280
Using NULL values in PyQGIS	282
Using generators for layer queries	283
Using alpha values to show data density	284
Using the __geo_interface__ protocol	288
Generating points along a line	289
Using expression-based labels	291
Creating dynamic forms in QGIS	292
Calculating length for all selected lines	295
Using a different status bar CRS than the map	296
Creating HTML labels in QGIS	297
Using OpenStreetMap's points of interest in QGIS	300
Visualizing data in 3D with WebGL	302
Visualizing data on a globe	305
Index	309

Preface

The open source geographic information system, QGIS, at version 2.6 now rivals even the most expensive commercial GIS software in both functionality and usability. It is also a showcase of the best geospatial open source technology available. It is not just a project in itself, but the marriage of dozens of open source projects in a single, clean interface.

Geospatial technology is not just the combined application of technology to geography. It is a symphony of geography, mathematics, computer science, statistics, physics, and other fields. The underlying algorithms implemented by QGIS are so complex that only a handful of people in the world can understand all of them. Yet, QGIS packages all this complexity so well that school children, city managers, disease researchers, geologists, and many other professionals wield this powerful software with ease to make decisions that improve life on earth.

However, this book is about another feature of QGIS that makes it the best choice for geospatial work. QGIS has one of the most deeply-integrated and well-designed Python interfaces of any software, period. In the latest version, there is virtually no aspect of the program that is off limits to Python, making it the largest geospatial Python library available. Almost without exception, the Python API, called PyQGIS, is consistent and predictable.

This book exploits the best features of QGIS to demonstrate over 140 reusable recipes, which you can use to automate workflows in QGIS or to build standalone GIS applications. Most recipes are very compact, and even if you can't find the exact solution that you are looking for, you should be able to get close. This book covers a lot of ground and pulls together fragmented ideas and documentation scattered throughout the Internet as well as the results of many hours of experimenting at the edges of the PyQGIS API.

What this book covers

Chapter 1, Automating QGIS, provides a brief overview of the different ways in which you can use Python with QGIS, including the QGIS Python console, standalone applications, plugins, and the Script Runner plugin. This chapter also covers how to set and retrieve application settings and a few other Python-specific features.

Chapter 2, Querying Vector Data, covers how to extract information from vector data without changing the data using Python. The topics covered include measuring, loading data from a database, filtering data, and other related processes.

Chapter 3, Editing Vector Data, introduces the topic of creating and updating data to add new information. It also teaches you how to break datasets apart based on spatial or database attributes as well as how to combine datasets. This chapter will also teach you how to convert data into different formats, change projections, simplify data, and more.

Chapter 4, Using Raster Data, demonstrates 25 recipes to use and transform raster data in order to create derivative products. This chapter highlights the capability of QGIS as a raster processing engine and not just a vector GIS.

Chapter 5, Creating Dynamic Maps, transitions into recipes to control QGIS as a whole in order to control map, project, and application-level settings. It includes recipes to access external web services and build custom map tools.

Chapter 6, Composing Static Maps, shows you how to create printed maps using the QGIS Map Composer. You will learn how to place reference elements on a map as well as design elements such as logos.

Chapter 7, Interacting with the User, teaches you how to control QGIS GUI elements created by the underlying Qt framework in order to create interactive input widgets for scripts, plugins, or standalone applications.

Chapter 8, QGIS Workflows, contains more advanced recipes, which result in a finished product or an extended capability. These recipes target actual tasks that geospatial analysts or programmers encounter on the job.

Chapter 9, Other Tips and Tricks, contains interesting recipes that fall outside the scope of the previous chapters. Many of these recipes demonstrate multiple concepts within a single recipe, which you may find useful for a variety of tasks.

What you need for this book

You will need the following software to complete all the recipes in this book; if a specific version is not available, use the most recent version:

- ▶ QGIS 2.6
- ▶ Python 2.7.6 (should be included with QGIS itself)
- ▶ IBM Java 7 Dev Kit
- ▶ Eclipse Luna 4.4.x
- ▶ Google Earth 7.1.2.2041

Who this book is for

If you are a geospatial analyst who wants to learn more about automating everyday GIS tasks or a programmer who is responsible for building GIS applications, this book is for you. Basic knowledge of Python is essential and some experience with QGIS will be an added advantage.

The short, reusable recipes make concepts easy to understand. You can build larger applications that are easy to maintain when they are put together.

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 styles of text 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:
"In the QGIS **Python Console**, we'll import the `random` module."

A block of code is set as follows:

```
proj = QgsProject.instance()
proj.setTitle("My QGIS Project")
proj.title()
proj.writeEntry("MyPlugin", "splash", "Geospatial Python Rocks!")
proj.readEntry("MyPlugin", "splash", "Welcome!") [0]
```

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

```
sudo easy_install PyPDF2
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "Enter information in the form and click on the **Send** button."



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 may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to feedback@packtpub.com, and mention the book title via 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 on 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 for all Packt books you have purchased from your account at <http://www.packtpub.com>. 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.

Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from http://www.packtpub.com/sites/default/files/downloads/49850S_ColoredImages.pdf.

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 would 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 on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from <http://www.packtpub.com/support>.

Piracy

Piracy of copyright 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

You can contact us at questions@packtpub.com if you are having a problem with any aspect of the book, and we will do our best to address it.

1

Automating QGIS

In this chapter, we will cover the following recipes:

- ▶ Installing QGIS for development
- ▶ Using the QGIS Python console
- ▶ Using Python's ScriptRunner plugin
- ▶ Setting up your QGIS IDE
- ▶ Debugging QGIS Python scripts
- ▶ Navigating the PyQGIS API
- ▶ Creating a QGIS plugin
- ▶ Distributing a plugin
- ▶ Building a standalone application
- ▶ Storing and reading global preferences
- ▶ Storing and reading project preferences
- ▶ Accessing the script path from within your script

Introduction

This chapter explains how to configure QGIS for automation using Python. In addition to setting up QGIS, we will also configure the free Eclipse **Integrated Development Environment (IDE)** with the PyDev plugin to make writing, editing, and debugging scripts easier. We will also learn the basics of different types of QGIS automated Python scripts through the PyQGIS API. Finally, we'll examine some core QGIS plugins that significantly extend the capability of QGIS.

Installing QGIS for development

QGIS has a set of Python modules and libraries that can be accessed from the Python console within QGIS. However, they can also be accessed from outside QGIS to write standalone applications. First, you must make sure that PyQGIS is installed for your platform, and then set up some required system environment variables.

In this recipe, we will walk you through the additional steps required beyond the normal QGIS installation to prepare your system for development. The steps for each platform are provided, which also include the different styles of Linux package managers.

Getting ready

QGIS uses slightly different installation methods for Windows, GNU/Linux, and Mac OS X. The Windows installers install everything you need for Python development, including Python itself.

However, on Linux distributions and Mac OS X, you may need to manually install the Python modules for the system installation of Python. On Mac OS X, you can download installers for some of the commonly used Python modules with QGIS from <http://www.kyngchaos.com/software/python>.

How to do it

On Linux, you have the option to compile from the source or you can just specify the Python QGIS interface to be installed through your package manager.

Installing PyQGIS using the Debian package manager

1. For Linux distributions based on the Debian Linux package manager, which includes Ubuntu and Debian, use the following command in a shell:
`sudo apt-get update`
2. Next, install the QGIS, PyQGIS, and QGIS GRASS plugins:
`sudo apt-get install qgis python-qgis qgis-plugin-grass`

Installing PyQGIS using the RPM package manager

1. For Linux distributions based on the **Red Hat Package Manager (RPM)**, first update the package manager, as follows:
`sudo yum update`
2. Then, install the packages for the QGIS, PyQGIS, and QGIS GRASS plugins:
`sudo yum install qgis qgis-python qgis-grass`

Setting the environment variables

Now, we must set the `PYTHONPATH` to the PyQGIS directory. At the same time, append the path to this directory to the `PATH` variable so that you can use the PyQGIS modules with an external IDE.

Setting the environment variables on Windows

1. Set the `PYTHONPATH` variable in a command prompt to the `bin` directory of the QGIS installation:

```
set PYTHONPATH="C:\Program Files\QGIS Brighton\bin"
```

2. Next, append QGIS's `bin` directories to the system's `PATH` variable:

```
set PATH="C:\Program Files\QGIS Brighton\bin";"C:\Program Files\QGIS Brighton\bin\apps\qgis\bin";%PATH%
```

Setting the environment variables on Linux

1. Set the `PYTHONPATH` variable in a command prompt to the `bin` directory of the QGIS installation:

```
export PYTHONPATH=/usr/share/qgis/python
```

2. Now, append the QGIS shared library directory to the runtime search path. Note that this location can vary depending on your particular system configuration:

```
export LD_LIBRARY_PATH=/usr/share/qgis/python
```

How it works...

The QGIS installation process and package managers set up the Python module's configuration internal to QGIS. When you use the Python console inside QGIS, it knows where all the PyQGIS modules are. However, if you want to use the PyQGIS API outside QGIS, using a system Python installation on Windows or Linux, it is necessary to set some system variables so that Python can find the required PyQGIS modules.

There's more...

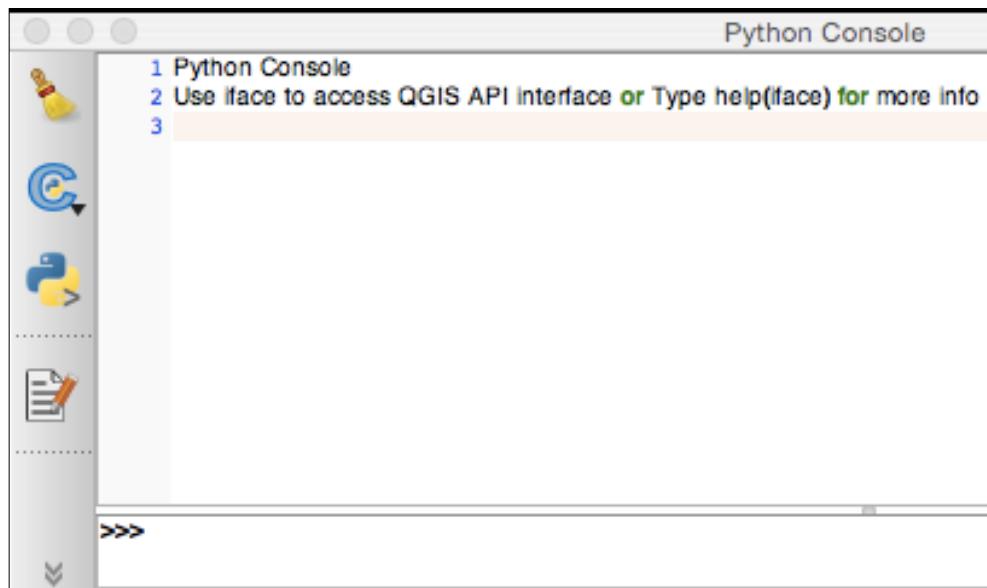
This recipe uses the default QGIS paths on each platform. If you aren't sure which PyQGIS path is for your system, you can figure this out from the Python console in QGIS.

Finding the PyQGIS path on Windows

The libraries on Windows are stored in a different location than in the case of other platforms. To locate the path, you can check the current working directory of the Python console:

1. Start QGIS.

2. Select **Python Console** from the **Plugins** menu, which appears in the lower-right corner of the QGIS application window, as shown in the following screenshot:



3. Use the `os` module to get the current working directory:

```
import os  
os.getcwd()
```

4. Verify that the current working directory of the Python console is returned.

Finding the location of the QGIS Python installation on other platforms

Perform the following steps to find the path needed for this recipe on all the platforms besides Windows:

1. Start QGIS.
2. Start the QGIS **Python Console**.

3. Use the `sys` module to locate the PyQGIS path:

```
import sys  
sys.path
```

4. Python will return a list of paths.

5. Find the path that ends in `/python`, which is the location of the Python installation used by QGIS

Using the QGIS Python console for interactive control

The QGIS Python console allows you to interactively control QGIS. You can test out ideas or just do some quick automation. The console is the simplest way to use the QGIS Python API.

How to do it...

In the following steps, we'll open the QGIS Python console, create a vector layer in memory, and display it on the map:

1. Start QGIS.
2. From the **Plugins** menu, select **Python Console**.
3. The following code will create a point on the map canvas:

```
layer = QgsVectorLayer('Point?crs=epsg:4326', 'MyPoint' ,  
'memory')  
pr = layer.dataProvider()  
pt = QgsFeature()  
point1 = QgsPoint(20,20)  
pt.setGeometry(QgsGeometry.fromPoint(point1))  
pr.addFeatures([pt])  
layer.updateExtents()  
QgsMapLayerRegistry.instance().addMapLayers([layer])
```

How it works...

This example uses a `memory` layer to avoid interacting with any data on disk or a network to keep things simple. Notice that when we declare the layer type, we add the parameter for the **Coordinate Reference System (CRS)** as EPSG:4326. Without this declaration, QGIS will prompt you to choose one. There are three parts or levels of abstraction to create even a single point on the map canvas, as shown here:

- ▶ First, create a layer that is of the type `geometry`. Next, set up a data provider to accept the data source.
- ▶ Then, create a generic feature object, followed by the point geometry.
- ▶ Next, stack the objects together and add them to the map.

The layer type is `memory`, meaning that you can define the geometry and the attributes inline in the code rather than in an external data source. In this recipe, we just define the geometry and skip the defining of any attributes.

Using the Python ScriptRunner plugin

The QGIS Python ScriptRunner plugin provides a middle ground for QGIS automation, between the interactive console and the overhead of plugins. It provides a script management dialog that allows you to easily load, create, edit, and run scripts for large-scale QGIS automation.

Getting ready

Install the **ScriptRunner** plugin using the QGIS plugin manager. Then, run the plugin from the **Plugin** menu to open the **ScriptRunner** dialog. Configure a default editor to edit scripts using the following steps:

1. Find the gear icon that represents the **ScriptRunner Preferences** settings dialog box and click on it.
2. In the **General Options** section, check the **Edit Scripts Using:** checkbox.
3. Click on the ... button to browse to the location of a text editor on your system.
4. Click on the **Open** button.
5. Click on the **OK** button in the **Preferences** dialog.

How to do it...

1. In the **ScriptRunner** dialog, click on the **New Script** icon, as shown in the following screenshot:



2. Browse to the directory where you can save your script, name the script, and save it.
3. Verify that the new script is loaded in **ScriptRunner**.
4. Right-click (or control-click on a Mac) on the script name in ScriptRunner and select **Edit Script in External Editor**.
5. In the editor, replace the template code with the following code:

```
from PyQt4.QtCore import *
from PyQt4.QtGui import *
from qgis.core import *
from qgis.gui import *
```

```
def run_script(iface):
    layer = QgsVectorLayer('Polygon?crs=epsg:4326', 'Mississippi'
, "memory")
    pr = layer.dataProvider()
    poly = QgsFeature()
    geom = QgsGeometry.fromWkt("POLYGON ((-88.82 34.99,-88.09
34.89,-88.39 30.34,-89.57 30.18,-89.73 31,-91.63 30.99,-90.87
32.37,-91.23 33.44,-90.93 34.23,-90.30 34.99,-88.82 34.99))")
    poly.setGeometry(geom)
    pr.addFeatures([poly])
    layer.updateExtents()
    QgsMapLayerRegistry.instance().addMapLayers([layer])
```

6. Click on the Run Script icon, which is represented by a green-colored arrow.
7. Close the **ScriptRunner** plugin.
8. Verify that the memory layer polygon was added to the QGIS map, as shown in the following screenshot:



How it works...

ScriptRunner is a simple but powerful idea. It allows you to build a library of automation scripts and use them from within QGIS, but without the overhead of building a plugin or a standalone application. All the Python and system path variables are set correctly and inherited from QGIS; however, you must still import the QGIS and Qt libraries.

Setting up your QGIS IDE

The Eclipse IDE with the PyDev plugin is cross-platform, has advanced debugging tools, and is free.



You can refer to http://pydev.org/manual_101_install.html in order to install PyDev correctly.



This tool makes an excellent PyQGIS IDE. Eclipse allows you to have multiple Python interpreters configured for different Python environments. When you install PyDev, it automatically finds the installed system Python installations. On Windows, you must also add the Python interpreter installed with PyQGIS. On all platforms, you must tell PyDev where the PyQGIS libraries are.

Getting ready

This recipe uses Eclipse and PyDev. You can use the latest version of either package that is supported by your operating system. All platforms besides Windows rely on the system Python interpreter. So, there is an extra step in Windows to add the QGIS Python interpreter.

How to do it...

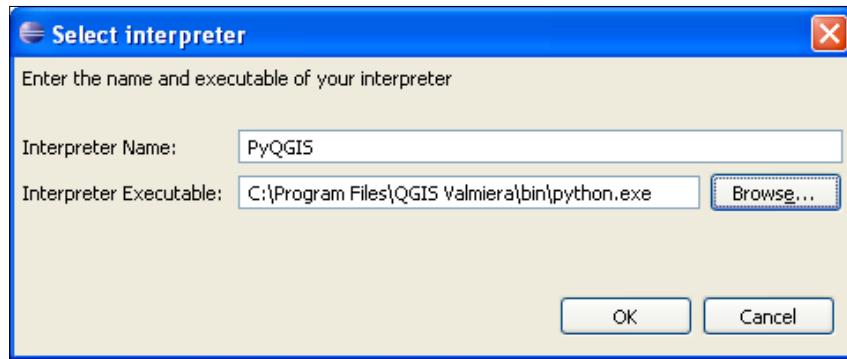
The following steps will walk you through how to add the QGIS-specific Python interpreter to Eclipse in order to support the running standalone QGIS applications or to debug QGIS plugins.

Adding the QGIS Python interpreter on Windows

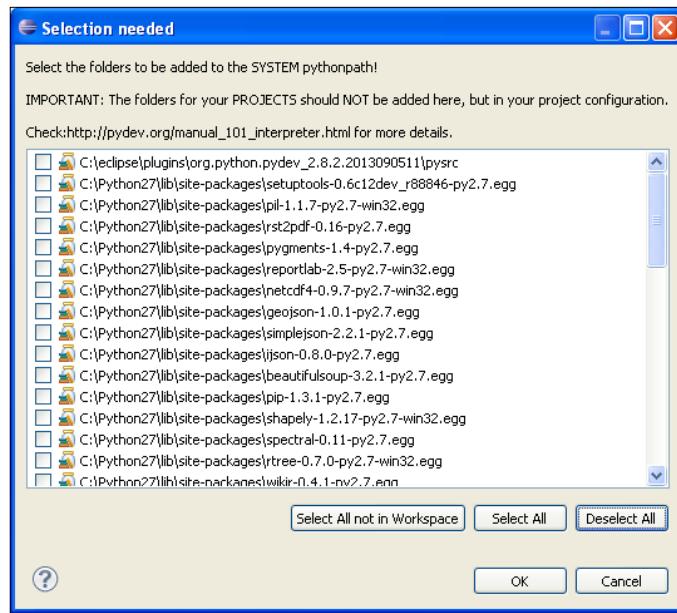
The process used to add the QGIS Python interpreter to Eclipse on Windows is different from the process used on Linux. The following steps describe how to set up the interpreter on the Windows version of Eclipse:

1. Open Eclipse.
2. From the **Window** menu, select **Preferences**. On OS X, you must click on the **Eclipse** menu to find the preferences menu.
3. In the pane on the left-hand side of the **Preferences** window, click on the plus sign next to **PyDev**.
4. From the list of PyDev preferences, select **Interpreter Python**.
5. In the pane labelled Python Interpreters, click on the **New** button.

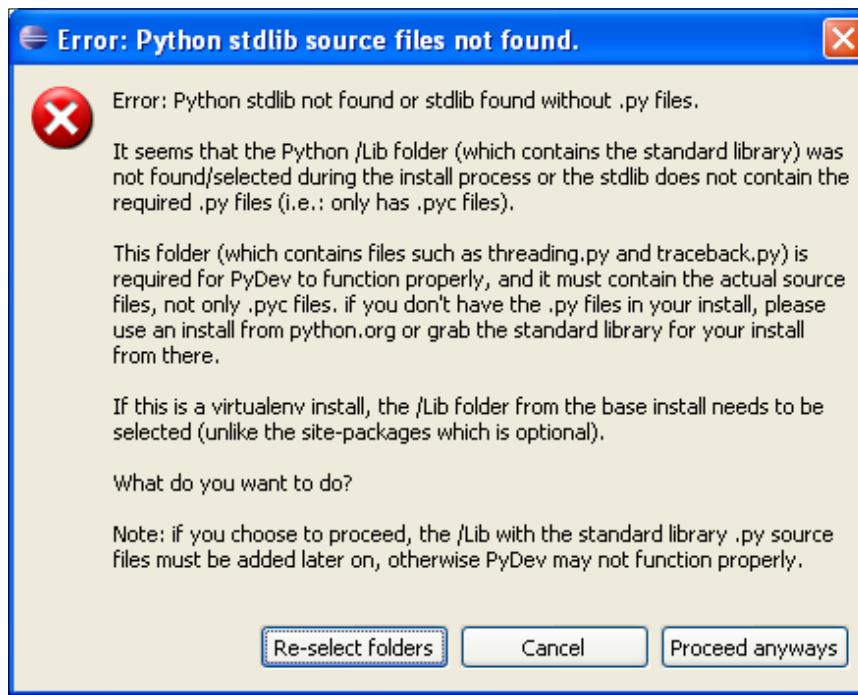
6. In the **Select interpreter** dialog, name the interpreter `PyQGIS`.
7. Browse to the location of the QGIS Python interpreter called `python.exe` within the `bin` folder of the QGIS program folder. On OS X and Linux, you can use the system Python installation. On Windows, Python is included with QGIS. The default location on Windows is `C:\Program Files\QGIS Brighton\bin\python.exe`, as shown in the following screenshot:



8. When you click on the **OK** button, Eclipse will attempt to automatically add every Python library it finds to the Python path for this interpreter configuration. We need to control which libraries are added to prevent conflicts. Click on the **Deselect All** button and then click on **OK**:



9. Eclipse will issue a warning dialog because you haven't selected any core libraries.
Click on the **Proceed anyways** button, as shown here:



Adding the PyQGIS module paths to the interpreter

Apart from adding the Python interpreter, you must also add the module paths needed by PyQGIS using the following steps. These steps will require you to switch back and forth between QGIS and Eclipse:

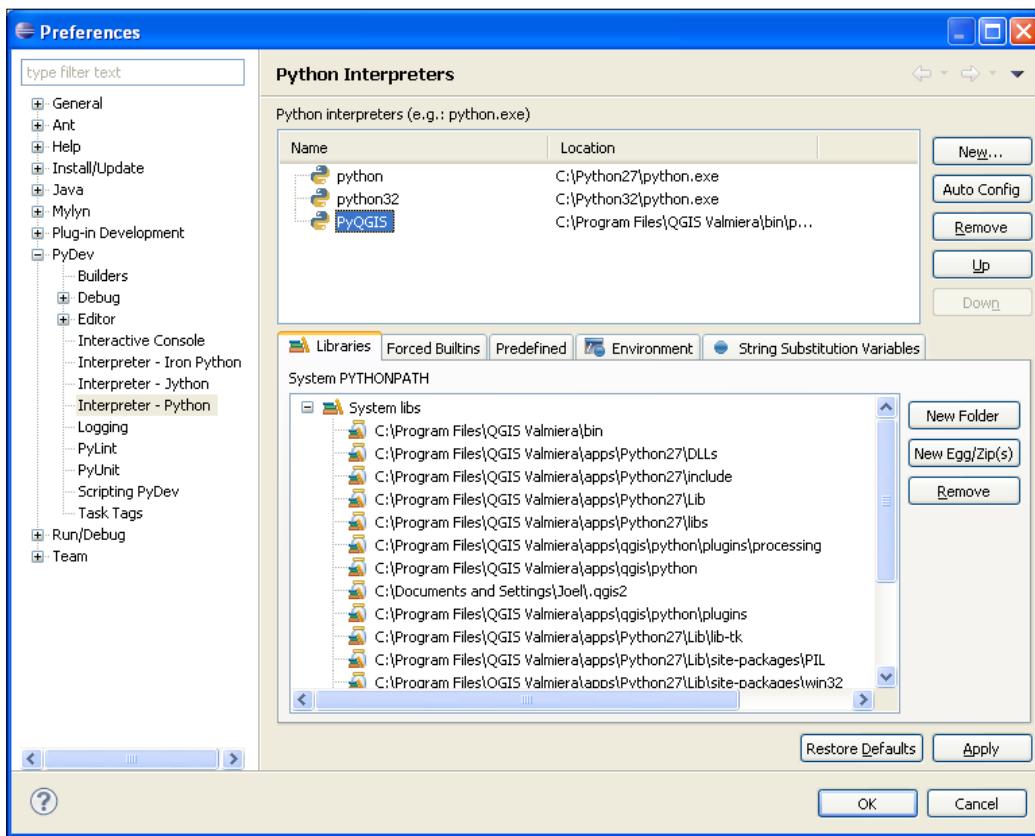
1. Start QGIS.
2. Start the QGIS **Python Console** from the **Plugins** menu.
3. Use the `sys` module to locate the PyQGIS Python path, as described in the previous recipe, *Setting the environment variables*:

```
import sys  
sys.path
```

4. We also want to add the PyQGIS API. Next, find that path using the QGIS **Python Console** by typing the following command:

```
qgis
```

5. For each path in the returned lists, click on the **New Folder** button in Eclipse's **Libraries** pane for your QGIS interpreter, and browse to that folder until all the paths have been added. If a given folder does not exist on your system, simply ignore it, as shown here:

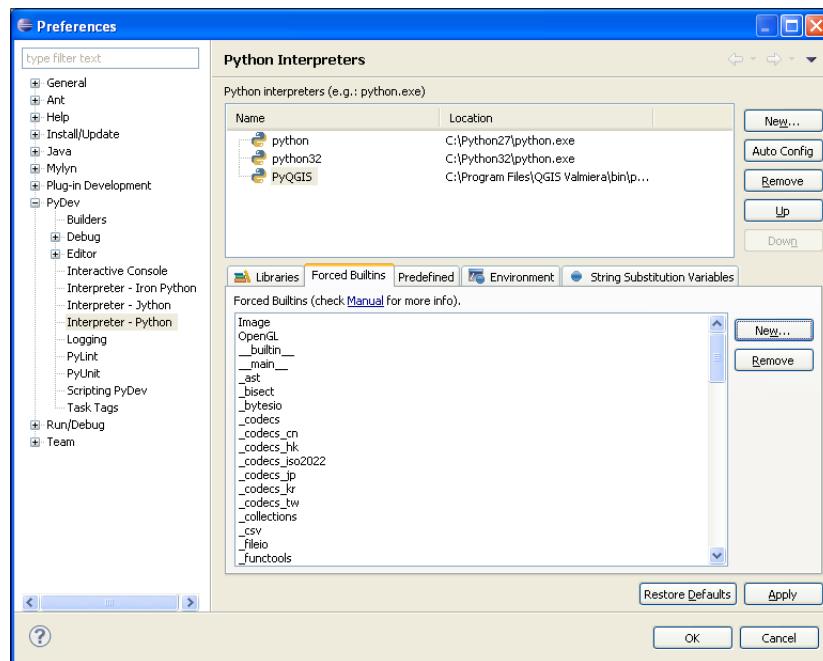


6. Click on the **OK** button in the **Preferences** dialog.

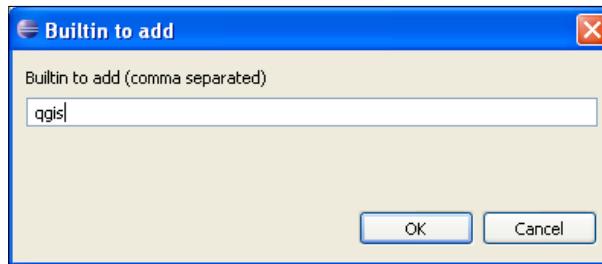
Adding the PyQGIS API to the IDE

To take full advantage of Eclipse's features, including code completion, we will add the QGIS and Qt4 modules to the PyQGIS Eclipse interpreter preferences. The following steps will allow Eclipse to suggest the possible methods and properties of QGIS objects as you type; this feature is known as **autocomplete**:

1. In the PyDev preferences for the PyQGIS Interpreter, select the **Forced Builtins** tab, as shown in the following screenshot:



2. Click on the **New** button.
3. In the **Builtin to add** dialog, type `qgis`:



4. Click on the **OK** button.

Adding environment variables

You will also need to create a `PATH` variable, which points to the QGIS binary libraries, DLLs on Windows, and other libraries needed by QGIS at runtime on all platforms.

1. In the **PyDev preferences** dialog, ensure that the **PyQGIS** interpreter is selected in the list of interpreters.
2. Select the **Environment** tab.
3. Click on the **New** button.

In the **Name** field, enter **PATH**.

1. For the **Value** field, add the path to the QGIS program directory and to any QGIS directories containing binaries separated by a semicolon. The following is an example from a Windows machine:

```
C:\Program Files\QGIS Brighton;C:\Program Files\QGIS Brighton\bin;C:\Program Files\QGIS Brighton\apps\qgis\bin;C:\Program Files\QGIS Brighton\apps\Python27\DLLs
```

How it works...

Eclipse and PyDev use only the information you provide to run a script in the Eclipse workspace. This approach is very similar to the popular Python tool **virtualenv**, which provides a clean environment when writing and debugging code to ensure that you don't waste time troubleshooting issues caused by the environment.

Debugging QGIS Python scripts

In this recipe, we will configure Eclipse to debug QGIS Python scripts.

How to do it...

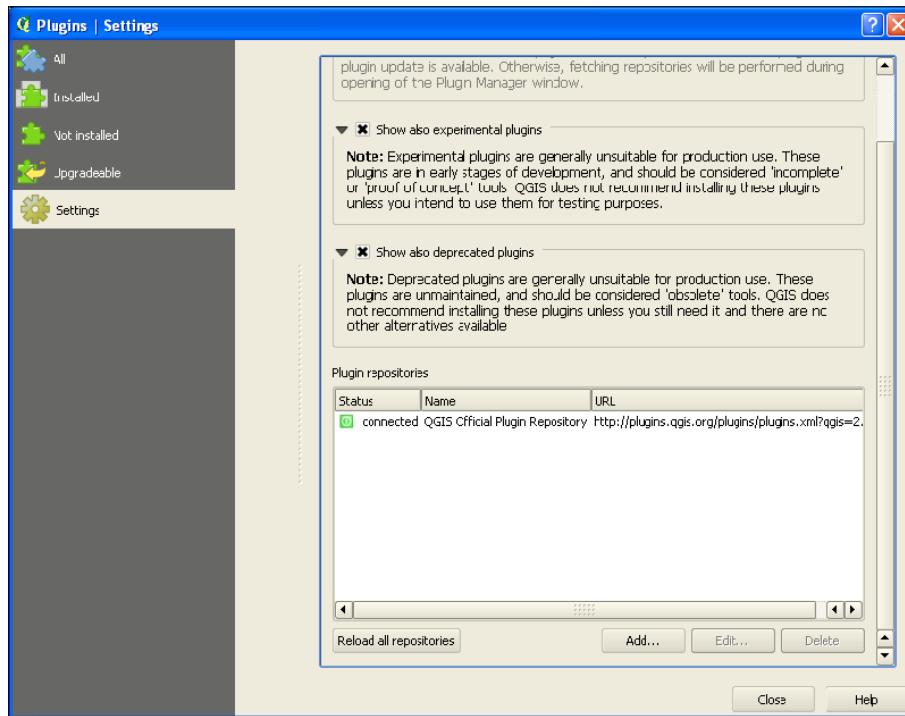
Both QGIS and Eclipse must be configured for debugging so that the two pieces of software can communicate. Eclipse **attaches** itself to QGIS in order to give you insights into the Python scripts running in QGIS. This approach allows you to run scripts in a controlled way that can pause execution while you monitor the program to catch bugs as they occur.

Configuring QGIS

The following steps will add two plugins to QGIS, which allows Eclipse to communicate with QGIS. One plugin, **Plugin Reloader**, allows you to reload a QGIS plugin into memory without restarting QGIS for faster testing. The second plugin, **Remote Debug**, connects QGIS to Eclipse.

Remote Debug is an experimental plugin, so you must ensure that experimental plugins are visible to the QGIS plugin manager in the list of available plugins.

1. Start QGIS.
2. Under the **Plugins** menu, select **Manage and Install Plugins...**
3. In the left pane of the **Plugins** dialog, select the **Settings** tab.
4. Scroll down in the **Settings** window and ensure that the **Show also experimental plugins** checkbox is checked, as shown in the following screenshot:



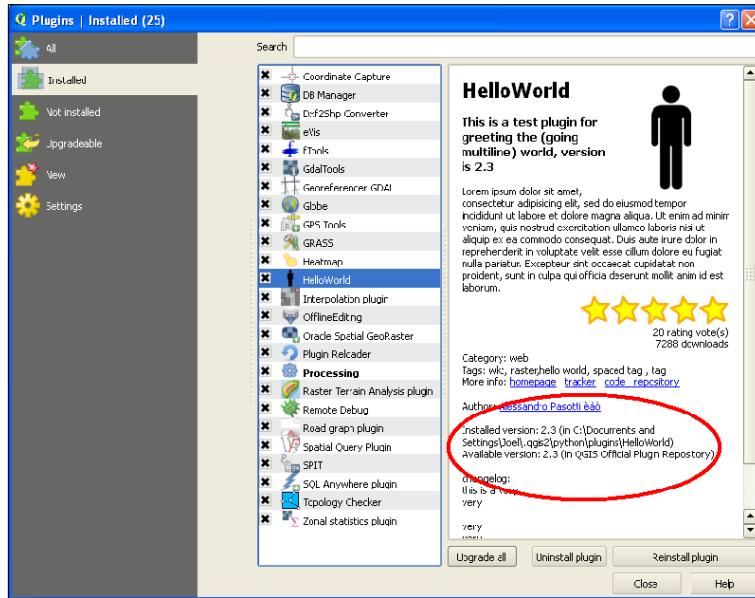
5. Click on the **OK** button.
6. Select the tab labeled **All** in the pane on the left-hand side of the **Plugins** window.
7. In the **Search** dialog at the top of the window, search for **Plugin Loader**.
8. Select **Plugin Loader** from the search results and then click on the **Install Plugin** button.
9. Next, search for the **Remote Debug** plugin and install it as well.
10. Finally, install the **HelloWorld** plugin as well.

Configuring Eclipse

Now that QGIS is configured for debugging in Eclipse, we will configure Eclipse to complete the debugging communication loop, as shown in the following steps:

1. Start Eclipse.
2. In the **File** menu, select **New** and then click on **Project**.
3. Select **General** and then click on **Project** from the **NewProject** dialog.
4. Click on the **Next>** button.
5. Give the project the name **HelloWorldPlugin**.
6. Click on the **Finish** button.
7. Select the new **HelloWorldPlugin** project in project explorer and select **New**; then, click on **Folder** from the **File** menu.
8. In the **New Folder** dialog, click on the **Advanced>>** button.
9. Choose the **Link to alternate location (Linked Folder)** radio button.
10. Click on the **Browse** button and browse to the location of the **HelloWorldPlugin** folder, as shown in the following screenshot:

[ You can find the location of the HelloWorld plugin from within the QGIS plugin manager.]

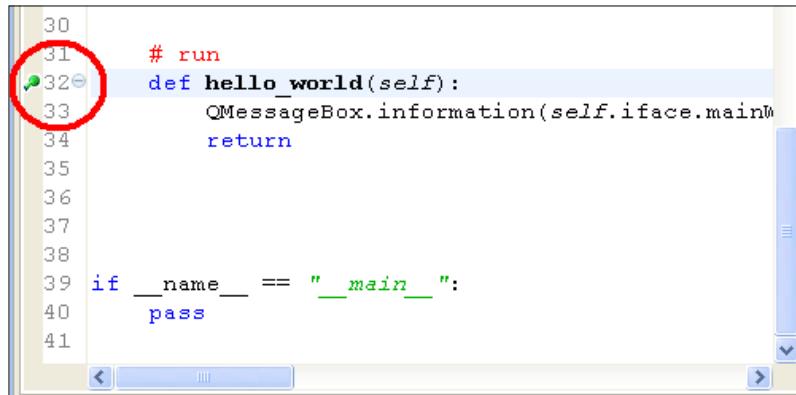


11. Click on the **Finish** button.

Testing the debugger

The previous parts of this recipe configured Eclipse and QGIS to work together in order to debug QGIS plugins. In this section, we will test the configuration using the simplest possible plugin, **HelloWorld**, to run Eclipse using the **Debug Perspective**. We will set up a break point in the plugin to pause the execution and then monitor plugin execution from within Eclipse, as follows:

1. Under the **HelloWorld** folder, open the file `HelloWorld.py`.
2. From the Eclipse **Window** menu, select **OpenPerspective** and then click on **Other...**
3. From the **OpenPerspective** dialog, select **Debug**.
4. Click on the **OK** button.
5. Scroll to the first line of the `hello_world()` function and double-click on the left-hand side of the line number to set a break point, which is displayed as a green-icon:



```

30
31
32 # run
33 def hello_world(self):
34     QMessageBox.information(self iface.mainWindow(), "Hello World", "Hello World")
35
36
37
38
39 if __name__ == "__main__":
40     pass
41

```

6. From the **Pydev** menu, select **Start Debug Server**.
7. Verify that the server is running by looking for a message in the Debug console at the bottom of the window, similar to the following:
`Debug Server at port: 5678`
8. Switch over to QGIS.
9. From the QGIS **Plugins** menu, select **RemoteDebug** and then select the **RemoteDebug** command.
10. Verify that the QGIS status bar in the lower-left corner of the window displays the following message:
`Python Debugging Active`

11. Now, select **HelloWorld** from the QGIS **Plugins** menu and then select **HelloWorld**.
12. Switch back to Eclipse.
13. Verify that the `hello_world()` function is highlighted at the break point.
14. From the **Run** menu, select **Resume**.
15. Switch back to QGIS.
16. Verify that the **HelloWorld** dialog box has appeared.

How it works...

The RemoteDebug plugin acts as a client to the PyDev debug server in order to send the Python script's execution status from QGIS to Eclipse. While it has been around for several versions of QGIS now, it is still considered experimental.

The PluginReloader plugin can reset plugins that maintain state as they run. The HelloWorld plugin is so simple that reloading is not needed to test it repeatedly. However, as you debug more complex plugins, you will need to run it in order to reset it before each test. This method is far more efficient and easier to use than closing QGIS, editing the plugin code, and then restarting.



You can find out more about debugging QGIS, including using other IDEs, at http://docs.qgis.org/2.6/en/docs/pyqgis_developer_cookbook/ide_debugging.html.

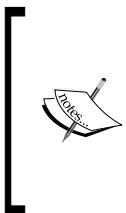


Navigating the PyQGIS API

The QGIS Python API, also known as PyQGIS, allows you to control virtually every aspect of QGIS. The ability to find the PyQGIS object you need in order to access a particular feature of QGIS is critical to automation.

Getting ready

The PyQGIS API is based on the QGIS C++ API. The C++ API is kept up to date online and is well-documented.



The QGIS API's web page is located at <http://qgis.org/api/2.6/modules.html>.

Notice the version number, 2.2, in the URL. You can change this version number to the version of QGIS you are using in order to find the appropriate documentation.



The PyQGIS API documentation is not updated frequently because it is nearly identical to the structure of the C++ API. However, the QGIS project on github.com maintains a list of all the PyQGIS classes for the latest version. The PyQGIS 2.6 API is located at https://github.com/qgis/QGIS/blob/master/python/qsci_apis/Python-2.6.api.

You can locate the documented class in the main C++ API and read about it. Then, look up the corresponding Python module and class using the PyQGIS API listing. In most cases, the C++ API name for a class is identical in Python.

In this recipe, we'll locate the PyQGIS class that controls labels in QGIS.

How to do it...

We will perform the following steps to see in which PyQGIS module the QGIS Label object and `QgsLabel` are located in:

1. Go to the QGIS API page at <http://qgis.org/api/2.6/index.html>.
2. Click on the **Modules** tab.
3. Click on the link **QGIS Core Library**.
4. Scroll down the list of modules in alphabetical order until you see **QgsLabel**.
5. Click on the **QgsLabel** link to access the label object documentation.
6. Now, go to the PyQGIS API listing at https://github.com/qgis/QGIS/blob/master/python/qsci_apis/Python-2.6.api.
7. Scroll down the alphabetical class listing until you see `qgis.core.QgsLabel.LabelField`.

How it works...

The QGIS API is divided into five distinct categories, as follows:

- ▶ Core
- ▶ GUI
- ▶ Analysis
- ▶ Map composer
- ▶ Network analysis

Most of the time, it's easy to find the class that targets the functionality you need with most of QGIS being contained in the catch-all **Core** module. The more you use the API, the quicker you'll be able to locate the objects you need for your scripts.

There's more...

If you're having trouble locating a class containing the keyword you need, you can use the search engine on the QGIS API website.



Beware, however, that the results returned by this search engine may contain items you don't need and can even send you looking in the wrong direction because of the similar keywords in different modules.

Creating a QGIS plugin

Plugins are the best way to extend QGIS, as they can be easily updated and reused by other people.

Getting ready

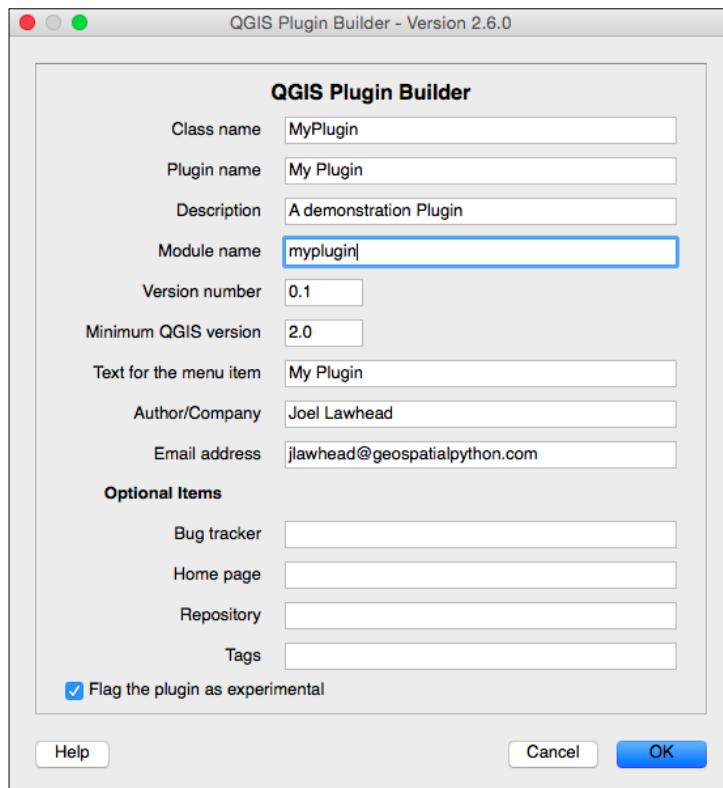
The easiest approach to creating a plugin is to use the **Plugin Builder** plugin to jumpstart development. You can find it in the main QGIS plugin repository and install it.

How to do it...

Perform the following steps to create a simple plugin that displays a dialog box with a custom message:

1. Start QGIS.
2. From the **Plugins** menu, select **Plugin Builder** and then click on **Plugin Builder** under the submenu.
3. In the **QGIS Plugin Builder** dialog, name the class `MyPlugin`.
4. Name the plugin `My Plugin`.
5. Type a short description, such as `A demonstration on building a QGIS Plugin.`
6. Enter `myplugin` for the **Module** name.
7. Leave the default version numbers as they are.
8. Enter `My Plugin` in the **Text for the menu item** field.
9. Enter your name and email address for author information.

10. Ensure that the checkbox labelled **Flag the plugin as experimental** is checked, as shown in the following screenshot:



11. Click on the **OK** button.
12. A file browser dialog will appear; you can choose a folder in which you want to create your plugin. Select one of the folders called `plugins` within the `python` folder in either the main user directory or the QGIS program directory. The following examples are from a Windows machine. You should use the folder in your user directory, which is the preferred place for third-party plugins. QGIS standard plugins go in the main program directory:
`C:\Documents and Settings\Joel\.qgis2\python\plugins`
`C:\Program Files\QGIS Brighton\apps\qgis\python\plugins`
13. Close the follow-on **Plugin Builder** information dialog by clicking on the **OK** button.
14. Using the command prompt, navigate to your new plugin template folder.
15. Use the `pyrcc4` command to compile the resource file:
`pyrcc4 -o resources_rc.py resources.qrc`



If you are on Windows, it is important to use the OSGEO4W shell, which is installed along with QGIS, for the Qt compilation tools to work properly.

16. In a text editor, such as Windows Notepad or vi on Linux, open the user interface XML file named `myplugin_dialog_base.ui`.

17. Insert the following XML for a custom label near line 31 and just before the last `</widget>` tag. Save the file after this edit:

```
<widget class=" QLabel" name="label">
<property name="geometry">
<rect>
<x>120</x>
<y>80</y>
<width>201</width>
<height>20</height>
</rect>
</property>
<property name="font">
<font>
<pointsize>14</pointsize>
</font>
</property>
<property name="text">
<string>Geospatial Python Rocks!</string>
</property>
</widget>
```

18. Now, compile the `ui` file using the `pyuic4` tool:

```
pyuic4 -o ui_myplugin.py ui_myplugin.ui
```

19. Your plugin is now ready. Restart QGIS.

20. Select **My Plugin** from the **Plugins** menu and then select **My Plugin** from the submenu to see the dialog you created within QGIS, as shown here:



How it works...

This recipe shows you the bare bones needed to make a working plugin. Although we haven't altered it, the code for the plugin's behavior is contained in `myplugin.py`. You can change the icon and the GUI, and just recompile any time you want. Note that we must compile the Qt4 portion of the plugin, which creates the dialog box. The entire QGIS GUI is built on the Qt4 library, so the `pyrcc4` compiler and `pyuic4` is included to compile the GUI widgets.

You can download the completed plugin with both the source and compiled ui and resource files at <https://geospatialpython.googlecode.com/svn/MyPlugin.zip>.



You can find out more about QGIS plugins, including the purpose of the other files in the directory, in the QGIS documentation at http://docs.qgis.org/testing/en/docs/pyqgis_developer_cookbook/plugins.html.

There's more...

We have edited the `myplugin_dialog_base.ui` XML file by hand to make a small change. However, there is a better way to use Qt Creator. Qt Creator is a fully-fledged, open source GUI designer for the Qt framework. It is an easy what-you-see-is-what-you-get editor for Qt Widgets, including PyQGIS plugins, which uses the included Qt Designer interface. On Windows, Qt Designer can be found in the QGIS program directory within the `bin` directory. It is named `designer.exe`. On other platforms, Qt Designer is included as part of the `qt4-devel` package.



You can also download Qt Creator, which includes Qt Designer, from <http://qt-project.org/downloads>.

When you run the installer, you can uncheck all the installation options, except the **Tools** category to install just the IDE.

Distributing a plugin

Distributing a QGIS plugin means placing the collection of files on a server as a `ZIP` file, with a special configuration file, in order to allow the QGIS plugin manager to locate and install the plugin. The QGIS project has an official repository, but third-party repositories are also permitted. The official repository is very strict regarding how the plugin is uploaded. So, for this recipe, we'll set up a simple third-party repository for a sample plugin and test it with the QGIS plugin manager to avoid polluting the main QGIS repository with a test project.

Getting ready

In order to complete this recipe, you'll need a sample plugin and a web-accessible directory. You'll also need a `zip` tool such as the free 7-zip program (<http://www.7-zip.org/download.html>). You can use the `MyPlugin` example from the *Creating a QGIS plugin* recipe as the plugin to distribute. For a web directory, you can use a Google Code repository, GitHub repository, or an other online directory you can access. Code repositories work well because they are a good place to store a plugin that you are developing.

How to do it...

In the following steps, we will package our plugin, create a server configuration file for it, and place it on a server to create a QGIS plugin repository:

1. First, zip up the plugin directory to create a `.ZIP` file.
2. Rename the `.ZIP` file to contain the plugin's version number:
`Myplugin.0.1.0.zip`
3. Upload this file to a publicly accessible web directory.
4. Upload the `icon.png` file from your plugin directory to the web directory.
5. Next, customize a `plugins.xml` metadata file for your plugin. Most of the data you need can be found in the `metatdata.txt` file in your plugin directory.

The following example provides some guidance:

```
<?xml version = '1.0' encoding = 'UTF-8'?>
<?xml-stylesheet type="text/xsl" href="" ?>
<plugins>
<pyqgis_plugin name="My Plugin"
    version="0.1.0"
    plugin_id="227">
<description>
<! [CDATA[Demonstration of a QGIS Plugin]]>
</description>
<about></about>
<version>0.1.0</version>
<qgis_minimum_version>1.8.0</qgis_minimum_version>
<qgis_maximum_version>2.9.9</qgis_maximum_version>
<homepage>
<! [CDATA[https://code.google.com/p/geospatialpython]]>
</homepage>
<file_name>MyPlugin.0.1.0.zip</file_name>
<icon>
http://geospatialpython.googlecode.com/svn/icon_227.png
</icon>
<author_name><! [CDATA[Joel Lawhead]]></author_name>
```

```
<download_url> http://geospatialpython.googlecode.com/svn/
MyPlugin.0.1.0.zip
</download_url>
<uploaded_by><! [CDATA[jll]]></uploaded_by>
<create_date>2014-05-16T15:31:19.824333</create_date>
<update_date>2014-07-15T15:31:19.824333</update_date>
<experimental>True</experimental>
<deprecated>False</deprecated>
<tracker>
<! [CDATA[http://code.google.com/p/geospatialpython/issues]]>
</tracker>
<repository>
<! [CDATA[https://geospatialpython.googlecode.com/svn/]]>
</repository>
<tags>
<! [CDATA[development,debugging,tools]]></tags>
<downloads>0</downloads>
<average_vote>0</average_vote>
<rating_votes>0</rating_votes>
</pyqgis_plugin>
</plugins>
```

6. Upload the `plugins.xml` file to your web directory.
7. Now, start QGIS and launch the plugins manager by going to the **Plugins** menu and selecting **Manage and Install Plugins...**
8. In the **Settings** tab of the **plugins settings** dialog, scroll down and click on the **Add...** button.
9. Give the plugin a name and then add the complete URL to your `plugins.xml` in the URL field.
10. Click on the **OK** button.
11. To make things easier, disable the other repositories by selecting the repository name, clicking on the **Edit** button, and unchecking the **Enable** checkbox.
12. Click on the **OK** button.
13. Click on the **Not Installed** tab.
14. Your test plugin should be the only plugin listed, so select it from the list.
15. Click on the **Install Plugin** button in the bottom-right corner of the window.
16. Click on the **Close** button.
17. Go to the **Plugins** menu and select your plugin to ensure that it works.

How it works...

The QGIS repository concept is simple and effective. The `plugins.xml` file contains a `download_url` tag that points to a ZIP file plugin on the same server or on a different server. The `name` attribute of the `pyqgis_plugin` tag is what appears in the QGIS plugin manager.

Creating a standalone application

QGIS is a complete desktop GIS application. However, with PyQGIS, it can also be a comprehensive geospatial Python library to build standalone applications. In this recipe, we will build a simple standalone script that creates a map with a line on it.

Getting ready

All you need to do to get ready is ensure that you have configured Eclipse and PyDev for PyQGIS development, as described in the *Setting up your QGIS IDE* recipe.

How to do it...

In PyDev, create a new project called `MyMap` with a Python script called `MyMap.py`, as follows:

1. In the Eclipse **File** menu, select **New** and then click on **PyDev Project**.
2. In the PyDev project's **Name** field, enter `MyMap`.
3. Next, select the **Python** radio button from the **Project Type** list.
4. From the **Interpreter** pull-down menu, select **PyQGIS**.
5. Leave the radio button checked for **Add project directory to the PYTHONPATH**.
6. Click on the **Finish** button.
7. Now, select the project in the PyDev package explorer.
8. From the **File** menu, select **New** and then click on **File**.
9. Name the file `myMap.py`.
10. Click on the **Finish** button.
11. Add the following code to the file that is open in the editor:

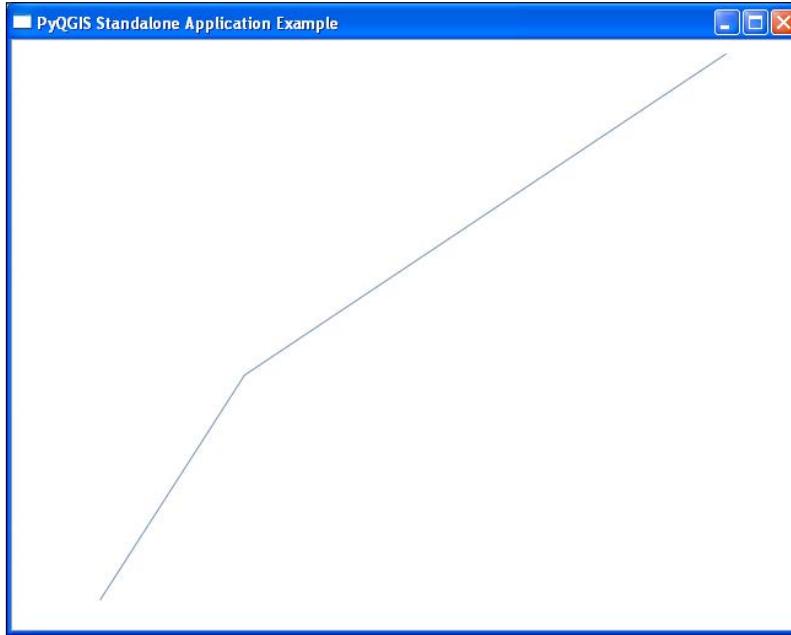
```
from qgis.core import *
from qgis.gui import *
from qgis.utils import *
from PyQt4.QtCore import *
from PyQt4.QtGui import *

app = QgsApplication([], True)
app.setPrefixPath("C:/Program Files/QGIS Brighton/apps/qgis",
True)
app.initQgis()
canvas = QgsMapCanvas()
canvas.setWindowTitle("PyQGIS Standalone Application Example")
```

```
canvas.setCanvasColor(Qt.white)
layer = QgsVectorLayer('LineString?crs=epsg:4326', 'MyLine',
"memory")
pr = layer.dataProvider()
linstr = QgsFeature()
geom = QgsGeometry.fromWkt("LINESTRING (1 1, 10 15, 40 35)")
linstr.setGeometry(geom)
pr.addFeatures([linstr])
layer.updateExtents()
QgsMapLayerRegistry.instance().addMapLayer(layer)
canvas.setExtent(layer.extent())
canvas.setLayerSet([QgsMapCanvasLayer(layer)])
canvas.zoomToFullExtent()
canvas.freeze(True)
canvas.show()
canvas.refresh()
canvas.freeze(False)
canvas.repaint()
exitcode = app._exec()
QgsApplication.exitQgis()
sys.exit(exitcode)
```

12. From the **Run** menu, select **Run**.

13. Verify that the standalone QGIS map appears in a new window, as shown here:



How it works...

This recipe uses as little code as possible to create a map canvas and to draw a line in order to demonstrate the skeleton of a standalone application, which can be built up further to add more functionality.

To create the line geometry, we use **Well-Known Text (WKT)**, which provides a simple way to define the line vertices without creating a bunch of objects. Towards the end of this code, we use a workaround for a bug in QGIS 2.2 by **freezing** the canvas. When the canvas is frozen, it does not respond to any events which, in the case of this bug, prevent the canvas from updating. Once we refresh the canvas, we unfreeze it and then repaint it to draw the line. This workaround will still work in QGIS 2.4 and 2.6 but is not necessary.

There's more...

The standalone application can be compiled into an executable that can be distributed without installing QGIS, using py2exe or PyInstaller:

You can find our more about py2exe at <http://www.py2exe.org>.

You can learn more about PyInstaller at <https://github.com/pyinstaller/pyinstaller/wiki>.

Storing and reading global preferences

PyQGIS allows you to store application-level preferences and retrieve them.

Getting ready

This code can be run in any type of PyQGIS application. In this example, we'll run it in the QGIS Python console for an easy demonstration. In this example, we'll change the default CRS for new projects and then read the value back from the global settings.

How to do it...

In this recipe, we will set the default projection used by QGIS for new projects using the Python console:

1. Start QGIS.
2. From the **Plugins** menu, select **Python Console**.

3. We will need to import the Qt core library, as follows:

```
from PyQt4.QtCore import *
```

4. In the Python console, run the following code:

```
settings = QSettings(QSettings.NativeFormat, QSettings.UserScope,
'QuantumGIS', 'QGis')
settings.setValue('/Projections/projectDefaultCrs', 'EPSG:2278')
settings.value('/Projections/projectDefaultCrs')
```

How it works...

This API is actually the Qt API that QGIS relies on for settings. In the `QSettings` object, we specify the `NativeFormat` for storage, which is the default format for the platform. On Windows, the format is the registry; on OS X, it's the `plist` files; and on Unix, it's the text files. The other `QSettings` parameters are the **organization** and the **application**, often used as a hierarchy to store information. Note that even after changing these settings, it may be that none of the properties in the QGIS GUI change immediately. In some cases, such as Windows, the system must be restarted for registry changes to take effect. However, everything will work programmatically.

There's more...

If you want to see all the options that you can change, call the `allKeys()` method of `QSettings`; this will return a list of all the setting names.

Storing and reading project preferences

The QGIS application settings are stored using the Qt API. However, QGIS project settings have their own object. In this recipe, we'll set and read the project title and then set and read a custom preference for a plugin.

Getting ready

We are going to set a plugin preference using the sample plugin created in the previous recipe, *Creating a QGIS plugin*. You can substitute the name of any plugin you want, however. We will also run this recipe in the QGIS Python console for quick testing, but this code will normally be used in a plugin.

How to do it...

In this recipe, we will first write and then read the title of the current project. Then, we will create a custom value for a plugin called `splash`, which can be used for the plugin startup splash screen if desired.

1. Start QGIS.
2. From the **Plugins** menu, select **Python Console**.
3. In the console, run the following code:

```
proj = QgsProject.instance()
proj.setTitle("My QGIS Project")
proj.title()
proj.writeEntry("MyPlugin", "splash", "Geospatial Python Rocks!")
proj.readEntry("MyPlugin", "splash", "Welcome!") [0]
```

How it works...

In the first two lines, we change the title of the current active project and then echo it back. In the next set of two lines, we set up and read custom settings for a plugin. Notice that the `readEntry()` method returns a tuple with the desired text and a boolean, acknowledging that the value is set. So, we extract the first index to get the text. The `read` method also allows the default text in case that property is not set (rather than throw an exception which must be handled) as well as the boolean value `False` to inform you that the default text was used because the property was not set. The values you set using this method are stored in the project's XML file when you save it.

There's more...

The `QgsProject` object has a number of methods and properties that may be useful. The QGIS API documentation details all of them at <http://qgis.org/api/2.6/classQgsProject.html>.

Accessing the script path from within your script

Sometimes, you need to know exactly where the current working directory is so that you can access external resources.

Getting ready

This code uses the Python built-in library and can be used in any context. We will run this recipe in the QGIS Python console.

How to do it...

In this recipe, we will get the current working directory of the Python console, which can change with configuration:

1. Start QGIS.
2. From the **Plugins** menu, select **Python Console**.
3. In the Python console, run the following code:

```
import os  
os.getcwd()
```

How it works...

QGIS relies heavily on file system paths to run the application and to manage external data. When writing cross-platform QGIS code, you cannot assume the working directory of your script.

There's more...

On his blog, one of the QGIS developers has an excellent post about the various aspects of path variables in QGIS beyond just the execution directory; you can check it out at <http://spatialgalaxy.net/2013/11/06/getting-paths-with-pyqgis/>.

2

Querying Vector Data

In this chapter, we will cover the following recipes:

- ▶ Loading a vector layer from a file
- ▶ Loading a vector layer from a geodatabase
- ▶ Examining vector layer features
- ▶ Examining vector layer attributes
- ▶ Filtering a layer by geometry
- ▶ Filtering a layer by attributes
- ▶ Buffering a feature
- ▶ Measuring the distance between two points
- ▶ Measuring the distance along a line
- ▶ Calculating the area of a polygon
- ▶ Creating a spatial index
- ▶ Calculating the bearing of a line

Introduction

This chapter demonstrates how to work with vector data through Python in QGIS. We will first work through loading different sources of vector data. Next, we'll move on to examining the contents of the data. Then, we'll spend the remainder of the chapter performing spatial and database operations on vector data.

Loading a vector layer from a file sample

This recipe describes the most common type of data used in QGIS, a file. In most cases, you'll start a QGIS project by loading a shapefile.

Getting ready

For ease of following the examples in this book, it is recommended that you create a directory called `qgis_data` in your root or user directory, which provides a short pathname. This setup will help prevent the occurrence of any frustrating errors resulting from path-related issues on a given system. In this recipe and others, we'll use a point shapefile of New York City museums, which you can download from https://geospatialpython.googlecode.com/svn/NYC_MUSEUMS_GEO.zip.

Unzip this file and place the shapfile's contents in a directory named `nyc` within your `qgis_data` directory.

How to do it...

Now, we'll walk through the steps of loading a shapefile and adding it to the map, as follows:

1. Start QGIS.
2. From the **Plugins** menu, select **Python Console**.
3. In the Python console, create the layer:

```
layer = QgsVectorLayer("/qgis_data/nyc/NYC_MUSEUMS_GEO.shp",
"New York City Museums", "ogr")
```

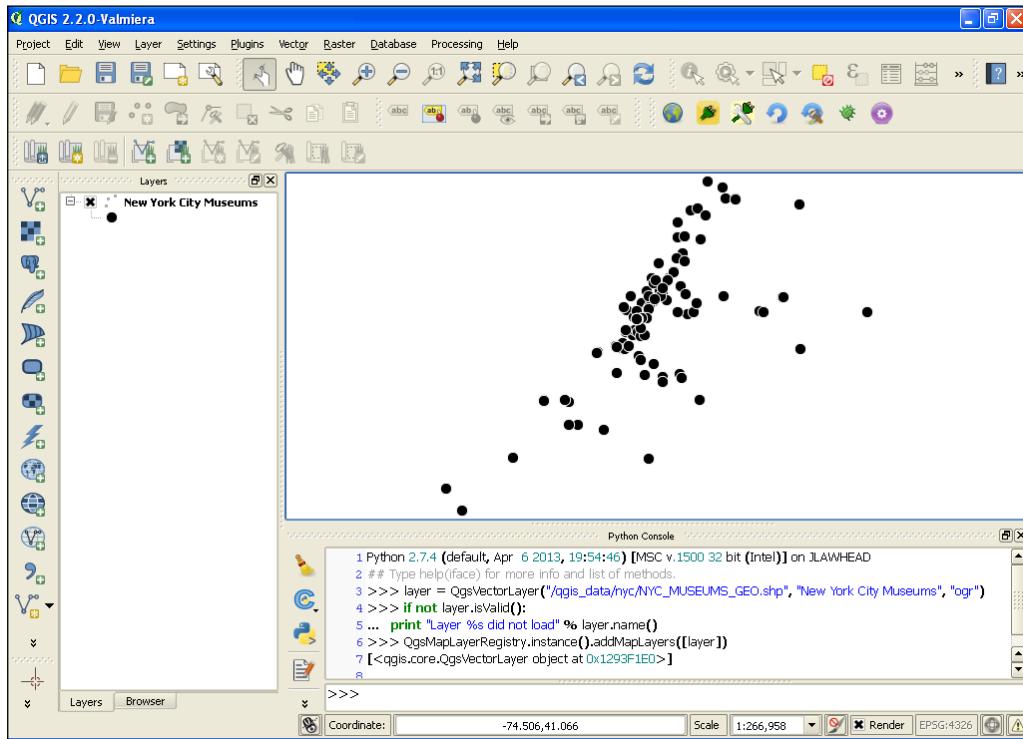
4. Next, ensure that the layer was created as expected:

```
if not layer.isValid():
    print "Layer %s did not load" % layer.name()
```

5. Finally, add the layer to the layer registry:

```
QgsMapLayerRegistry.instance().addMapLayers([layer])
```

Verify that your QGIS map looks similar to the following image:



How it works...

The `QgsVectorLayer` object requires the location of the file, a name for the layer in QGIS, and a data provider that provides the right parser and capabilities managed for the file format. Most vector layers are covered by the `ogr` data provider, which attempts to guess the format from the file name extension in order to use the appropriate driver. The formats available with this data provider are listed at http://www.gdal.org/ogr_formats.html.

Once we have created the `QgsVector` object, we do a quick check using the `layer.isValid()` method to see whether the file was loaded properly. We won't use this method in every recipe to keep the code short, but this method is often very important. It's usually the only indication that something has gone wrong. If you have a typo in the filename or you try to connect to an online data source but have no network connection, you won't see any errors. Your first indication will be another method failing further into your code, which will make tracking down the root cause more difficult.

In the last line, we add the vector layer to the `QgsMapLayerRegistry`, which makes it available on the map. The registry keeps track of all the layers in the project. The reason why QGIS works this way is so you can load multiple layers, style them, filter them, and do other operations before exposing them to the user on the map.

Loading a vector layer from a spatial database

The PostGIS geodatabase is based on the open source Postgres database. The geodatabase provides powerful geospatial data management and operations. PyQGIS fully supports PostGIS as a data source. In this recipe, we'll add a layer from a PostGIS database.

Getting ready

Installing and configuring PostGIS is beyond the scope of this book, so we'll use a sample geospatial database interface from the excellent service www.QGISCloud.com. www.QGISCloud.com has its own Python plugin called **QGIS Cloud**. You can sign up for free and create your own geodatabase online by following the site's instructions, or you can use the example used in the recipe.

How to do it...

Perform the following steps to load a PostGIS layer into a QGIS map:

1. First, create a new `DataSourceURI` instance:

```
uri = QgsDataSourceURI()
```

2. Next, create the database connection string:

```
uri.setConnection("spacialdb.com", "9999", "lzmjzm_hwpqlf",
"lzmjzm_hwpqlf", "0e9fcc39")
```

3. Now, describe the data source:

```
uri.setDataSource("public", "islands", "wkb_geometry", "")
```

4. Then, create the layer:

```
layer = QgsVectorLayer(uri.uri(), "Islands", "postgres")
```

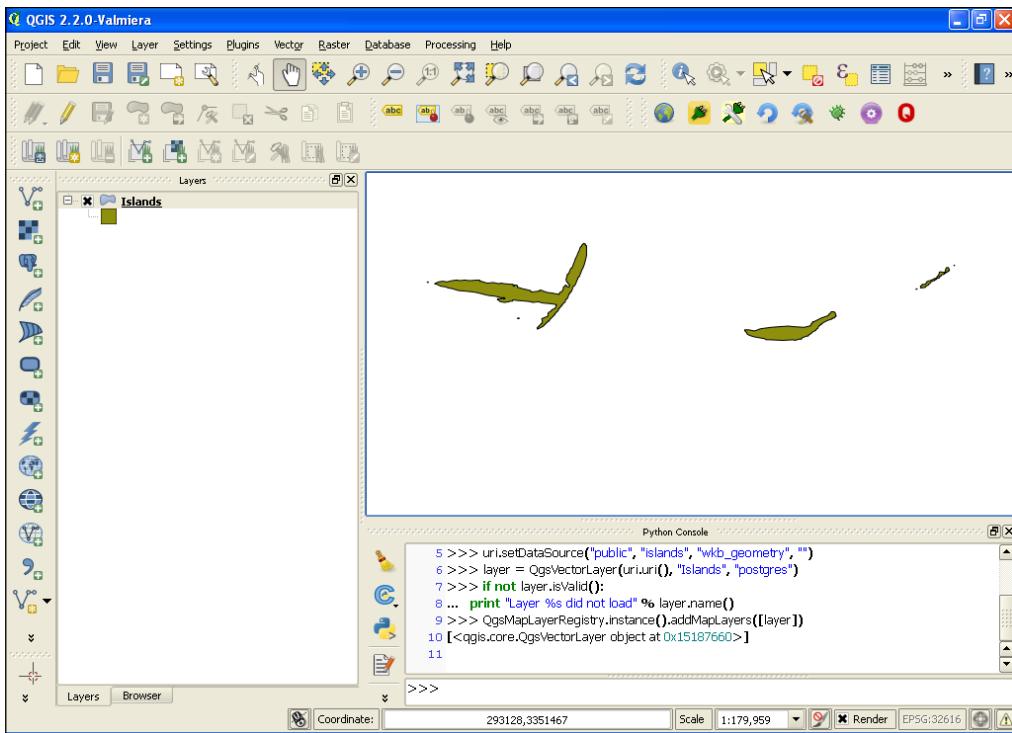
- Just to be safe, make sure everything works:

```
if not layer.isValid():
    print "Layer %s did not load" % layer.name()
```

- Finally, add the layer to the map if everything is okay:

```
QgsMapLayerRegistry.instance().addMapLayers([layer])
```

You can see the `islands` layer in the map, as shown in the following screenshot:



How it works...

PyQGIS provides an object in the API to create a PostGIS data source in `QgsDataSourceURI()`. The connection string parameters in the second line of code are the database server, port, database name, user, and password. In the example, the database, username, and password are randomly generated unique names. The data source parameters are the schema name, table name, geometry column, and an optional SQL WHERE to subset the layer as needed.

Examining vector layer features

Once a vector layer is loaded, you may want to investigate the data. In this recipe, we'll load a vector point layer from a shapefile and take a look at the x and y values of the first point.

Getting ready

We'll use the same New York City Museums layer from *Loading a vector layer from a file* recipe in this chapter. You can download the layer from https://geospatialpython.googlecode.com/svn/NYC_MUSEUMS_GEO.zip.

Unzip that file and place the shapefile's contents in a directory named `nyc` within your `qgis_data` directory, within your root or home directory.

How to do it...

In this recipe, we will load the layer, get the features, grab the first feature, obtain its geometry, and take a look at the values for the first point:

1. First, load the layer:

```
layer =  
QgsVectorLayer("/qgis_data/nyc/NYC_MUSEUMS_GEO.shp", "New  
York City Museums", "ogr")
```

2. Next, get an iterator of the layer's features:

```
features = layer.getFeatures()
```

3. Now, get the first feature from the iterator:

```
f = features.next()
```

4. Then, get the feature's geometry:

```
g = f.geometry()
```

5. Finally, get the point's values:

```
g.asPoint()
```

6. Verify that the Python console output is similar to the following `QgsPoint` object:

```
(-74.0138, 40.7038)
```

How it works...

When you access a layer's features or geometry using the previously demonstrated methods, PyQGIS returns a Python iterator. The iterator data structure allows Python to work efficiently with very large data sets without keeping the entire dataset in memory.

Examining vector layer attributes

A true GIS layer contains both spatial geometry and database attributes. In this recipe, we'll access a vector point layer's attributes in PyQGIS. We'll use a file-based layer from a shapefile, but once a layer is loaded in QGIS, every vector layer works the same way.

Getting ready

Once again, we'll use the same New York City Museums layer from the *Loading a vector layer from a file* recipe in this chapter. You can download the layer from https://geospatialpython.googlecode.com/svn/NYC_MUSEUMS_GEO.zip.

Unzip that file and place the shapefile's contents in a directory named `nyc` within your `qgis_data` directory, within your root or home directory.

How to do it...

In the following steps, we'll load the layer, access the `features` iterator, grab the first feature, and then view the attributes as a Python list:

1. First, load the shapefile as a vector layer:

```
layer =  
QgsVectorLayer("/qgis_data/nyc/NYC_MUSEUMS_GEO.shp", "New  
York City Museums", "ogr")
```

2. Next, get the features iterator:

```
features = layer.getFeatures()
```

3. Now, grab the first feature from the iterator:

```
f = features.next()
```

4. Finally, examine the attributes as a Python list:

```
f.attributes()
```

5. Verify that the Python console's output resembles the following list:

```
[u'Alexander Hamilton U.S. Custom House', u'(212) 514-3700',  
u'http://www.oldnycustomhouse.gov/', u'1 Bowling Grn', NULL, u'New  
York', 10004.0, -74.013756, 40.703817]
```

How it works...

Examining attributes is consistent with accessing the point values of a layer's geometry. Note that all string attribute values are returned as unicode strings, which is the case for all QGIS strings. Unicode allows the internationalization (that is, translation) of QGIS for other languages besides English.

There's more...

The attribute values don't mean much without the knowledge of what those values represent. You will also need to know the fields. You can get the fields as a list by accessing the `fields` iterator and calling the `name()` method for each field. This operation is easily accomplished with a Python list comprehension:

```
[c.name() for c in f.fields().toList()]
```

This example returns the following result:

```
[u'NAME', u'TEL', u'URL', u'ADDRESS1', u'ADDRESS2', u'CITY', u'ZIP',
u'XCOORD', u'YCOORD']
```

Filtering a layer by geometry

In this recipe, we'll perform a spatial operation to select a subset of a point layer based on the points contained in an overlapping polygon layer. We'll use shapefiles in both cases, with one being a point layer and the other a polygon. This kind of subset is one of the most common GIS operations.

Getting ready

We will need two new shapefiles that have not been used in previous recipes. You can download the point layer from https://geospatialpython.googlecode.com/files/MSCities_Geo_Pts.zip.

Similarly, you can download the geometry layer from https://geospatialpython.googlecode.com/files/GIS_CensusTract.zip.

Unzip these shapefiles and place them in a directory named `ms` within your `qgis_data` directory, within your root or home directory.

How to do it...

In this recipe, we will perform several steps to select features in the point layer that fall within the polygon layer, as follows:

1. First, load the point layer:

```
lyrPts = QgsVectorLayer("/qgis_data/ms/MSCities_Geo_Pts.shp",
    "MSCities_Geo_Pts", "ogr")
```

2. Next, load the polygon layer:

```
lyrPoly = QgsVectorLayer("/qgis_data/ms/GIS_CensusTract_poly.shp",
    "GIS_CensusTract_poly", "ogr")
```

3. Add the layers to the map using a list:

```
QgsMapLayerRegistry.instance().addMapLayers([lyrPts, lyrPoly])
```

4. Access the polygon layer's features:

```
ftsPoly = lyrPoly.getFeatures()
```

5. Now, iterate through the polygon's features:

```
for feat in ftsPoly:
```

6. Grab each feature's geometry:

```
geomPoly = feat.geometry()
```

7. Access the point features and filter the point features by the polygon's bounding box:

```
featsPnt = lyrPts.getFeatures(QgsFeatureRequest().
    setFilterRect(geomPoly.boundingBox()))
```

8. Iterate through each point and check whether it's within the polygon itself:

```
for featPnt in featsPnt:
    if featPnt.geometry().within(geomPoly):
```

9. If the polygon contains the point, print the point's ID and select the point:

```
print featPnt.id()
lyrPts.select(featPnt.id())
```

10. Now, set the polygon layer as the active map layer:

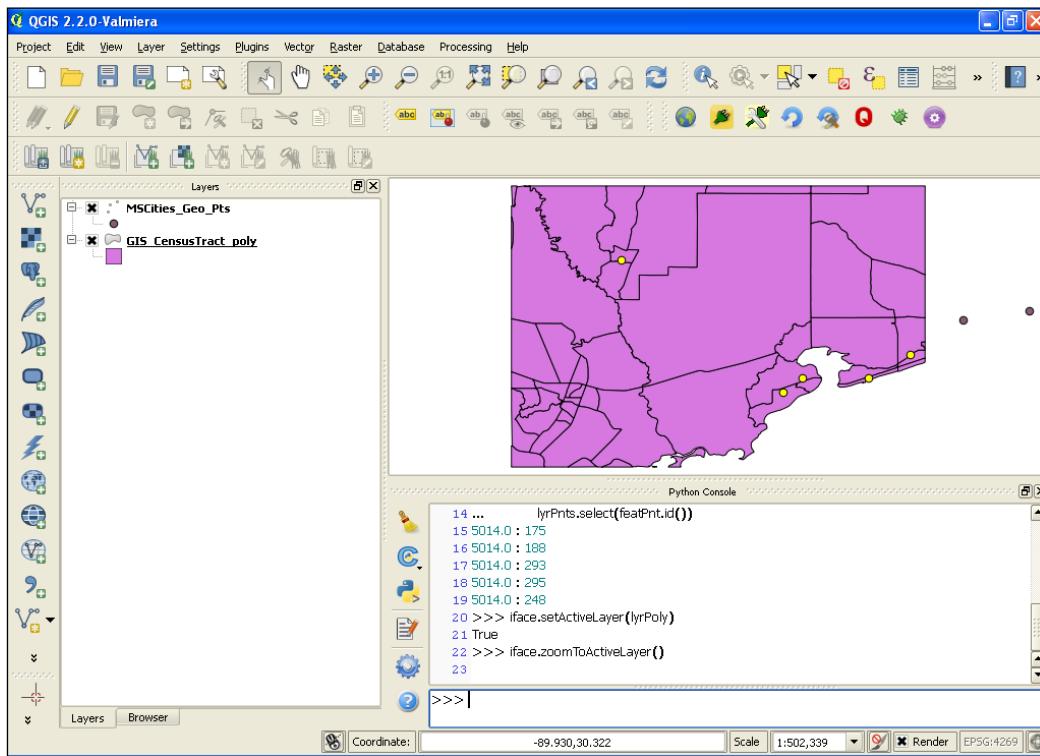
```
iface.setActiveLayer(lyrPoly)
```

11. Zoom to the polygon layer's maximum extent:

```
iface.zoomToActiveLayer()
```

Querying Vector Data

Verify that your map looks similar to the following image:



How it works...

While QGIS has a number of tools for spatial selection, PyQGIS doesn't have a dedicated API for these types of functions. However, there are just enough methods in the API, thanks to the underlying `ogr`/GEOS library, that you can easily create your own spatial filters for two layers. Step 7 isn't entirely necessary, but we gain some efficiency using the bounding box of the polygon to limit the number of point features we're examining. Calculations involving rectangles are far quicker than detailed point-in-polygon queries. So, we quickly reduce the number of points we need to iterate through for the more expensive spatial operations.

Filtering a layer by attributes

In addition to the spatial queries outlined in the previous recipe, we can also subset a layer by its attributes. This type of query resembles a more traditional relational database query and in fact uses SQL statements. In this recipe, we will filter a point shapefile-based layer by an attribute.

Getting ready

We'll use the same New York City Museums layer used in the previous recipes in this chapter. You can download the layer from https://geospatialpython.googlecode.com/svn/ NYC_MUSEUMS_GEO.zip.

Unzip that file and place the shapefile's contents in a directory named `nyc` within your `qgis_data` directory, within your root or home directory.

How to do it...

In this recipe, we'll filter the layer by an attribute, select the filtered features, and zoom to them, as follows:

1. First, we load the point layer:

```
lyrPts = QgsVectorLayer("/qgis_data/nyc/NYC_MUSEUMS_GEO.shp",
    "Museums", "ogr")
```

2. Next, we add the layer to the map in order to visualize the points:

```
QgsMapLayerRegistry.instance().addMapLayers([lyrPts])
```

3. Now, we filter the point layer to points with attributes that match a specific zip code:

```
selection = lyrPts.getFeatures(QgsFeatureRequest().
    setFilterExpression(u'"ZIP" = 10002'))
```

4. Then, we use a list comprehension to create a list of feature IDs that are fed to the feature selection method:

```
lyrPts.setSelectedFeatures([s.id() for s in selection])
```

5. Finally, we zoom to the selection:

```
iface.mapCanvas().zoomToSelected()
```

Verify that the point layer has three selected features, shown in yellow.

How it works...

This recipe takes advantage of QGIS filter expressions, highlighted in step 3. These filter expressions are a subset of SQL. The `QgsFeatureRequest` handles the query expression as an optional argument to return an iterator with just the features you want. These queries also allow some basic geometry manipulation. This recipe also introduces the `mapCanvas().zoomToSelected()` method, which is a convenient way to set the map's extent to the features of interest.

Buffering a feature intermediate

Buffering a feature creates a polygon around a feature as a selection geometry or just a simple visualization. In this recipe, we'll buffer a point in a point feature and add the returned polygon geometry to the map.

Getting ready

Once again, we'll use the same New York City Museums layer. You can download the layer from https://geospatialpython.googlecode.com/svn/NYC_MUSEUMS_GEO.zip.

Unzip that file and place the shapefile's contents in a directory named `nyc` within your `qgis_data` directory, within your root or home directory.

How to do it...

This recipe involves both a spatial operation and multiple visualizations. To do this, perform the following steps:

1. First, load the layer:

```
lyr = QgsVectorLayer("/qgis_data/nyc/NYC_MUSEUMS_GEO.shp",
" Museums", "ogr")
```

2. Next, visualize the layer on the map:

```
QgsMapLayerRegistry.instance().addMapLayers([lyr])
```

3. Access the layer's features:

```
fts = lyr.getFeatures()
```

4. Grab the first feature:

```
ft = fts.next()
```

5. Select this feature:

```
lyr.setSelectedFeatures([ft.id()])
```

6. Create the buffer:

```
buff = ft.geometry().buffer(.2,8)
```

7. Set up a memory layer for the buffer's geometry:

```
buffLyr = QgsVectorLayer('Polygon?crs=EPSG:4326', 'Buffer',
'memory')
```

8. Access the layer's data provider:

```
pr = buffLyr.dataProvider()
```

9. Create a new feature:

```
b = QgsFeature()
```

10. Set the feature's geometry with the buffer geometry:

```
b.setGeometry(buff)
```

11. Add the feature to the data provider:

```
pr.addFeatures([b])
```

12. Update the buffer layer's extents:

```
buffLyr.updateExtents()
```

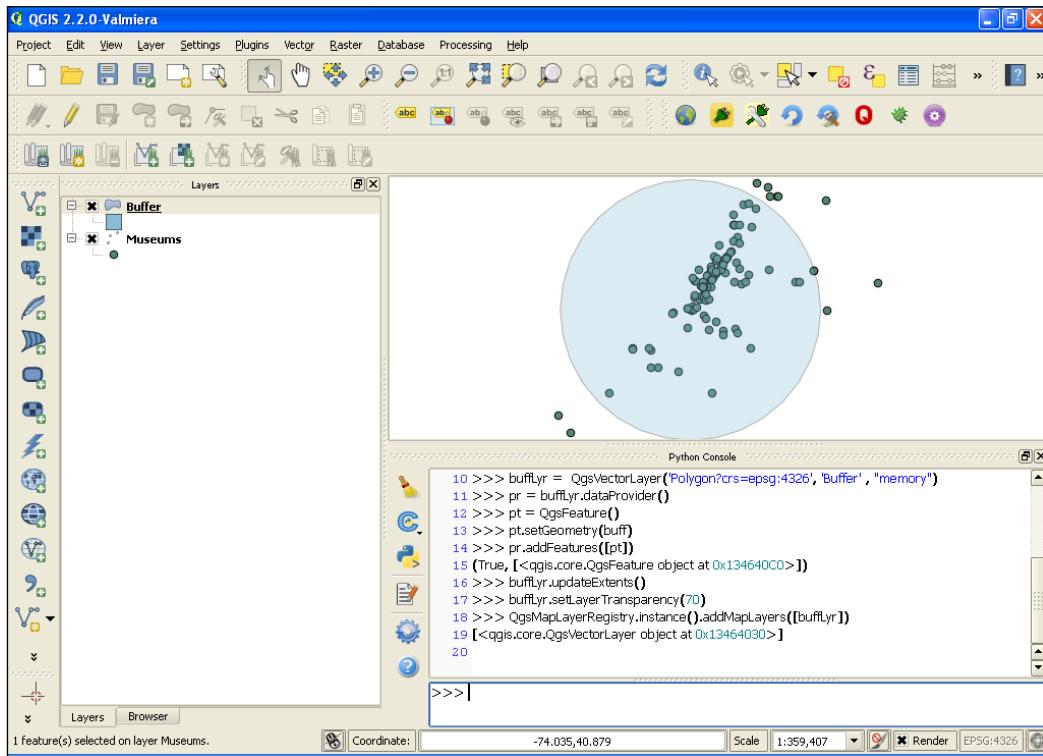
13. Set the buffer layer's transparency so that you can see other features as well:

```
buffLyr.setLayerTransparency(70)
```

14. Add the buffer layer to the map:

```
QgsMapLayerRegistry.instance().addMapLayers([buffLyr])
```

Verify that your map looks similar to this screenshot:



How it works...

The interesting portion of this recipe starts with Step 6, which creates the buffer geometry. The parameters for the `buffer()` method are the distance in map units for the buffer followed by the number of straight line segments used to approximate curves. The more segments you specify, the more the buffer appears like a circle. However, more segments equals greater geometric complexity and therefore slower rendering, as well as slower geometry calculations. The other interesting feature of this recipe is Step 13, in which we set the transparency of the layer to 70 percent. We also introduce the way to create a new layer, which is done in memory. Later chapters will go more in depth on creating data.

Measuring the distance between two points

In the `QgsDistanceArea` object, PyQGIS has excellent capabilities for measuring the distance. We'll use this object for several recipes, starting with measuring the distance between two points.

Getting ready

If you don't already have the New York City Museums layer used in the previous recipes in this chapter, download the layer from https://geospatialpython.googlecode.com/svn/NYC_MUSEUMS_GEO.zip.

Unzip that file and place the shapefile's contents in a directory named `nyc` within your `qgis_data` directory, within your root or home directory.

How to do it...

In the following steps, we'll extract the first and last points in the layer's point order and measure their distance:

1. First, import the library that contains the QGIS contents:

```
from qgis.core import Qgs
```

2. Then, load the layer:

```
lyr = QgsVectorLayer("/qgis_data/nyc/NYC_MUSEUMS_GEO.shp",
    "Museums", "ogr")
```

3. Access the features:

```
fts = lyr.getFeatures()
```

4. Get the first feature:

```
first = fts.next()
```

5. Set a placeholder for the last feature:

```
last = fts.next()
```

6. Iterate through the features until you get the last one:

```
for f in fts:  
    last = f
```

7. Create a measurement object:

```
d = QgsDistanceArea()
```

8. Measure the distance:

```
m = d.measureLine(first.geometry().asPoint(),  
last.geometry().asPoint())
```

9. Convert the measurement value from decimal degrees to meters:

```
d.convertMeasurement(m, 2, 0, False)
```

10. Ensure that your Python console output looks similar to this tuple:

```
(4401.1622240174165, 0)
```

How it works...

The `QgsDistanceArea` object accepts different types of geometry as input. In this case, we use two points. The map units for this layer are in decimal degrees, which isn't meaningful for a distance measurement. So, we use the `QgsDistanceArea.convertMeasurement()` method to convert the output to meters. The parameters for the method are the measurement output, the input units (in decimal degrees), the output units (meters), and a boolean to denote whether this conversion is an area calculation versus a linear measurement.

The returned tuple is the measurement value and the units. The value 0 tells us that the output is in meters.

Measuring the distance along a line sample

In this recipe, we'll measure the distance along a line with multiple vertices.

Getting ready

For this recipe, we'll use a line shapefile with two features. You can download the shapefile as a `.ZIP` file from <https://geospatialpython.googlecode.com/svn/paths.zip>

Unzip the shapefile into a directory named `qgis_data/shapes` within your root or home directory.

How to do it...

The steps for this recipe are fairly straightforward. We'll extract the geometry from the first line feature and pass it to the measurement object, as shown here:

1. First, we must load the QGIS constants library:

```
from qgis.core import Qgs
```

2. Load the line layer:

```
lyr = QgsVectorLayer("/qgis_data/shapes/paths.shp", "Route",  
"ogr")
```

3. Grab the features:

```
fts = lyr.getFeatures()
```

4. Get the first feature:

```
route = fts.next()
```

5. Create the measurement object instance:

```
d = QgsDistanceArea()
```

6. Then, we must configure the QgsDistanceArea object to use the ellipsoidal mode for accurate measurements in meters:

```
d.setEllipsoidalMode(True)
```

7. Pass the line's geometry to the measureLine method:

```
m = d.measureLine(route.geometry().asPolyline())
```

8. Convert the measurement output to miles:

```
d.convertMeasurement(m, Qgs.Meters, Qgs.NauticalMiles, False)
```

Ensure that your output looks similar to the following:

```
(2314126.583384674, 7)
```

How it works...

The QgsDistanceArea object can perform any type of measurement, based on the method you call. When you convert the measurement from meters (represented by 0) to miles (identified by the number 7), you will get a tuple with the measurement in miles and the unit identifier. The QGIS API documentation shows the values for all the unit constants

(<http://qgis.org/api/classQGis.html>).

Calculating the area of a polygon

This recipe simply measures the area of a polygon.

Getting ready

For this recipe, we'll use a single-feature polygon shapefile, which you can download from <https://geospatialpython.googlecode.com/files/Mississippi.zip>

Unzip the shapefile and put it in a directory named **qgis_data/ms** within your root or home directory.

How to do it...

Perform the following steps to measure the area of a large polygon:

1. First, import the QGIS constants library, as follows:

```
from qgis.core import Qgs
```

2. Load the layer:

```
lyr = QgsVectorLayer("/qgis_data/ms/mississippi.shp",
"Mississippi", "ogr")
```

3. Access the layer's features:

```
fts = lyr.getFeatures()
```

4. Get the boundary feature:

```
boundary = fts.next()
```

5. Create the measurement object instance:

```
d = QgsDistanceArea()
```

6. Pass the polygon list to the `measureArea()` method:

```
m = d.measurePolygon(boundary.geometry().asPolygon()[0])
```

7. Convert the measurement from decimal degrees to miles:

```
d.convertMeasurement(m, QGis.Degrees, QGis.NauticalMiles,
True)
```

8. Verify that your output looks similar to the following:

```
(42955.47889640281, 7)
```

How it works...

PyQGIS has no `measureArea()` method, but it has a `measurePolygon()` method in the `QgsDistanceArea` object. The method accepts a list of points. In this case, when we convert the measurement output from decimal degrees to miles, we also specify `True` in the `convertMeasurement()` method so that QGIS knows that it is an area calculation. Note that when we get the boundary geometry as a polygon, we use an index of 0, suggesting that there is more than one polygon. A polygon geometry can have inner rings, which are specified as additional polygons. The outermost ring, in this case the only ring, is the first polygon.

Creating a spatial index

Until now, the recipes in this book used the raw geometry for each layer of operations. In this recipe, we'll take a different approach and create a spatial index for a layer before we run operations on it. A spatial index optimizes a layer for spatial queries by creating additional, simpler geometries that can be used to narrow down the field of possibilities within the complex geometry.

Getting ready

If you don't already have the New York City Museums layer used in the previous recipes in this chapter, download the layer from https://geospatialpython.googlecode.com/svn/NYC_MUSEUMS_GEO.zip.

Unzip that file and place the shapefile's contents in a directory named `nyc` within your `qgis_data` directory, within your root or home directory.

How to do it...

In this recipe, we'll create a spatial index for a point layer and then we'll use it to perform a spatial query, as follows:

1. Load the layer:

```
lyr = QgsVectorLayer("/qgis_data/nyc/NYC_MUSEUMS_GEO.shp",
" Museums", "ogr")
```

2. Get the features:

```
fts = lyr.getFeatures()
```

3. Get the first feature in the set:

```
first = fts.next()
```

4. Now, create the spatial index:

```
index = QgsSpatialIndex()
```

5. Begin loading the features:

```
index.insertFeature(first)
```

6. Insert the remaining features:

```
for f in fts:  
    index.insertFeature(f)
```

7. Now, select the IDs of 3 points nearest to the first point. We use the number 4 because the starting point is included in the output:

```
hood = index.nearestNeighbor(first.geometry().asPoint(), 4)
```

How it works...

The index speeds up spatial operations. However, you must add each feature one by one. Also, note that the `nearestNeighbor()` method returns the ID of the starting point as part of the output. So, if you want 4 points, you must specify 5.

Calculating the bearing of a line

Sometimes, you need to know the compass bearing of a line to create specialized symbology or use as input in a spatial calculation. Even though its name only mentions distance and area, the versatile `QgsDistanceArea` object includes this function as well. In this recipe, we'll calculate the bearing of the end points of a line. However, this recipe will work with any two points.

Getting ready

We'll use the line shapefile used in a previous recipe. You can download the shapefile as a `.ZIP` file from <https://geospatialpython.googlecode.com/svn/paths.zip>

Unzip the shapefile into a directory named `qgis_data/shapes` within your root or home directory.

How to do it...

The steps to be performed are as simple as getting the two points we need and running them through the bearing function, converting from radians to degrees, and then converting to a positive compass bearing:

1. First, import the Python math module:

```
import math
```

2. Next, load the layer:

```
lyr = QgsVectorLayer("/qgis_data/shapes/paths.shp", "Route",  
"ogr")
```

3. Now, grab the features:

```
fts = lyr.getFeatures()
```

4. Then, grab the first line feature:

```
route = fts.next()
```

5. Create the measurement object:

```
d = QgsDistanceArea()
```

6. You must set the ellipsoidal mode to True in order to project the data before calculating the bearing:

```
d.setEllipsoidalMode(True)
```

7. Get all the points as a list:

```
points = route.geometry().asPolyline()
```

8. Get the first point:

```
first = points[0]
```

9. Grab the last point:

```
last = points[-1]
```

10. Calculate the bearing in radians:

```
r = d.bearing(first, last)
```

11. Now convert radians to degrees:

```
b = math.degrees(r)
```

12. Ensure that the bearing is positive:

```
if b < 0: b += 360
```

13. View the output:

```
print b
```

Verify that the bearing is close to the following number:

320.3356091875395

How it works...

The default output of the bearing calculation is in radians. However, the Python `math` module makes conversion a snap of the fingers. If the conversion of degrees results in a negative number, most of the time we will want to add that number to 360 in order to get a compass bearing, as we did here.

Loading data from a spreadsheet

Spreadsheets are one of the most common methods used to collect and store simple geographic data. QGIS can work with text files called CSV or comma-separated values files. Any spreadsheet can be converted to a CSV using the spreadsheet program. As long as the CSV data has a column representing x values, one column representing y values, and other columns representing data with the first row containing field names, QGIS can import it. Many organizations distribute geographic information as a CSV, so sooner or later you will find yourself importing a CSV. Moreover, PyQGIS lets you do it programmatically. Note that a CSV can be delimited by any character as long as it is consistent. Also, the file extension of the CSV file doesn't matter as long as you specify the file type for QGIS.

Getting ready

We'll use a sample CSV file with point features representing points of interest in a region. You can download this sample from https://geospatialpython.googlecode.com/svn/MS_Features.txt.

Save this to your `qgis_data/ms` directory in your root or home directory.

How to do it...

We will build a URI string to load the CSV as a vector layer. All of the parameters used to describe the structure of the CSV are included in the URI, as follows:

1. First, we build the base URI string with the filename:

```
uri = "file:///qgis_data/ms/MS_Features.txt?"
```

2. Next, we tell QGIS that the file is a CSV file:

```
uri += "type=csv&"
```

3. Now, we specify our delimiter, which is a pipe ("|"), as a URL-encoded value:

```
uri += "delimiter=%7C&"
```

4. Next, we tell QGIS to trim any spaces at the ends of the fields:

```
uri += "trimFields=Yes&"
```

5. Now, the most important part, we specify the x field:

```
uri += "xField=PRIM_LONG_DEC&"
```

6. Then, we specify the y field:

```
uri += "yField=PRIM_LAT_DEC&"
```

7. We decline the spatial index option:

```
uri += "spatialIndex=no&"
```

8. We decline the subset option:

```
uri += "subsetIndex=no&"
```

9. We tell QGIS not to watch the file for changes:

```
uri += "watchFile=no&"
```

10. Finally, we complete the uri with the CRS of the layer:

```
uri += "crs=epsg:4326"
```

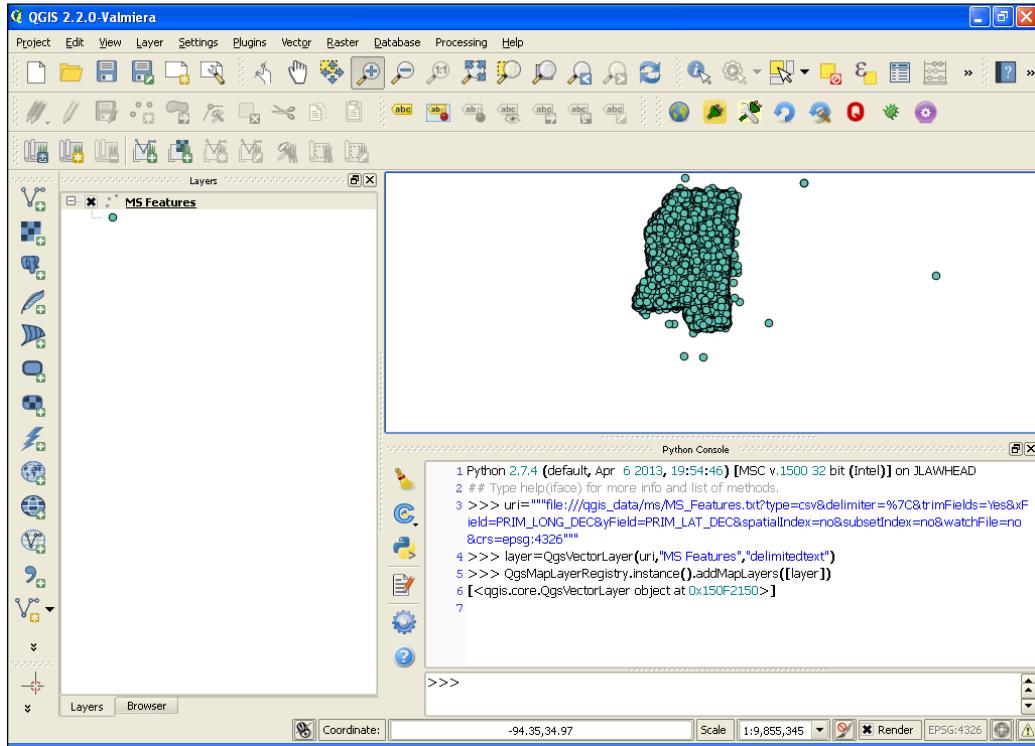
11. We load the layer using the delimitedtext data provider:

```
layer = QgsVectorLayer(uri, "MS Features", "delimitedtext")
```

12. Finally, we add it to the map:

```
QgsMapLayerRegistry.instance().addMapLayers([layer])
```

Verify that your map looks similar to the map shown in the following screenshot:



How it works...

The URI is quite extensive, but necessary to give QGIS enough information to properly load the layer. We used strings in this simple example, but using the `QUrl` object is safer, as it handles the encoding for you. The documentation for the `QUrl` class is in the Qt documentation at <http://qt-project.org/doc/qt-4.8/qurl.html>.

Note that in the URI, we tell QGIS that the type is **CSV**, but when we load the layer, the type is **delimitedtext**. QGIS will ignore empty fields as long as all of the columns are balanced.

There's more...

If you're having trouble loading a layer, you can use the **QGIS Add Delimited Text Layer...** dialog under the **Layer** menu to figure out the correct parameters. Once the layer is loaded, you can take a look at its metadata to see the URI QGIS constructed to load it. You can also get the correct parameters from a loaded, delimited text layer using the `layer.source()` method programmatically. And, of course, both of these methods work with any type of layer, not just delimited text. Unlike other layer types, however, you cannot edit delimited text layers in QGIS.

3

Editing Vector Data

In this chapter, we will cover the following recipes:

- ▶ Creating a vector layer in memory
- ▶ Adding a point feature to a vector layer
- ▶ Adding a line feature to a vector layer
- ▶ Adding a polygon feature to a vector layer
- ▶ Adding a set of attributes to a vector layer
- ▶ Adding a field to a vector layer
- ▶ Joining a shapefile attribute table to a CSV file
- ▶ Moving vector layer geometry
- ▶ Changing a vector layer attribute
- ▶ Deleting vector layer geometry
- ▶ Deleting a vector layer field
- ▶ Deleting vector layer attributes
- ▶ Reprojecting a vector layer
- ▶ Converting a shapefile to Keyhole Markup Language (KML)
- ▶ Merging shapefiles
- ▶ Splitting a shapefile
- ▶ Generalizing a vector layer
- ▶ Dissolving vector shapes
- ▶ Performing a union on vector shapes
- ▶ Rasterizing a vector layer

Introduction

This chapter details how to edit QGIS vector data using the Python API. The `QgsVectorLayer` object contains the basics of adding, editing, and deleting features. All other geospatial operations are accessed through the **Processing Toolbox** or even through custom scripts.

Creating a vector layer in memory

Sometimes, you need to create a temporary data set for quick output or as an intermediate step in a more complex operation without the overhead of actually writing a file to disk. PyQGIS employs **memory layers** that allow you to create a complete vector data set, including the geometry, fields, and attributes, virtually. Once the memory layer is created, you can work with it in the same way you would work with a vector layer loaded from disk.

Getting ready

This recipe entirely runs inside the PyQGIS console, so no preparation or external resources are required.

How to do it...

We will create a Point vector layer, named `Layer 1` with a few fields and then validate it:

1. Start QGIS.
2. From the **Plugins** menu, select **Python Console**.
3. In the Python console, create a `QgsVectorLayer`, including fields, and specify it as a memory data provider:

```
vectorLyr =  
QgsVectorLayer('Point?crs=epsg:4326&field=city:string(25)&  
field=population:nt', 'Layer 1' , "memory")
```

4. Now, validate the layer and ensure that the console returns `True`:

```
vectorLyr.isValid()
```

How it works...

The QgsVectorLayer requires three arguments. The last argument specifies the type, which in this case is `memory`. The second argument specifies the layer name. Normally, the first argument is the path to the file on disk, which is used to create the layer. In the case of the memory layer, the first argument becomes the construction string for the layer. The format uses query parameters that follow the convention `key = value`. We first specify the coordinate reference system and then specify the fields we want. In this case, we specify the first field, a string for city names, and then an integer field for population.

There's more...

You can easily see how describing a layer's attribute table structure in a string can become unwieldy. You can also use a Python-ordered dictionary to build the string dynamically, as shown in the following steps.

1. First, you need to import the `OrderedDict` container, which remembers the order in which keys are inserted:

```
from collections import OrderedDict
```

2. Then, build an ordered dictionary that contains attribute names and types:

```
fields =
OrderedDict([('city','str(25)'),('population','int')))
```

3. Next, build a string by joining the output of a Python list comprehension that loops through the ordered dictionary:

```
path = '&'.join(['field={}:{}'.format(k,v) for k,v in
fields.items()])
```

4. Finally, use this string to define the layer:

```
vectorLyr = QgsVectorLayer('Point?crs=epsg:4326&' + path,
'Layer 1' , "memory")
```

Adding a point feature to a vector layer

This recipe performs the simplest possible edit to a vector layer instantiated from a shapefile. We will add a point to an existing point layer.

Getting ready

For this recipe, download the zipped shapefile from https://geospatialpython.googlecode.com/svn/NYC_MUSEUMS_GEO.zip.

Extract the .shp, .shx, and .dbf files to the /qgis_data/nyc directory.

How to do it...

We will load the vector layer from the shapefile, create a new geometry object as a point, create a new feature, set the geometry, and add it to the layer's data provider. Finally, we will update the extent of the layer to make sure that the bounding box of the layer encapsulates the new point:

1. Start QGIS.
2. From the **Plugins** menu, select **Python Console**.
3. First, load the layer:

```
vectorLyr =  
QgsVectorLayer('/qgis_data/nyc/NYC_MUSEUMS_GEO.shp', 'Museums'  
, "ogr")
```

4. Now, will access the layer's data provider:

```
vpr = vectorLyr.dataProvider()
```

5. Next, create a new point using the `QgsGeometry` object:

```
pnt = QgsGeometry.fromPoint(QgsPoint(-74.80, 40.549))
```

6. Now, will create a new `QgsFeature` object to house the geometry:

```
f = QgsFeature()
```

7. Next, set the geometry of the feature using our point:

```
f.setGeometry(pnt)
```

8. Then, place the features into the layer's feature list:

```
vpr.addFeatures([f])
```

9. Finally, update the layer's extent to complete the addition:

```
vectorLyr.updateExtents()
```

How it works...

PyQGIS abstracts the points within a layer into four levels. At the lowest level is the `QgsPoint` object, which contains nothing more than the coordinates of the point. This object is added to an abstract `QgsGeometry` object. This object becomes the geometric part of a `QgsFeature` object, which also has the ability to store and manage attributes. All the features are managed by the `QgsDataProvider` object. The data provider manages the geospatial aspect of a layer to separate that aspect from styling and other presentation-related portions. QGIS has another editing approach in Python, which is called an **editing buffer**. When you use an editing buffer, the changes can be displayed, but they are not permanent until you commit them. The most common use case for this editing method is in GUI applications where the user may decide to roll back the changes by cancelling the editing session. The *PyQGIS Developer Cookbook* has an example of using and editing buffers in Python, and is available at http://docs.qgis.org/2.6/en/docs/pyqgis_developer_cookbook/vector.html.

Adding a line feature to a vector layer

Adding a line to a vector layer in QGIS is identical to adding a single point, but here you just have to add more points to the `QgsGeometry` object.

Getting ready

For this recipe, you will need to download a zipped line shapefile that contains two line features from <https://geospatialpython.googlecode.com/svn/paths.zip>.

Extract the ZIP file to a directory named `paths` in your `/qgis_data` directory.

How to do it...

In this recipe, we will load the line layer from the shapefile, build a list of points, create a new geometry object, and add the points as a line. We will also create a new feature, set the geometry, and add it to the layer's data provider. Finally, we will update the extent of the layer to make sure that the bounding box of the layer encapsulates the new feature:

1. Start QGIS.
2. From the **Plugins** menu, select **Python Console**.
3. First, load the line layer and ensure that it is valid:

```
vectorLyr = QgsVectorLayer('/qgis_data/paths/paths.shp',
    'Paths' , "ogr")
vectorLyr.isValid()
```

4. Next, access the layer's data provider:

```
vpr = vectorLyr.dataProvider()
```

5. Now, build our list of points for a new line:

```
points = []
points.append(QgsPoint(430841,5589485))
points.append(QgsPoint(432438,5575114))
points.append(QgsPoint(447252,5567663))
```

6. Then, create a geometry object from the line:

```
line = QgsGeometry.fromPolyline(points)
```

7. Create a feature and set its geometry to the line:

```
f = QgsFeature()
f.setGeometry(line)
```

8. Finally, add the feature to the layer data provider and update the extent:

```
vpr.addFeatures([f])
vectorLyr.updateExtents()
```

How it works...

As with all the geometry in QGIS, we use the four-step process of building points, geometry, feature, and data provider to add the line. Interestingly, the `QgsGeometry` object accepts Python lists for the collection of points instead of creating a formal object, as is done with the `QgsPoint` object.

Adding a polygon feature to a vector layer

In this recipe, we'll add a polygon to a layer. A polygon is the most complex kind of geometry. However, in QGIS, the API is very similar to a line.

Getting ready

For this recipe, we'll use a simple polygon shapefile, which you can download as a ZIP file from <https://geospatialpython.googlecode.com/files/polygon.zip>.

Extract this shapefile to a folder called `polygon` in your `/qgis_data` directory.

How to do it...

This recipe will follow the standard PyQGIS process of loading a layer, building a feature, and adding it to the layer's data provider, as follows:

1. Start QGIS.
2. From the **Plugins** menu, select **Python Console**.
3. First, load the layer and validate it:

```
vectorLyr =  
QgsVectorLayer('/qgis_data/polygon/polygon.shp', 'Polygon'  
, "ogr")  
vectorLyr.isValid()
```

4. Next, access the layer's data provider:

```
vpr = vectorLyr.dataProvider()
```
5. Now, build a list of points for the polygon:

```
points = []  
points.append(QgsPoint(-123.26,49.06))  
points.append(QgsPoint(-127.19,43.07))  
points.append(QgsPoint(-120.70,35.21))  
points.append(QgsPoint(-115.89,40.02))  
points.append(QgsPoint(-113.04,48.47))  
points.append(QgsPoint(-123.26,49.06))
```

6. Next, create a geometry object and ingest the points as a polygon. We nest our list of points in another list because a polygon can have inner rings, which will consist of additional lists of points being added to this list:

```
poly = QgsGeometry.fromPolygon([points])
```

7. Next, build the feature object and add the points:

```
f = QgsFeature()  
f.setGeometry(poly)
```

8. Finally, add the feature to the layer's data provider and update the extents:

```
vpr.addFeatures([f])
```

How it works...

Adding a polygon is very similar to adding a line, with one key difference that is a common pitfall. The last point must be identical to the first point in order to close the polygon. If you don't repeat the first point, you won't receive any errors, but the polygon will not be displayed in QGIS, which can be difficult to troubleshoot.

Adding a set of attributes to a vector layer

Each QGIS feature has two parts, the geometry and the attributes. In this recipe, we'll add an attribute for a layer from an existing dataset.

Getting ready

We will use a point shapefile with museum data for New York City, which you can download as a ZIP file from https://geospatialpython.googlecode.com/svn/NYC_MUSEUMS_GEO.zip.

Extract this shapefile to the `/qgis_data/nyc` directory.

How to do it...

A feature must have geometry, but it does not require attributes. So, we will create a new feature, add some attributes, and then add everything to the layer, as follows:

1. Start QGIS.
2. From the **Plugins** menu, select **Python Console**.
3. First, load the layer and validate it:

```
vectorLyr = QgsVectorLayer('/qgis_data/nyc/NYC_MUSEUMS_GEO.shp',
    'Museums' , "ogr")
vectorLyr.isValid()
```
4. Next, access the layer's data provider so that we can get the list of fields:

```
vpr = vectorLyr.dataProvider()
```
5. Now, create a point geometry, which in this case is a new museum:

```
pnt = QgsGeometry.fromPoint(QgsPoint(-74.13401,40.62148))
```
6. Next, get the `fields` object for the layer that we'll need to create a new feature for:

```
fields = vpr.fields()
```

7. Then, create a new feature and initialize the attributes:

```
f = QgsFeature(fields)
```

8. Now, set the geometry of our new museum feature:

```
f.setGeometry(pnt)
```

9. Now, we are able to add a new attribute. Adding an attribute is similar to updating a Python dictionary, as shown here:

```
f['NAME'] = 'Python Museum'
```

10. Finally, we add the feature to the layer and update the extents:

```
vpr.addFeatures([f])
vectorLyr.updateExtents()
```

How it works...

PyQGIS attributes are defined as an ordered array. The syntax for referencing a field is similar to the syntax for a Python dictionary. We use the layer's data provider object to perform the actual editing. When we use this approach, no signals are triggered at the layer object level. If we are just trying to edit data on the filesystem, that's okay, but if the layer is going to be added to the map canvas for display or user interaction, then you should use the editing buffer in the `QgsVectorLayer` object. This editing buffer allows you to commit or roll back changes and also keeps track of the state of the layer when things are changed.

Adding a field to a vector layer

This recipe demonstrates how to add a new field to a layer. Each field represents a new column in a dataset for which each feature has a new attribute. When you add a new attribute, all the features are set to `NULL` for that field index.

Getting ready

We will use the New York City museums' shapefile used in other recipes, which you can download as a ZIP file from https://geospatialpython.googlecode.com/svn/NYC_MUSEUMS_GEO.zip.

Extract this shapefile to `/qgis_data/nyc`.

How to do it...

All the data management for a layer is handled through the layer's data provider and the fields are no different. We will load the layer, access the data provider, define the new field, and finalize the change, as follows:

1. Start QGIS.
2. From the **Plugins** menu, select **Python Console**.
3. First, you must import the Qt library's data types, which PyQGIS uses to specify the layer field's data types:

```
from PyQt4.QtCore import QVariant
```
4. Next, load and validate the layer:

```
vectorLyr =  
QgsVectorLayer('/qgis_data/nyc/NYC_MUSEUMS_GEO.shp', 'Museums'  
, "ogr")  
vectorLyr.isValid()
```

5. Then, access the layer data provider:

```
vpr = vectorLyr.dataProvider()
```
6. Now, add a Python list of QgsField objects, which defines the field name and type. In this case, we'll add one field named Admission as a Double:

```
vpr.addAttribute([QgsField("Admission", QVariant.Double)])
```
7. Finally, update the fields to complete the change:

```
vectorLyr.updateFields()
```

How it works...

The nomenclature used for the fields and attributes in QGIS is a little inconsistent and can be confusing if you've used other GIS packages. In QGIS, a column is a field that has a name and a type. The attribute table holds a value for each field column and each feature row. However, in the QgsVectorDataProvider object, you use the `addAttributes()` method to add a new field column. Also, in other GIS software, you may see the use of the word `field` and `attribute` reversed.

Joining a shapefile attribute table to a CSV file

Joining attribute tables to other database tables allows you to use a spatial dataset in order to reference a dataset without any geometry, using a common key between the data tables. A very common use case for this is to join a vector dataset of census attributes to a more detailed census attribute dataset. The use case we will demonstrate here links a US census tract file to a detailed CSV file that contains more in-depth information.

Getting ready

For this recipe, you will need a census tract shapefile and a CSV file containing the appropriate census data for the shapefile. You can download the sample data set from <https://geospatialpython.googlecode.com/svn/census.zip>.

Extract this data to a directory named /qgis_data/census.

How to do it...

The join operation is quite involved. We'll perform this operation and save the layer as a new shapefile with the joined attributes. Then we'll load the new layer and compare the field count to the original layer to ensure that the join occurred. We'll use the terms `target layer` and `join layer`. The `target layer` will be the shapefile, and the `join layer` will be a CSV with some additional fields we want to add to the shapefile. To do this, perform the following steps:

1. Start QGIS.
2. From the **Plugins** menu, select **Python Console**.
3. First, load the county's census tract layer and validate it:

```
vectorLyr =
QgsVectorLayer('/qgis_data/census/hancock_tracts.shp',
'Hancock' , "ogr")

vectorLyr.isValid()
```

4. Now, load the CSV file as a layer and validate it as well:

```
infoLyr =
QgsVectorLayer('/qgis_data/census/ACS_12_5YR_S1901_with_ann
.csv' , 'Census' , "ogr")

infoLyr.isValid()
```

5. Once this is done, you must add both the layers to the map registry for the two layers to interact for the join. However, set the visibility to False, so the layers do not appear on the map:

```
QgsMapLayerRegistry.instance().addMapLayers([vectorLyr, infoLyr], False)
```

6. Next, you must create a special join object:

```
info = QgsVectorJoinInfo()
```

7. The join object needs the layer ID of the CSV file:

```
info.joinLayerId = infoLyr.id()
```

8. Next, specify the key field from the CSV file whose values correspond to the values in the shapefile:

```
info.joinFieldName = "GEOid2"
```

9. Then, specify the corresponding field in the shapefile:

```
info.targetFieldName = "GEOID"
```

10. Set the memoryCache property to True in order to speed up access to the joined data:

```
info.memoryCache = True
```

11. Add the join to the layer now:

```
vectorLyr.addJoin(info)
```

12. Next, write out the joined shapefile to a new file on disk:

```
QgsVectorFileWriter.writeAsVectorFormat(vectorLyr, "/qgis_data/census/joined.shp", "CP120", None, "ESRI Shapefile")
```

13. Now, load the new shapefile back in as a layer for verification:

```
joinedLyr = QgsVectorLayer('/qgis_data/census/joined.shp', 'Joined', 'ogr')
```

14. Verify that the field count in the original layer is 12:

```
vectorLyr.dataProvider().fields().count()
```

15. Finally, verify that the new layer has a field count of 142 from the join:

```
joinedLyr.dataProvider().fields().count()
```

How it works...

This recipe reaches out to the very edge of the PyQGIS API, forcing you to use some workarounds. Most recipes for data manipulation can be performed programmatically without writing data to disk or loading layers onto the map, but joins are different. Because the `QgsVectorJoinInfo` object needs the layer ID of the CSV layer, we must add both the layers to the map layer registry. Fortunately, we can do this without making them visible, if we are just trying to write a data manipulation script. A join is designed to be a temporary operation to query a dataset. Oddly, PyQGIS lets you create the join, but you cannot query it. This limitation is the reason why if you want to work with the joined data, you must write it to a new shapefile and reload it. Fortunately, PyQGIS allows you to do that.

There's more...

You can find an alternate method that works around the PyQGIS limitation in a Processing Toolbox script, which manually matches the joined data in Python, at <https://github.com/rldhont/Quantum-GIS/blob/master/python/plugins/processing/algss/qgis/JoinAttributes.py>.

Moving vector layer geometry

Sometimes, you need to change the location of a feature. You can do this by deleting and re-adding the feature, but PyQGIS provides a simple way to change the geometry.

Getting ready

You will need the New York City museums' shapefile, which you can download as a ZIP file from https://geospatialpython.googlecode.com/svn/NYC_MUSEUMS_GEO.zip.

Extract this shapefile to `/qgis_data/nyc`.

How to do it...

We will load the shapefile as a vector layer, validate it, define the feature ID we want to change, create the new geometry, and change the feature in the layer. To do this, perform the following steps:

1. Start QGIS.
2. From the **Plugins** menu, select **Python Console**.

3. First, load the layer and validate it:

```
vectorLyr =  
QgsVectorLayer('/qgis_data/nyc/NYC_MUSEUMS_GEO.shp',  
'Museums' , "ogr")  
  
vectorLyr.isValid()
```

4. Next, define the feature ID we are interested in changing:

```
feat_id = 22
```

5. Now, create the new point geometry, which will become the new location:

```
geom = QgsGeometry.fromPoint(QgsPoint(-74.20378,40.89642))
```

6. Finally, change the geometry and replace it with our new geometry, specifying the feature ID:

```
vectorLyr.dataProvider().changeGeometryValues({feat_id : geom})
```

How it works...

The `changeGeometryValues()` method makes editing a snap of the fingers. If we had to delete and then re-add the feature, we would have to go through the trouble of reading the attributes, preserving them, and then re-adding them with the new feature. You must, of course, know the feature ID of the feature you want to change. How you determine this ID depends on your application. Typically, you will query the attributes to find a specific value, or you can do a spatial operation of some sort.

Changing a vector layer feature's attribute

The process to change an attribute in a feature is straightforward and well-supported by the PyQGIS API. In this recipe, we'll change a single attribute, but you can change as many attributes of a feature as desired at once.

Getting ready

You will need the New York City museums' shapefile used in other recipes, which you can download as a ZIP file from https://geospatialpython.googlecode.com/svn/NYC_MUSEUMS_GEO.zip.

Extract this shapefile to `/qgis_data/nyc`.

How to do it...

We will load the shapefile as a vector layer, validate it, define the feature IDs of the fields we want to change, get the index of the field names that we will change, define the new attribute value as an attribute index and value, and change the feature in the layer. To do this, we need to perform the following steps:

1. Start QGIS.

2. From the **Plugins** menu, select **Python Console**.

3. First, load the layer and validate it:

```
vectorLyr =  
QgsVectorLayer('/qgis_data/nyc/NYC_MUSEUMS_GEO.shp',  
'Museums' , "ogr")  
vectorLyr.isValid()
```

4. Next, define the feature IDs you want to change:

```
fid1 = 22  
fid2 = 23
```

5. Then, get the index of the fields you want to change, which are the telephone number and city name:

```
tel = vectorLyr.fieldNameIndex("TEL")  
city = vectorLyr.fieldNameIndex("CITY")
```

6. Now, create the Python dictionary for the attribute index and the new value, which in this case is an imaginary phone number:

```
attr1 = {tel:"(555) 555-1111", city:"NYC"}  
attr2 = {tel:"(555) 555-2222", city:"NYC"}
```

7. Finally, use the layer's data provider to update the fields:

```
vectorLyr.dataProvider().changeAttributeValues({fid1:attr1,  
fid2:attr2})
```

How it works...

Changing attributes is very similar to changing the geometry within a feature. We explicitly name the feature IDs in this example, but in a real-world program, you would collect these IDs as a part of some other process output, such as a spatial selection. An example of this type of spatial selection is available in the *Filtering a layer by Geometry* recipe, in *Chapter 2, Querying Vector Data*.

Deleting a vector layer feature

In this recipe, we'll completely remove a feature, including the geometry and attributes, from a layer.

Getting ready

You will need the New York City museums' shapefile used in other recipes, which you can download as a ZIP file from https://geospatialpython.googlecode.com/svn/NYC_MUSEUMS_GEO.zip.

Extract this shapefile to /qgis_data/nyc.

How to do it...

All we need to do is load the layer and then delete the desired features by ID, using the layer's data provider:

1. Start QGIS.
2. From the **Plugins** menu, select **Python Console**.
3. First, load and validate the layer:

```
vectorLyr =  
QgsVectorLayer('/qgis_data/nyc/NYC_MUSEUMS_GEO.shp',  
'Museums' , "ogr")  
vectorLyr.isValid()
```

4. Next, specify a Python list containing feature IDs. In this case, we have two:

```
vectorLyr.dataProvider().deleteFeatures([ 22, 95 ])
```

How it works...

This operation cannot be simpler or better designed. There are a number of ways in which we can programmatically fill a Python list with feature IDs. For example, we can use the *Chapter 2, Filtering a Layer by Attributes* in this recipe. Then, we just pass this list to the layer's data provider and we are done.

Deleting a vector layer attribute

In this recipe, we'll wipe out an entire attribute and all the feature fields for a vector layer.

Getting ready

You will need the New York City museums' shapefile used in other recipes, which you can download as a ZIP file from https://geospatialpython.googlecode.com/svn/NYC_MUSEUMS_GEO.zip.

Extract this shapefile to `/qgis_data/nyc`.

How to do it...

This operation is straight forward. We'll load and validate the layer, use the layer's data provider to delete the attribute by index, and finally, we will update all the fields to remove the orphaned values. To do this, we need to perform the following steps:

1. Start QGIS.
2. From the **Plugins** menu, select **Python Console**.
3. First, load and validate the layer:

```
vectorLyr =  
QgsVectorLayer('/qgis_data/nyc/NYC_MUSEUMS_GEO.shp',  
'Museums' , "ogr")  
vectorLyr.isValid()
```

4. Then, delete the first attribute:

```
vectorLyr.dataProvider().deleteAttributes([1])
```

5. Finally, update the fields:

```
vectorLyr.updateFields()
```

How it works...

Because we are changing the actual structure of the layer data, we must call the `updateFields()` method of the layer to remove the field values which no longer have an attribute.

Reprojecting a vector layer

We will use the Processing Toolbox in QGIS to reproject a layer to a different coordinate system.

Getting ready

For this recipe, we'll need the Mississippi cities' shapefile in the Mississippi Trans Mercator projection (EPSG 3814), which can be downloaded as a ZIP file from https://geospatialpython.googlecode.com/files/MSCities_MSTM.zip.

Extract the zipped shapefile to a directory named `/qgis_data/ms`.

How to do it...

To reproject the layer, we'll simply call the `qgis:reprojectlayer` processing algorithm, specifying the input shapefile, the new projection, and the output file name. To do this, perform the following steps:

1. Start QGIS.
2. From the **Plugins** menu, select **Python Console**.
3. First, you need to import the processing module:
`import processing`
4. Next, run the reprojection alogoritm, as follows:

```
processing.runalg("qgis:reprojectlayer",
  "/qgis_data/ms/MSCities_MSTM.shp", "epsg:4326",
  "/qgis_data/ms/MSCities_MSTM_4326.shp")
```

How it works...

The source data starts out in EPSG 3814, but we want to project it to WGS 84 Geographic, which is commonly used to deal with global datasets and is usually the default coordinate reference system for GPS devices. The target EPSG code is 4326. Dealing with map projections can be quite complex. This QGIS tutorial has some more examples and explains more about map projections at http://manual.linfiniti.com/en/vector_analysis/reproject_transform.html.

Converting a shapefile to KML

In this recipe, we'll convert a layer to KML. KML is an **Open Geospatial Consortium (OGC)** standard and is supported by the underlying OGR library used by QGIS.

Getting ready

For this recipe, download the following zipped shapefile and extract it to a directory named `/qgis_data/hancock`:

<https://geospatialpython.googlecode.com/files/hancock.zip>

How to do it...

To convert a shapefile to the KML XML format, we'll load the layer and then use the `QgsVectorFileWriter` object to save it as KML:

1. Start QGIS.
2. From the **Plugins** menu, select **Python Console**.
3. First load the layer and validate it:

```
vectorLyr =  
QgsVectorLayer('/qgis_data/hancock/hancock.shp', 'Hancock'  
, "ogr")  
vectorLyr.isValid()
```

4. Then, establish the destination CRS. KML should always be in EPS:4326:

```
dest_crs = QgsCoordinateReferenceSystem(4326)
```
5. Next, use the file writer to save it as a KML file by specifying the file type as KML:

```
QgsVectorFileWriter.writeAsVectorFormat(vectorLyr,  
"/qgis_data/hancock/hancock.kml", "utf-8", dest_crs, "KML")
```

How it works...

You will end up with a KML file in the directory next to your shapefile. KML supports styling information. QGIS uses some default styling information that you can change, either by hand using a text editor, or programmatically using an XML library such as Python's ElementTree. KML is one of many standard vector formats you can export using this method.

Merging shapefiles

Merging shapefiles with matching projections and attribute structures is a very common operation. In QGIS, the best way to merge vector datasets is to use another GIS system included with QGIS on Windows and OSX called **SAGA**. On other platforms, you must install SAGA separately and activate it in the Processing Toolbox configuration. In PyQGIS, you can access SAGA functions through the Processing Toolbox. **SAGA** is yet another open source GIS that is similar to QGIS. However, both packages have strengths and weaknesses. By using SAGA through the Processing Toolbox, you can have the best of both systems.

Getting ready

In this recipe, we'll merge some building footprint shapefiles from adjoining areas into a single shapefile. You can download the sample dataset from https://geospatialpython.googlecode.com/files/tiled_footprints.zip.

Extract the zipped shapefiles to a directory named `/qgis_data/tiled_footprints`.

How to do it...

We will locate all the `.shp` files in the data directory and hand them to the `saga:mergeshapeslayers` object in order to merge them.

1. Start QGIS.
2. From the **Plugins** menu, select **Python Console**.
3. Import the Python `glob` module for wildcard file matching:

```
import glob
```

4. Next, import the processing module for the merge algorithm:

```
import processing
```

5. Now, specify the path of our data directory:

```
pth = "/qgis_data/tiled_footprints/"
```

6. Locate all the `.shp` files:

```
files = glob.glob(pth + "/*.shp")
```

7. Then, specify the output name of the merged shapefile:

```
out = pth + "merged.shp"
```

8. Finally, run the algorithm that will load the merged shapefile on to the map:

```
processing.runandload("saga:mergeshapeslayers", files.pop(0), ;
".join(files), out)
```

How it works...

The algorithm accepts a base file and then a semicolon-separated list of additional files to be merged, and it finally accepts the output filename. The `glob` module creates a list of the files. To get the base file, we use the list `pop()` method to get the first filename. Then, we use the Python string's `join()` method to make the required delimited list for the rest.

There's more...

QGIS has its own merge method available through the processing module called `qgis:mergevectorlayers`, but it is limited because it only merges two files. The SAGA method allows any number of files to be merged.

Splitting a shapefile

Sometimes, you need to split a shapefile in order to break a larger dataset into more manageable sizes or to isolate a specific area of interest. There is a script in the Processing Toolbox that splits a shapefile by attribute. It is very useful, even though it is provided as an example of how to write processing scripts.

Getting ready

We will split a census tract shapefile by county. You can download the sample zipped shapefile from https://geospatialpython.googlecode.com/files/GIS_CensusTract.zip.

1. Extract the zipped shapefile to a directory named `/qgis_data/census`.
2. You also need the following script for the Processing Toolbox:
https://geospatialpython.googlecode.com/svn/Split_vector_layer_by_attribute.py
3. Next, use the following steps to add the script to the Processing Toolbox:
4. Download the script to your `/qgis_data/` directory.
5. In the QGIS **Processing Toolbox**, open the **Scripts** tree menu and then go to the **Tools** submenu.
6. Then, double-click on the **Add script from file** command.
7. In the **File** dialog, navigate to the script. Select the **Script** and click on the **Open** button.

The stage is set now. Perform the steps in the next section to split the shapefile.

How to do it...

This recipe is as simple as running the algorithm and specifying the filename and data attribute. Perform the following steps:

1. Start QGIS.
2. From the **Plugins** menu, select **Python Console**.
3. Import the processing module:
`import processing`
4. Define your data directory as a variable to shorten the processing command:
`pth = "/qgis_data/census/"`
5. Finally, run the algorithm:

```
processing.runalg("script:splitvectorlayerbyattribute",pth +  
"GIS_CensusTract_poly.shp","COUNTY_8",pth + "split")
```

How it works...

The algorithm will dump the split files in the data directory, numbered sequentially.

Generalizing a vector layer

Generalizing the geometry, also known as simplifying, removes points from a vector layer to reduce the space required to store the data on disk, the bandwidth needed to move it over a network, and the processing power required to perform analysis with it or display it in QGIS. In many cases, the geometry of a layer contains redundant points along with straight lines that can be removed without changing the spatial properties of a layer, with the exception of topology constraints.

Getting ready

For this recipe, we will use a boundary file for the state of Mississippi, which you can download from <https://geospatialpython.googlecode.com/files/Mississippi.zip>.

Extract the zipped shapefile to a directory named /qgis_data/ms.

How to do it...

Generalizing is native to QGIS, but we will access it in PyQGIS through the Processing Toolbox using the `qgis:simplifygeometries` algorithm, as follows:

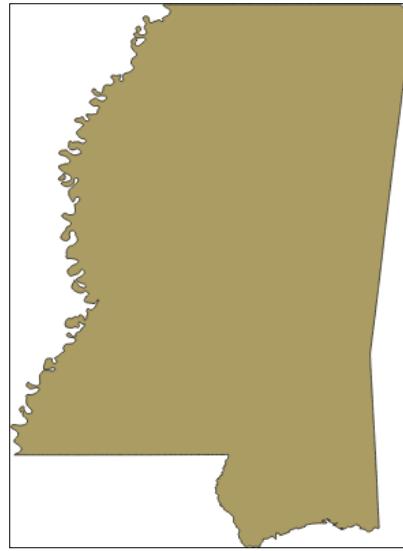
1. Start QGIS.
2. From the **Plugins** menu, select **Python Console**.
3. Import the processing module:
`import processing`
4. Now, run the processing algorithm, specifying the algorithm name, input data, tolerance value, spacing between points — which defines how close two points are in map units before one is deleted — and the output dataset's name:

```
processing.runandload("qgis:simplifygeometries", "/qgis_data/ms/  
mississippi.shp", 0.3, "/qgis_data/ms/generalize.shp")
```

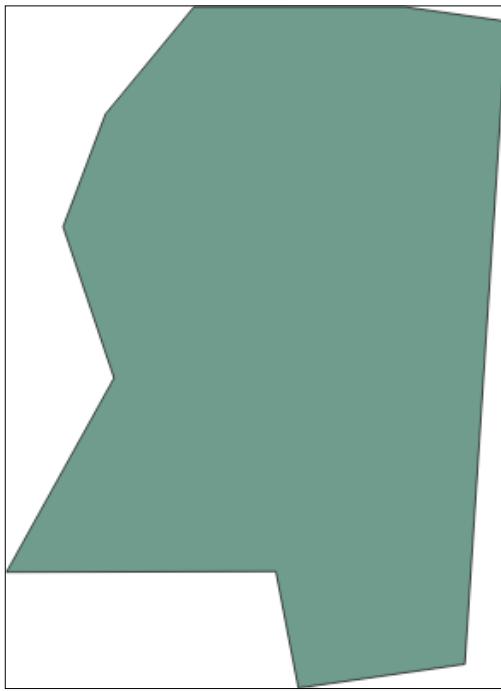
How it works...

The simplicity of the `simplifygeometries` command makes the operation look simple. However, the simplification is itself quite complex. The same settings rarely produce desirable results across multiple datasets.

The shapefile in this recipe starts out quite complex with hundreds of points, as seen in the following visualization:



The simplified version has only 10 points, as seen in the following image:



Dissolving vector shapes

Dissolving shapes can take two different forms. You can combine a group of adjoining shapes by the outermost boundary of the entire dataset, or you can also group the adjoining shapes with the same attribute value.

Getting ready

Download the GIS census tract shapefile, which contains tracts for several counties from https://geospatialpython.googlecode.com/files/GIS_CensusTract.zip.

Extract it to your /qgis_data directory, in a directory called census.

How to do it...

We will use the Processing Toolbox for this recipe and specifically a native QGIS algorithm called `dissolve`, as follows:

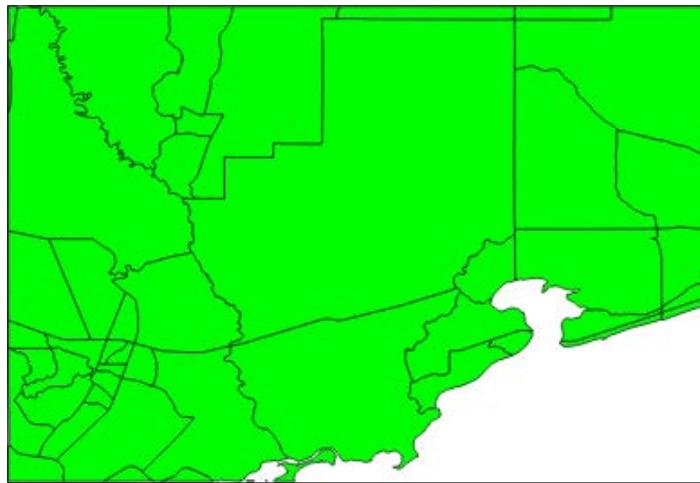
1. Start QGIS.
2. From the **Plugins** menu, select **Python Console**.
3. Import the processing module:
`import processing`
4. Next, run the `dissolve` algorithm, specifying the input data—`False` to specify that we don't want to dissolve all the shapes into one but to use an attribute instead—the attribute we want to use, and the output filename:

```
processing.runandload("qgis:dissolve","/qgis_data/census/  
GIS_CensusTract_poly.shp",False,"COUNTY_8","/qgis_data/census/  
dissolve.shp")
```

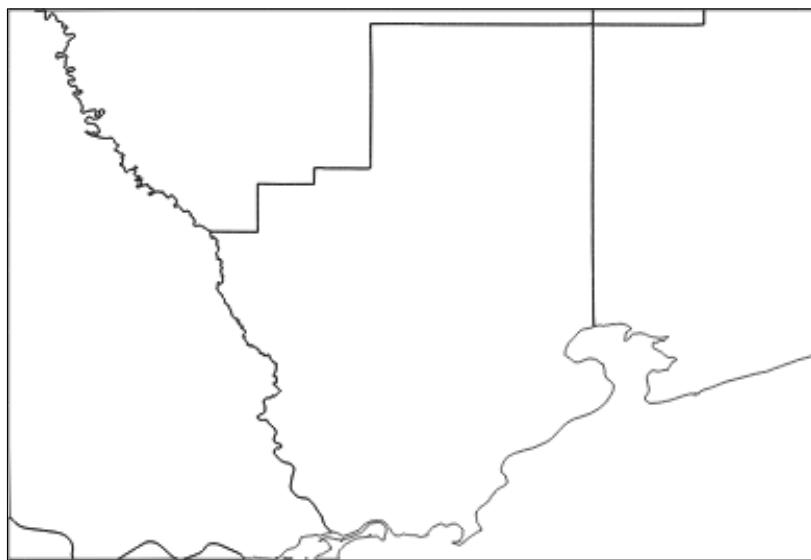
How it works...

By only changing the boolean in the statement to `True`, we can dissolve all adjoining shapes into one. It is also important to note that QGIS will assign the fields of the first shape it encounters in each group to the final shape. In most cases, this will make the attributes virtually useless. This operation is primarily a spatial task.

You can see that each county boundary has a number of census tracts in the original layer, as shown in the following image:



Once the shapes are dissolved, you are left with only the county boundaries, as shown in this image:



Performing a union on vector shapes

A union turns two overlapping shapes into one. This task can be easily accomplished with the Processing Toolbox. In this recipe, we'll merge the outline of a covered building with the footprint of the main building.

Getting ready

You can download the building files from <https://geospatialpython.googlecode.com/svn/union.zip> and extract them to a directory named /qgis_data/union.

How to do it...

All we need to do is run the `qgis:union` algorithm, as follows:

1. Start QGIS.
2. From the **Plugins** menu, select **Python Console**.
3. Import the processing module:
`import processing`

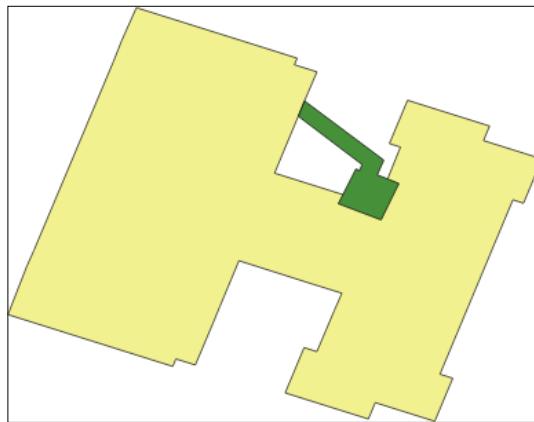
4. Now, run the algorithm by specifying the two input shapes and a single output file:

```
processing.runandload("qgis:union","/qgis_data/union/building.  
shp","/qgis_data/union/walkway.shp","/qgis_data/union/union.shp")
```

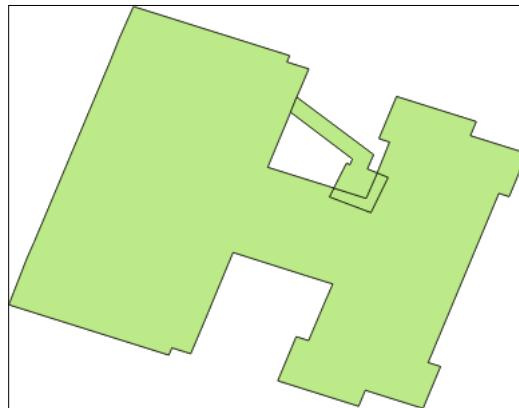
How it works...

As you can tell from the structure of the command, this tool can only combine two shapes at once. It finds where the two shapes meet and then removes the overlap, joining them at the meeting point.

In the original data, the shapefile starts out as two distinct shapes, as shown in this image:



Once the union is complete, the shapes are now one shapefile, with the overlap being a separate feature, as shown in this image:



Rasterizing a vector layer

Sometimes, a raster dataset is the most efficient way to display a complex vector that is merely a backdrop in a map. In these cases, you can rasterize a vector layer to turn it into an image.

Getting ready

We will demonstrate how to rasterize a vector layer using the following contour shapefile, which you can download from <https://geospatialpython.googlecode.com/svn/contour.zip>.

Extract it to your `/qgis_data/rasters` directory.

How to do it...

We will run the `gdalogr:rasterize` algorithm to convert this vector data to a raster, as follows:

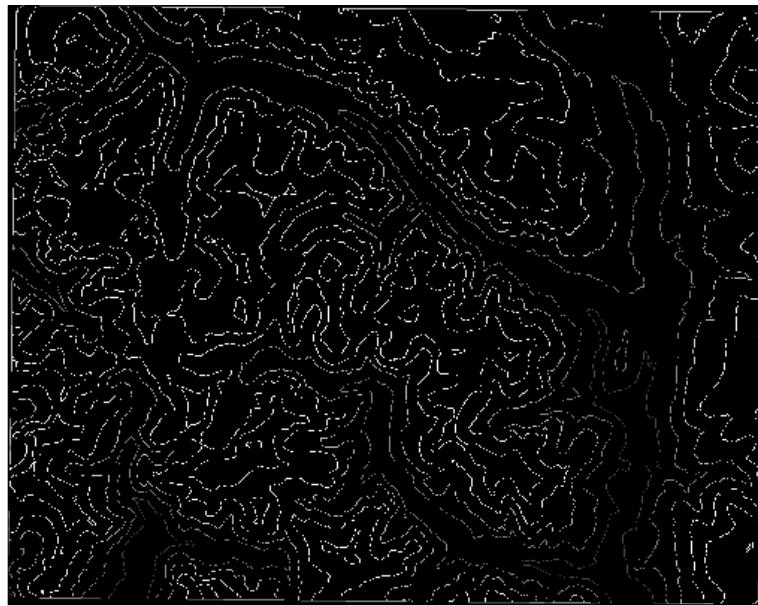
1. Start QGIS.
2. From the **Plugins** menu, select **Python Console**.
3. Import the processing module:
`import processing`
4. Run the algorithm, specifying the input data, the attribute from which raster values need to be drawn, 0 in order to specify pixel dimensions for the output instead of map dimensions, width and height, and finally the output raster name:

```
processing.runalg("gdalogr:rasterize","/qgis_data/rasters/contour.shp","ELEV",0,1000,1000,"/qgis_data/rasters/contour.tif")
```

How it works...

If you want to specify the output dimensions in map units, use 1 instead of 0. Note that the symbology of the layer becomes frozen once you convert it to a raster. The raster is also no longer dynamically scalable.

The following image shows the rasterized output of the elevation contour shapefile:



4

Using Raster Data

In this chapter, we will cover the following recipes:

- ▶ Loading a raster layer
- ▶ Getting the cell size of a raster layer
- ▶ Obtaining the width and height of a raster
- ▶ Counting raster bands
- ▶ Swapping raster bands
- ▶ Querying the value of a raster at a specified point
- ▶ Reprojecting a raster
- ▶ Creating an elevation hillshade
- ▶ Creating vector contours from elevation data
- ▶ Sampling a raster dataset using a regular grid
- ▶ Adding elevation data to line using a digital elevation model
- ▶ Creating a common extent for rasters
- ▶ Resampling raster resolution
- ▶ Counting the unique values in a raster
- ▶ Mosaicing rasters
- ▶ Converting a TIFF image to a JPEG image
- ▶ Creating pyramids for a raster
- ▶ Converting a pixel location to a map coordinate

- ▶ Converting a map coordinate to a pixel location
- ▶ Creating a KML image overlay for a raster
- ▶ Classifying a raster
- ▶ Converting a raster to a vector
- ▶ Georeferencing a raster from ground control points
- ▶ Clipping a raster using a shapefile

Introduction

This chapter shows you how to bring raster data into a GIS and create derivative raster products using QGIS and Python. QGIS is equally adept at working with raster data as with vector data, by incorporating leading-edge open source libraries and algorithms, including GDAL, SAGA, and the Orfeo Toolbox. QGIS provides a consistent interface to a large array of remote sensing tools. We will switch back and forth between visually working with raster data and using QGIS as a processing engine via the Processing Toolbox, to completely automating remote sensing workflows.

Raster data consists of rows and columns of cells or pixels, with each cell representing a single value. The easiest way to think of raster data is as images, which is how they are typically represented by software. However, raster datasets are not necessarily stored as images. They can also be ASCII text files or **binary large objects (BLOBs)** in databases.

Another difference between geospatial raster data and regular digital images is their resolution. Digital images express resolution as dots-per-inch, if they are printed in full size. Resolution can also be expressed as the total number of pixels in the image, defined as megapixels. However, geospatial raster data uses the ground distance that each cell represents. For example, a raster dataset with a two-feet resolution means that a single cell represents two feet on the ground. This also means that only objects larger than two feet can be identified visually in the dataset.

Raster datasets may contain multiple bands, meaning that different wavelengths of light can be collected at the same time over the same area. Often, this range is from 3 to 7 bands wide, but it can be several hundred bands wide in hyperspectral systems. These bands are viewed individually or swapped in and out as the RGB bands of an image. They can also be recombined using mathematics into a derived single band image and then recolored using a set number of classes, representing similar values within the dataset.

Loading a raster layer

The QGSRasterLayer API provides a convenient, high-level interface to raster data. To use this interface, we must load a layer into QGIS. The API allows you to work with a layer without adding it to the map. In this way, we'll load layer and then add it to the map.

Getting ready

As with the other recipes in this book, you need to create a directory called `qgis_data` in our root or user directory, which provides a short pathname without spaces. This setup will help prevent any frustrating errors that result from path-related issues on a given system. In this recipe, and the others, we'll use a Landsat satellite image of the Mississippi Gulf Coast, which you can download from <https://geospatialpython.googlecode.com/files/SatImage.zip>.

Unzip the `SatImage.tif` and `SatImage.tfw` files and place them in a directory named `rasters` within your `qgis_data` directory.

How to do it...

Now, we'll go through how to load a raster layer and then step by step add it to the map

1. Start QGIS.
2. From the **Plugins** menu, select **Python Console**.
3. Then, in the **Python Console**, create the layer by specifying the source file and a layer name:

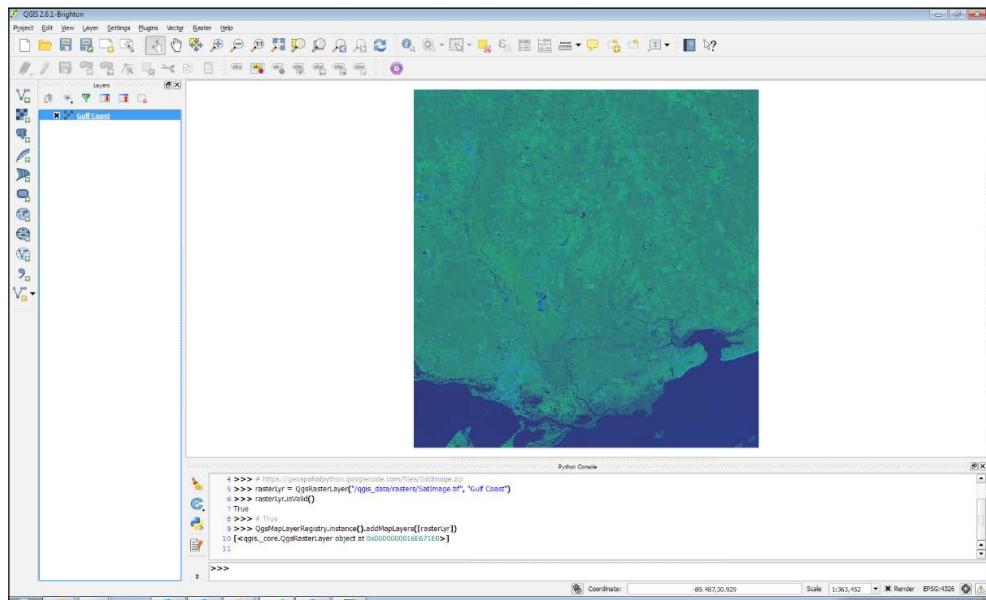
```
rasterLyr = QgsRasterLayer("/qgis_data/rasters/SatImage.tif",
    "Gulf Coast")
```

4. Next, ensure that the layer is created as expected. The following command should return True:

```
rasterLyr.isValid()
```
5. Finally, add the layer to the layer registry:

```
QgsMapLayerRegistry.instance().addMapLayers([rasterLyr])
```

6. Verify that your QGIS map looks similar to the following image:



QGIS zooms to the extent of the raster layer when it is loaded as shown in this example of a Landsat satellite image of the Mississippi Gulf Coast

How it works...

The `QgsRasterLayer` object requires the location of the file and a name for the layer in QGIS. The underlying GDAL library determines the appropriate method of loading the layer. This approach contrasts with the `QgsVectorLayer()` method, which requires you to specify a data provider. Raster layers also have a data provider, but unlike vector layers, all raster layers are managed through GDAL. One of the best features of QGIS is that it combines the best of breed open source geospatial tools into one package. GDAL can be used as a library as we are using it here from Python or as a command-line tool.

Once we have created the `QgsRasterLayer` object, we do a quick check using the `rasterLayer.isValid()` method to see whether the file was loaded properly. This method will return `True` if the layer is valid. We won't use this method in every recipe; however, it is a best practice, especially when building dynamic applications that accept user input. Because most of the PyQGIS API is built around C libraries, many methods do not throw exceptions if an operation fails. You must use specialized methods to verify the output.

Finally, we add the layer to the map layer registry, which makes it available on the map and in the legend. The registry keeps track of all the loaded layers by separating, loading, and visualizing the layers. QGIS allows you to work behind the scenes in order to perform unlimited intermediate processes on a layer before adding the final product to the map.

Getting the cell size of a raster layer

The first key element of a geospatial raster is the width and height, in pixels. The second key element is the ground distance of each pixel, also called the pixel size. Once you know the cell size and a coordinate somewhere on the image (usually the upper-left corner), you can begin using remote sensing tools on the image. In this recipe, we'll query the cell size of a raster.

Getting ready

Once again, we will use the SatImage raster available at <https://geospatialpython.googlecode.com/files/SatImage.zip>.

Place this raster in your `/qgis_data/rasters` directory.

How to do it...

We will load the raster as a layer and then use the `QgsRasterLayer` API to get the cell size for the x and y axis. To do this, we need to perform the following steps:

1. Start QGIS.
2. From the **Plugins** menu, select **Python Console**.
3. Load the layer and validate it:

```
rasterLyr = QgsRasterLayer("/qgis_data/rasters/satimage.tif",
"Sat Image")
rasterLyr.isValid()
```
4. Now, call the x distance method, which should return 0.00029932313140079714:

```
rasterLyr.rasterUnitsPerPixelX()
```
5. Then, call the y distance, which should be 0.00029932313140079714:

```
rasterLyr.rasterUnitsPerPixelY()
```

How it works...

GDAL provides this information, which is passed through to the layer API. Note that while the x and y values are essentially the same in this case, it is entirely possible for the x and y distances to be different—especially if an image is projected or warped in some way.

Obtaining the width and height of a raster

All raster layers have a width and height in pixels. Because remote sensing data can be considered an image as well as an array or matrix, you will often see different terms used, including columns and rows or pixels and lines. These different terms surface many times within the QGIS API.

Getting ready

We will use the SatImage raster again, which is available at <https://geospatialpython.googlecode.com/files/SatImage.zip>.

Place this raster in your /qgis_data/rasters directory.

How to do it...

1. Start QGIS.
2. From the **Plugins** menu, select **Python Console**.
3. In the Python Console, load the layer and ensure that it is valid:

```
rasterLyr = QgsRasterLayer("/qgis_data/rasters/satimage.tif",
"satimage")
rasterLyr.isValid()
```

Check the name of SatImage after unzipping.

4. Obtain the layer's width, which should be 2592:

```
rasterLyr.width()
```

5. Now, get the raster's height, which will return 2693:

```
rasterLyr.height()
```

How it works...

The width and height of a raster are critical pieces of information for many algorithms, including calculating the map units that the raster occupies.

Counting raster bands

A raster might have one or more bands. Bands represent layers of information within a raster. Each band has the same number of columns and rows.

Getting ready

We will again use the SatImage raster available at <https://geospatialpython.googlecode.com/files/SatImage.zip>.

Place this raster in your /qgis_data/rasters directory.

How to do it...

We will load the layer and then print the band count to the console. To do this, we need to perform the following steps:

1. Start QGIS.
2. From the **Plugins** menu, select **Python Console**.
3. In the Python Console, load the layer and ensure that it is valid:

```
rasterLyr = QgsRasterLayer("/qgis_data/rasters/satimage.tif",
    "Sat Image")
rasterLyr.isValid()
```
4. Now, get the band count, which should be 3 in this case:

```
rasterLyr.bandCount()
```

How it works...

It is important to note that raster bands are not zero-based indexes. When you want to access the first band, you reference it as 1 instead of 0. Most sequences within a programming context start with 0.

Swapping raster bands

Computer displays render images in the visible spectrum of red, green, and blue light (RGB). However, raster images may contain bands outside the visible spectrum. These types of rasters make poor visualizations, so you will often want to recombine the bands to change the RGB values.

Getting ready

For this recipe, we will use a false-color image, which you can download from <https://geospatialpython.googlecode.com/files/FalseColor.zip>.

Unzip this `.tif` file and place it in your `/qgis_data/rasters` directory.

How to do it...

We will load this raster and swap the order of the first and second bands. Then, we will add it to the map. To do this, we need to perform the following steps:

1. Start QGIS.
2. From the **Plugins** menu, select **Python Console**.
3. In the **Python Console**, load the layer and ensure that it is valid:

```
rasterLyr =  
QgsRasterLayer("/qgis_data/rasters/FalseColor.tif", "Band  
Swap")  
rasterLyr.isValid()
```

4. Now, we must access the layer renderer in order to manipulate the order of the bands displayed. Note that this change does not affect the underlying data:

```
ren = rasterLyr.renderer()
```

5. Next, we will set the `red` band to band 2:

```
ren.setRedBand(2)
```

6. Now, we will set the `green` band to band 1:

```
ren.setGreenBand(1)
```

7. Finally, add the altered raster layer to the map:

```
QgsMapLayerRegistry.instance().addMapLayers([rasterLyr])
```

How it works...

Load the source image into QGIS as well to compare the results. In the false-color image, vegetation appears red, while in the band-swapped image, trees appear a more natural green and the water is blue. QGIS uses the RGB order to allow you to continue to reference the bands by number. Even though band 2 is displayed first, it is still referenced as band 2. Also, notice that the band order is controlled by a `QgsMultiBandColorRenderer` object instantiated by the layer rather than the layer itself. The type of renderer that is needed is determined at load time by the data type and number of bands.

There's more...

The `QgsMultiBandColorRenderer()` method has other methods to control contrast enhancement for each band, such as `setRedContrastEnhancement()`. You can learn more about raster renderers for different types of data in the QGIS API documentation at <http://qgis.org/api/classQgsRasterRenderer.html>.

Querying the value of a raster at a specified point

A common remote sensing operation is to get the raster data value at a specified coordinate. In this recipe, we'll query the data value in the center of the image. It so happens that the raster layer will calculate the center coordinate of its extent for you.

Getting ready

As with many recipes in this chapter, we will again use the `SatImage` raster, which is available at <https://geospatialpython.googlecode.com/files/SatImage.zip>.

Place this raster in your `/qgis_data/rasters` directory.

How to do it...

We will load the layer, get the center coordinate, and then query the value. To do this, we need to perform the following steps:

1. First, load and validate the layer:

```
rasterLyr = QgsRasterLayer("/qgis_data/rasters/satimage.tif",
    "Sat Image")
rasterLyr.isValid()
```

2. Next, get the layer's center point from its `QgsRectangle extent` object, which will return a tuple with the x and y values:

```
c = rasterLyr.extent().center()
```

3. Now, using the layer's data provider, we can query the data value at that point using the `identify()` method:

```
qry = rasterLyr.dataProvider().identify(c,  
QgsRaster.IdentifyFormatValue)
```

4. Because a query error won't throw an exception, we must validate the query:

```
qry.isValid()
```

5. Finally, we can view the query results, which will return a Python dictionary with each band number as the key pointing to the data values in that band:

```
qry.results()
```

6. Verify that you get the following output:

```
{1: 17.0, 2: 66.0, 3: 56.0}
```

How it works...

This recipe is short compared to others, however, we have touched upon several portions of the PyQGIS raster API. First start with a raster layer and get the extents; we then calculate the center and create a point at the center coordinates, and lastly we query the raster at that point. If we were to perform this same, seemingly simple operation using the Python API of the underlying GDAL library, which does the work, this example would have been approximately seven times longer.

Reprojecting a raster

A core requirement for all geospatial analysis is the ability to change the map projection of data in order to allow different layers to be open on the same map. Reprojection can be challenging, but QGIS makes it a snap of the fingers. Starting with this recipe, we will begin using the powerful QGIS Processing Toolbox. The Processing Toolbox wraps over 600 algorithms into a highly consistent API, available to Python and also as interactive tools. This toolbox was originally a third-party plugin named SEXTANTE, but is now a standard plugin distributed with QGIS.

Getting ready

As with many recipes in this chapter, we will use the SatImage raster available at <https://geospatialpython.googlecode.com/files/SatImage.zip>.

Place this raster in your `/qgis_data/rasters` directory.

How to do it...

In this recipe, we will use the `gdal warp` algorithm of the processing module to reproject our image from EPSG 4326 to 3722. To do this, we need to perform the following steps:

1. Start QGIS.
2. From the **Plugins** menu, select **Python Console**.
3. The first line of code is used to import the processing module:

```
import processing
```
4. Next, we load our raster layer and validate it:

```
rasterLyr = QgsRasterLayer("/qgis_data/rasters/SatImage.tif",
    "Reproject")
rasterLyr.isValid()
```
5. Finally, we run the `gdal warp` algorithm by inserting the correct parameters, including the layer reference, current projection, desired projection, None for changes to the resolution, 0 to represent nearest neighbor resampling, None for additional parameters, 0 -Byte output raster data type (1 for int16), and an output name for the reprojected image:

```
processing.runalg("gdalogr:warpreproject", rasterLyr,
    "EPSG:4326", "EPSG:3722", None, 0, None, "/0,
    qgis_data/rasters/warped.tif")
```
6. Verify that the output image, `warped.tif`, was properly created in the filesystem.

How it works...

The Processing Toolbox is essentially a wrapper for command-line tools. However, unlike the tools it accesses, the toolbox provides a consistent and mostly predictable API. Users familiar with Esri's ArcGIS ArcToolbox will find this approach familiar. Besides consistency, the toolbox adds additional validation of parameters and logging, making these tools more user friendly. It is important to remember that you must explicitly import the `processing` module. PyQGIS automatically loads the QGIS API, but this module is not yet included. Remember that it was a third-party plugin until fairly recently.

There's more...

The `runalg()` method, short for the `run` algorithm, is the most common way to run processing commands. There are other processing methods that you can use though. If you want to load the output of your command straight into QGIS, you can swap `runalg()` for the `runandload()` method. All arguments to the method remain the same. You can also get a list of processing algorithms with descriptions by running `processing.alglist()`. For any given algorithm, you can run the `alghelp()` command to see the types of input it requires, such as `processing.alghelp("gdalogr:warpproject")`. You can also write your own processing scripts based on combinations of algorithms and add them to the processing toolbox. There is also a visual modeler for chaining processing commands together.

Creating an elevation hillshade

A hillshade, or shaded relief, is a technique to visualize elevation data in order to make it photorealistic for presentation as a map. This capability is part of GDAL and is available in QGIS in two different ways. It is a tool in the **Terrain Analysis** menu under the **Raster** menu and it is also an algorithm in the Processing Toolbox.

Getting ready

You will need to download a DEM from <https://geospatialpython.googlecode.com/files/dem.zip>.

Unzip the file named `dem.asc` and place it in your `/qgis_data/rasters` directory.

How to do it...

In this recipe, we will load the DEM layer and run the `Hillshade` processing algorithm against it. To do this, we need to perform the following steps:

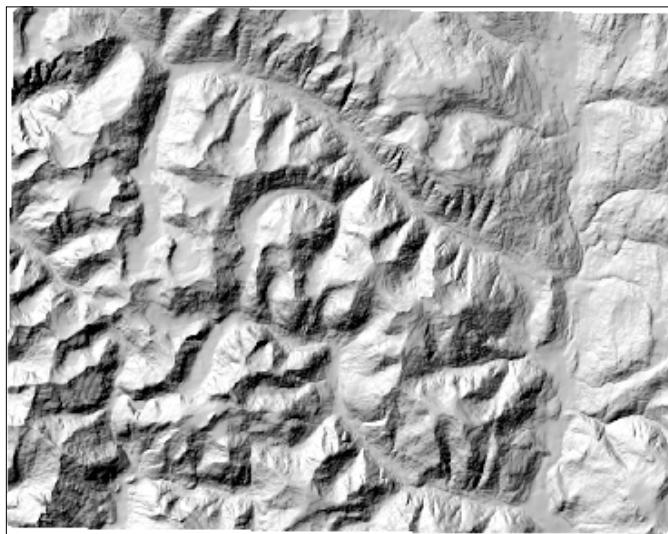
1. Start QGIS.
2. From the **Plugins** menu, select **Python Console**.
3. Import the processing module:
`import processing`
4. Load and validate the layer:

```
rasterLyr = QgsRasterLayer("/qgis_data/rasters/dem.asc",
    "Hillshade")
rasterLyr.isValid()
```

5. Run the Hillshade algorithm, providing the algorithm name, layer reference, band number, compute edges option, zevenbergen option for smoother terrain, z-factor elevation exaggeration number, scaling ratio of vertical to horizontal units, azimuth (angle of the light source), altitude (height of the light source), and output image's name:

```
processing.runandload("gdalogr:hillshade", rasterLyr, 1,  
False, False, 1.0, 1.0, 315.0, 45.0,  
"/qgis_data/rasters/hillshade.tif")
```

6. Verify that the output image, hillshade.tif, looks similar to the following image in QGIS. It should be automatically loaded into QGIS via the processing.runandload() method:



How it works...

The Hillshade algorithm simulates a light source over an elevation dataset to make it more visually appealing. Most of the time, the only variables in the algorithm you need to alter are the z-factor, azimuth, and altitude to get different effects. However, if the resulting image doesn't look right, you may need to alter the scale. According to the GDAL documentation, if your DEM is in degrees, you should set a scale of 111120, and if it is in meters, you should set a scale of 370400. This dataset covers a small area such that a scale of 1 is sufficient. For more information on these values, see the gdaldem documentation at <http://www.gdal.org/gdaldem.html>.

Creating vector contours from elevation data

Contours provides an effective visualization of terrain data by tracing a line along the same elevation to form a loop at set intervals in the dataset. Similar to the hillshade capability in QGIS, the **Contour** tool is provided by GDAL both as a menu option under the **Raster** menu in the **Extraction category** as well as a Processing Toolbox algorithm.

Getting ready

This recipe uses the DEM from <https://geospatialpython.googlecode.com/files/dem.zip>, which is used in the other recipes as well.

Unzip the file named `dem.asc` and place it in your `/qgis_data/rasters` directory.

How to do it...

In this recipe, we will load and validate the DEM layer, add it to the map, and then produce and load the contour vector as a layer. To do this, we need to perform the following steps:

1. Start QGIS.
2. From the **Plugins** menu, select **Python Console**.
3. Import the processing module.

```
import processing
```

4. Load and validate the DEM:

```
rasterLyr = QgsRasterLayer("/qgis_data/rasters/dem.asc",
"DEM")
rasterLyr.isValid()
```

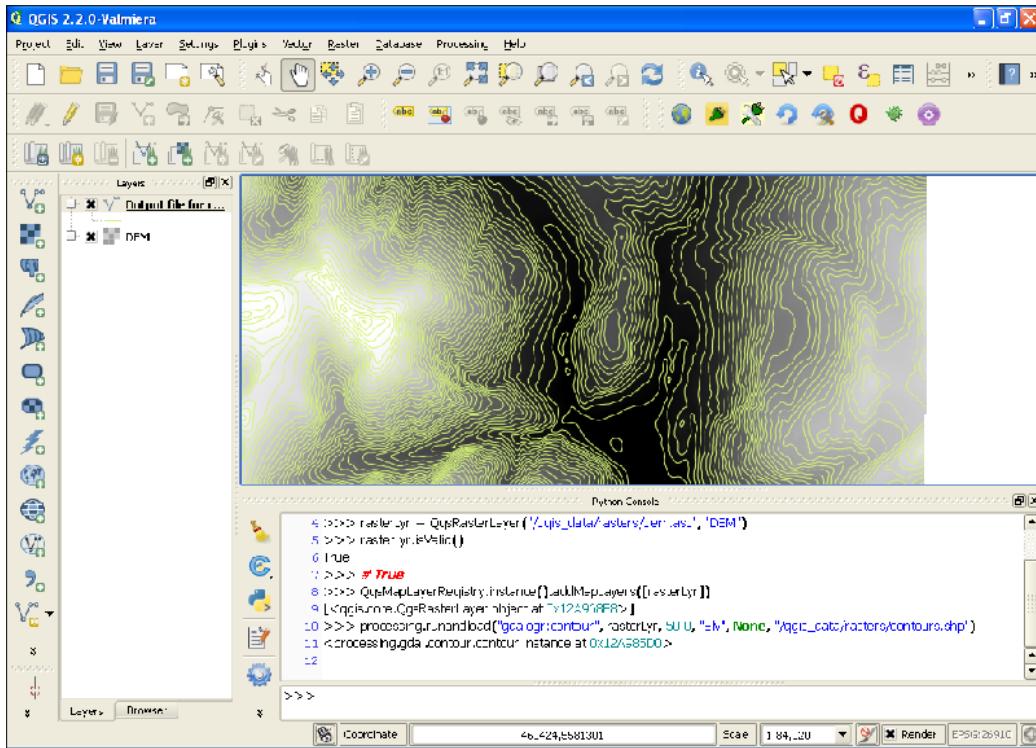
5. Add the DEM to the map using the `mapLayerRegistry` method:

```
QgsMapLayerRegistry.instance().addMapLayers([rasterLyr])
```

6. Run the contour algorithm and draw the results on top of the DEM layer, specifying the algorithm name, layer reference, interval between contour lines in map units, name of the vector data attribute field that will contain the elevation value, any extra parameters, and output filename:

```
processing.runandload("gdalogr:contour", rasterLyr, 50.0,
"Elv", None, "/qgis_data/rasters/contours.shp")
```

7. Verify that the output in QGIS looks similar to the following screenshot:



This recipe overlays the resulting elevation contours over the DEM as a way to convert elevation data into a vector data set.

How it works...

The contour algorithm creates a vector dataset, that is a shapefile. The layer attribute table contains the elevation values for each line. Depending on the resolution of the elevation dataset, you may need to change the contour interval to stop the contours from becoming too crowded or too sparse at your desired map resolution. Usually, autogenerated contours like this are a starting point, and you must manually edit the result to make it visually appealing. You may want to smoothen lines or remove unnecessary small loops.

Sampling a raster dataset using a regular grid

Sometimes, you need to sample a raster dataset at regular intervals in order to provide summary statistics or for quality assurance purposes on the raster data. A common way to accomplish this regular sampling is to create a point grid over the dataset, query the grid at each point, and assign the results as attributes to those points. In this recipe, we will perform this type of sampling over a satellite image. QGIS has a tool to perform this operation called regular points, which is in the **Vector** menu under **Research Tools**. However, there is no tool in the QGIS API to perform this operation programmatically. However, we can implement this algorithm directly using Python's `numpy` module.

Getting ready

In this recipe, we will use the previously used SatImage raster, available at <https://geospatialpython.googlecode.com/files/SatImage.zip>.

Place this raster in your `/qgis_data/rasters` directory.

How to do it...

The order of operation for this recipe is to load the raster layer, create a vector layer in memory, add points at regular intervals, sample the raster layer at these points, and then add the sampling data as attributes for each point. To do this, we need to perform the following steps:

1. Start QGIS.
2. From the **Plugins** menu, select **Python Console**.
3. We will need to import the `numpy` module, which is included with QGIS, as well as the Qt core module:

```
import numpy  
from PyQt4.QtCore import *
```

4. Now, we will create a `spacing` variable to control how far apart the points are in map units:

```
spacing = .1
```

5. Next, we will create an `inset` variable to determine how close to the edge of the image the points start, in map units:

```
inset = .04
```

6. Now, we load and validate the raster layer:

```
rasterLyr = QgsRasterLayer("/qgis_data/rasters/satimage.tif",
    "Sat Image")
rasterLyr.isValid()
```

7. Next, we collect the coordinate reference system and extent from the raster layer in order to transfer it to the point layer:

```
rpr = rasterLyr.dataProvider()
epsg = rasterLyr.crs().postgisSrid()
ext = rasterLyr.extent()
```

8. Now, we create an in-memory vector point layer, which won't be written to disk:

```
vectorLyr = QgsVectorLayer('Point?crs=epsg:%s' % epsg, 'Grid',
    "memory")
```

9. In order to add points to the vector layer, we must access its data provider:

```
vpr = vectorLyr.dataProvider()
qd = QVariant.Double
```

10. Next, we create the attributes' fields to store the raster data samples:

```
vpr.addAttribute([QgsField("Red", qd), QgsField("Green", qd),
    QgsField("Blue", qd)])
vectorLyr.updateFields()
```

11. We use the `inset` variable to set up the layer's extents inside the raster layer:

```
xmin = ext.xMinimum() + inset
xmax = ext.xMaximum()
ymin = ext.yMinimum() + inset
ymax = ext.yMaximum() - inset
```

12. Now, we use the `numpy` module to efficiently create the coordinates of the points in our regular grid:

```
pts = [(x,y) for x in (i for i in numpy.arange(xmin, xmax,
    spacing)) for y in (j for j in numpy.arange(ymin, ymax,
    spacing))]
```

13. Then, we create a list to store the point features we will create:

```
feats = []
```

14. In one loop, we create the point features, query the raster, and then update the attribute table. We store the points in a list for now:

```
for x,y in pts:
    f = QgsFeature()
    f.initAttributes(3)
    p = QgsPoint(x,y)
    qry = rasterLyr.dataProvider().identify(p,
QgsRaster.IdentifyFormatValue)
    r = qry.results()
    f.setAttribute(0, r[1])
    f.setAttribute(1, r[2])
    f.setAttribute(2, r[3])
    f.setGeometry(QgsGeometry.fromPoint(p))
    feats.append(f)
```

15. Next, we pass the list of points to the data provider of the points layer:

```
vpr.addFeatures(feats)
```

16. Now, we update the layer's extents:

```
vectorLyr.updateExtents()
```

17. Then, we add both the raster and vector layers to the map in the list. The last item in the list is on top:

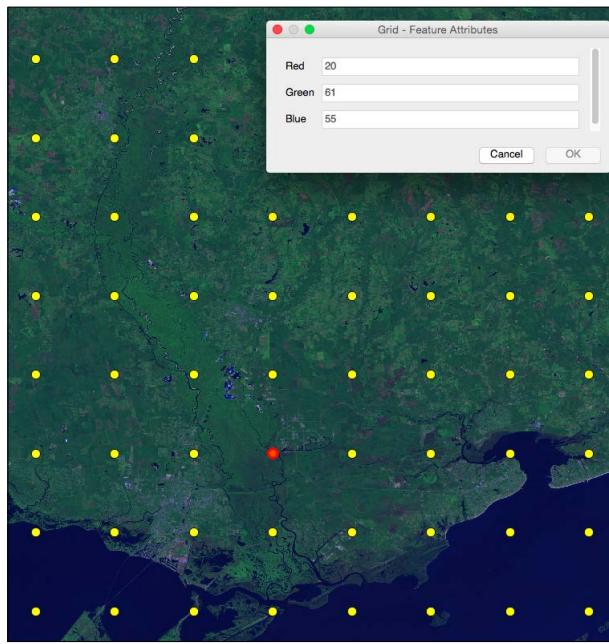
```
QgsMapLayerRegistry.instance().addMapLayers([rasterLyr,vector
lyr])
```

18. Finally, we refresh the map to see the result:

```
canvas = iface.mapCanvas()
canvas.setExtent(rasterLyr.extent())
canvas.refresh()
```

How it works...

The following screenshot shows the end result, with one of the points in the grid identified using the **Identify Features** map tool. The results dialog shows the raster values of the selected point:



When you use the QGIS Identification Tool to click on one of the points, the results dialog shows the extracted Red, Green, and Blue values from the image.

Using memory layers in QGIS is an easy way to perform quick, one-off operations without the overhead of creating files on disk. Memory layers also tend to be fast if your machine has the resources to spare.

There's more...

In this example, we used a regular grid, but we could have just as easily modified the numpy-based algorithm to create a random points grid, which in some cases is more useful. However, the Processing Toolbox also has a simple algorithm for random points called `grass:v.random`.

Adding elevation data to line vertices using a digital elevation model

If you have a transportation route through some terrain, it is useful to know the elevation profile of that route. This operation can be accomplished using the points that make up the line along the route to query a DEM and to assign elevation values to that point. In this recipe, we'll do exactly that.

Getting ready

You will need an elevation grid and a route. You can download this dataset from <https://geospatialpython.googlecode.com/svn/path.zip>.

Unzip the `path` directory containing a shapefile and the elevation grid. Place the whole `path` directory in your `qgis_data/rasters` directory.

How to do it...

We will need two processing algorithms to complete this recipe. We will load the raster and vector layers, convert the line feature to points, and then use these points to query the raster. The resulting point dataset will serve as the elevation profile for the route. To do this, we need to perform the following steps:

1. Start QGIS.
2. From the **Plugins** menu, select **Python Console**.
3. Import the processing module:
`import processing`
4. Set up the filenames as variables, so they can be used throughout the script:
`pth = "/qgis_data/rasters/path/"
rasterPth = pth + "elevation.asc"
vectorPth = pth + "path.shp"
pointsPth = pth + "points.shp"
elvPointsPth = pth + "elvPoints.shp"`
5. Load and validate the source layers:
`rasterLyr = QgsRasterLayer(rasterPth, "Elevation")
rasterLyr.isValid()
vectorLyr = QgsVectorLayer(vectorPth, "Path", "ogr")
vectorLyr.isValid()`

6. Add the layers to the map:

```
QgsMapLayerRegistry.instance().addMapLayers([vectorLyr,  
rasterLyr])
```

7. Create an intermediate point dataset from the line using a SAGA algorithm in the Processing Toolbox:

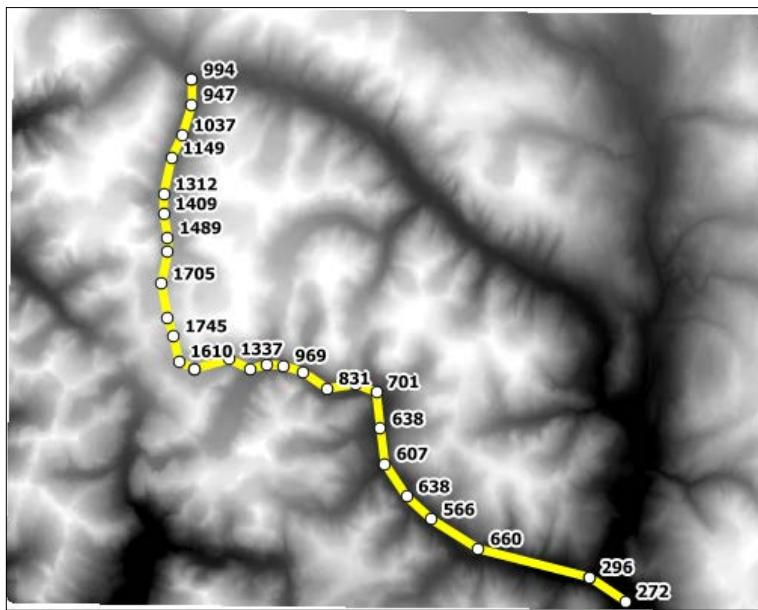
```
processing.runalg("saga:convertlinestopoints", vectorLyr,  
False, 1, pointsPth)
```

8. Finally, use another processing algorithm from SAGA to create the final dataset with the grid values assigned to the points:

```
processing.runandload("saga:addgridvaluestopoints", pointsPth,  
rasterPth, 0, elvPointsPth)
```

How it works...

The following image saved from QGIS shows the DEM, route line, and elevation points with elevation labels, all displayed on the map, with some styling:



It is necessary to convert the lines to points because a line feature can only have one set of attributes. You can perform the same operation with a polygon as well.

There's more...

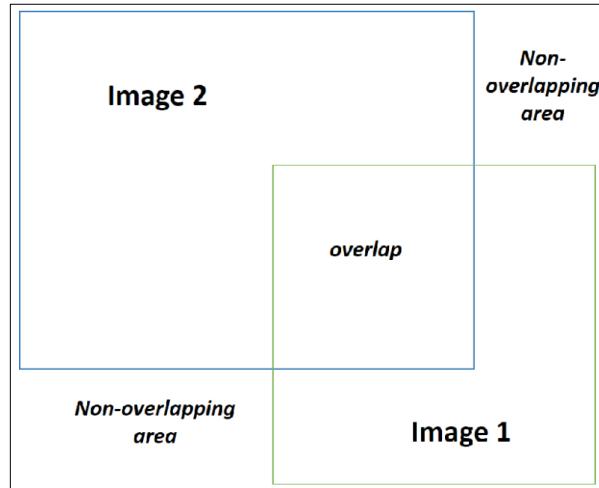
Instead of running two algorithms, we can build a processing script that combines these two algorithms into one interface and then add it to the toolbox. In the Processing Toolbox, there is a category called **Scripts**, which has a tool called **Create new script**. Double-clicking on this tool will bring up an editor that lets you build your own processing scripts. Depending on your platform, you may need to install or configure SAGA to use this algorithm. You can find binary packages for Linux at <http://sourceforge.net/p/saga-gis/wiki/Binary%20Packages/>.

Also, on Linux, you may need to change the following option:

1. In the **Processing** menu, select **Options....**
2. In the **Options** dialog, open the **Providers** tree menu and then open the **Saga** tree menu.
3. Uncheck **the Use 2.0.8 syntax** option.

Creating a common extent for rasters

If you are trying to compare two raster images, it is important that they have the same extent and resolution. Most software packages won't even allow you to attempt to compare images if they don't have the same extent. Sometimes, you have images that overlap but do not share a common extent and/or are of different resolutions. The following illustration is an example of this scenario:



In this recipe, we'll take two overlapping images and give them the same extents.

Getting ready

You can download two overlapping images from <https://geospatialpython.googlecode.com/svn/overlap.zip>.

Unzip the images and place them in your `/qgis_data/rasters` directory.

You will also need to download the following processing script from:

https://geospatialpython.googlecode.com/svn/unify_extents.zip

Unzip the contents and place the scripts in your `\.qgis2\processing\scripts` directory, found within your user directory. For example, on a Windows 64-bit machine, the directory will be `C:\Users\<username>\.qgis2\processing\scripts`, replacing `<username>` with your username.

Make sure you restart QGIS. This script is a modified version of the one created by Yury Ryabov on his blog at <http://ssrebelious.blogspot.com/2014/01/unifying-extent-and-resolution-of.html>.

The original script used a confirmation dialog that required user interaction. The modified script adheres to the Processing Toolbox programming conventions and allows you to use it programmatically as well.

How to do it...

The only step in QGIS is to run the newly created processing command. To do this, we need to perform the following steps:

1. Start QGIS.
2. From the **Plugins** menu, select **Python Console**.
3. Import the processing module:
`import processing`
4. Run the newly added processing algorithm, specifying the algorithm name, path to the two images, an optional no data value, an output directory for the unified images, and a Boolean flag to load the images into QGIS:
`processing.runalg("script:unifyextentandresolution","/qgis_data/rasters/Image2.tif;/qgis_data/rasters/Imagen1.tif",-9999,"/qgis_data/rasters",True)`
5. In the QGIS table of contents, verify that you have two images named:

`Imagen1_unified.tif`
`Image2_unfied.tif`

How it works...

The following screenshot shows the common extent for the rasters, by setting the transparency of `Image1_unified.tif` to the pixel `0,0,0`:



If you don't use the transparency setting, you will see that both images fill the non-overlapping areas with no data within the minimum bounding box of both extents. The no data values, specified as `-9999`, will be ignored by other processing algorithms.

Resampling raster resolution

Resampling an image allows you to change the current resolution of an image to a different resolution. Resampling to a lower resolution, also known as downsampling, requires you to remove pixels from the image while maintaining the geospatial referencing integrity of the dataset. In the QGIS Processing Toolbox, the `gdalogr:warpproject` algorithm is used, which is the same as the algorithm used for reprojection.

Getting ready

We will again use the SatImage raster available at <https://geospatialpython.googlecode.com/files/SatImage.zip>.

Place this raster in your `/qgis_data/rasters` directory.

How to do it...

There's an extra step in this process, where we will get the current pixel resolution of the raster as a reference to calculate the new resolution and pass it to the algorithm. To do this, we need to perform the following steps:

1. Start QGIS.
2. From the **Plugins** menu, select **Python Console**.
3. Import the processing module:
`import processing`
4. Load and validate the raster layer:
`rasterLyr = QgsRasterLayer("/qgis_data/rasters/SatImage.tif",
"Resample")
rasterLyr.isValid()`
5. The algorithm requires projection information. We are not changing it, so just assign the current projection to a variable:
`epsg = rasterLyr.crs().postgisSrid()
srs = "EPSG:%s" % epsg`
6. Get the current pixel's ground distance and multiply it by 2 to calculate half the ground resolution. We only use the X distance because in this case, it is identical to the Y distance:
`res = rasterLyr.rasterUnitsPerPixelX() * 2`
7. Run the resampling algorithm, specifying the algorithm name, layer reference, input and then output spatial reference system, desired resolution, resampling algorithm (0 is the nearest neighbor), any additional parameters, 0 for output raster data type, and the output filename:
`processing.runalg("gdalogr:warpreproject", rasterLyr, srs,
srs, res, 0, None, 0, "/qgis_data/rasters/resampled.tif")`
8. Verify that the `resampled.tif` image was created in your `/qgis_data/rasters` directory.

How it works...

It is counterintuitive at first to reduce the resolution by multiplying it. However, by increasing the spatial coverage of each pixel, it takes less pixels to cover the extent of the raster. You can easily compare the difference between the two in QGIS visually by loading both the images and zooming to an area with buildings or other detailed structures and then turning one layer off or on.

Counting the unique values in a raster

Remotely-sensed images are not just pictures; they are data. The value of the pixels has meaning that can be automatically analyzed by a computer. The ability to run statistical algorithms on a dataset is key to remote sensing. This recipe counts the number of unique combinations of pixels across multiple bands. A use case for this recipe will be to assess the results of image classification, which is a recipe that we'll cover later in this chapter. This recipe is in contrast to the typical histogram function, which totals the unique values and the frequency of each value per band.

Getting ready

We will use the SatImage raster available at <https://geospatialpython.googlecode.com/files/SatImage.zip>.

Place this raster in your `/qgis_data/rasters` directory.

How to do it...

This algorithm relies completely on the `numpy` module, which is included with PyQGIS. Numpy can be accessed through the GDAL package's `gdalnumeric` module. To do this, we need to perform the following steps:

1. Start QGIS.
2. From the **Plugins** menu, select **Python Console**.
3. First, we must import the bridge module called `gdalnumeric`, which connects GDAL to Numpy in order to perform an array math on geospatial images:

```
import gdalnumeric

4. Now, we will load our raster image directly into a multidimensional array:
```

```
a = gdalnumeric.LoadFile("/qgis_data/rasters/satimage.tif")
```

5. The following code counts the number of pixel combinations in the image:

```
b = a.T.ravel()
c=b.reshape((b.size/3,3))
order = gdalnumeric.numpy.lexsort(c.T)
c = c[order]
diff = gdalnumeric.numpy.diff(c, axis=0)
ui = gdalnumeric.numpy.ones(len(c), 'bool')
ui[1:] = (diff != 0).any(axis=1)
u = c[ui]
```

6. Now, we can take a look at the size of the resulting one-dimensional array to get the unique values count:

```
u.size
```

Lastly, verify that the result is 16085631.

How it works...

The `numpy` module is an open source equivalent of the commercial package Matlab. You can learn more about Numpy at: <http://Numpy.org>.

When you load an image using Numpy, it is loaded as a multidimensional array of numbers. Numpy allows you to do an array math on the entire array using operators and specialized functions, in the same way you would on variables containing a single numeric value.

Mosaicing rasters

Mosaicing rasters is the process of fusing multiple geospatial images with the same resolution and map projection into one raster. In this recipe, we'll combine two overlapping satellite images into a single dataset.

Getting ready

You will need to download the overlapping dataset from <https://geospatialpython.googlecode.com/svn/overlap.zip> if you haven't downloaded it from a previous recipe.

Place the two images in your `/qgis_data/rasters/` directory.

How to do it...

This process is relatively straightforward and has a dedicated algorithm within the Processing Toolbox. Perform the following steps:

1. Start QGIS.
2. From the **Plugins** menu, select **Python Console**.
3. Run the `gdalogr:merge` algorithm, specifying the process name, two images, a boolean to use the pseudocolor palette from the first image, a boolean to stack each image into a separate band, and the output filename:

```
processing.runalg("gdalogr:merge", "C:/qgis_data/rasters/Image2.tif;C:/qgis_data/rasters/Image1.tif", False, False, "/qgis_data/rasters/merged.tif")
```
4. Verify that the `merged.tif` image has been created and displays the two images as a single raster within QGIS.

How it works...

The **merge** processing algorithm is a simplified version of the actual `gdal_merge` command-line utility. This algorithm is limited to the GDAL output and aggregates the extent of input rasters. It can only merge two rasters at a time. The `gdal_merge` tool has far more options, including additional output formats, the ability to merge more than two rasters at once, the ability to control the extent, and more. You can also use the GDAL API directly to take advantage of these features, but it will take far more code than what is used in this simple example.

Converting a TIFF image to a JPEG image

Image format conversion is a part of nearly every geospatial project. Rasters come in dozens of different specialized formats, making conversion to a more common format a necessity. The GDAL utilities include a tool called `gdal_translate` specifically for format conversion. Unfortunately, the algorithm in the Processing Toolbox is limited in functionality. For format conversion, it is easier to use the core GDAL API.

Getting ready

We will use the SatImage raster available at <https://geospatialpython.googlecode.com/files/SatImage.zip>.

Place this raster in your `/qgis_data/rasters` directory.

How to do it...

In this recipe, we'll open a TIFFimage using GDAL and copy it to a new dataset as a JPEG2000 image, which allows you to use the common JPEG format while maintaining geospatial information. To do this, we need to perform the following steps:

1. Start QGIS.
2. From the **Plugins** menu, select **Python Console**.
3. Import the gdal module:
`from osgeo import gdal`
4. Get a GDAL driver for our desired format:
`drv = gdal.GetDriverByName("JP2OpenJPEG")`

5. Open the source image:

```
src = gdal.Open("/qgis_data/rasters/satimage.tif")
```

6. Copy the source dataset to the new format:

```
tgt = drv.CreateCopy("/qgis_data/rasters/satimage.jp2", src)
```

How it works...

For the straight format conversion of an image format, the core GDAL library is extremely fast and simple. GDAL supports the creation of over 60 raster formats and the reading of over 130 raster formats.

Creating pyramids for a raster

Pyramids, or overview images, sacrifice the disk space for map rendering speed by storing resampled, lower-resolution versions of images in the file alongside the full resolution image. Once you have finalized a raster, building pyramid overviews is a good idea.

Getting ready

For this recipe, we will use a false-color image, that you can download from <https://geospatialpython.googlecode.com/files/FalseColor.zip>.

Unzip this TIF file and place it in your /qgis_data/rasters directory.

How to do it...

The Processing Toolbox has a dedicated algorithm for building pyramid images. Perform the following steps to create pyramids for a raster

1. Start QGIS.
2. From the **Plugins** menu, select **Python Console**.
3. Import the processing module:

```
import processing
```

4. Run the gdalogr:overviews algorithm, specifying the process name, input image, overview levels, the option to remove existing overviews, resampling method (0 is the nearest neighbor), and overview format (1 is internal):

```
processing.runalg("gdalogr:overviews","/qgis_data/rasters/FalseColor.tif","2 4 8 16",True,0,1)
```

5. Now, load the raster into QGIS by dragging and dropping it from the filesystem onto the map canvas.
6. Double-click on the layer name in the map's legend to open the **Layer Properties** dialog.
7. In the **Layer Properties** dialog, click on the **Pyramids** tab and verify that the layer has multiple resolutions listed.

How it works...

The concept of overview images is quite simple. You resample the images several times, and then a viewer chooses the most appropriate, smallest file to load on the map, depending on scale. The overviews can be stored in the header of the file for certain formats or as an external file format. The level of overviews needed depends largely on the file size and resolution of your current image, but is really arbitrary. In this example, we double the scale by a factor of 2, which is common practice. Most of the zoom tools in the applications will double the scale when you click to zoom in. The factor of 2 gives you enough zoom levels, so that you usually won't zoom to a level where there is no pyramid image. There is a point of diminishing returns if you create too many levels because pyramids take up additional disk space. Usually 4 to 5 levels is effective.

Converting a pixel location to a map coordinate

The ability to view rasters in a geospatial context relies on the conversion of pixel locations to coordinates on the ground. Sooner or later when you use Python to write geospatial programs, you'll have to perform this conversion yourself.

Getting ready

We will use the SatImage raster available at:

<https://geospatialpython.googlecode.com/files/SatImage.zip>

Place this raster in your /qgis_data/rasters directory.

How to do it...

We will use GDAL to extract the information needed to convert pixels to coordinates and then use pure Python to perform the calculation. We'll use the center pixel of the image as the location to convert.

1. Start QGIS.

2. From the **Plugins** menu select **Python Console**

3. We need to import the gdal module:

```
from osgeo import gdal
```

4. Then, we need to define the reusable function that does the conversion accepting a GDAL GeoTransform object containing the raster georeferencing information and the pixel's x,y values:

```
def Pixel2world(geoMatrix, x, y):  
    ulX = geoMatrix[0]  
    ulY = geoMatrix[3]  
    xDist = geoMatrix[1]  
    yDist = geoMatrix[5]  
    coorX = (ulX + (x * xDist))  
    coorY = (ulY + (y * yDist))  
    return (coorX, coorY)
```

5. Now, we'll open the image in GDAL

```
src = gdal.Open("/qgis_data/rasters/Satimage.tif")
```

6. Next, get the GeoTransform object from the image:

```
geoTrans = src.GetGeoTransform()
```

7. Now, calculate the center pixel of the image:

```
centerX = src.RasterXSize/2  
centerY = src.RasterYSize/2
```

8. Finally, perform the conversion by calling our function:

```
Pixel2world(geoTrans, centerX, centerY)
```

9. Verify the coordinates returned are close to the following output:

```
(-89.59486002580364, 30.510227817850406)
```

How it works...

Pixel conversion is just a scaling ratio between two planes, the image coordinate system and the Earth coordinate system. When dealing with large areas, this conversion can become a more complex projection because the curvature of the Earth comes into play. The GDAL website has a nice tutorial about the geotransform object at the following URL: http://www.gdal.org/gdal_tutorial.html

Converting a map coordinate to a pixel location

When you receive a map coordinate as user input or from some other source, you must be able to convert it back to the appropriate pixel location on a raster.

Getting ready

We will use the SatImage raster available at:

<https://geospatialpython.googlecode.com/files/SatImage.zip>

Place this raster in your /qgis_data/rasters directory.

How to do it...

Similar to the previous recipe, we will define a function, extract the GDAL GeoTransform object from our raster, and use it for the conversion.

1. Start QGIS.
2. From the **Plugins** menu select **Python Console**
3. We need to import the gdal module:

```
from osgeo import gdal
```
4. Then, we need to define the reusable function that does the coordinate to pixel conversion. We get the GDAL GeoTransform object containing the raster georeferencing information and the map x,y coordinates:

```
def world2Pixel(geoMatrix, x, y):  
    ulX = geoMatrix[0]  
    ulY = geoMatrix[3]  
    xDist = geoMatrix[1]
```

```
yDist = geoMatrix[5]
rtnX = geoMatrix[2]
rtnY = geoMatrix[4]
pixel = int((x - ulX) / xDist)
line = int((y - ulY) / yDist)
return (pixel, line)
```

5. Next, we open the source image:

```
src = gdal.Open("/qgis_data/rasters/satimage.tif")
```

6. Now, get the GeoTransform object:

```
geoTrans = src.GetGeoTransform()
```

7. Finally, perform the conversion:

```
world2Pixel(geoTrans, -89.59486002580364, 30.510227817850406)
```

8. Verify your output is the following:

```
(1296, 1346)
```

How it works...

This conversion is very reliable over small areas, but as the area of interest expands you must account for elevation as well, which requires a far more complex transformation depending on how an image was generated.



The following presentation from the University of Massachusetts does an excellent job of explain the challenges of georeferencing data:

http://courses.umass.edu/nrc592g-cschweik/pdfs/Class_3_Georeferencing_concepts.pdf

Creating a KML image overlay for a raster

GoogleEarth is one of the most widely available geospatial viewers in existence. The XML data format used by GoogleEarth for geospatial data is called **KML**. The Open Geospatial Consortium adopted KML as a data standard. Converting rasters into a **KML** overlay compressed in a **KMZ** archive file is a very popular way to make data available to end users who know how to use GoogleEarth.

Getting ready

We will use the SatImage raster again available at the following URL if you haven't downloaded it from previous recipes:

<https://geospatialpython.googlecode.com/files/SatImage.zip>

Place this raster in your /qgis_data/rasters directory.

How to do it...

In this recipe, we'll create a **KML** document describing our image. Then we'll convert the image to a **JPEG** in memory using GDAL's specialized virtual file system and write all of the contents directly to a **KMZ** file using Python's `zipfile` module.

1. Start QGIS.

2. From the **Plugins** menu select **Python Console**

3. We need to import the `gdal` module as well as the Python `zipfile` module:

```
from osgeo import gdal  
import zipfile
```

4. Next, we'll open our satellite image in `gdal`:

```
srcf = "/qgis_data/rasters/SatImage.tif"
```

5. Now, we'll create a variable with our virtualized file name, using the GDAL virtual file naming convention beginning with `vismem`:

```
vfn = "/vsimem/satimage.jpg"
```

6. We create the JPEG `gdal` driver object for the output format:

```
drv = gdal.GetDriverByName('JPEG')
```

7. Now, we can open the source file:

```
src = gdal.Open(srcf)
```

8. Then, we can copy that source file to our virtual JPEG:

```
tgt = drv.CreateCopy(vfn, src)
```

9. Now, we are going to create a raster layer in QGIS for our raster, just for the benefit of it calculating the image's extent:

```
rasterLyr = QgsRasterLayer(srcf, "SatImage")
```

10. Next, we get the layer's extent:

```
e = rasterLyr.extent()
```

11. Next, we format our KML document template and insert the image extents:

```
kml = """<?xml version="1.0" encoding="UTF-8"?>
<kml xmlns="http://www.opengis.net/kml/2.2">
  <Document>
    <name>QGIS KML Example</name>
    <GroundOverlay>
      <name>SatImage</name>
      <drawOrder>30</drawOrder>
      <Icon>
        <href>SatImage.jpg</href>
      </Icon>
      <LatLonBox>
        <north>%s</north>
        <south>%s</south>
        <east>%s</east>
        <west>%s</west>
      </LatLonBox>
    </GroundOverlay>
  </Document>
</kml>""" % (e.yMaximum(), e.yMinimum(), e.xMaximum(),
e.xMinimum())
```

12. Now, we open our virtual JPEG in GDAL and prepare it for reading:

```
vsifile = gdal.VSIFOpenL(vfn, 'r')
gdal.VSIFSeekL(vsifile, 0, 2)
vsileng = gdal.VSIFTellL(vsifile)
gdal.VSIFSeekL(vsifile, 0, 0)
```

13. Finally, we write our KML document and virtual JPEG into a zipped KMZ file:

```
z = zipfile.ZipFile("/qgis_data/rasters/satimage.kmz", "w",
zipfile.ZIP_DEFLATED)
z.writestr("doc.kml", kml)
z.writestr("SatImage.jpg", gdal.VSIFReadL(1, vsileng, vsifile))
z.close()
```

14. Now, open the KMZ file in GoogleEarth and verify it looks like the following screenshot:



How it works...

KML is a straightforward XML format. There are entire libraries in Python dedicated to reading and writing it, but for a simple export to share an image or two, the PyQGIS console is more than adequate. While we run this script in the QGIS Python interpreter, it could be run outside of QGIS using just GDAL.

There's more...

The **Orfeo Toolbox** has a processing algorithm called `otb:imagetokmzexport` which has a much more sophisticated KMZ export tool for images.

Classifying a raster

Image classification is one of the most complex aspects of remote sensing. While QGIS is able to color pixels based on values for visualization, it stops short of doing much classification. It does provide a Raster Calculator tool where you can perform arbitrary math formulas on an image, however it does not attempt to implement any common algorithms. The Orfeo Toolbox is dedicated purely to remote sensing and includes an automated classification algorithm called K-means clustering, which groups pixels into an arbitrary number of similar classes to create a new image. We can do a nice demonstration of image classification using this algorithm.

Getting ready

For this recipe, we will use a false color image which you can download here:

<https://geospatialpython.googlecode.com/files/FalseColor.zip>

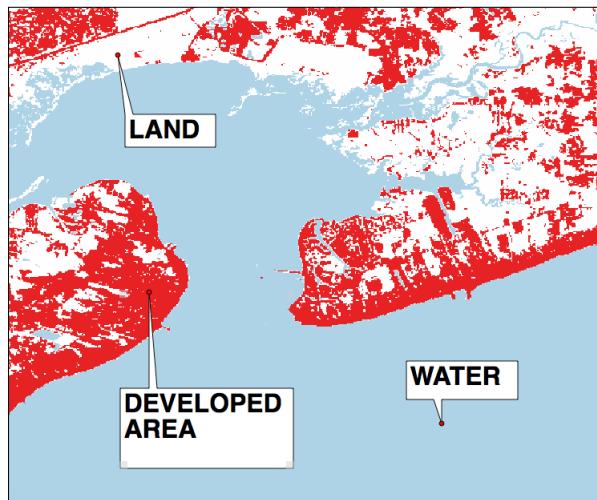
Unzip this TIFF file and place it in your /qgis_data/rasters directory.

How to do it...

All we need to do is run the algorithm on our input image. The important parameters are the second, third, sixth, and tenth parameters. They define the input image name, the amount of RAM to dedicate to the task, the number of classes, and the output name respectively.

1. First, import the processing module in the QGIS **Python Console**:
`import processing`
2. Next, run the otb algorithm using the `processing.runandload()` method to display the output in QGIS:
`processing.runandload("otb:unsupervisedkmeansimageclassification", "/qgis_data/rasters/FalseColor.tif", 768, None, 10000, 3, 1000, 0.95, "/qgis_data/rasters/classify.tif", None)`
3. When the image loads in QGIS, double click the layer name in the **Table of Contents**.
4. In the **Layer Properties** dialog, choose **Style**.
5. Change the **Render type** menu to **Singleband pseudocolor**.

6. Change the **color map** menu on the right to **Spectral**.
7. Click the **Classify** button.
8. Choose the **Ok** button at the bottom of the window.
9. Verify your image looks similar to the following image, except without the class labels:



How it works...

Keeping the class number low allows the automated classification algorithm to focus on the major features in the image and helps us to achieve a very high level of accuracy determining overall land use. Additional automated classification would require supervised analysis with training data sets and more in-depth preparation. But the overall concept would remain the same. QGIS has a nice plugin for semi-automatic classification. You can learn more about it at the following URL:

<https://plugins.qgis.org/plugins/SemiAutomaticClassificationPlugin/>

Converting a raster to a vector

Raster datasets represent real-world features efficiently but can have limited usage for geospatial analysis. Once you have classified an image into a manageable data set, you can convert those raster classes into a vector data set for more sophisticated GIS analysis. GDAL has a function for this operation called **polygonize**.

Getting ready

You will need to download the following classified raster and place it in your `/qgis_data/rasters` directory:

https://geospatialpython.googlecode.com/svn/landuse_bay.zip

How to do it...

Normally, you would save the output of this recipe as a shapefile. We won't specify an output file name. The Processing Toolbox will assign it a temporary filename and return that filename. We'll simply load the temporary file into QGIS. The algorithm allows you to write to a shapefile by specifying it as the last parameter.

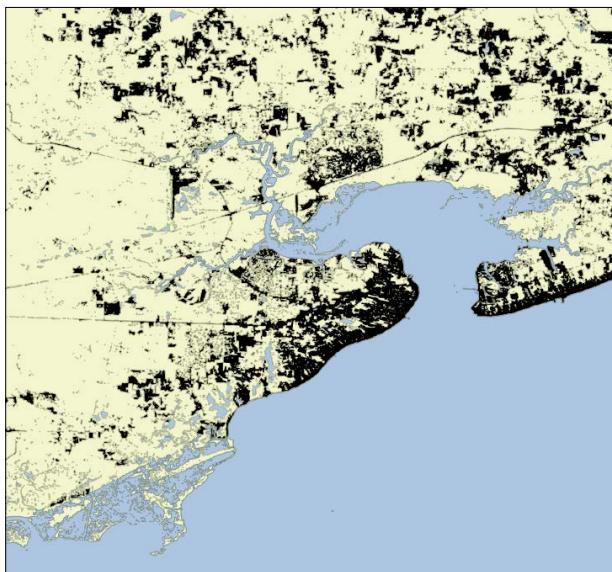
1. In the QGIS **Python Console**, import the processing module:

```
import processing
```

2. Next, run the algorithm specifying the process name, input image, the field name for the class number, and optionally the output shapefile:

```
processing.runalg("gdalogr:polygonize", "C:/qgis_data/rasters/landuse_bay.tif", "DN", None)
```

3. You should get a vector layer with three classes, defined as polygons, denoting developed areas. In the sample image below, we have assigned unique colors to each class: developed area (darkest), water (midtones), and land (lightest color):



How it works...

GDAL looks for clusters of pixels and creates polygons around them. It is important to have as few classes as possible. If there is too much variation in the pixels, then GDAL will create a polygon around each pixel in the image. You turn this recipe into a true analysis product by using the recipe in *Chapter 1, Calculating the Area of a Polygon* to quantify each class of land use.

Georeferencing a raster from control points

Sometimes a raster that represents features on the earth is just an image with no georeferencing information. That is certainly the case with historical scanned maps. However, you can use a referenced data set of the same area to create tie points, or ground control points, and then use an algorithm to warp the image to fit the model of the earth. It is common for data collection systems to just store the **ground control points (GCP)** along with the raster to keep the image in as raw a format as possible. Each change to an image holds the possibility of losing data. So georeferencing an image on demand is often the best approach.

In this recipe, we'll georeference a historical survey map of the Louisiana and Mississippi Gulf Coast from 1853. The control points were manually created with the QGIS Georeferencer plugin and saved to a standardized control point file.

Getting ready

Download the following zip file, unzip the contents, and put the `georef` directory in `/qgis_data/rasters`:

<https://geospatialpython.googlecode.com/svn/georef.zip>

How to do it...

We will use a low-level module of the processing API to access some specialized GDAL utility functions.

1. In the QGIS **Python Console**, import the `GdalUtils` module:

```
from processing.algs.gdal.GdalUtils import GdalUtils
```

2. Now, we will set up some path names for source and target data, which will be used multiple times:

```
src = "/qgis_data/rasters/georef/1853survey.jpg"
points = "/qgis_data/rasters/georef/1853Survey.points"
trans = "/qgis_data/rasters/georef/1835survey_trans.tif"
final = "/qgis_data/rasters/georef/1835survey_georef.tif"
```

3. Next, we will open up our GCP file and read past the header line:

```
gcp = open(points, "rb")
hdr = gcp.readline()
```

4. Then, we can begin building our first gdal utility command:

```
command = ["gdal_translate"]
```

5. Loop through the GCP file and append the points to the command arguments:

```
for line in gcp:
    x,y,col,row,e = line.split(",")
    command.append("-gcp")
    command.append("%s" % col)
    command.append("%s" % abs(float(row)))
    command.append("%s" % x)
    command.append("%s" % y)
```

6. Now, add the input and output file to the command:

```
command.append(src)
command.append(trans)
```

7. Next, we can execute the first command:

```
GdalUtils.runGdal(command, None)
```

8. Next, we change the command to warp the image:

```
command = ["gdalwarp"]
command.extend(["-r", "near", "-order", "3", "-co",
"COMPRESS=None", "-dstalpha"])
```

9. Add the output of the last command as the input and use the final image path as the output:

```
command.append(trans)
command.append(final)
```

10. Now, run the warp command to complete the task:

```
GdalUtils.runGdal(command, None)
```

How it works...

The GdalUtils API exposes the underlying tools used by the Processing Toolbox algorithm, yet provides a robust cross-platform approach that is better than other traditional methods of accessing external programs from Python. If you pull the output image into QGIS and compare it to the USGS coastline shapefile, you can see the results are fairly accurate and could be improved with additional control points and referenced data. The number of GCPs required for a given image is a matter of trial and error. Adding more GCPs won't necessarily lead to better results. You can find out more about creating GCPs in the QGIS documentation:

http://docs.qgis.org/2.6/en/docs/user_manual/plugins/plugins_georeferencer.html

Clipping a raster using a shapefile

Sometimes you need to use a subset of an image which covers an area of interest for a project. In fact, areas of an image outside your area of interest can distract your audience from the idea you are trying to convey. Clipping a raster to a vector boundary allows you to only use the portions of the raster you need. It can also save processing time by eliminating areas outside your area of interest.

Getting ready

We will use the SatImage raster again available at the following URL if you haven't downloaded it from previous recipes:

<https://geospatialpython.googlecode.com/files/SatImage.zip>

Place this raster in your /qgis_data/rasters directory.

How to do it...

Clipping is a common operation and GDAL is well suited for it.

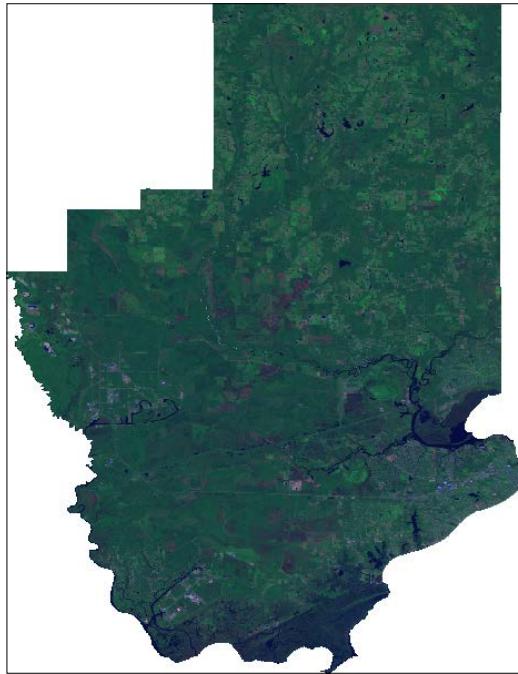
1. First, in the QGIS **Python Console**, run import the processing module:

```
import processing
```

2. Next, run the processing command specifying the input image name as the second argument and the output image as the seventh argument:

```
processing.runandload("gdalogr:cliprasterbymasklayer","/qgis_data/rasters/SatImage.tif","/qgis_data/hancock/hancock.shp","none",False,False,"","/qgis_data/rasters/clipped.tif")
```

3. Verify your output raster looks like the following screenshot:



How it works...

GDAL creates a no data mask outside the shapefile boundary. To the extent of the original image remains the same, however you no longer visualize it and processing algorithms will ignore the no data values.

5

Creating Dynamic Maps

In this chapter, we will cover the following recipes:

- ▶ Accessing the map canvas
- ▶ Changing the map units
- ▶ Iterating over layers
- ▶ Symbolizing a vector layer
- ▶ Rendering a single band raster using a color ramp algorithm
- ▶ Creating a complex vector layer symbol
- ▶ Using icons as vector layer symbols
- ▶ Creating a graduated vector layer symbol
- ▶ Creating a categorized vector layer symbol
- ▶ Creating a map bookmark
- ▶ Navigating to a map bookmark
- ▶ Setting scale-based visibility for a layer
- ▶ Using SVG for layer symbols
- ▶ Using pie charts for symbols
- ▶ Using the OpenStreetMap service
- ▶ Using the Bing aerial image service
- ▶ Adding real-time weather data from OpenWeatherMap
- ▶ Labeling a feature
- ▶ Changing map layer transparency
- ▶ Adding standard map tools to the canvas
- ▶ Using a map tool to draw points on the canvas

- ▶ Using a map tool to draw polygons or lines on the canvas
- ▶ Building a custom selection tool
- ▶ Creating a mouse coordinate tracking tool

Introduction

In this chapter, we'll programmatically create dynamic maps using Python to control every aspect of the QGIS map canvas. We'll learn how to use custom symbology, labels, map bookmarks, and even real-time data. We'll also go beyond the canvas to create custom map tools. You will see that every aspect of QGIS is up for grabs with Python, to write your own application. Sometimes, the PyQGIS API may not directly support your application goal, but there is nearly always a way to accomplish what you set out to do with QGIS.

Accessing the map canvas

Maps in QGIS are controlled through the map canvas. In this recipe, we'll access the canvas and then check one of its properties to ensure that we have control over the object.

Getting ready

The only thing you need to do for this recipe is to open QGIS and select **Python Console** from the **Plugins** menu.

How to do it...

We will assign the map canvas to a variable named `canvas`. Then, we'll check the `size` property of the canvas to get its size in pixels. To do this, perform the following steps:

1. Enter the following line in the QGIS **Python Console**:

```
canvas = qgis.utils.iface.mapCanvas()
```

2. Now, to ensure that we have properly accessed the canvas, check its size in pixels using the following line of code:

```
canvas.size()
```

3. Verify that QGIS returns a `QSize` object that contains the canvas's pixel size, similar to the following format:

```
PyQt4.QtCore.QSize(698, 138)
```

How it works...

Everything in QGIS centers on the canvas. The canvas is part of the QGIS interface or iface API. Anything you see on the screen when using QGIS is generated through the iface API. Note that the iface object is only available to scripts and plugins. When you are building a standalone application, you must initialize your own QgsMapCanvas object.

Changing the map units

Changing the units of measurement on a map, or map units, is a very common operation, depending on the purpose of your map or the standards of your organization or country. In this recipe, we'll read the map units used by QGIS and then change them for your project.

Getting ready

The only preparation you need for this recipe is to open QGIS and select **Python Console** from the **Plugins** menu.

How to do it...

In the following steps, we'll access the map canvas, check the map unit type, and then alter it to a different setting.

1. First, access the map canvas, as follows:

```
canvas = iface.mapCanvas()
```

2. Now, get the map units type. By default, it should be the number **2**:

```
canvas.mapUnits()
```

3. Now, let's set the map units to meters using the built-in enumerator:

```
canvas.setMapUnits(Qgis.Meters)
```

How it works...

QGIS has seven different map units, which are enumerated in the following order:

0 Meters

1 Feet

2 Degrees

3 UnknownUnit

- 4 DecimalDegrees
- 5 DegreesMinutesSeconds
- 6 DegreesDecimalMinutes
- 7 NauticalMiles

It is important to note that changing the map units just changes the unit of measurement for the measurement tool and the display in the status bar; it does not change the underlying map projection. You'll notice this difference if you try to run an operation in the Processing Toolbox, which depends on projected data in meters, if the data is unprojected. The most common use case for changing map units is to switch between imperial and metric units, depending on the user's preference.

Iterating over layers

For many GIS operations, you need to loop through the map layers to look for specific information or to apply a change to all the layers. In this recipe, we'll loop through the layers and get information about them.

Getting ready

We'll need two layers in the same map projection to perform this recipe. You can download the first layer as a ZIP file from https://geospatialpython.googlecode.com/files/MSCities_Geo_Pts.zip.

You can download the second zipped layer from <https://geospatialpython.googlecode.com/files/Mississippi.zip>.

Unzip both of these layers into a directory named `ms` within your `qgis_data` directory.

How to do it...

We will add the layers to the map through the map registry. Then, we will iterate through the map layers and print each layer's title. To do this, perform the following steps:

1. First, let's open the polygon and the point layer using the QGIS **Python Console**:

```
lyr_1 =
QgsVectorLayer("/Users/joellawhead/qgis_data/ms/mississippi.sh
p", "Mississippi", "ogr")

lyr_2 =
QgsVectorLayer("/Users/joellawhead/qgis_data/ms/MSCities_Geo_P
ts.shp", "Cities", "ogr")
```

2. Next, get the map layer registry instance:

```
registry = QgsMapLayerRegistry.instance()
```

3. Now add the vector layers to the map:

```
registry.addMapLayers([lyr_2, lyr_1])
```

4. Then, we retrieve the layers as an iterator:

```
layers = registry.mapLayers()
```

5. Finally, we loop through the layers and print the titles:

```
for l in layers:  
    print l.title()
```

6. Verify that you can read the layer titles in the **Python Console**, similar to the following format:

```
Cities20140904160234792
```

```
Mississippi20140904160234635
```

How it works...

Layers in QGIS are independent of the map canvas until you add them to the map layer registry. They have an ID as soon as they are created. When added to the map, they become part of the canvas, where they pick up titles, symbols, and many other attributes. In this case, you can use the map layer registry to iterate through them and access them to change the way they look or to add and extract data.

Symbolizing a vector layer

The appearance of the layers on a QGIS map is controlled by its symbology. A layer's symbology includes the renderer and one or more symbols. The renderer provides rules dictating the appearance of symbols. The symbols describe properties, including color, shape, size, and linewidth. In this recipe, we'll load a vector layer, change its symbology, and refresh the map.

Getting ready

Download the following zipped shapefile and extract it to your `qgis_data` directory into a folder named `ms` from <https://geospatialpython.googlecode.com/files/Mississippi.zip>.

How to do it...

We will load a layer, add it to the map layer registry, change the layer's color, and then refresh the map. To do this, perform the following steps:

1. First, using the QGIS **Python Console**, we must import the `QtGui` library in order to access the `QColor` object that is used to describe colors in the PyQGIS API:

```
from PyQt4.QtGui import *
```

2. Next, we create our vector layer, as follows:

```
lyr =  
QgsVectorLayer("/Users/joellawhead/qgis_data/ms/mississippi.sh  
p", "Mississippi", "ogr")
```

3. Then, we add it to the map layer registry:

```
QgsMapLayerRegistry.instance().addMapLayer(lyr)
```

4. Now, we access the layer's symbol list through the layer's renderer object:

```
symbols = lyr.rendererV2().symbols()
```

5. Next, we reference the first symbol, which in this case is the only symbol:

```
sym = symbols[0]
```

6. Once we have the symbol, we can set its color:

```
sym.setColor(QColor.fromRgb(255,0,0))
```

7. We must remember to repaint the layer in order to force the update:

```
lyr.triggerRepaint()
```

How it works...

Changing the color of a layer sounds simple, but remember that in QGIS, anything you see must be altered through the canvas API. Therefore, we add the layer to the map and access the layer's symbology through its renderer. The map canvas is rendered as a raster image. The renderer is responsible for turning the layer data into a bitmap image, so the presentation information for a layer is stored with its renderer.

Rendering a single band raster using a color ramp algorithm

A color ramp allows you to render a raster using just a few colors to represent different ranges of cell values that have similar meaning, in order to group them. The approach that will be used in this recipe is the most common way to render elevation data.

Getting ready

You can download a sample DEM from <https://geospatialpython.googlecode.com/files/dem.zip>, which you can unzip in a directory named `rasters` in your `qgis_data` directory.

How to do it...

In the following steps, we will set up objects to color a raster, create a list establishing the color ramp ranges, apply the ramp to the layer renderer, and finally add the layer to the map. To do this, we need to perform the following steps:

1. First, we import the `QtGui` library for color objects in the QGIS **Python Console**:

```
from PyQt4 import QtGui
```

2. Next, we load the raster layer, as follows:

```
lyr =
QgsRasterLayer("/Users/joellawhead/qgis_data/rasters/dem.asc",
"DEM")
```

3. Now, we create a generic raster shader object:

```
s = QgsRasterShader()
```

4. Then, we instantiate the specialized ramp shader object:

```
c = QgsColorRampShader()
```

5. We must name a type for the ramp shader. In this case, we use an `INTERPOLATED` shader:

```
c.setColorRampType(QgsColorRampShader.INTERPOLATED)
```

6. Now, we'll create a list of our color ramp definitions:

```
i = []
```

7. Then, we populate the list with color ramp values that correspond to elevation value ranges:

```
i.append(QgsColorRampShader.ColorRampItem(400,
QtGui.QColor('#d7191c'), '400'))
i.append(QgsColorRampShader.ColorRampItem(900,
QtGui.QColor('#fdae61'), '900'))
i.append(QgsColorRampShader.ColorRampItem(1500,
QtGui.QColor('#ffffbf'), '1500'))
i.append(QgsColorRampShader.ColorRampItem(2000,
QtGui.QColor('#abdda4'), '2000'))
i.append(QgsColorRampShader.ColorRampItem(2500,
QtGui.QColor('#2b83ba'), '2500'))
```

8. Now we assign the color ramp to our shader:

```
c.setColorRampItemList(i)
```

9. Now, we tell the generic raster shader to use the color ramp:

```
s.setRasterShaderFunction(c)
```

10. Next, we create a raster renderer object with the shader:

```
ps = QgsSingleBandPseudoColorRenderer(lyr.dataProvider(), 1,
s)
```

11. We assign the renderer to the raster layer:

```
lyr.setRenderer(ps)
```

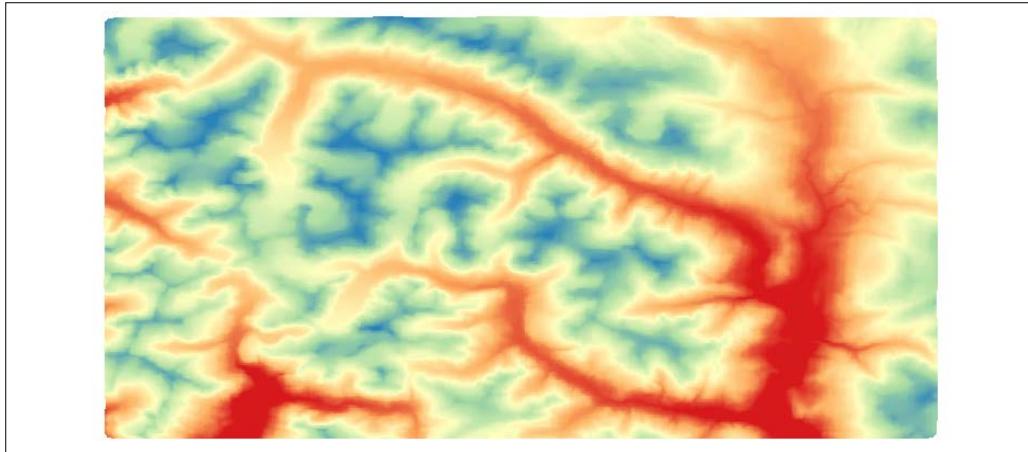
12. Finally, we add the layer to the canvas in order to view it:

```
QgsMapLayerRegistry.instance().addMapLayer(lyr)
```

How it works...

While it takes a stack of four objects to create a color ramp, this recipe demonstrates how flexible the PyQGIS API is. Typically, the more objects it takes to accomplish an operation in QGIS, the richer the API is, giving you the flexibility to make complex maps.

Notice that in each `ColorRampItem` object, you specify a starting elevation value, the color, and a label as the string. The range for the color ramp ends at any value less than the following item. So, in this case, the first color will be assigned to the cells with a value between 400 and 899. The following screenshot shows the applied color ramp.



Creating a complex vector layer symbol

The true power of QGIS symbology lies in its ability to stack multiple symbols in order to create a single complex symbol. This ability makes it possible to create virtually any type of map symbol you can imagine. In this recipe, we'll merge two symbols to create a single symbol and begin unlocking the potential of complex symbols.

Getting ready

For this recipe, we will need a line shapefile, which you can download and extract from <https://geospatialpython.googlecode.com/svn/paths.zip>.

Add this shapefile to a directory named `shapes` in your `qgis_data` directory.

How to do it...

Using the **QGISPythonConsole**, we will create a classic railroad line symbol by placing a series of short, rotated line markers along a regular line symbol. To do this, we need to perform the following steps:

1. First, we load our line shapefile:

```
lyr =  
QgsVectorLayer("/Users/joellawhead/qgis_data/shapes/paths.shp"  
, "Route", "ogr")
```

2. Next, we get the symbol list and reference the default symbol:

```
symbolList = lyr.rendererV2().symbols()  
symbol = symbolList[0]
```

3. Then, we create a shorter variable name for the symbol layer registry:

```
symLyrReg = QgsSymbolLayerV2Registry
```

4. Now, we set up the line style for a simple line using a Python dictionary:

```
lineStyle = {'width': '0.26', 'color': '0,0,0'}
```

5. Then, we create an abstract symbol layer for a simple line:

```
symLyr1Meta =  
symLyrReg.instance().symbolLayerMetadata("SimpleLine")
```

6. We instantiate a symbol layer from the abstract layer using the line style properties:

```
symLyr1 = symLyr1Meta.createSymbolLayer(lineStyle)
```

7. Now, we add the symbol layer to the layer's symbol:

```
symbol.appendSymbolLayer(symLyr1)
```

8. Now, in order to create the rails on the railroad, we begin building a marker line style with another Python dictionary, as follows:

```
markerStyle = {}  
markerStyle['width'] = '0.26'  
markerStyle['color'] = '0,0,0'  
markerStyle['interval'] = '3'  
markerStyle['interval_unit'] = 'MM'  
markerStyle['placement'] = 'interval'  
markerStyle['rotate'] = '1'
```

9. Then, we create the marker line abstract symbol layer for the second symbol:

```
symLyr2Meta =
symLyrReg.instance().symbolLayerMetadata("MarkerLine")
```

10. We instantiate the symbol layer, as shown here:

```
symLyr2 = symLyr2Meta.createSymbolLayer(markerStyle)
```

11. Now, we must work with a subsymbol that defines the markers along the marker line:

```
sybSym = symLyr2.subSymbol()
```

12. We must delete the default subsymbol:

```
sybSym.deleteSymbolLayer(0)
```

13. Now, we set up the style for our rail marker using a dictionary:

```
railStyle = {'size':'2', 'color':'0,0,0', 'name':'line',
'angle':'0'}
```

14. Now, we repeat the process of building a symbol layer and add it to the subsymbol:

```
railMeta =
symLyrReg.instance().symbolLayerMetadata("SimpleMarker")
rail = railMeta.createSymbolLayer(railStyle)
sybSym.appendSymbolLayer(rail)
```

15. Then, we add the subsymbol to the second symbol layer:

```
symbol.appendSymbolLayer(symLyr2)
```

16. Finally, we add the layer to the map:

```
QgsMapLayerRegistry.instance().addMapLayer(lyr)
```

How it works...

First, we must create a simple line symbol. The marker line by itself will render correctly, but the underlying simple line will be a randomly chosen color. We must also change the subsymbol of the marker line because the default subsymbol is a simple circle.

Using icons as vector layer symbols

In addition to the default symbol types available in QGIS, you can also use TrueType fonts as map symbols. TrueType fonts are scalable vector graphics that can be used as point markers. In this recipe, we'll create this type of symbol.

Getting ready

You can download the point shapefile used in this recipe from
https://geospatialpython.googlecode.com/files/NYC_MUSEUMS_GEO.zip.

Extract it to your `qgis_data` directory in a folder named `nyc`.

How to do it...

We will load a point shapefile as a layer and then use the character G in a freely-available font called Webdings, which is probably already on your system, to render a building icon on each point in the layer. To do this, we need to perform the following steps:

1. First, we'll define the path to our point shapefile:

```
src = "/qgis_data/nyc/NYC_MUSEUMS_GEO.shp"
```

2. Then, we'll load the vector layer:

```
lyr = QgsVectorLayer(src, "Museums", "ogr")
```

3. Now, we'll use a Python dictionary to define the font properties:

```
fontStyle = {}  
fontStyle['color'] = '#000000'  
fontStyle['font'] = 'Webdings'  
fontStyle['chr'] = 'G'  
fontStyle['size'] = '6'
```

4. Now, we'll create a font symbol layer:

```
symLyr1 = QgsFontMarkerSymbolLayerV2.create(fontStyle)
```

5. Then, we'll change the default symbol layer of the vector layer to our font's symbol information:

```
lyr.rendererV2().symbols()[0].changeSymbolLayer(0, symLyr1)
```

6. Finally, we'll add the layer to the map:

```
QgsMapLayerRegistry.instance().addMapLayer(lyr)
```

How it works...

The font marker symbol layer is just another type of marker layer; however, the range of possibilities with vector fonts is far broader than the built-in fonts in QGIS. Many industries define standard cartographic symbols using customized fonts as markers.

Creating a graduated vector layer symbol renderer

A graduated vector layer symbol renderer is the vector equivalent of a raster color ramp. You can group features into similar ranges and use a limited set of colors to visually identify these ranges. In this recipe, we'll render a graduated symbol using a polygon shapefile.

Getting ready

You can download a shapefile containing a set of urban area polygons from https://geospatialpython.googlecode.com/files/MS_UrbanAnC10.zip.

Extract this file to a directory named `ms` in your `qgis_data` directory.

How to do it...

We will classify each urban area by population size using a graduated symbol, as follows:

1. First, we import the `QColor` object to build our color range.

```
from PyQt4.QtGui import QColor
```

2. Next, we load our polygon shapefile as a vector layer:

```
lyr = QgsVectorLayer("/qgis_data/ms/MS_UrbanAnC10.shp", "Urban Areas", "ogr")
```

3. Now, we build some nested Python tuples that define the symbol graduation. Each item in the tuple contains a range label, range start value, range end value, and a color name, as shown here:

```
population = (
    ("Village", 0.0, 3159.0, "cyan"),
    ("Small town", 3160.0, 4388.0, "blue"),
    ("Town", 43889.0, 6105.0, "green"),
    ("City", 6106.0, 10481.0, "yellow"),
    ("Large City", 10482.0, 27165, "orange"),
    ("Metropolis", 27165.0, 1060061.0, "red"))
```

4. Then, we establish a Python list to hold our QGIS renderer objects:

```
ranges = []
```

5. Next, we loop through our range list, build the QGIS symbols, and add them to the renderer list:

```
for label, lower, upper, color in population:  
    sym = QgsSymbolV2.defaultSymbol(lyr.geometryType())  
    sym.setColor(QColor(color))  
    rng = QgsRendererRangeV2(lower, upper, sym, label)  
    ranges.append(rng)
```

6. Now, reference the field name containing the population values in the shapefile attributes:

```
field = "POP"
```

7. Then, we create the renderer:

```
renderer = QgsGraduatedSymbolRendererV2(field, ranges)
```

8. We assign the renderer to the layer:

```
lyr.setRendererV2(renderer)
```

9. Finally, we add the map to the layer:

```
QgsMapLayerRegistry.instance().addMapLayer(lyr)
```

How it works...

The approach to using a graduated symbol for a vector layer is very similar to the color ramp shader for a raster layer. You can have as many ranges as you'd like by extending the Python tuple that is used to build the ranges. Of course, you can also build your own algorithms by programmatically examining the data fields first and then dividing up the values in equal intervals or some other scheme.

Creating a categorized vector layer symbol

A categorized vector layer symbol allows you to create distinct categories with colors and labels for unique features. This approach is typically used for datasets with a limited number of unique types of features. In this recipe, we'll categorize a vector layer into three different categories.

Getting ready

For this recipe, we'll use a land use shapefile, which you can download from https://geospatialpython.googlecode.com/svn/landuse_shp.zip.

Extract it to a directory named hancock in your `qgis_data` directory.

How to do it...

We will load the vector layer, create three categories of land use, and render them as categorized symbols. To do this, we need to perform the following steps:

1. First, we need to import the `QColor` object for our category colors:

```
from PyQt4.QtGui import QColor
```

2. Then, we load the vector layer:

```
lyr =
QgsVectorLayer("Users/joellawhead/qgis_data/hancock/landuse.sh
p", "Land Use", "ogr")
```

3. Next, we'll create our three land use categories using a Python dictionary with a field value as the key, color name, and label:

```
landuse = {
    "0": ("yellow", "Developed"),
    "1": ("darkcyan", "Water"),
    "2": ("green", "Land")}
```

4. Now, we can build our categorized renderer items:

```
categories = []
for terrain, (color, label) in landuse.items():
    sym = QgsSymbolV2.defaultSymbol(lyr.geometryType())
    sym.setColor(QColor(color))
    category = QgsRendererCategoryV2(terrain, sym, label)
    categories.append(category)
```

5. We name the field containing the land use value:

```
field = "DN"
```

6. Next, we build the renderer:

```
renderer = QgsCategorizedSymbolRendererV2(field, categories)
```

7. We add the renderer to the layer:

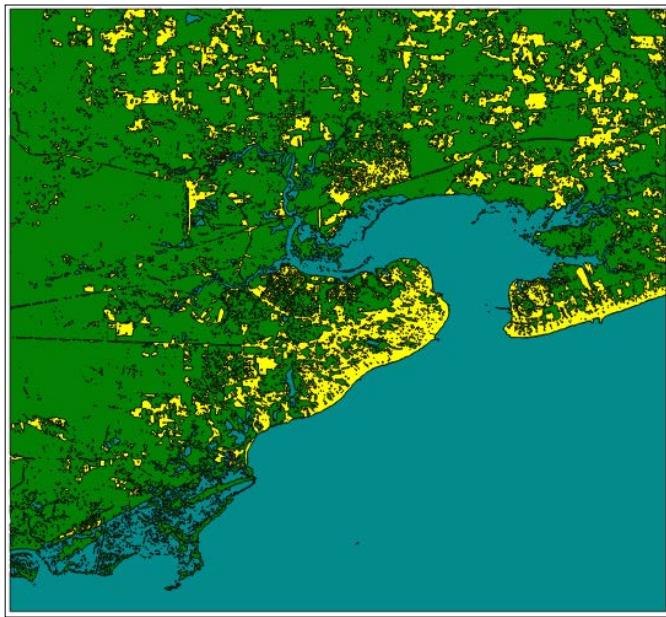
```
lyr.setRendererV2(renderer)
```

8. Finally, we add the categorized layer to the map:

```
QgsMapLayerRegistry.instance().addMapLayer(lyr)
```

How it works...

There are only slight differences in the configurations of the various types of renderers in QGIS. Setting them up by first defining the properties of the renderer using native Python objects makes your code easier to read and ultimately manage. The following map image illustrates the categorized symbol in this recipe:



Creating a map bookmark

Map bookmarks allow you to save a location on a map in QGIS, so you can quickly jump to the points you need to view repeatedly without manually panning and zooming the map. PyQGIS does not contain API commands to read, write, and zoom to bookmarks. Fortunately, QGIS stores the bookmarks in an SQLite database. Python has a built-in SQLite library that we can use to manipulate bookmarks using the database API.

Getting ready

You can download a census tract polygon shapefile to use with this recipe from https://geospatialpython.googlecode.com/files/GIS_CensusTract.zip.

Extract it to your `qgis_data` directory. We are going to create a bookmark that uses an area of interest within this shapefile, so you can manually load the bookmark in order to test it out.

How to do it...

We will access the QGIS configuration variables to get the path of the user database, which stores the bookmarks. Then, we'll connect to this database and execute a SQL query that inserts a bookmark. Finally, we'll commit the changes to the database, as follows:

1. First, using the QGIS **PythonConsole**, we must import Python's built-in SQLite library:

```
import sqlite3
```

2. Next, get the path to the database:

```
dbPath = QgsApplication.qgisUserDbFilePath()
```

3. Now, we connect to the database:

```
db = sqlite3.connect(dbPath)
```

4. Then, we need a database cursor to manipulate the database:

```
cursor = db.cursor()
```

5. Now, we can execute the SQL query, which is a string. In the **VALUES** portion of the query, we will leave the bookmark ID as **NULL** but give it a name, then we leave the project name **NULL** and set the extents, as follows:

```
cursor.execute("""INSERT INTO tbl_bookmarks (
    bookmark_id, name, project_name,
    xmin, ymin, xmax, ymax,
    projection_srid)
VALUES (NULL, "BSL", NULL,
        -89.51715550010032,
        30.233838337125075,
        -89.27257255649518,
        30.381717490617945,
        4269)""")
```

6. Then, we commit the changes:

```
db.commit()
```

7. To test the map bookmark, load the census tract layer onto the map by dragging and dropping it from your filesystem into QGIS.

8. Next, click on the **View** menu in QGIS and select **ShowBookmarks**.

9. Then, select the **BSL bookmark** and click on the **ZoomTo** button.

10. Verify that the map snapped to an area of interest close to the polygons, with OBJECTIDs from 4625 to 4627.

How it works...

Even when QGIS doesn't provide a high-level API, you can almost always use Python to dig deeper and access the information you want. QGIS is built on open source software, therefore no part of the program is truly off-limits.

Navigating to a map bookmark

Map bookmarks store important locations on a map, so you can quickly find them later. You can programmatically navigate to bookmarks using the Python `sqlite3` library in order to access the bookmarks database table in the QGIS user database and then use the PyQGIS canvas API.

Getting ready

We will use a census tract layer to test out the bookmark navigation. You can download the zipped shapefile from https://geospatialpython.googlecode.com/files/GIS_CensusTract.zip.

Manually load this layer into QGIS after extracting it from the ZIP file. Also, make sure that you have completed the previous recipe, *Creating a map bookmark*. You will need a bookmark named `BSL` for an area of interest in this shapefile.

How to do it...

We will retrieve a bookmark from the QGIS user database and then set the map's extents to this bookmark. To do this, perform the following steps:

1. First, import the Python `sqlite3` library:

```
import sqlite3
```

2. Next, get the location of the user database from the QGIS data:

```
dbPath = QgsApplication.qgisUserDbFilePath()
```

3. Now, we connect to the database:

```
db = sqlite3.connect(dbPath)
```

4. Then, we need a database cursor to run queries:

```
cursor = db.cursor()
```

5. Now, we can get the bookmark information for the bookmark named **BSL**:

```
cursor.execute("""SELECT * FROM tbl_bookmarks WHERE  
name='BSL'""")
```

6. Now, we'll get the complete results from the query:

```
row = cursor.fetchone()
```

7. Then, we split the values of the result into multiple variables:

```
id,mark_name,project,xmin,ymin,xmax,ymax,srid = row
```

8. Now, we can use the bookmark to create a QGIS extent rectangle:

```
rect = QgsRectangle(xmin, ymin, xmax, ymax)
```

9. Next, we reference the map canvas:

```
canvas = qgis.utils.iface.mapCanvas()
```

10. Finally, we set the extent of the canvas to the rectangle and then refresh the canvas:

```
canvas.setExtent(rect)
```

```
canvas.refresh()
```

How it works...

Reading and writing bookmarks with SQLite is straightforward even though its not a part of the main PyQGIS API. Notice that bookmarks have a placeholder for a project name, which you can use to filter bookmarks by project if needed.

Setting scale-based visibility for a layer

Sometimes, a GIS layer only makes sense when it is displayed at a certain scale, for example, a complex road network. PyQGIS supports scale-based visibility to programmatically set the scale range, in which a layer is displayed. In this recipe, we'll investigate scale-dependent layers.

Getting ready

You will need the sample census tract shapefile available as a ZIP file from https://geospatialpython.googlecode.com/files/GIS_CensusTract.zip.

Extract the zipped layer to a directory named `census` in your `qgis_data` directory.

How to do it...

We will load the vector layer, toggle scale-based visibility, set the visibility range, and then add the layer to the map. To do this, perform the following steps:

1. First, we load the layer:

```
lyr =  
QgsVectorLayer("/Users/joellawhead/qgis_data/census/GIS_Census  
Tract_poly.shp", "Census", "ogr")
```

2. Next, we toggle scale-based visibility:

```
lyr.toggleScaleBasedVisibility(True)
```

3. Then, we set the minimum and maximum map scales at which the layer is visible:

```
lyr.setMinimumScale(22945.0)
```

```
lyr.setMaximumScale(1000000.0)
```

4. Now, we add the layer to the map:

```
QgsMapLayerRegistry.instance().addMapLayer.lyr)
```

5. Finally, manually zoom in and out of the map to ensure that the layer disappears and reappears at the proper scales.

How it works...

The map scale is a ratio of map units to physical map size, expressed as a floating-point number. You must remember to toggle scale-dependent visibility so that QGIS knows that it needs to check the range each time the map scale changes.

Using SVG for layer symbols

Scalable Vector Graphics (SVG) are an XML standard that defines vector graphics that can be scaled at any resolution. QGIS can use SVG files as markers for points. In this recipe, we'll use Python to apply one of the SVG symbols included with QGIS to a point layer.

Getting ready

For this recipe, download the following zipped point shapefile layer from https://geospatialpython.googlecode.com/files/NYC_MUSEUMS_GEO.zip.

Extract it to your `qgis_data` directory.

How to do it...

In the following steps, we'll load the vector layer, build a symbol layer and renderer, and add it to the layer, as follows:

1. First, we'll define the path to the shapefile:

```
src =  
"/Users/joellawhead/qgis_data/NYC_MUSEUMS_GEO/NYC_MUSEUMS_GEO.  
shp"
```

2. Next, we'll load the layer:

```
lyr = QgsVectorLayer(src, "Museums", "ogr")
```

3. Now, we define the properties of the symbol, including the location of the SVG file as a Python dictionary:

```
svgStyle = {}  
svgStyle['fill'] = '#0000ff'  
svgStyle['name'] = 'landmark/tourism=museum.svg'  
svgStyle['outline'] = '#000000'  
svgStyle['outline-width'] = '6.8'  
svgStyle['size'] = '6'
```

4. Then, we create an SVG symbol layer:

```
symLyr1 = QgsSvgMarkerSymbolLayerV2.create(svgStyle)
```

5. Now, we change the layer renderer's default symbol layer:

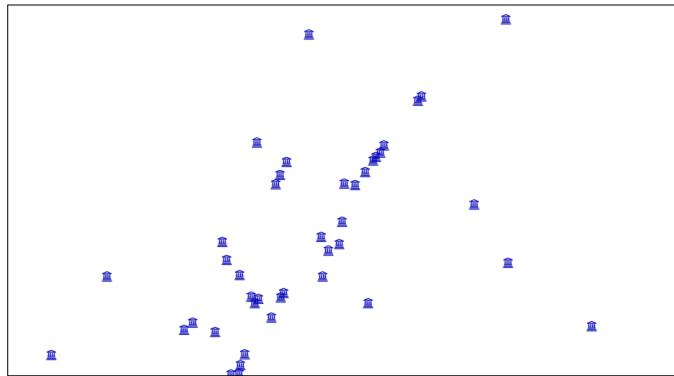
```
lyr.rendererV2().symbols()[0].changeSymbolLayer(0, symLyr1)
```

6. Finally, we add the layer to the map in order to view the SVG symbol:

```
QgsMapLayerRegistry.instance().addMapLayer(lyr)
```

How it works...

The default SVG layers are stored in the QGIS application directory. There are numerous graphics available that cover many common uses. You can also add your own graphics as well. The following map image shows the recipe's output:



Using pie charts for symbols

QGIS has the ability to use dynamic pie charts as symbols describing the statistics in a given region. In this recipe, we'll use pie chart symbols on a polygon layer in QGIS.

Getting ready

For this recipe, download the following zipped shapefile and extract it to a directory named `ms` in your `qgis_data` directory from <https://geospatialpython.googlecode.com/svn/County10PopnHou.zip>.

How to do it...

As with other renderers, we will build a symbol layer, add it to a renderer, and display the layer on the map. The pie chart diagram renderers are more complex than other renderers but have many more options. Perform the following steps to create a pie chart map:

1. First, we import the PyQt GUI library:

```
from PyQt4.QtGui import *
```

2. Then, we load the layer:

```
lyr =
QgsVectorLayer("/Users/joellawhead/qgis_data/ms/County10PopnHo
u.shp", "Population", "ogr")
```

3. Next, we set up categories based on attribute names:

```
categories = [u'PCT_WHT', u'PCT_BLK', u'PCT_AMIND',
u'PCT_ASIAN', u'PCT_HAW', u'PCT_ORA', u'PCT_MR', u'PCT_HISP']
```

4. Now, we set up a list of corresponding colors for each category:

```
colors =
['#3727fa', '#01daae', '#f849a6', '#268605', '#6810ff', '#453990',
'#630f2f', '#07dd45']
```

5. Next, we convert the hex color values to QColor objects:

```
qcolors = []
for c in colors:
    qcolors.append(QColor(c))
```

6. Now, we reference the map canvas:

```
canvas = iface.mapCanvas()
```

7. Then, we create a pie diagram object:

```
diagram = QgsPieDiagram()
```

8. Then, we create a diagram settings object:

```
ds = QgsDiagramSettings()
```

9. Now, we define all the diagram settings that will be used for the renderer:

```
ds.font = QFont("Helvetica", 12)
ds.transparency = 0
ds.categoryColors = qcolors
ds.categoryAttributes = categories
ds.size = QSizeF(100.0, 100.0)
ds.sizeType = 0
ds.labelXPlacementMethod = 1
ds.scaleByArea = True
ds.minimumSize = 0
ds.BackgroundColor = QColor(255,255,255,0)
ds.PenColor = QColor("black")
ds.penWidth = 0
```

10. Now, we can create our diagram renderer:

```
dr = QgsLinearlyInterpolatedDiagramRenderer()
```

11. We must set a few size parameters for our diagrams:

```
dr.setLowerValue(0.0)
dr.setLowerSize(QSizeF(0.0, 0.0))
dr.setUpperValue(2000000)
dr.setUpperSize(QSizeF(40, 40))
dr.setClassificationAttribute(6)
```

12. Then, we can add our diagram to the renderer:

```
dr.setDiagram(diagram)
```

13. Next, we add the renderer to the layer:

```
lyr.setDiagramRenderer(dr)
```

14. Now, we apply some additional placement settings at the layer level:

```
dls = QgsDiagramLayerSettings()
dls.dist = 0
dls.priority = 0
dls.xPosColumn = -1
dls.yPosColumn = -1
dls.placement = 0
lyr.setDiagramLayerSettings(dls)
```

15. In QGIS 2.6, the diagram renderer is tied to the new PAL labeling engine, so we need to activate this engine:

```
label = QgsPalLayerSettings()
label.readFromLayer(lyr)
label.enabled = True
label.writeToLayer(lyr)
```

16. Next, we delete any cached images that are rendered and force the layer to repaint:

```
if hasattr_lyr, "setCacheImage":
    lyr.setCacheImage(None)

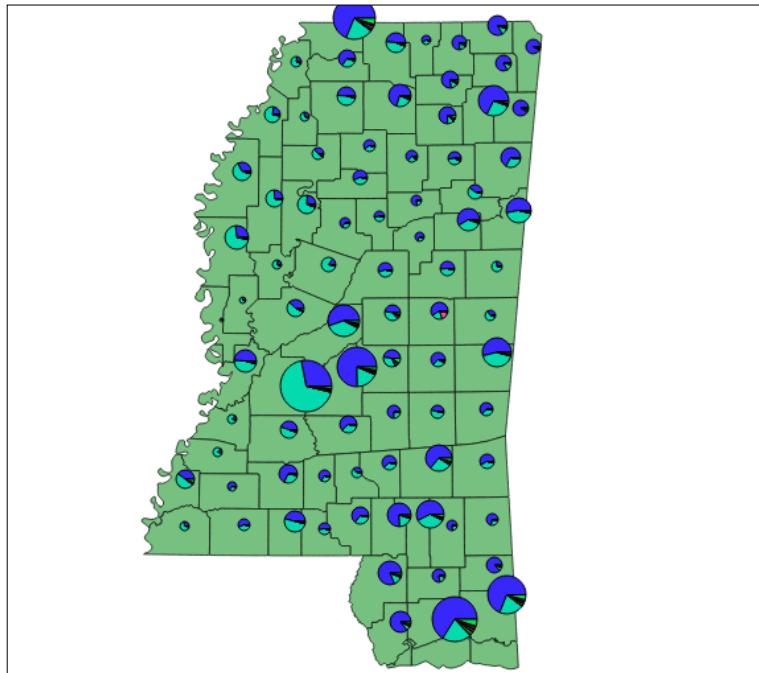
lyr.triggerRepaint()
```

17. Finally, we add our diagram layer to the map:

```
QgsMapLayerRegistry.instance().addMapLayer(lyr)
```

How it works...

The basics of pie chart diagram symbols are straightforward and work in a similar way to other types of symbols and renderers. However, it gets a little confusing as we need to apply settings at three different levels – the diagram level, the render level, and the layer level. It turns out they are actually quite complex. Most of the settings are poorly documented, if at all. Fortunately, most of them are self-explanatory. The following screenshot shows an example of the completed pie chart diagram map:



There's more...

To learn more about what is possible with pie chart diagram symbols, you can experiment with this recipe in the Script Runner plugin, where you can change or remove settings and quickly re-render the map. You can also manually change the settings using the QGIS dialogs and then export the style to an XML file and see what settings are used. Most of them map to the Python API well.

Using the OpenStreetMap service

Cloud-based technology is moving more and more data to the Internet, and GIS is no exception. QGIS can load web-based data using Open GIS Consortium standards, such as **Web Map Service (WMS)**. The easiest way to add WMS layers is using the **Geospatial Data Abstraction Library (GDAL)** and its virtual filesystem feature to load a tiled layer.

Getting ready

You don't need to do any preparation for this recipe, other than opening the Python console plugin within QGIS.

How to do it...

We will create an XML template that describes the tiled web service from OpenStreetMap we want to import. Then, we'll turn it into a GDAL virtual file and load it as a QGIS raster layer. To do this, we need to perform the following steps:

1. First, we import the GDAL library:

```
from osgeo import gdal
```

2. Next, we'll create our XML template, describing the OpenStreetMap tiled web service:

```
xml = """<GDAL_WMS>
<Service name="TMS">
<ServerUrl>http://tile.openstreetmap.org/${z}/${x}/${y}.png</
ServerUrl>
</Service>
<DataWindow>
<UpperLeftX>-20037508.34</UpperLeftX>
<UpperLeftY>20037508.34</UpperLeftY>
<LowerRightX>20037508.34</LowerRightX>
<LowerRightY>-20037508.34</LowerRightY>
<TileLevel>18</TileLevel>
<TileCountX>1</TileCountX>
<TileCountY>1</TileCountY>
<YOrigin>top</YOrigin>
</DataWindow>
<Projection>EPSG:900913</Projection>
<BlockSizeX>256</BlockSizeX>
```

- ```
<BlockSizeY>256</BlockSizeY>
<BandsCount>3</BandsCount>
<Cache />
</GDAL_WMS>"""

3. Now, we'll create the path for our GDAL virtual filesystem's file:
vfn = "/vsimem/osm.xml"

4. Next, we use GDAL to create the virtual file using the path and the XML document:
gdal.FileFromMemBuffer(vfn, xml)

5. Now, we can create a raster layer from the virtual file:
rasterLyr = QgsRasterLayer(vfn, "OSM")

6. Before we add the layer to the map, we'll make sure that it's valid:
rasterLyr.isValid()

7. Finally, add the layer to the map:
QgsMapLayerRegistry.instance().addMapLayers([rasterLyr])
```

## How it works...

There are other ways to load tiled map services such as OpenStreetMap into QGIS programmatically, but GDAL is by far the most robust. The prefix `vsimem` tells GDAL to use a virtual file in order to manage the tiles. This approach frees you from the need to manage downloaded tiles on disk directly and allows you to focus on your application's functionality.

## Using the Bing aerial image service

While there are many services that provide street map tiles, there are far fewer services that provide imagery services. One excellent free service for both maps and, more importantly, imagery is Microsoft's Bing map services. We can access Bing imagery programmatically in QGIS using GDAL's WMS capability coupled with virtual files.

## Getting ready

You don't need to do any preparation for this recipe other than opening the Python console plugin within QGIS.

## How to do it...

Similar to the approach used for the previous *Using the OpenStreetMap service* recipe, we will create an XML file as a string to describe the service, turn it into a GDAL virtual file, and load it as a raster in QGIS. To do this, we need to perform the following steps:

1. First, we import the GDAL library:

```
from osgeo import gdal
```

2. Next, we create the XML file, describing the Bing service as a string:

```
xml = """<GDAL_WMS>
 <Service name="VirtualEarth">
 <ServerUrl>
 http://a${server_num}.ortho.tiles.virtualearth.net/tiles/
 a${quadkey}.jpeg?g=90
 </ServerUrl>
 </Service>
 <MaxConnections>4</MaxConnections>
 <Cache/>
</GDAL_WMS>"""
```

3. Now, we create the virtual file path for the XML file:

```
vfn = "/vsimem/bing.xml"
```

4. Then, we turn the XML file into a GDAL virtual file:

```
gdal.FileFromMemBuffer(vfn, xml)
```

5. Now, we can add the file as a QGIS raster layer and check its validity:

```
rasterLyr = QgsRasterLayer(vfn, "BING")
rasterLyr.isValid()
```

6. Finally, we add the layer to the map:

```
QgsMapLayerRegistry.instance().addMapLayers([rasterLyr])
```

## How it works...

GDAL has drivers for various map services. The service name for Bing is `VirtualEarth`. The  `${ }`  clauses in the server URL provide placeholders, which will be replaced with instance-specific data when GDAL downloads styles. When using this data, you should be aware that it has copyright restrictions. Be sure to read the Bing usage agreement online.

## Adding real-time weather data from OpenWeatherMap

Real-time data is one of the most exciting data types you can add to a modern map. Most data producers make data available through **Open GIS Consortium** standards. One such example is OpenWeatherMap, which offers an OGC **Web Map Service (WMS)** for different real-time weather data layers. In this recipe, we'll access this service to access a real-time weather data layer.

### Getting ready

To prepare for this recipe, you just need to open the QGIS **Python Console** by clicking on the **Plugins** menu and selecting **Python Console**.

### How to do it...

We will add a WMS weather data layer for precipitation to a QGIS map, as follows:

1. First, we specify the parameters for the service:

```
service =
'crs=EPSG:900913&dpiMode=7&featureCount=10&format=image/png&la
yers=precipitation&styles=&url=http://wms.openweathermap.org/s
ervice'
```

2. Next, we create the raster layer, specifying `wms` as the type:

```
rlayer = QgsRasterLayer(service, "precip", "wms")
```

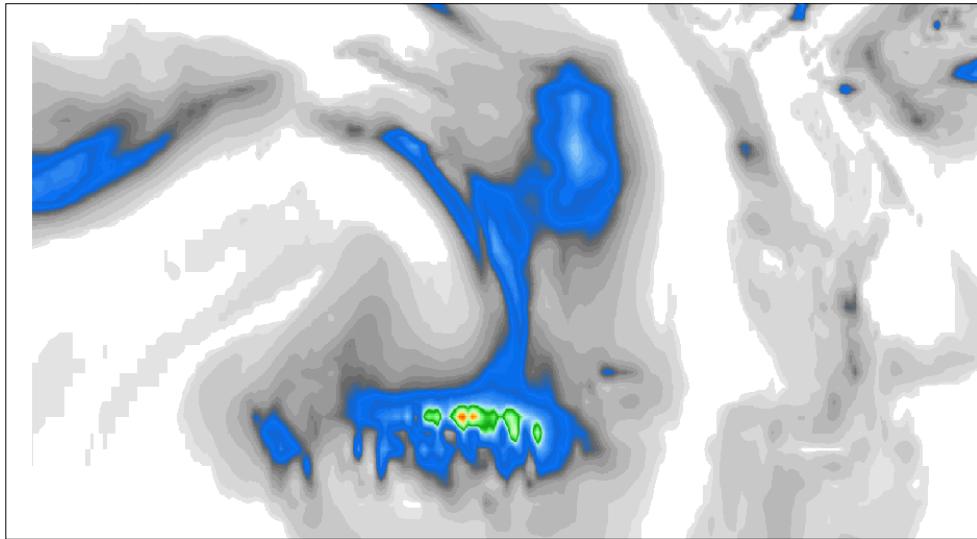
3. Finally, we add the precipitation layer to the map:

```
QgsMapLayerRegistry.instance().addMapLayers([rlayer])
```

### How it works...

A WMS request is typically an HTTP GET request with all of the parameters as part of the URL. In PyQGIS, you use a URL-encoded format and specify the parameters separately from the URL.

The following map image shows the output of the precipitation layer in QGIS:



## Labeling features

Once your map layers are styled, the next step to creating a complete map is labeling features. We'll explore the basics of labeling in this recipe.

### Getting ready

Download the following zipped shapefile from [https://geospatialpython.googlecode.com/files/MSCities\\_Geo\\_Pts.zip](https://geospatialpython.googlecode.com/files/MSCities_Geo_Pts.zip).

Extract the shapefile to a directory named `ms` in your `qgis_data` shapefile.

### How to do it...

We will load the point shapefile layer, create a label object, set its properties, apply it to the layer, and then add the layer to the map. To do this, we need to perform the following steps:

1. First, to save space, we'll specify the path to the shapefile:

```
src = "/Users/joellawhead/qgis_data/ms/MSCities_Geo_Pts.shp"
```

2. Next, we'll load the layer:

```
lyr = QgsVectorLayer(src, "Museums", "ogr")
```

3. Then, we'll create the labeling object:

```
label = QgsPalLayerSettings()
```

4. Now, we'll configure the labels, starting with the current layer settings being read:

```
label.readFromLayer(lyr)
```

```
label.enabled = True
```

5. Then, we specify the attribute for the label data:

```
label.fieldName = 'NAME10'
```

6. Then, we can set the placement and size options:

```
label.placement= QgsPalLayerSettings.AroundPoint
```

```
label.setDataDefinedProperty(QgsPalLayerSettings.Size, True, True, '8', '')
```

7. Next, we commit the changes to the layer:

```
label.writeToLayer(lyr)
```

8. Finally, we can add the layer to the map to view the labels:

```
QgsMapLayerRegistry.instance().addMapLayers([lyr])
```

## How it works...

An interesting part of labeling is the round-trip read and write process to access the layer data and the assignment of the labeling properties. Labeling can be quite complex, but this recipe covers the basics needed to get started.

## Changing map layer transparency

Map layer transparency allows you to change the opacity of a layer, so the items behind it are visible to some degree. A common technique is to make a vector layer polygon partially transparent in order to allow the underlying imagery or elevation data to add texture to the data.

## Getting ready

In a directory called `ms`, in your `qgis_data` directory, download and extract the following shapefile from

<https://geospatialpython.googlecode.com/files/Mississippi.zip>.

## How to do it...

The process is extremely simple. Transparency is just a method:

1. First, we load the shapefile layer:

```
lyr =
QgsVectorLayer("/Users/joellawhead/qgis_data/ms/mississippi.sh
p", "Mississippi", "ogr")
```

2. Next, we set the layer's transparency to 50 percent:

```
lyr.setLayerTransparency(50)
```

3. Finally, we add this layer to the map:

```
QgsMapLayerRegistry.instance().addMapLayer(lyr)
```

## How it works...

If you set the transparency to 100 percent, the layer is completely opaque. If you set it to 0, the layer becomes completely invisible.

## Adding standard map tools to the canvas

In this recipe, you'll learn how to add standard map navigation tools to a standalone map canvas. Creating the simplest possible interactive application provides a framework to begin building specialized geospatial applications using QGIS as a library.

## Getting ready

Download the following zipped shapefile and extract it to your `qgis_data` directory into a folder named `ms` from <https://geospatialpython.googlecode.com/files/Mississippi.zip>.

## How to do it...

We will walk through the steps required to create a map canvas, add a layer to it, and then add some tools to zoom and pan the map, as follows:

1. First, because we are working outside the QGIS Python interpreter, we need to import some QGIS and Qt libraries:

```
from qgis.gui import *
from qgis.core import *
```

```
from PyQt4.QtGui import *
from PyQt4.QtCore import SIGNAL, Qt
import sys, os
```

2. Then, we must set the location of our main QGIS application directory. This setting is platform-dependent:

```
OSX:
QgsApplication.setPrefixPath("/Applications/QGIS.app/Contents/
MacOS/", True)

Windows:
app.setPrefixPath("C:/Program Files/QGIS
Valmiera/apps/qgis", True)
```

3. Next, we begin initializing the class:

```
class MyWnd(QMainWindow):
 def __init__(self):
```

4. Now, we can initialize the application and create the map canvas:

```
QMainWindow.__init__(self)
QgsApplication.setPrefixPath("/Applications/QGIS.app/Contents/
MacOS/", True)
QgsApplication.initQgis()
self.canvas = QgsMapCanvas()
self.canvas.setCanvasColor(Qt.white)
```

5. Then, we can load the shapefile layer and add it to the canvas:

```
self.lyr = QgsVectorLayer("/Users/joellawhead/qgis_data/ms/
mississippi.shp", "Mississippi", "ogr")
QgsMapLayerRegistry.instance().addMapLayer(self.lyr)
self.canvas.setExtent(self.lyr.extent())
self.canvas.setLayerSet([QgsMapCanvasLayer(self.lyr)])
self.setCentralWidget(self.canvas)
```

6. Next, we define the buttons that will be visible on the toolbar:

```
actionZoomIn = QAction("Zoom in", self)
actionZoomOut = QAction("Zoom out", self)
actionPan = QAction("Pan", self)
actionZoomIn.setCheckable(True)
actionZoomOut.setCheckable(True)
actionPan.setCheckable(True)
```

7. Now, we connect the signal created when the buttons are clicked to the Python methods that will provide each tool's functionality:

```
actionZoomIn.triggered.connect(self.zoomIn)
actionZoomOut.triggered.connect(self.zoomOut)
actionPan.triggered.connect(self.pan)
```

8. Next, we create our toolbar and add the buttons:

```
self.toolbar = self.addToolBar("Canvas actions")
(actionZoomIn)
self.toolbar.addAction(actionZoomOut)
self.toolbar.addAction(actionPan)
```

9. Then, we connect the buttons to the applications states:

```
self.toolPan = QgsMapToolPan(self.canvas)
self.toolPan.setAction(actionPan)

self.toolZoomIn = QgsMapToolZoom(self.canvas, False) # false = in
self.toolZoomIn.setAction(actionZoomIn)

self.toolZoomOut = QgsMapToolZoom(self.canvas, True) # true = out
self.toolZoomOut.setAction(actionZoomOut)
```

10. Then, we define which button will be selected when the application loads:

```
self.pan()
```

11. Now, we define the Python methods that control the application's behavior for each tool:

```
def zoomIn(self):
 self.canvas.setMapTool(self.toolZoomIn)

def zoomOut(self):
 self.canvas.setMapTool(self.toolZoomOut)

def pan(self):
 self.canvas.setMapTool(self.toolPan)
```

12. Then, we create a Qt application that uses our application window class:

```
class MainApp(QApplication):
 def __init__(self):
 QApplication.__init__(self, [], True)
 wdg = MyWnd()
 wdg.show()
 self.exec_()
```

13. Finally, we enter the program's main loop:

```
if __name__ == "__main__":
 import sys
 app = MainApp()
```

## How it works...

An application is a continuously running program loop that ends only when we quit the application. QGIS is based on the Qt windowing library, so our application class inherits from the main window class that provides the canvas and the ability to create toolbars and dialogs. This is a lot of setup, even for an extremely simple application, but once the framework for an application is complete, it becomes much easier to extend it.

# Using a map tool to draw points on the canvas

QGIS contains a built-in functionality to zoom and pan the map in custom applications. It also contains the basic hooks to build your own interactive tools. In this recipe, we'll create an interactive point tool that lets you mark locations on the map by clicking on a point.

## Getting ready

We will use the application framework from the previous *Adding standard map tools to the canvas* recipe, so complete that recipe first. We will extend that application with a new tool. The complete version of this application is available in the code samples provided with this book.

## How to do it...

We will set up the button, signal trigger, and actions as we do with all map tools. However, because we are building a new tool, we must also define a class to define exactly what the tool does. To do this, we need to perform the following actions:

1. First, we define our point tool's button in the actions portion of our application. Place this line after the `QAction("Pan")` method:

```
actionPoint = QAction("Point", self)
```

2. In the next section, we make sure that when we click on the button, it stays selected:

```
actionPoint.setCheckable(True)
```

3. In the section after that, we define the method that is used when the button is triggered:

```
self.connect(actionPoint, SIGNAL("triggered()"), self.point)
```

4. Now, we add the button to the toolbar along with the other buttons:

```
self.toolbar.addAction(actionPoint)
```

5. Then, we link the application to our specialized tool class:

```
self.toolPoint = PointMapTool(self.canvas)
self.toolPoint.setAction(actionPoint)
```

6. We set the point tool to be selected when the application loads:

```
self.point()
```

7. Now, we define the method in the main application class for our tool:

```
def point(self):
 self.canvas.setMapTool(self.toolPoint)
```

8. Now, we create a class that describes the type of tool we have and the output it provides. The output is a point on the canvas, defined in the canvasPressEvent method, that receives the button-click event. We will inherit from a generic tool called the QgsMapToolEmitPoint in order to create points:

```
class PointMapTool(QgsMapToolEmitPoint):
 def __init__(self, canvas):
 self.canvas = canvas
 QgsMapToolEmitPoint.__init__(self, self.canvas)
 self.point = None

 def canvasPressEvent(self, e):
 self.point = self.toMapCoordinates(e.pos())
 print self.point.x(), self.point.y()
 m = QgsVertexMarker(self.canvas)
 m.setCenter(self.point)
 m.setColor(QColor(0, 255, 0))
 m.setIconSize(5)
 m.setIconType(QgsVertexMarker.ICON_BOX) # or ICON_CROSS,
 ICON_X
 m.setPenWidth(3)
```

## How it works...

For custom tools, PyQGIS provides a set of generic tools for the common functions that you can extend and piece together. In this case, the EmitPoint tool handles the details of the events and map functionality when you click on a location on the map.

## Using a map tool to draw polygons or lines on the canvas

In this recipe, we'll create a tool to draw polygons on the canvas. This tool is an important tool because it opens the doors to even more advanced tools. Once you have a polygon on the canvas, you can do all sorts of operations that involve querying and geometry.

### Getting ready

We will use the application framework from the *Adding standard map tools to the canvas* recipe, so complete that recipe. We will extend that application with a new tool. The complete version of this application is available in the code samples provided with this book.

### How to do it...

We will add a new tool to the toolbar and also create a class that describes our polygon tool, as follows:

1. First, we define our polygon tool's button in the actions portion of our application. Place this line after the `QAction("Pan")` method:  
`actionPoly = QAction("Polygon", self)`
2. In the next section, we make sure that when we click on the button, it stays selected:  
`actionPoly.setCheckable(True)`
3. In the section after that, we define the method used when the button is triggered:  
`self.connect(actionPoly, SIGNAL("triggered()"), self.poly)`
4. Now, we add the button to the toolbar along with the other buttons:  
`self.toolbar.addAction(actionPoly)`

5. Then, we link the application to our specialized tool class:

```
self.toolPoly = PolyMapTool(self.canvas)
self.toolPoly.setAction(actionPoly)
```

6. We set the point tool to be selected when the application loads:

```
self.poly()
```

7. Now, we define the method in the main application class for our tool:

```
def poly(self):
 self.canvas.setMapTool(self.toolPoly)
```

Now, we create a class that describes the type of tool we have and the output it provides. The output is a point on the canvas defined in the `canvasPressEvent` method, which receives the button-click event and the `showPoly` method. We will inherit from a generic tool in order to create points called the `QgsMapToolEmitPoint`; we will also use an object called `QgsRubberBand` for handling polygons:

```
class PolyMapTool(QgsMapToolEmitPoint):
 def __init__(self, canvas):
 self.canvas = canvas
 QgsMapToolEmitPoint.__init__(self, self.canvas)
 self.rubberband = QgsRubberBand(self.canvas, QGis.Polygon)
 self.rubberband.setColor(Qt.red)
 self.rubberband.setWidth(1)
 self.point = None
 self.points = []

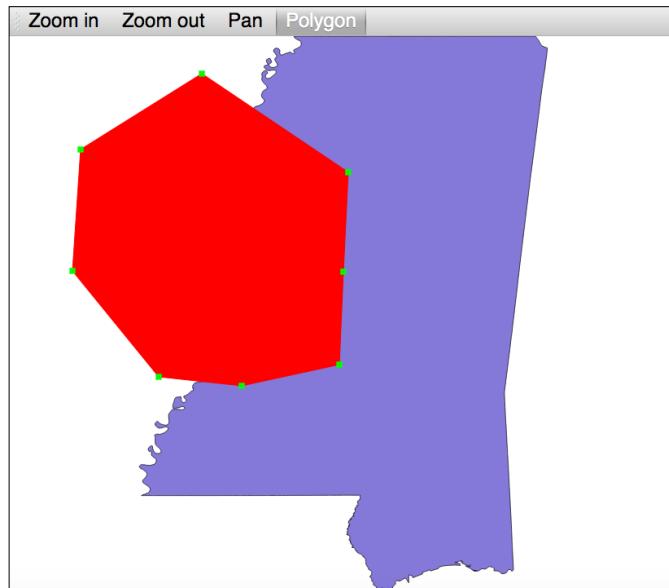
 def canvasPressEvent(self, e):
 self.point = self.toMapCoordinates(e.pos())
 m = QgsVertexMarker(self.canvas)
 m.setCenter(self.point)
 m.setColor(QColor(0, 255, 0))
 m.setIconSize(5)
 m.setIconType(QgsVertexMarker.ICON_BOX)
 m.setPenWidth(3)
```

```
self.points.append(self.point)
self.isEmittingPoint = True
self.showPoly()

def showPoly(self):
 self.rubberband.reset(QGis.Polygon)
 for point in self.points[:-1]:
 self.rubberband.addPoint(point, False)
 self.rubberband.addPoint(self.points[-1], True)
 self.rubberband.show()
```

## How it works...

All the settings for the polygon are contained in the custom class. There is a key property, called **EmittingPoint**, which we use to detect whether we are still adding points to the polygon. This value starts out as `false`. If this is the case, we reset our polygon object and begin drawing a new one. The following screenshot shows a polygon drawn with this tool on a map:



## Building a custom selection tool

In this recipe, we will build a custom tool that both draws a shape on the map and interacts with other features on the map. These two basic functions are the basis for almost any map tool you would want to build, either in a standalone QGIS application like this one, or by extending the QGIS desktop application with a plugin.

### Getting ready

We will use the application framework from the *Adding standard map tools to the canvas* recipe, so complete that recipe first. We will extend that application with a new tool. The complete version of this application is available in the code samples provided with this book. It will also be beneficial to study the other two tool-related recipes, *A map tool to draw polygons or lines on the canvas* and *A map tool to draw points on the canvas*, as this recipe builds on them as well.

You will also need the following zipped shapefile from [https://geospatialpython.googlecode.com/files/NYC\\_MUSEUMS\\_GEO.zip](https://geospatialpython.googlecode.com/files/NYC_MUSEUMS_GEO.zip).

Download and extract it to your `qgis_data` directory.

### How to do it...

We will add a new tool to the toolbar and also create a class describing our selection tool, including how to draw the selection polygon and how to select the features. To do this, we need to perform the following steps:

1. First, we define our polygon tool's button in the actions portion of our application. Place this line after the `QAction("Pan")` method:  
`actionSelect = QAction("Select", self)`
2. In the next section, we make sure that when we click on the button, it stays selected:  
`actionSelect.setCheckable(True)`
3. In the section after that, we define the method used when the button is triggered:  
`self.connect(actionSelect, SIGNAL("triggered()"), self.select)`
4. Now, we add the button to the toolbar along with the other buttons:  
`self.toolbar.addAction(actionSelect)`

5. Then, we link the application to our specialized tool class:

```
self.toolSelect = SelectMapTool(self.canvas, self.lyr)
self.toolSelect.setAction(actionSelect)
```

6. We set the point tool to be selected when the application loads:

```
self.select()
```

7. Now, we define the method in the main application class for our tool:

```
def select(self):
 self.canvas.setMapTool(self.toolSelect)
```

8. Next, we create a class that describes the type of tool we have and how it works. The output is a point on the canvas defined in the canvasPressEvent method, which receives the button click-event and the selectPoly method. We will inherit from a generic tool to create points called the QgsMapToolEmitPoint; we will also use an object called QgsRubberBand to handle polygons. However, we must also perform the selection process to highlight the features that fall within our selection polygon:

```
class SelectMapTool(QgsMapToolEmitPoint):
 def __init__(self, canvas, lyr):
 self.canvas = canvas
 self.lyr = lyr
 QgsMapToolEmitPoint.__init__(self, self.canvas)
 self.rubberband = QgsRubberBand(self.canvas, QGis.Polygon)
 self.rubberband.setColor(QColor(255,255,0,50))
 self.rubberband.setWidth(1)
 self.point = None
 self.points = []

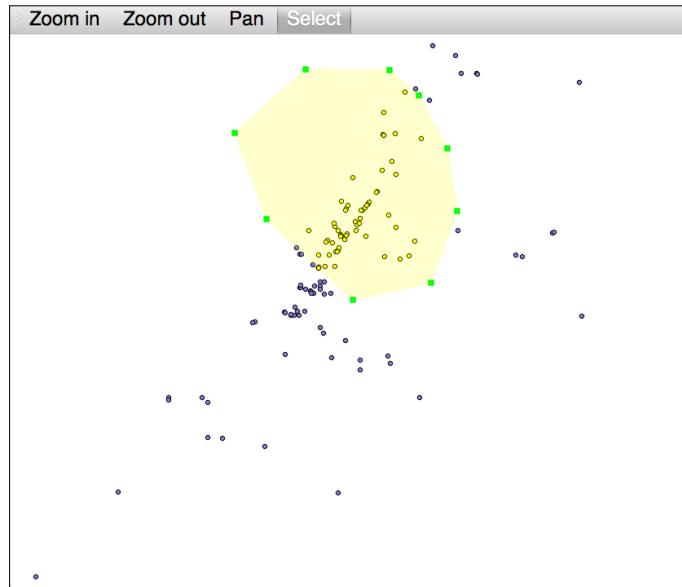
 def canvasPressEvent(self, e):
 self.point = self.toMapCoordinates(e.pos())
 m = QgsVertexMarker(self.canvas)
 m.setCenter(self.point)
 m.setColor(QColor(0,255,0))
 m.setIconSize(5)
 m.setIconType(QgsVertexMarker.ICON_BOX)
 m.setPenWidth(3)
 self.points.append(self.point)
 self.isEmittingPoint = True
```

```
self.selectPoly()

def selectPoly(self):
 self.rubberband.reset(QGis.Polygon)
 for point in self.points[:-1]:
 self.rubberband.addPoint(point, False)
 self.rubberband.addPoint(self.points[-1], True)
 self.rubberband.show()
 if len(self.points) > 2:
 g = self.rubberband.asGeometry()
 featsPnt = self.lyr.getFeatures(QgsFeatureRequest().
 setFilterRect(g.boundingBox()))
 for featPnt in featsPnt:
 if featPnt.geometry().within(g):
 self.lyr.select(featPnt.id())
```

## How it works...

QGIS has a generic tool for highlighting features, but in this case, we can use the standard selection functionality, which simplifies our code. With the exception of a dialog to load new layers and the ability to show attributes, we have a very basic but nearly complete standalone GIS application. The following screenshot shows the selection tool in action:



## Creating a mouse coordinate tracking tool

In this recipe, we'll build a tool that tracks and displays the mouse coordinates in real time. This tool will also demonstrate how to interact with the status bar of a QGIS application.

### Getting ready

We will use the application framework from the *Adding standard map tools to the canvas* recipe, so complete that recipe first. We will extend that application with the coordinate tracking tool. A complete version of this application is available in the code samples provided with this book. It will also be beneficial to study the other two tool-related recipes in this chapter, *A map tool to draw polygons or lines on the canvas* and *A map tool to draw points on the canvas*, as this recipe builds on them as well.

### How to do it...

We will add an event filter to the basic standalone QGIS application and use it to grab the current mouse coordinates as well as update the status bar. To do this, we need to perform the following steps:

1. As the last line of our application's `__init__` method, insert the following line to create a default status bar message when the application loads:

```
self.statusBar().showMessage(u"x: --, y: --")
```

2. Immediately after the application's `__init__` method, we will add the following event filter method:

```
defeventFilter(self, source, event):
 ifevent.type() == QEvent.MouseMove:
 ifevent.buttons() == Qt.NoButton:
 pos = event.pos()
 x = pos.x()
 y = pos.y()
 p = self.canvas.getCoordinateTransform().toMapCoordinates(x, y)
 self.statusBar().showMessage(u"x: %s, y: %s" % (p.x(), p.y()))
 else:
 pass
 return QMainWindow.eventFilter(self, source, event)
```

3. In the MainApp class, as the second-last line, we must install the event filter using the following code:

```
self.installEventFilter(wdg)
```

## How it works...

In the Qt framework, in order to watch out for mouse events, we must insert an event filter that allows us to monitor all the events in the application. Within the default event filter method, we can then process any event we want. In this case, we watch for any movements of the mouse.

# 6

## Composing Static Maps

In this chapter, we will cover the following recipes:

- ▶ Creating the simplest map renderer
- ▶ Using the map composer
- ▶ Adding labels to a map for printing
- ▶ Adding a scale bar to the map
- ▶ Adding a north arrow to the map
- ▶ Adding a logo to the map
- ▶ Adding a legend to the map
- ▶ Adding a custom shape to the map
- ▶ Adding a grid to the map
- ▶ Adding a table to the map
- ▶ Saving a map to a PNG image
- ▶ Adding a world file to a map image
- ▶ Saving a map to a project
- ▶ Loading a map from a project

### Introduction

In this chapter, we'll create maps using PyQGIS, Qt image objects, and QGIS Map Composer to create map layouts that can be exported as documents or images. The QGIS Map Composer is designed to create static map layouts with decorative and reference elements, for printing or inclusion in another document.

## Creating the simplest map renderer

In order to turn a dynamic GIS map into a static map image or document, you must create a renderer to **freeze** the map view and create a graphic version of it. In this recipe, we'll render a map to a JPEG image and save it.

### Getting ready

You will need to download the following zipped shapefile and extract it to your `qgis_data` directory, to a subdirectory named `hancock`:

<https://geospatialpython.googlecode.com/svn/hancock.zip>

You will also need to open the **Python Console** under the **Plugins** menu in QGIS. You can run these lines of code inside the console.

### How to do it...

In this recipe, we will load our shapefile, add it to the map, create a blank image, set up the map view, render the map image, and save it. To do this, we need to perform the following steps:

1. First, we need to import the underlying Qt libraries required for image handling:

```
from PyQt4.QtGui import *
from PyQt4.QtCore import *
```

2. Next, we load the layer and add it to the map:

```
lyr = QgsVectorLayer("/qgis_data/hancock/hancock.shp", "Hancock",
"ogr")
reg = QgsMapLayerRegistry.instance()
reg.addMapLayer(lyr)
```

3. Now, we create a blank image to accept the map image:

```
i = QImage(QSize(600,600), QImage.Format_ARGB32_Premultiplied)
c = QColor("white")
i.fill(c.rgb())
p = QPainter()
p.begin(i)
```

4. Then, we access the map renderer:

```
r = QgsMapRenderer()
```

5. Now, we get the IDs of the map layers:

```
lyrs = reg.mapLayers().keys()
```

6. Then, we use the newly initialized renderer layers in the map:

```
r.setLayerSet(lyrs)
```

7. Now, we get the full extent of the map as a rectangle:

```
rect = QgsRectangle(r.fullExtent())
```

8. Then, we set a scale for the renderer. Smaller numbers produce a larger map scale, and larger numbers produce a smaller map scale. We can change the map scale to create a buffer around the map image:

```
rect.scale(1.1)
```

9. Next, we set the extent of the renderer to the rectangle:

```
r.setExtent(rect)
```

10. Now we set the output size and resolution of the image. The resolution is automatically calculated:

```
r.setOutputSize(i.size(), i.logicalDpiX())
```

11. Now, we can render the map and finalize the image:

```
r.render(p)
```

```
p.end()
```

12. Finally, we save the map image:

```
i.save("/qgis_data/map.jpg", "jpg")
```

13. Verify that you have a map image in your `qgis_data` directory, similar to the map displayed in QGIS.

## How it works...

QGIS uses the underlying Qt GUI library to create common image types. We haven't used any of the QGIS composer objects to render the image; however, this rendering technique is used to save maps created with the QGIS composer.

## There's more...

The QImage object supports other image formats as well. To save a map image to a PNG, replace the last step in the *How to do it...* section with the following code:

```
i.save("/qgis_data/map.png", "png")
```

## Using the map composer

The QGIS Map Composer allows you to combine a map with nonspatial elements that help enhance our understanding of the map. In this recipe, we'll create a basic map composition. A composition requires you to define the physical paper size and output format. Even a simple composition example such as this has over 20 lines of configuration options.

## Getting ready

You will need to download the following zipped shapefile and extract it to your `qgis_data` directory, to a subdirectory named `hancock`:

```
https://geospatialpython.googlecode.com/svn/hancock.zip
```

You will also need to open the **Python Console** under the **Plugins** menu in QGIS. You can run this recipe in the console or wrap it in a script for the **Script Runner** plugin, using the template provided with the plugin.

## How to do it...

In this recipe, the major steps are to load the shapefile into a map, build the map composition, and render it to an image, described as follows:

1. First, we need to import the Qt libraries for image handling:

```
from PyQt4.QtGui import *
from PyQt4.QtCore import *
```

2. Next, we load the layer and add it to the map:

```
lyr = QgsVectorLayer("/qgis_data/hancock/hancock.shp", "Hancock",
"ogr")
reg = QgsMapLayerRegistry.instance()
reg.addMapLayer(lvr)
```

3. Now, we create a blank image to accept the map image:

```
i = QImage(QSize(600,600), QImage.Format_ARGB32_Premultiplied)
c = QColor("white")
i.fill(c.rgb())
p = QPainter()
p.begin(i)
```

4. Next, we get the IDs of the map layers:

```
lyrs = reg.mapLayers().keys()
```

5. Then, we access the map renderer:

```
mr = iface.mapCanvas().mapRenderer()
```

6. We then use the newly initialized renderer layers in the map:

```
mr.setLayerSet(lyrs)
```

7. Now, we get the full extent of the map as a rectangle:

```
rect = QgsRectangle(lyr.extent())
```

8. Then, we set the scale for the renderer. Smaller numbers produce a larger map scale, and larger numbers produce a smaller map scale to add an image buffer around the map:

```
rect.scale(1.2)
```

9. Now, we set the map renderer's extent to the full map's extent:

```
mr.setExtent(rect)
```

10. Next, we begin using the QGIS composer by creating a new composition and assigning it the map renderer:

```
c = QgsComposition(mr)
```

11. Then, we set the composition style. We will define it as Print, which will allow us to create both PDF documents and images. The alternative is to define it as a postscript, which is often used for direct output to printer devices:

```
c.setPlotStyle(QgsComposition.Print)
```

12. Now, we define our paper size, which is specified in millimeters. In this case, we will use the equivalent of an 8.5 x 11 inch sheet of paper, which is the US letter size:

```
c.setPaperSize(215.9, 279.4)
```

13. Next, we'll calculate dimensions for the map so that it takes up approximately half the page and is centered:

```
w, h = c.paperWidth() * .50, c.paperHeight() * .50
x = (c.paperWidth() - w) / 2
y = ((c.paperHeight() - h)) / 2
```

14. Then, we create the map composer object and set its extent:

```
composerMap = QgsComposerMap(c,x,y,w,h)
composerMap.setNewExtent(rect)
```

15. Next, we give the map a frame around its border and add it to the page:

```
composerMap.setFrameEnabled(True)
c.addItem(composerMap)
```

16. Now, we ensure that the resolution of the composition is set. The resolution defines how much detail the output contains. Lower resolutions contain less detail and create smaller files. Higher resolutions provide more image detail but create larger files:

```
dpi = c.printResolution()
c.setPrintResolution(dpi)
```

17. We now convert the dots-per-inch resolution to dots-per-millimeter:

```
mm_in_inch = 25.4
dpmm = dpi / mm_in_inch
width = int(dpmm * c.paperWidth())
height = int(dpmm * c.paperHeight())
```

18. Next, we initialize the image:

```
image = QImage(QSize(width, height), QImage.Format_ARGB32)
image.setDotsPerMeterX(dpmm * 1000)
image.setDotsPerMeterY(dpmm * 1000)
image.fill(0)
```

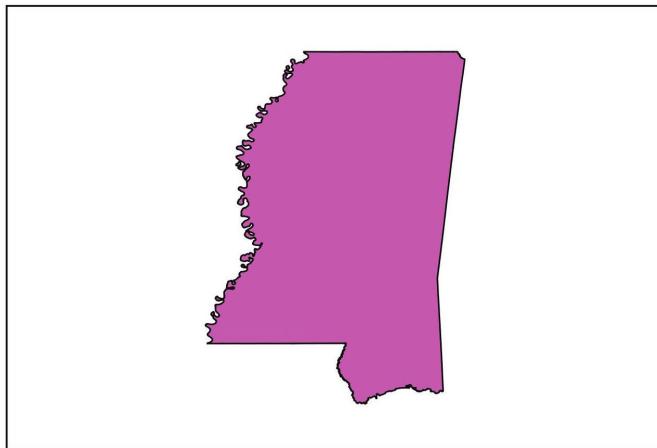
19. Now, we render the composition:

```
imagePainter = QPainter(image)
sourceArea = QRectF(0, 0, c.paperWidth(), c.paperHeight())
targetArea = QRectF(0, 0, width, height)
c.render(imagePainter, targetArea, sourceArea)
imagePainter.end()
```

20. Finally, we save the composition as a JPEG image:

```
image.save("/Users/joellawhead/qgis_data/map.jpg", "jpg")
```

Verify that the output image resembles the following sample image:



### How it works...

Map compositions are very powerful, but they can also be quite complex. You are managing the composition that represents a virtual sheet of paper. On that composition, you place objects, such as the map. Then, you must also manage the rendering of the composition as an image. All these items are independently configurable, which can sometimes lead to unexpected results with the sizing or visibility of items.

### There's more...

In the upcoming versions of QGIS, the map composer class may be renamed as the `print_layout` class. You can find out more information about this proposed change at <https://github.com/qgis/QGIS-Enhancement-Proposals/pull/9>

## Adding labels to a map for printing

The `QgsComposition` object allows you to place arbitrary text anywhere in the composition. In this recipe, we'll demonstrate how to add a label to a map composition.

### Getting ready

You will need to download the following zipped shapefile and extract it to your `qgis_data` directory, to a subdirectory named `hancock`:

<https://geospatialpython.googlecode.com/svn/hancock.zip>

In addition to the shapefile, you will also need the `MapComposer` class. This class encapsulates the boilerplate composer code in a reusable way to make adding other elements easier. You can download it from <https://geospatialpython.googlecode.com/svn/MapComposer.py>.

This file must be accessible from the QGIS Python console by ensuring that it is in the Python path directory. Place the file in the `.qgis2/python` directory within your home directory.

## How to do it...

To add a label to a composition, we'll first build the map composition, create a label, and then save the composition as an image. To do this, we need to perform the following steps:

1. First, we need to import the Qt GUI libraries and the `MapComposer` class:

```
from PyQt4.QtGui import *
from PyQt4.QtCore import *
import MapComposer
```

2. Next, we create a layer with the shapefile, setting the path to the shapefile in order to match your system:

```
lyr = QgsVectorLayer("/Users/joellawhead/qgis_data/hancock/
hancock.shp", "Hancock", "ogr")
```

3. Now, we add this layer to the map:

```
reg = QgsMapLayerRegistry.instance()
reg.addMapLayer(lyr)
```

4. Next, we access the map renderer:

```
mr = iface.mapCanvas().mapRenderer()
```

5. Then, we create a `MapComposer` object, passing in the map layer registry and the map renderer:

```
qc = MapComposer.MapComposer(qmlr=reg, qmr=mr)
```

6. Now, we create a new label object:

```
qc.label = QgsComposerLabel(qc.c)
```

7. We can set the label text to any string:

```
qc.label.setText("Hancock County")
```

8. We can automatically set the size of the label container to fit the string we used:

```
qc.label.adjustSizeToText()
```

9. Now, we add a frame around the label box:

```
qc.label.setFrameEnabled(True)
```

10. Then, we set the position of the label on the page, which is at the top-left corner of the map:

```
qc.label.setItemPosition(qc.x, qc.y-10)
```

11. Next, we add the label to the map composition now that it is configured:

```
qc.c.addItem(qc.label)
```

12. Finally, we save the composition image:

```
qc.output("/Users/joellawhead/qgis_data/map.jpg", "jpg")
```

13. Verify that your output image has a text label in a frame at the top-left corner of the map.

## How it works...

In this case, we created a very simple label based on defaults. However, labels can be customized to change the font, size, color, and style for print-quality compositions. Also, note that the x,y values used to place items in a composition start in the upper-left corner of the page. As you move an item down the page, the y value increases.

## Adding a scale bar to the map

A scale bar is one of the most important elements of a map composition, as it defines the scale of the map to determine the ground distance on the map. QGIS composer allows you to create several different types of scale bars from a simple text scale ratio to a graphical, double scale bar with two measurement systems. In this recipe, we'll create a scale bar that measures in kilometres.

## Getting ready

You will need to download the following zipped shapefile and extract it to your `qgis_data` directory, to a subdirectory named `ms`:

<https://geospatialpython.googlecode.com/svn/mississippi.zip>

In addition to the shapefile, you will also need the `MapComposer` class. This class encapsulates the boilerplate composer code in a reusable way to make adding other elements easier. You can download it from <https://geospatialpython.googlecode.com/svn/MapComposer.py>.

This file must be accessible from the QGIS Python console; ensure that it is in the Python path directory. Place the file in the `.qgis2/python` directory within your home directory.

For the scale bar to display correctly, you must ensure that QGIS is set to automatically reproject data on the fly. In QGIS, go to the **Settings** menu and select **Options**. In the **Options** dialog, select the **CRS** panel. In the **Default CRS for new projects** section, check the **Enable 'on the fly' reprojection by default** radio button. Click on the **OK** button to confirm the setting.

## How to do it...

First, we will generate the map, then we'll generate the composition, and finally we'll create the scale bar and place it in the lower-right corner of the map. To do this, we need to perform the following steps:

1. First, we need to import the libraries we'll need:

```
from PyQt4.QtCore import *
from PyQt4.QtGui import *
from qgis.core import *
from qgis.gui import *
import MapComposer
```

2. Then, we'll build the map renderer using the shapefile:

```
lyr = QgsVectorLayer("/Users/joellawhead/qgis_data/ms/
mmississippi.shp", "Mississippi", "ogr")
reg = QgsMapLayerRegistry.instance()
reg.addMapLayer(lyr)
mr = iface.mapCanvas().mapRenderer()
```

3. Next, we'll create the `MapComposer` object using the layer registry and map renderer:

```
qc = MapComposer.MapComposer(qmlr=reg, qmr=mr)
```

4. Now, we'll initialize the scale bar object:

```
qc.scalebar = QgsComposerScaleBar(qc.c)
```

5. Then, we define the scale bar type. The default is a text scale, but we'll create a more traditional box scale bar:

```
qc.scalebar.setStyle('Single Box')
```

6. Next, we apply the scale bar to the map and set the scale bar graphic to the default size:

```
qc.scalebar.setComposerMap(qc.composerMap)
qc.scalebar.applyDefaultSize()
```

7. We use the scale bar size, map size, and map position to calculate the desired position of the scale bar, in the lower-right corner of the map:

```
sbw = qc.scalebar.rect().width()
sbh = qc.scalebar.rect().height()
mcw = qc.composerMap.rect().width()
mch = qc.composerMap.rect().height()
sbx = qc.x + (mcw - sbw)
sby = qc.y + mch
```

8. Then, we set the calculated position of the scale bar and add it to the composition:

```
qc.scalebar.setItemPosition(sbx, sby)
qc.c.addItem(qc.scalebar)
```

9. Finally, we save the composition to an image:

```
qc.output("/Users/joellawhead/qgis_data/map.jpg", "jpg")
```

## How it works...

The scale bar will display in kilometres if the map projection is set correctly, which is why it is important to have automatic reprojection enabled in the QGIS settings. The location of the scale bar within the composition is not important, as long as the `composerMap` object is applied to it.

## Adding a north arrow to the map

North arrows are another common cartographic element found even in ancient maps, which show the orientation of the map relative to either true, grid, or magnetic north. Sometimes, these symbols can be quite elaborate. However, QGIS provides a basic line arrow element that we will use in combination with a map label to make a basic north arrow.

### Getting ready

You will need to download the following zipped shapefile and extract it to your `qgis_data` directory, to a subdirectory named `ms`:

<https://geospatialpython.googlecode.com/svn/Mississippi.zip>

In addition to the shapefile, you will also need the `MapComposer` class to simplify the code needed to add this one element. If you haven't already used it in a previous recipe, you can download it from <https://geospatialpython.googlecode.com/svn/MapComposer.py>.

This file must be accessible from the QGIS Python Console; for this, you need to ensure that it is in the Python path directory. Place the file in the `.qgis2/python` directory within your home directory.

## How to do it...

In this recipe, we will create a map composition, draw an arrow to the right of the map, and then place a label with a capital letter N below the arrow. To do this, we need to perform the following steps:

1. First, we import the Qt and MapComposer Python libraries:

```
from PyQt4.QtCore import *
from PyQt4.QtGui import *
from qgis.core import *
from qgis.gui import *
import MapComposer
```

2. Next, we create the map composition object:

```
lyr = QgsVectorLayer("/qgis_data/ms/mississippi.shp",
"Mississippi", "ogr")
reg = QgsMapLayerRegistry.instance()
reg.addMapLayer(lyr)
mr = iface.mapCanvas().mapRenderer()
qc = MapComposer.MapComposer(qmlr=reg, qmr=mr)
```

3. Now, we calculate the position of the arrow along the right-hand side of the map, set its position, and then add it to the composition:

```
mcw = qc.composerMap.rect().width()
mch = qc.composerMap.rect().height()
ax = qc.x + mcw + 10
ay = (qc.y + mch) - 10
afy = ay - 20
qc.arrow = QgsComposerArrow(QPointF(ax, ay), QPointF(ax, afy),
qc.c)
qc.c.addItem(qc.arrow)
```

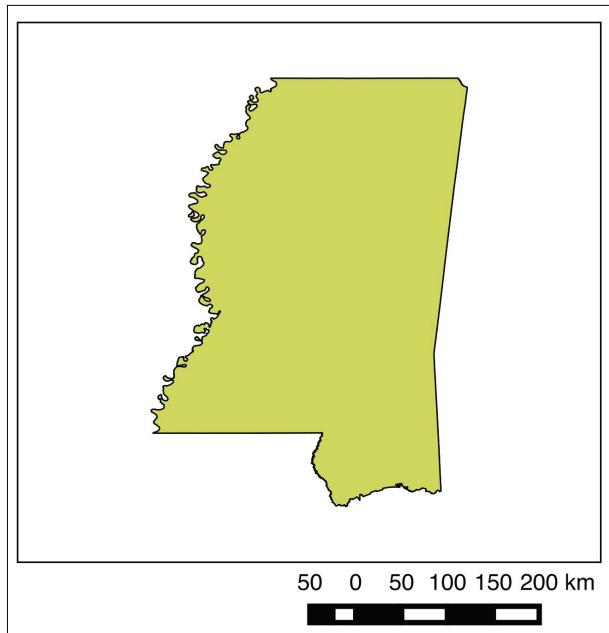
4. Then, we create a capital letter N label and add it to the composition just below the arrow:

```
f = QFont()
f.setBold(True)
f.setFamily("Times New Roman")
f.setPointSize(30)
qc.labelNorth = QgsComposerLabel(qc.c)
qc.labelNorth.setText("N")
qc.labelNorth.setFont(f)
qc.labelNorth.adjustSizeToText()
qc.labelNorth.setFrameEnabled(False)
qc.labelNorth.setItemPosition(ax - 5, ay)
qc.c.addItem(qc.labelNorth)
```

5. Finally, we save the composition to an image:

```
qc.output("/qgis_data/map.jpg", "jpg")
```

Verify that your output image looks similar to the following:



## How it works...

The QGIS composer doesn't have a dedicated north arrow or compass rose object. However, it is quite simple to construct one, as demonstrated in the preceding section. The arrow is just a graphic. The direction of the arrow is controlled by the location of the start point and the end point listed, respectively, when you create the `QgsComposerArrow` object.

## There's more...

You can extend this example to have an arrow point in multiple compass directions. You can also use an image of a more elaborate compass rose added to the composition. We'll demonstrate how to add images in the next recipe. Note that the arrow element can also be used to point to items on the map with an associated label.

## Adding a logo to the map

An important part of customizing a map is to add your logo or other graphics to the composition. In this recipe, we'll add a simple logo to the map.

### Getting ready

You will need to download the following zipped shapefile and extract it to your `qgis_data` directory, to a subdirectory named `ms`:

<https://geospatialpython.googlecode.com/svn/Mississippi.zip>

You will also need a logo image, which you can download from

<https://geospatialpython.googlecode.com/svn/trunk/logo.png>.

Place the image in your `qgis_data/rasters` directory.

If you haven't already done so in the previous recipe, download the `MapComposer` library from <https://geospatialpython.googlecode.com/svn/MapComposer.py>, to simplify the creation of the map composition.

Place the file in the `.qgis2/python` directory within your home directory.

## How to do it...

In this recipe, we will create the map composition, add the logo image, and save the map as an image. To do this, we need to perform the following steps:

1. First, we need to import the Qt GUI, core QGIS, QGIS GUI, and MapComposer libraries:

```
from PyQt4.QtCore import *
from PyQt4.QtGui import *
from qgis.core import *
from qgis.gui import *
import MapComposer
```

2. Next, we will build a basic map composition using the shapefile:

```
lyr = QgsVectorLayer("/qgis_data/ms/mississippi.shp",
"Mississippi", "ogr")
reg = QgsMapLayerRegistry.instance()
reg.addMapLayer(lyr)
mr = iface.mapCanvas().mapRenderer()
qc = MapComposer.MapComposer(qmlr=reg, qmr=mr)
```

3. Now, we initialize the picture object:

```
qc.logo = QgsComposerPicture(qc.c)
```

4. Then, we set the path of the picture to our image file:

```
qc.logo.setPictureFile("/qgis_data/rasters/logo.png")
```

5. We must set the size of the box or scene rectangle such that it is large enough to contain the logo. Otherwise, the picture will appear cropped:

```
qc.logo.setSceneRect(QRectF(0,0,50,70))
```

6. Next, we calculate the position of the logo relative to the map image. We'll place the logo near the top-left corner of the map:

```
lx = qc.x + 50
ly = qc.y - 120
```

7. Now, we set the logo's position and add it to the map composition:

```
mcw = qc.composerMap.rect().width()
mch = qc.composerMap.rect().height()
lx = qc.x
ly = qc.y - 20
```

8. Finally, we save the composition as an image:

```
qc.output("/qgis_data/map.jpg", "jpg")
```

## How it works...

This recipe is very straight forward, as the `QgsComposerPicture` is an extremely simple object. You can use JPG, PNG, or SVG images. This technique can be used to add custom north arrows or other cartographic elements as well.

## Adding a legend to the map

A map legend decodes the symbology used in a thematic GIS map for the reader. Legends are tightly integrated into QGIS, and in this recipe, we'll add the default legend from the map to the print composition.

## Getting ready

Download the shapefile for this map from <https://geospatialpython.googlecode.com/svn/Mississippi.zip> and extract it to your `qgis_data` directory in a subdirectory named `ms`.

As with the previous recipes in this chapter, we will use the `MapComposer` library from <https://geospatialpython.googlecode.com/svn/MapComposer.py> to simplify the creation of the map composition.

Place the file in the `.qgis2/python` directory within your home directory.

## How to do it...

This recipe is as simple as creating the map, adding the automatically generated legend, and saving the output to an image. To do this, we need to perform the following steps:

1. First, we will need to load the Qt and QGIS GUI libraries followed by the `MapComposer` library:

```
from PyQt4.QtCore import *
from PyQt4.QtGui import *
from qgis.core import *
from qgis.gui import *
import MapComposer
```

2. Next, we will load the shapefile as a layer and create the map composition with the MapComposer library, passing it the map layer registry and map renderer:

```
lyr = QgsVectorLayer("/qgis_data/ms/mississippi.shp",
"Mississippi", "ogr")
reg = QgsMapLayerRegistry.instance()
reg.addMapLayer(lyr)
mr = iface.mapCanvas().mapRenderer()
qc = MapComposer.MapComposer(qmlr=reg, qmr=mr)
```

3. Now, we initialize the legend object:

```
qc.legend = QgsComposerLegend(qc.c)
```

4. We now tell the legend which layer set we want to use:

```
qc.legend.model().setLayerSet(qc.qmr.layerSet())
```

5. Then, we set the legend's position to the left-hand side of the map and add it to the composition:

```
qc.legend.setItemPosition(5, qc.y)
qc.c.addItem(qc.legend)
```

6. Finally, we output the composition to the map:

```
qc.output("/qgis_data/map.jpg", "jpg")
```

## How it works...

Adding a legend is quite simple. QGIS will carry over the styling that is autogenerated when the layer is loaded or manually set by the user. Of course, you can also save layer styling, which is loaded with the layer and used by the legend. However, if you're generating a composition in the background such as in a standalone application, for example, every aspect of the legend is customizable through the PyQGIS API.

## Adding a custom shape to the map

The QGIS composer has an object for drawing and styling nonspatial shapes, including rectangles, ellipses, and triangles. In this recipe, we'll add some rectangles filled with different colors, which will resemble a simple bar chart, as an example of using shapes.

## Getting ready

Download the zipped shapefile for this map from <https://geospatialpython.googlecode.com/svn/Mississippi.zip> and extract it to your `qgis_data` directory, to in a subdirectory named `ms`.

We will also use the `MapComposer` library from <https://geospatialpython.googlecode.com/svn/MapComposer.py> to simplify the creation of the map composition.

Place the file in the `.qgis2/python` directory within your home directory.

## How to do it...

First, we will create a simple map composition with the shapefile. Then, we will define the style properties for our rectangles. Next, we will draw the rectangles, apply the symbols, and render the composition. To do this, we need to perform the following steps:

1. First, we must import the PyQGIS and Qt GUI libraries as well as the `MapComposer` library, as follows:

```
from PyQt4.QtCore import *
from PyQt4.QtGui import *
from qgis.core import *
from qgis.gui import *
import MapComposer
```

2. Next, we create the map composition by using the shapefile:

```
lyr = QgsVectorLayer("/qgis_data/ms/mississippi.shp",
"Mississippi", "ogr")
reg = QgsMapLayerRegistry.instance()
reg.addMapLayer(lyr)
mr = iface.mapCanvas().mapRenderer()
qc = MapComposer.MapComposer(qmlr=reg, qmr=mr)
```

3. Now, we create three basic fill symbols by building Python dictionaries with color properties and initialize the symbols with these dictionaries:

```
red = {'color':'255,0,0,255','color_border':'0,0,0,255'}
redsym = QgsFillSymbolV2.createSimple(red)
blue = {'color':'0,0,255,255','color_border':'0,0,0,255'}
```

- ```
bluesym = QgsFillSymbolV2.createSimple(blue)
yellow = {'color': '255,255,0,255', 'color_border': '0,0,0,255'}
yellowsym = QgsFillSymbolV2.createSimple(yellow)

4. Then, we calculate the y position of the first shape, relative to the map:
mch = qc.composerMap.rect().height()
sy = qc.y + mch

5. We create the first shape and set it to the type 1, which is a rectangle:
qc.shape1 = QgsComposerShape(10,sy-25,10,25,qc.c)
qc.shape1.setShapeType(1)

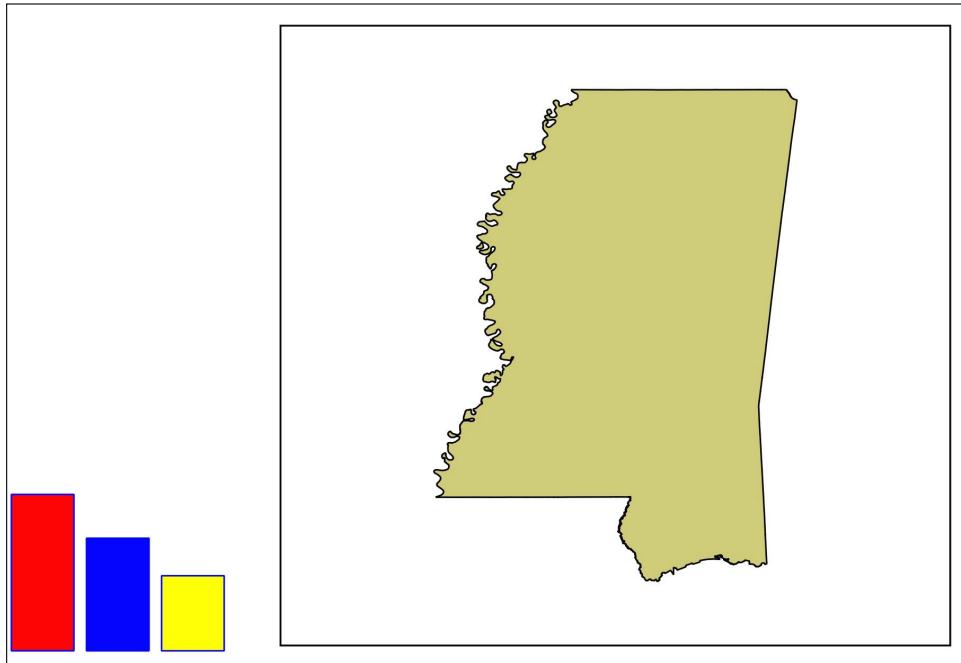
6. Next, we tell the shape to use a symbol, set the symbol for one of our three fill
symbols, and add the shape to the composition:
qc.shape1.setUseSymbolV2(True)
qc.shape1.setShapeStyleSymbol(redsym)
qc.c.addItem(qc.shape1)

7. We repeat the process with two other shapes, changing their position, size,
and symbols to make them look different:
qc.shape2 = QgsComposerShape(22,sy-18,10,18,qc.c)
qc.shape2.setShapeType(1)
qc.shape2.setUseSymbolV2(True)
qc.shape2.setShapeStyleSymbol(bluesym)
qc.c.addItem(qc.shape2)

qc.shape3 = QgsComposerShape(34,sy-12,10,12,qc.c)
qc.shape3.setShapeType(1)
qc.shape3.setUseSymbolV2(True)
qc.shape3.setShapeStyleSymbol(yellowsym)
qc.c.addItem(qc.shape3)

8. Finally, we output the composition as an image:
qc.output("/qgis_data/map.jpg", "jpg")
```

Verify that your output image looks similar to the following:



How it works...

This simple graphical output is nearly 40 lines of code. While there may be some limited uses for dealing with these shapes, in most cases, the simpler route will be to just import images. However, it provides a good foundation for a richer graphics API, as QGIS continues to evolve.

There's more...

If you are using fill symbols within a Python plugin in a QGIS version less than 2.6, you must ensure that the symbols are defined in the global scope, or QGIS will crash due to a bug. The easiest way to include the variables in the global scope is to define them immediately after the import statements. It also affects scripts that are run in the Script Runner plugin. This bug was fixed in version 2.6 and subsequent versions.

Adding a grid to the map

An annotated reference grid is useful for map products used to locate features. This recipe teaches you how to add both reference lines on a map and annotations for the lines around the edges of the map.

Getting ready

You will need a shapefile for this map from <https://geospatialpython.googlecode.com/svn/Mississippi.zip>, and you need to extract it to your `qgis_data` directory, to a subdirectory named `ms`.

As with the previous recipes in this chapter, we will use the `MapComposer` library from <https://geospatialpython.googlecode.com/svn/MapComposer.py> to simplify the creation of the map composition.

Place the file in the `.qgis2/python` directory within your home directory.

How to do it...

In this recipe, the general steps are to create the map composition, establish the overall grid parameters, define the grid line placement, and then style the grid and annotations. To do this, we need to perform the following steps:

1. First, we need to import all the GUI libraries and the `MapComposer` library:

```
from PyQt4.QtCore import *
from PyQt4.QtGui import *
from qgis.core import *
from qgis.gui import *
import MapComposer
```

2. Next, we create the map composition using the shapefile:

```
lyr = QgsVectorLayer("/qgis_data/ms/mmississippi.shp",
"Mississippi", "ogr")
reg = QgsMapLayerRegistry.instance()
reg.addMapLayer(lyr)
mr = iface.mapCanvas().mapRenderer()
qc = MapComposer.MapComposer(qmlr=reg, qmr=mr)
```

3. Now, we are going to create some variables to shorten some unusually long method and object names:

```
setGridAnnoPos = qc.composerMap.setGridAnnotationPosition  
setGridAnnoDir = qc.composerMap.setGridAnnotationDirection  
qcm = QgsComposerMap
```

4. Then, we enable the grid, set the line spacing, and use solid lines for the grid:

```
qc.composerMap.setGridEnabled(True)  
qc.composerMap.setGridIntervalX(.75)  
qc.composerMap.setGridIntervalY(.75)  
qc.composerMap.setGridStyle(qcm.Solid)
```

5. Next, we enable the annotation numbers for coordinates and set the decimal precision to 0 for whole numbers:

```
qc.composerMap.setShowGridAnnotation(True)  
qc.composerMap.setGridAnnotationPrecision(0)
```

6. Now, we go around the map composition frame and define locations and directions for each set of grid lines, using our shorter variable names from the previous steps:

```
setGridAnnoPos(qcm.OutsideMapFrame, qcm.Top)  
setGridAnnoDir(qcm.Horizontal, qcm.Top)  
setGridAnnoPos(qcm.OutsideMapFrame, qcm.Bottom)  
setGridAnnoDir(qcm.Horizontal, qcm.Bottom)  
setGridAnnoPos(qcm.OutsideMapFrame, qcm.Left)  
setGridAnnoDir(qcm.Vertical, qcm.Left)  
setGridAnnoPos(qcm.OutsideMapFrame, qcm.Right)  
setGridAnnoDir(qcm.Vertical, qcm.Right)
```

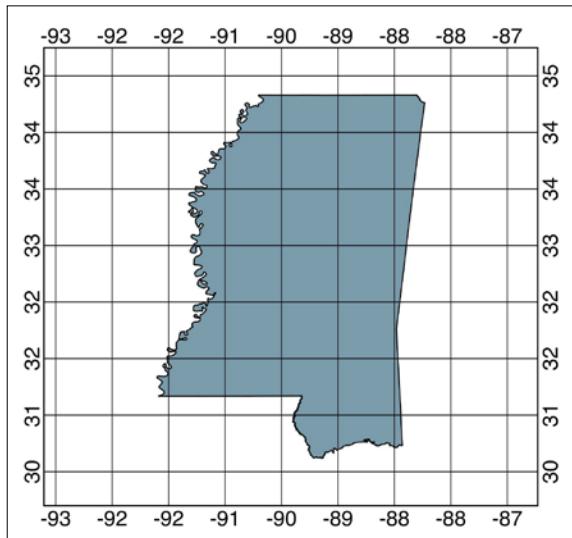
7. Finally, we set some additional styling for the grid lines and annotations before adding the whole map to the overall composition:

```
qc.composerMap.setAnnotationFrameDistance(1)  
qc.composerMap.setGridPenWidth(.2)  
qc.composerMap.setGridPenColor(QColor(0, 0, 0))  
qc.composerMap.setAnnotationFontColor(QColor(0, 0, 0))  
qc.c.addComposerMap(qc.composerMap)
```

8. We output the composition to an image:

```
qc.output("/qgis_data/map.jpg", "jpg")
```

Verify that your output image looks similar to the following:



How it works...

This recipe has a lot of steps because the grids are customizable. The order of operations is important as well. Notice that we do not add the map to the composition until the very end. Often, you will make what seem to be minor changes and the grid may not render. Hence, modify this recipe carefully.

Adding a table to the map

QGIS composer provides an object to add a table to a composition, representing either the attributes of a vector layer or an arbitrary text table you create. In this recipe, we'll add a table to the composition with the attributes of our map layer shapefile.

Getting ready

Download the shapefile for this map from <https://geospatialpython.googlecode.com/svn/Mississippi.zip> and extract it to your `qgis_data` directory, to a subdirectory named `ms`.

As with the previous recipes in this chapter, we will use the `MapComposer` library from <https://geospatialpython.googlecode.com/svn/MapComposer.py> to simplify the creation of the map composition.

Place the file in the `.qgis2/python` directory within your home directory.

How to do it...

The following steps will create a map composition, add the table, and output the composition to an image:

1. First, we import our GUI libraries and the MapComposer library:

```
from PyQt4.QtCore import *
from PyQt4.QtGui import *
from qgisfromqgis.core import *
from qgisfromqgis.gui import *
import MapComposer
```

2. Next, we create the map composition:

```
lyr = QgsVectorLayer("/qgis_data/ms/mississippi.shp",
"Mississippi", "ogr")
reg = QgsMapLayerRegistry.instance()
reg.addMapLayer(lyr)
mr = iface.mapCanvas().mapRenderer()
qc = MapComposer.MapComposer(qmlr=reg, qmr=mr)
```

3. Now, we can initialize the table object:

```
qc.table = QgsComposerAttributeTable(qc.c)
```

4. Then, we reference the related map:

```
qc.table.setComposerMap(qc.composerMap)
```

5. Next, we can specify the layer whose attributes we want to display in the table:

```
qc.table.setVectorLayer(lyr)
```

6. Now, we can position the table below the map and add it to the composition:

```
mch = qc.composerMap.rect().height()
qc.table.setItemPosition(qc.x, qc.y + mch + 20)
qc.c.addItem(qc.table)
```

7. Finally, we output the composition to an image:

```
qc.output("/qgis_data/map.jpg", "jpg")
```

How it works...

The table object is very straight forward. Using the attributes of a vector layer is automatic. You can also build the table cell by cell if you want to display customized information.

Adding a world file to a map image

Exporting a map as an image removes all of its spatial information. However, you can create an external text file called a **world file**, which provides the georeferencing information for the raster image, so that it can be used by GIS software, including QGIS, as a raster layer. In this recipe, we'll export a map composition as an image and create a world file with it.

Getting ready

You will need to download the zipped shapefile from <https://geospatialpython.googlecode.com/svn/Mississippi.zip> and extract it to your `qgis_data` directory, to a subdirectory named `ms`.

In addition to the shapefile, you will also need the `MapComposer` class to simplify the code needed to add this one element. If you have not already used it in a previous recipe, you can download it from <https://geospatialpython.googlecode.com/svn/MapComposer.py>.

This file must be accessible from the QGIS Python console; for this, you need to ensure that it is in the python path directory. Place the file in the `.qgis2/python` directory within your home directory.

How to do it...

First, we'll create the map composition, then we'll save it as an image, and finally we'll generate the world file. To do this, we need to perform the following steps:

1. First, we need to import the GUI and `MapComposer` libraries:

```
from PyQt4.QtCore import *
from PyQt4.QtGui import *
from qgisfromqgis.core import *
from qgisfromqgis.gui import *
import MapComposer
```

2. Next, we'll create the map's composition using the MapComposer libraries:

```
lyr = QgsVectorLayer("/qgis_data/ms/mississippi.shp",
"Mississippi", "ogr")

reg = QgsMapLayerRegistry.instance()

reg.addMapLayer(lyr)

mr = iface.mapCanvas().mapRenderer()

qc = MapComposer.MapComposer(qmlr=reg, qmr=mr)
```

3. Now, we'll define the name of our output file:

```
output = "/qgis_data/map"
```

4. Then, we can export the composition as an image:

```
qc.output(output + ".jpg", "jpg")
```

5. Now, we'll create an object that contains the world file's information:

```
qc.c.setWorldFileMap(qc.composerMap)
qc.c.setGenerateWorldFile(True)
wf = qc.c.computeWorldFileParameters()
```

6. Finally, we'll open a text file and write each line of the text file:

```
with open(output + ".jgw", "w") as f:
    f.write("%s\n" % wf[0])
    f.write("%s\n" % wf[1])
    f.write("%s\n" % wf[3])
    f.write("%s\n" % wf[4])
    f.write("%s\n" % wf[2])
    f.write("%s\n" % wf[5])
```

How it works...

The world file contains the ground distance per pixel and the upper-left coordinate of the map image. The QGIS composer automatically generates this information based on the referenced map. The world file's name must be the same as the image with an extension that uses the first and last letter of the image file extension plus the letter w. For example, a .TIFF image file will have a world file with the extension .TFW. You can learn more about what the world file variables in each line mean at http://en.wikipedia.org/wiki/World_file.

Saving a map to a project

Saving a project automatically can be useful for autosave features or as part of a process to autogenerate projects from dynamically updated data. In this recipe, we'll save a QGIS project to a .qgs project file.

Getting ready

You will need to download the following zipped shapefile and extract it to your `qgis_data` directory, to a subdirectory named `ms`:

<https://geospatialpython.googlecode.com/svn/Mississippi.zip>

How to do it...

We will create a simple QGIS project by loading a shapefile layer, then we'll access the project object, and save the map project to a file, as follows:

1. First, we need the Qt core library in the QGIS Python console:

```
from PyQt4.QtCore import *
```

2. Next, we load the shapefile and add it to the map:

```
lyr = QgsVectorLayer("/Users/joellawhead/qgis_data/ms/mississippi.
shp", "Mississippi", "ogr")
reg = QgsMapLayerRegistry.instance()
reg.addMapLayer(lyr)
```

3. Then, we create a `file` object to save our project:

```
f = QFileInfo("/Users/joellawhead/qgis_data/myProject.qgs")
```

4. Now, we can access the QGIS project object instance:

```
p = QgsProject.instance()
```

5. Finally, we can save the project by writing it to the file object:

```
p.write(f)
```

How it works...

QGIS simply creates an XML document with all the project settings and GIS map settings. You can read and even modify the XML output by hand.

Loading a map from a project

This recipe demonstrates how to load a project from a .qgs XML file. Loading a project will set up the map and project settings for a previously saved project within QGIS.

Getting ready

You will need to complete the previous recipe, *Saving a map to a project*, so that you have a project named `myProject.qgs` in your `qgis_data` folder.

How to do it...

For this recipe, you need to set up a file object, set a resource path, and then read the file object that references the project file. To do this, you need to perform the following steps:

1. First, we import the core Qt library for the file object:

```
from PyQt4.QtCore import *
```

2. Next, we initiate the file object with the path to the project file:

```
f = QFileInfo("/Users/joellawhead/qgis_data/myProject.qgs")
```

3. Now, we access the project object:

```
p = QgsProject.instance()
```

4. Then, we set the resource path for QGIS to find data and other files, in case the project was saved with relative paths instead of absolute paths:

```
p.readPath("/Users/joellawhead/qgis_data/")
```

5. Finally, we tell the project object to read the project file in order to load the map:

```
p.read(f)
```

How it works...

QGIS has a setting to save references to data and other files either as relative paths, which are relative to the project file, or absolute paths, which contain the full path. If the saved paths are absolute, PyQGIS will be unable to locate data sources. Setting the read path to the full system path of the project file ensures that QGIS can find all the referenced files in the project file, if they are saved as relative paths.

7

Interacting with the User

In this chapter, we will cover the following recipes:

- ▶ Using log files
- ▶ Creating a simple message dialog
- ▶ Creating a warning dialog
- ▶ Creating an error dialog
- ▶ Displaying a progress bar
- ▶ Creating a simple text input dialog
- ▶ Creating a file input dialog
- ▶ Creating a combobox
- ▶ Creating radio buttons
- ▶ Creating checkboxes
- ▶ Creating tabs
- ▶ Stepping the user through a wizard
- ▶ Keeping dialogs on top

Introduction

QGIS has been built using the comprehensive graphical user interface framework called Qt. Both QGIS and Qt have Python APIs. In this chapter, we'll learn how to interact with the user in order to collect and display information outside the default QGIS interface. Qt has excellent documentation of its own, and since QGIS is built on top of Qt, all of this documentation applies to QGIS. You can find the Qt documentation at <http://qt-project.org>.

Using log files

Log files provide a way to track exactly what is going on in a Python plugin or script, by creating messages that are available even if the script or QGIS crashes. These log messages make troubleshooting easier. In this recipe, we'll demonstrate two methods used for logging. One method is using actual log files on the filesystem, and the other is using the QGIS **Log Messages** window, which is available by clicking on the yellow triangle with an exclamation point at the bottom-right corner of the QGIS application window, or by selecting **View** menu, then clicking on **Panels**, and then checking **Log Messages**.

Getting ready

To use log files, we must configure the `QGIS_LOG_FILE` environment variable by performing the following steps so that QGIS knows where to write log messages:

1. From the QGIS **Settings** menu, select **Options**.
2. In the **Options** dialog, select **System** panel.
3. In the **System** panel, scroll down to the **Environment** section.
4. In the **Environment** section, check the **Use custom variables** checkbox.
5. Click on the **Add** button.
6. In the **Variable** field, enter `QGIS_LOG_FILE`.
7. In the **Value** field, enter `/qgis_data/log.txt` or the path to another directory where you have write permissions.
8. Click on the **OK** button to close the **Options** dialog.
9. Restart QGIS for the environment variable to take effect.

How to do it...

We will write a message to our custom log file configured in the previous section, and then write a message to the tabbed QGIS **Log Messages** window. To do this, we need to perform the following steps:

1. First, open the **Python Console** in QGIS.
2. Next, we'll write the following log file message:

```
QgsLogger.logMessageToFile("This is a message to a log file.")
```

3. Then, we'll write a message to the QGIS **Log Messages** window, specifying the message as the first argument and a name for the tab in which the message will appear:

```
QgsMessageLog.logMessage("This is a message from the Python Console", "Python Console")
```

4. Now, open the log file and check whether the message has appeared.
5. Finally, open the QGIS **Log Messages** window, click on the **Python Console** tab, and verify that the second log message appears.

How it works...

The traditional log file provides a simple and portable way to record information from QGIS using Python. The **Log Messages** window is a more structured way to view information from many different sources, with a tabbed interface and a convenient timestamp on each message. In most cases, you'll probably want to use the **Log Messages** window because QGIS users are familiar with it. However, use it sparingly. It's OK to log lots of messages when testing code, but restrict logging for plugins or applications to serious errors only. Heavy logging – for example, logging messages while looping over every feature in a layer – can slow down QGIS or even cause it to crash.

Creating a simple message dialog

Message dialogs pop up to grab the user's attention and to display important information. In this recipe, we'll create a simple information dialog.

Getting ready

Open the QGIS **Python Console** by going to the **Plugins** menu and selecting **Python Console**.

How to do it...

We will create a message dialog and display some text in it, as follows:

1. First, we need to import the GUI library:

```
from PyQt4.QtGui import *
```

2. Then, we'll create the message dialog:

```
msg = QMessageBox()
```

3. Next, we'll set the message we want to display:

```
msg.setText("This is a simple information message.")
```

4. Finally, we call the execution method to display the message dialog:

```
msg.show()
```

How it works...

Note that we are directly using the underlying Qt framework from which QGIS is built. QGIS API's objects begin with `Qgs`, while Qt objects begin with just the letter `Q`.

There's more...

A message dialog box should also be used sparingly because it is a popup that can become annoying to the user or can get lost in the array of open windows and dialogs on a user's desktop. The preferred method for a QGIS information message is to use the `QgsMessageBar()` method, which is well-documented in the PyQGIS Developer Cookbook found at http://docs.qgis.org/testing/en/docs/pyqgis_developer_cookbook/communicating.html

Creating a warning dialog

Sometimes, you need to notify a user when an issue is detected, which might lead to problems if the user continues. This situation calls for a warning dialog, which we will demonstrate in this recipe.

Getting ready

Open the QGIS **Python Console** by going to the **Plugins** menu and selecting **Python Console**.

How to do it...

In this recipe, we will create a dialog, set the warning message and a warning icon, and display the dialog, as follows:

1. First, we import the GUI library:

```
from PyQt4.QtGui import *
```

2. Next, we initialize the warning dialog:

```
msg = QMessageBox()
```

3. Then, we set the warning message:

```
msg.setText("This is a warning...")
```

4. Now, add a warning icon to the dialog that has an enumeration index of 2:

```
msg.setIcon(QMessageBox.Warning)
```

5. Finally, we call the execution method to display the dialog:

```
msg.show()
```

How it works...

Message dialogs should be used sparingly because they interrupt the user experience and can easily become annoying. However, sometimes it is important to prevent a user from taking an action that may cause data corruption or a program to crash.

Creating an error dialog

You can issue an error dialog box when you need to end a process due to a serious error. In this recipe, we'll create an example of an error dialog.

Getting ready

Open the **QGIS Python Console** by selecting the **Plugins** menu and then clicking on **Python Console**.

How to do it...

In this recipe, we will create a dialog, assign an error message, set an error icon, and display the dialog, as follows:

1. First, we need to import the GUI library:

```
from PyQt4.QtGui import *
```

2. Next, we initialize the dialog:

```
msg = QMessageBox()
```

3. Then, we set the error message:

```
msg.setText("This is an error!")
```

4. Subsequently, we set an icon number for the error icon:

```
msg.setIcon(QMessageBox.Critical)
```

5. Finally, we execute the error dialog:

```
msg.show()
```

How it works...

An important feature of modal windows is that they always stay on top of the application, regardless of whether the user changes the window's focus. This feature ensures that the user addresses the dialog before they proceed.

Displaying a progress bar

A progress bar is a dynamic dialog that displays the percentage complete bar for a running process that the user must wait for before continuing. A progress bar is more advanced than a simple dialog because it needs to be updated continuously. In this recipe, we'll create a simple progress dialog based on a timer.

Getting ready

No groundwork is required for this recipe.

How to do it...

The steps for this recipe include creating a custom class based on the `QProgressBar`, initializing the dialog and setting its size and title, creating a timer, connecting the progress bar to the timer, starting the time, and displaying the progress. To do this, we need to perform the following steps:

1. First, we must import both the GUI and QGIS core libraries:

```
from PyQt4.QtGui import *
from PyQt4.QtCore import *
```

2. Next, we create a custom class for our progress bar, including a method to increase the value of the progress bar:

```
class Bar(QProgressBar):
    value = 0
    def increaseValue(self):
        self.setValue(self.value)
        self.value = self.value+1
```

3. Now, we set the progress bar:

```
bar = Bar()
```

4. Next, we set the progress bar's size and title:

```
bar.resize(300,40)
bar.setWindowTitle('Working...')
```

5. Then, we initialize the timer, which will serve as the process we monitor:

```
timer = QTimer()
```

6. Now, connect the the timer's `timeout` signal to the `increaseValue` method, which we created earlier. Whenever the timer finishes its countdown, it will emit the `timeout` signal and notify the `increaseValue` method.

```
timer.timeout.connect(bar.increaseValue)
```

7. Now, we will start the timer, specifying an interval of 500 milliseconds. The timer will call its `timeout()` signal every 0.5 seconds:

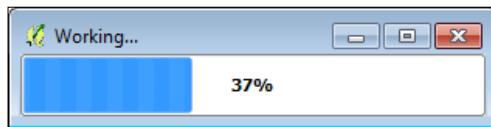
```
timer.start(500)
```

8. Finally, we show the progress bar and start the progress meter:

```
bar.show()
```

How it works...

The progress bar will stop when its value reaches 100, but our timer will continue to run until the `stop()` method is called. In a more realistic implementation, you will need a way to determine whether the monitored process is complete. The indicator might be the creation of a file, or even better, a signal. The Qt framework uses the concept of signals and slots to connect GUI elements. A GUI is event-based, with multiple events occurring at different times, including user actions and other triggers. The signal/slot system allows you to define reactions to events when they occur, without writing code to continuously monitor changes. In this recipe, we use the predefined signal from the timer and create our own slot. A slot is just a method identified as a slot by passing it to a signal's `connect()` method. The following screenshot shows an example of the progress bar:



There's more...

In a complex GUI application such as QGIS, you will end up with multiple signals that trigger multiple slots simultaneously. You must take care that a rapidly updating element such as a progress bar doesn't slow down the application. Using a thread to only update the progress bar when something has truly changed is more efficient. For an example of this technique, take a look at <http://snorf.net/blog/2013/12/07/multithreading-in-qgis-python-plugins/>.

Using the `QgsMessageBar` object is preferred to display informative messages, but it can also accept widgets such as the progress bar. The PyQGIS Developer Cookbook has an example that shows how to place the progress bar in the `QgsMessageBar` object (http://docs.qgis.org/testing/en/docs/pyqgis_developer_cookbook/communicating.html)

Creating a simple text input dialog

In this recipe, we'll demonstrate one of the simplest methods used for accepting input from a user, a text input dialog.

Getting ready

Open the **QGIS Python Console** by selecting the **Plugins** menu and then clicking on **Python Console**.

How to do it...

In this recipe, we will initialize the dialog and then configure its title and label. We'll set the editing mode and the default text. When you click on the **OK** button, the text will be printed to the **Python Console**. To do this, we need to perform the following steps:

1. First, we need to import the GUI library:

```
from PyQt4.QtGui import *
```

2. Next, we initialize the dialog:

```
qid = QInputDialog()
```

3. Now, we set the window's title, label text, editing mode, and default text:

```
title = "Enter Your Name"  
label = "Name: "  
mode = QLineEdit.Normal  
default = "<your name here>"
```

4. We configure the dialog while capturing the user input and the return code in variables:

```
text, ok = QInputDialog.getText(qid, title, label, mode, default)
```

5. When the dialog appears, type in some text and click on the **OK** button.

6. Now, we print the user input to the console:

```
print text
```

7. Finally, verify that the correct text is printed to the **Python Console**.

How it works...

The editing mode differentiates between **normal**, which we used here, and **password**, to obscure typed passwords. Although we haven't used it in this example, the return code is a Boolean, which can be used to verify that the user input occurred.

Creating a file input dialog

The best way to get a filename from the user is to have them browse to the file using a dialog. You can have the user type in a filename using the text input dialog, but this method is prone to errors. In this recipe, we'll create a file dialog and print the chosen filename to the console.

Getting ready

Open the QGIS **Python Console** by selecting the **Plugins** menu and then clicking on **Python Console**.

How to do it...

In this recipe, we will create and configure the dialog, browse to a file, and print the chosen filename, as follows:

1. First, we import the GUI library:

```
from PyQt4.QtGui import *
```

2. Next, we initialize the file dialog and specify its window title:

```
qfd = QFileDialog()  
title = 'Open File'
```

3. Now, we specify a path to the directory we want the file dialog to start in:

```
path = "/Users/joellawhead/qgis_data"
```

4. Then, we configure the file dialog with the preceding parameters and assign the output to a variable:

```
f = QFileDialog.getOpenFileName(qfd, title, path)
```

5. When the dialog appears, browse to a file, select it, and click on the **OK** button.

6. Finally, we print the chosen filename to the console:

```
print f
```

How it works...

The file dialog simply provides a filename. After the user selects the file, you must open it or perform some other operation on it. If the user cancels the file dialog, the file variable is just an empty string. You can use the `QFileInfo` object to get the path of the selected file:

```
from PyQt4.QtCore import *  
path = QFileInfo(f).path()
```

Then, you can save this path in the project settings, as demonstrated in *Chapter 1, Automating QGIS*. This way, next time when you open a file dialog, you will start in the same directory location as the previous file, which is usually more convenient.

There's more...

You can also use the `QFileDialog()` method to get the filenames to be saved. You can use the `FileMode` enumeration to restrict the user to selecting directories as well.

Creating a combobox

A combobox provides a drop-down list to limit the user's selection to a defined set of choices. In this recipe, we'll create a simple combobox.

Getting ready

Open the **QGIS Python Console** by selecting the **Plugins** menu and then clicking on **Python Console**.

How to do it...

In this recipe, we will initialize the combobox widget, add choices to it, resize it, display it, and then capture the user input in a variable for printing to the console. To do this, we need to perform the following steps:

1. First, we import the GUI library:

```
from PyQt4.QtGui import *
```

2. Now, we create our combobox object:

```
cb = QComboBox()
```

3. Next, we add the items that we want the user to choose from:

```
cb.addItems(["North", "South", "West", "East"])
```

4. Then, we resize the widget:

```
cb.resize(200,35)
```

5. Now we can display the widget to the user:

```
cb.show()
```

6. Next, we need to select an item from the list.

7. Now, we set the user's choice to a variable:

```
text = cb.currentText()
```

8. Finally, we can print the selection:

```
print text
```

9. Verify that the selection is printed to the console.

How it works...

Items added to the combobox are a Python list. This feature makes it easy to dynamically generate choices using Python as the result of a database query or other dynamic data. You may also want the index of the object in the list, which you can access with the `currentIndex` property.

Creating radio buttons

Radio buttons are good for user input when you want the user to select an exclusive choice from a list of options, as opposed to checkboxes, which let a user select many or all of the options available. For longer lists of choices, a combobox is a better option. Once a radio button is selected, you can unselect it only by choosing another radio button.

Getting ready

Open the **QGIS Python Console** by selecting the **Plugins** menu and then clicking on **Python Console**.

How to do it...

Radio buttons are easier to manage as part of a class, so we'll create a custom class that also includes a textbox to view which radio button is selected. To do this, perform the following steps:

1. First, we'll import both the GUI and the core QGIS libraries:

```
from PyQt4.QtCore import *
from PyQt4.QtGui import *
```

2. Next, we'll create the `RadioButton` class and set up the radio buttons and the textbox:

```
class RadioButton(QWidget):
    def __init__(self, parent=None):
        QWidget.__init__(self, parent)
```

3. We must also define a layout to manage the placement of the widgets, as follows:

```
self.layout = QVBoxLayout()
self.rb1 = QRadioButton('Option 1')
self.rb2 = QRadioButton('Option 2')
self.rb3 = QRadioButton('Option 3')
self.textbox = QLineEdit()
```

4. Now, we'll connect the toggled signal of each radio button to the methods you'll define in just a moment, in order to detect when a radio button is selected:

```
self.rb1.toggled.connect(self.rb1_active)
self.rb2.toggled.connect(self.rb2_active)
self.rb3.toggled.connect(self.rb3_active)
```

5. Then, we'll add the radio buttons and the textbox to the layout:

```
self.layout.addWidget(self.rb1)
self.layout.addWidget(self.rb2)
self.layout.addWidget(self.rb3)
self.layout.addWidget(self.textbox)
```

6. Now, we can define the layout for the custom widget we are building:

```
self.setLayout(self.layout)
```

7. Next, we can define the methods to indicate which radio button is selected. You can also define these options in a single method, but for a better understanding, three methods are easier:

```
def rb1_active(self, on):
    if on:
        self.textbox.setText('Option 1 selected')
def rb2_active(self, on):
    if on:
        self.textbox.setText('Option 2 selected')
def rb3_active(self, on):
    if on:
        self.textbox.setText('Option 3 selected')
```

8. We are now ready to initialize our class and display the radio buttons:

```
buttons = RadioButton()
buttons.show()
```

9. Finally, click on each of the three radio buttons and verify that the text in the textbox changes to indicate that the radio button you clicked on is selected.

How it works...

Radio buttons are almost always grouped together as a single object because they are related options. Many GUI frameworks expose them as a single object in the API; however, Qt keeps them as separate objects for maximum control.

Creating checkboxes

Checkboxes are closely related to radio buttons, in that they offer options around a single theme. However, unlike radio buttons, checkboxes can be selected or unselected. You can also select more than one checkbox at a time. In this recipe, we'll create a dialog with checkboxes and some textboxes to programmatically track which checkboxes are selected.

Getting ready

Open the QGIS **Python Console** by selecting the **Plugins** menu and then clicking on **Python Console**.

How to do it...

In this recipe, we'll use a class to manage the checkboxes and the textbox widgets, as follows:

1. First, we import the GUI and QGIS core libraries:

```
from PyQt4.QtCore import *
from PyQt4.QtGui import *
```

2. Next, we create our custom class for the checkboxes and textboxes:

```
class CheckBox(QWidget):
    def __init__(self, parent=None):
        QWidget.__init__(self, parent)
```

3. Next, we'll need a layout object to manage the placement of the widgets:

```
self.layout = QVBoxLayout()
```

4. Now, we'll add three checkboxes and three textboxes:

```
self.cb1 = QCheckBox('Option 1')
self.cb2 = QCheckBox('Option 2')
self.cb3 = QCheckBox('Option 3')
self.textbox1 = QLineEdit()
self.textbox2 = QLineEdit()
self.textbox3 = QLineEdit()
```

5. Then, we'll connect the status signals of the checkboxes to the methods that we'll define later:

```
self.cb1.toggled.connect(self.cb1_active)
self.cb2.toggled.connect(self.cb2_active)
self.cb3.toggled.connect(self.cb3_active)
```

6. Next, we must add the widgets to the layout:

```
    self.layout.addWidget(self.cb1)
    self.layout.addWidget(self.cb2)
    self.layout.addWidget(self.cb3)
    self.layout.addWidget(self.textbox1)
    self.layout.addWidget(self.textbox2)
    self.layout.addWidget(self.textbox3)
```

7. Now, we set our custom class's layout to the layout we created:

```
    self.setLayout(self.layout)
```

8. We then create the methods that change the textboxes each time a checkbox is toggled:

```
# First checkbox
def cb1_active(self, on):
    if on:
        self.textbox1.setText('Option 1 selected')
    else: self.textbox1.setText('')

# Second checkbox
def cb2_active(self, on):
    if on:
        self.textbox2.setText('Option 2 selected')
    else: self.textbox2.setText('')

# Third checkbox
def cb3_active(self, on):
    if on:
        self.textbox3.setText('Option 3 selected')
    else: self.textbox3.setText('')
```

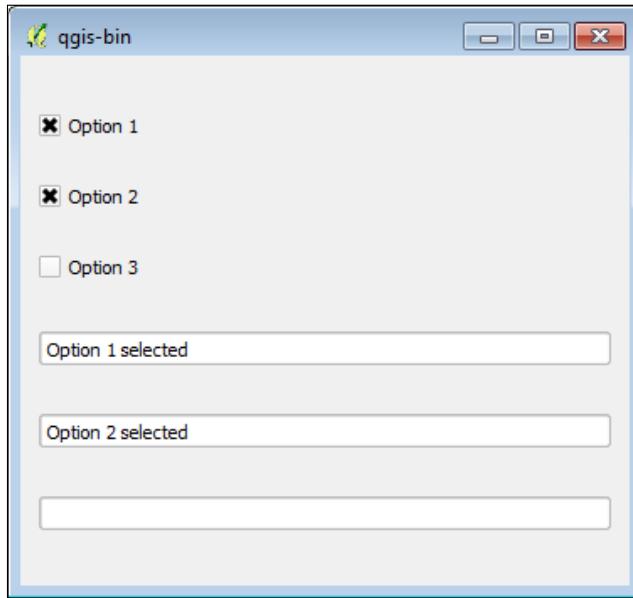
9. Now, we are ready to initialize our custom class and display the dialog:

```
buttons = CheckBox()
buttons.show()
```

10. Toggle the checkboxes separately and simultaneously and then verify that the textboxes reflect the changes.

How it works...

Textboxes allow you to verify that you are programmatically catching the signal from the checkboxes as they are toggled. You can also use a single checkbox as a Boolean for an option with only two choices. When you run this recipe, the result should look similar to the following screenshot:



Creating tabs

Tabs allow you to condense the information from several screens into a relatively small place. Tabs provide titles at the top of the window, which present an individual widget layout for each title when clicked. In this recipe, we'll create a simple tabbed interface.

Getting ready

Open the **QGIS Python Console** by selecting the **Plugins** menu and then clicking on **Python Console**.

How to do it...

We will create an overarching tab widget. Then, we'll create three generic widgets to represent our tabs. We'll set up layouts with three different GUI widgets and assign each layout to our tab widgets. Finally, we'll add our tabs to the tab widget and display it. To do this, we need to perform the following steps:

1. First, we need to import the GUI and QGIS core libraries:

```
from PyQt4.QtCore import *
from PyQt4.QtGui import *
```

2. Next, we create our tab and configure its title and size:

```
qtw = QTabWidget()
qtw.setWindowTitle("PyQGIS Tab Example")
qtw.resize(400,300)
```

3. Now, we initialize our tab widgets:

```
tab1 = QWidget()
tab2 = QWidget()
tab3 = QWidget()
```

4. Then, we'll set up a widget and a layout with a rich text input box, using HTML tags for bold text for our first tab:

```
layout1 = QVBoxLayout()
layout1.addWidget(QTextEdit("<b>Type text here</b>"))
tab1.setLayout(layout1)
```

5. Now, we'll set up a simple button for our second tab, following the same format as the first tab:

```
layout2 = QVBoxLayout()
layout2.addWidget(QPushButton("Button"))
tab2.setLayout(layout2)
```

6. Next, we'll create the widget and the layout for our third tab with a simple text label:

```
layout3 = QVBoxLayout()
layout3.addWidget(QLabel("Label text example"))
tab3.setLayout(layout3)
```

7. Then, we'll add the tabs to the tab window:

```
qtw.addTab(tab1, "First Tab")
qtw.addTab(tab2, "Second Tab")
qtw.addTab(tab3, "Third Tab")
```

8. Finally, we'll display the tab window:

```
qtw.show()
```

9. Verify that you can click on each tab and interact with the widgets.

How it works...

The key to this recipe is the `QTabWidget().method`. Everything else is just arbitrary layouts and widgets, which are ultimately contained in the tab widget.



The general rule of thumb for tabs is to keep the information in them independently.



There is no way to predict how the user will interact with a tabbed interface, and if the information across tabs is dependent, problems will arise.

Stepping the user through a wizard

A wizard is a series of dialogs that lead the user through a sequence of steps. The information on each page of a wizard might relate in some way to the information on other pages. In this recipe, we'll create a simple three-page wizard to collect some information from the user and display it back to them.

Getting ready

Open the **QGIS Python Console** by selecting the **Plugins** menu and then clicking on **Python Console**.

How to do it...

We will create three classes, each representing a page of our wizard. The first two pages will collect information and the third page will display it back to the user. We will create a `QWizard` object to tie the page classes together. We will also use the concept of wizard fields to pass information among the pages.

To do this, we need to perform the following steps:

1. First, we import the GUI and QGIS core libraries:

```
from PyQt4.QtCore import *
from PyQt4.QtGui import *
```

2. Next, we create the class for the first page of our wizard and add a textbox to collect the user's name as the `uname` variable:

```
class Page1(QWizardPage):
    def __init__(self, parent=None):
        super(Page1, self).__init__(parent)
        self.setTitle("What's Your Name?")
        self.setSubTitle("Please enter your name.")
        self.label = QLabel("Name:")
        self.uname = QLineEdit("<enter your name>")
```

3. Now, we register the `uname` field so that we'll be able to access the entered value later on, without having to keep track of the variable itself:

```
self.registerField("uname", self.uname)
```

4. Then, we set up the layout for the page:

```
layout = QVBoxLayout()
layout.addWidget(self.label)
layout.addWidget(self.uname)
self.setLayout(layout)
```

5. Next, we'll set the class for our second page:

```
class Page2(QWizardPage):
    def __init__(self, parent=None):
        super(Page2, self).__init__(parent)
        self.setTitle("When's Your Birthday?")
        self.setSubTitle("Select Your Birthday.")
```

6. Then, we'll add a calendar widget to get the user's birthday:

```
self.cal = QCalendarWidget()
```

7. We'll register the selected date as a field, to be accessed later on:

```
self.registerField("cal", self.cal, "selectedDate")
```

8. Then, we'll set up the layout for this page:

```
layout = QVBoxLayout()
layout.addWidget(self.cal)
self.setLayout(layout)
```

9. We are now ready to set up the third page, which will display the user's information. We'll use simple labels, which are dynamically populated in the next step:

```
class Page3(QWizardPage):
    def __init__(self, parent=None):
        super(Page3, self).__init__(parent)
        self.setTitle("About You")
        self.setSubTitle("Here is Your Information:")
        self.name_lbl = QLabel()
        self.date_lbl = QLabel()
        layout = QVBoxLayout()
        layout.addWidget(self.name_lbl)
        layout.addWidget(self.date_lbl)
        self.setLayout(layout)
```

10. Now, we set up the initialization of the page. We will first access the fields registered from the previous pages to grab the user input:

```
def initializePage(self):
    uname = self.field("uname")
    date = self.field("cal").toString()
```

11. Then, all we have to do is set those values to the text for the labels using Python string formatting:

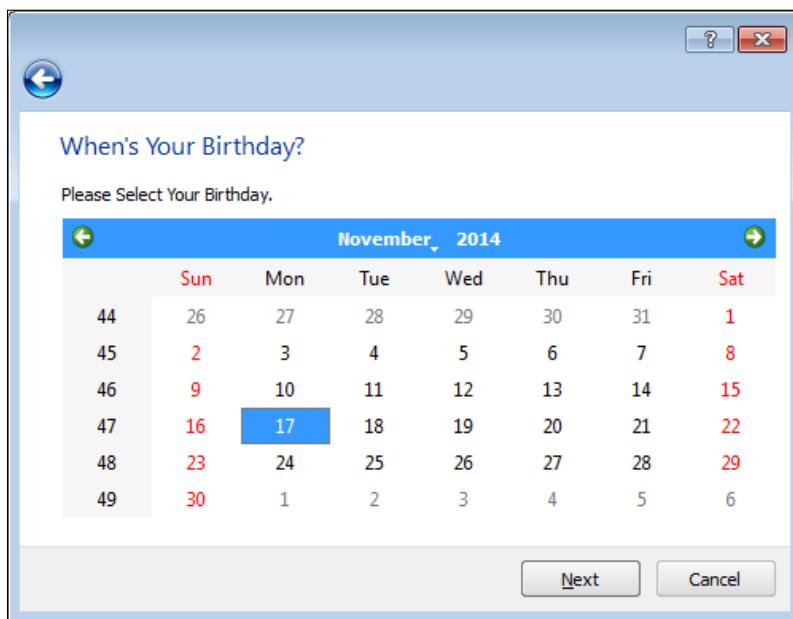
```
    self.name_lbl.setText("Your name is %s" % uname)
    self.date_lbl.setText("Your birthday is %s" % date)
```

12. Finally, we create our wizard widget, add pages, and display the wizard:

```
wiz = QWizard()
wiz.addPage(Page1())
wiz.addPage(Page2())
wiz.addPage(Page3())
wiz.show()
```

How it works...

The wizard interface shares many traits with the tab widget, with some important differences. The wizard only allows the user to move back and forth in a linear progression based on the page order. It can share information among pages if the information is registered as fields, which then makes the pages global to the scope of the wizard. However, the `field()` method is a protected method, so your pages must be defined as classes inherited from the `QWizardPage` object for the registered fields to work as expected. The following screenshot shows the calendar screen of the wizard:



Keeping dialogs on top

It's easy to lose track of windows that pop up in front of QGIS. As soon as the user changes focus to move the main QGIS application window, your dialog can disappear behind it, forcing the user to rearrange their whole desktop to find the smaller window again. Fortunately, Qt has a window setting called `hint`, which allows you to force a window to stay on top. This type of dialog is called a modal dialog. In this recipe, we'll create a message dialog using `hint`.

Getting ready

Open the **QGIS Python Console** by selecting the **Plugins** menu and then clicking on **Python Console**.

How to do it...

In this recipe, we will create a simple message dialog and set it to stay on top, as follows:

1. First, we import the Qt GUI and QGIS core libraries:

```
from PyQt4.QtGui import *
from PyQt4.QtCore import *
```

2. Next, we create the text for our message:

```
msg = " This window will always stay on top."
```

3. Now, we create our dialog and specify the message and hint:

```
lbl = QLabel(msg, None, Qt.WindowStaysOnTopHint)
```

4. We can resize and show the dialog:

```
lbl.resize(400,400)
lbl.show()
```

5. Click on the main QGIS application window to change the window focus and verify that the dialog stays on top of QGIS.

How it works...

This simple technique can help to ensure that a user addresses an important dialog before moving on.

8

QGIS Workflows

In this chapter, we will cover the following recipes:

- ▶ Creating an NDVI
- ▶ Geocoding addresses
- ▶ Creating raster footprints
- ▶ Performing network analysis
- ▶ Routing along streets
- ▶ Tracking a GPS
- ▶ Creating a mapbook
- ▶ Finding the least cost path
- ▶ Performing nearest neighbor analysis
- ▶ Creating a heat map
- ▶ Creating a dot density map
- ▶ Collecting field data
- ▶ Computing road slope using elevation data
- ▶ Geolocating photos on the map
- ▶ Image change detection

Introduction

In this chapter, we'll use Python to perform a variety of common geospatial tasks in QGIS, which may be complete workflows in themselves or key pieces of larger workflows.

Creating an NDVI

A **Normalized Difference Vegetation Index (NDVI)** is one of the oldest remote sensing algorithms used to detect green vegetation in an area of interest, using the red and near-infrared bands of an image. The chlorophyll in plants absorbs visible light, including the red band, while the cell structures of plants reflect near-infrared light. The NDVI formula provides a ratio of near-infrared light to the total incoming radiation, which serves as an indicator of vegetation density. This recipe will use Python to control the QGIS raster calculator in order to create an NDVI using a multispectral image of a farm field.

Getting ready

Download the image from <https://geospatialpython.googlecode.com/svn/farm-field.tif> and place it in your `qgis_data` to a directory named `rasters`.

How to do it...

We will load the raster as a QGIS raster layer, perform the NDVI algorithm, and finally apply a color ramp to the raster so that we can easily visualize the green vegetation in the image. To do this, we need to perform the following steps:

1. In the QGIS **Python Console**, import the following libraries:

```
from PyQt4.QtGui import *
from PyQt4.QtCore import *
from qgis.analysis import *
```

2. Now, load the raster image as a layer using the following code:

```
rasterName = "farm"
raster = QgsRasterLayer("/Users/joellawhead/qgis_data/\
rasters/farm-field.tif", rasterName)
```

3. Then, create entries in the QGIS raster calculator for the two bands using the following code:

```
ir = QgsRasterCalculatorEntry()
r = QgsRasterCalculatorEntry()
```

4. Now, using the following lines of code, assign the raster layer as the raster component of each calculator entry:

```
ir.raster = raster  
r.raster = raster
```

5. Select the appropriate band for each entry, so the calculator will use the data we need for the NDVI. The red and infrared band numbers are typically listed in the raster's metadata:

```
ir.bandNumber = 2  
r.bandNumber = 1
```

6. Next, assign a reference ID to each entry using the special QGIS naming convention, as shown here, with the name of the layer as a prefix followed by an @ symbol and the band number as a suffix:

```
ir.ref = rasterName + "@2"  
r.ref = rasterName + "@1"
```

7. Build the raster calculator expression with the following code:

```
references = (ir.ref, r.ref, ir.ref, r.ref)  
exp = "1.0 * (%s - %s) / 1.0 + (%s + %s)" % references
```

8. Then, specify the output name of the NDVI image:

```
output = "/Users/joellawhead/qgis_data/rasters/ndvi.tif"
```

9. Set up the variables for the rest of the raster calculator call by defining the raster's extent, its width and height in columns and rows, and the raster entries we defined in the previous steps:

```
e = raster.extent()  
w = raster.width()  
h = raster.height()  
entries = [ir,r]
```

10. Now, create the NDVI using our expression:

```
ndvi = QgsRasterCalculator(exp, output, "GTiff", e, w, h,  
entries)  
ndvi.processCalculation()
```

11. Next, load the NDVI output as a raster layer:

```
lyr = QgsRasterLayer(output, "NDVI")
```

-
12. We must perform a histogram stretch on the image, otherwise the differences in values will be difficult to see. A stretch is performed using a QGIS contrast enhancement algorithm:

```
algorithm = QgsContrastEnhancement.StretchToMinimumMaximum  
limits = QgsRaster.ContrastEnhancementMinMax  
lyr.setContrastEnhancement(algorithm, limits)
```

13. Next, build a color ramp shader to colorize the NDVI, as follows:

```
s = QgsRasterShader()  
c = QgsColorRampShader()  
c.setColorRampType(QgsColorRampShader.INTERPOLATED)
```

14. Then, add entries for each color in the image. Each entry consists of a lower value range, a color, and a label. The color in an entry will continue from the lower value until it encounters a higher value or the maximum value. Note that we will use a variable alias for the extremely long name of the QGIS ColorRampItem object:

```
i = []  
qri = QgsColorRampShader.ColorRampItem  
i.append(qri(0, QColor(0,0,0,0), 'NODATA'))  
i.append(qri(214, QColor(120,69,25,255), 'Lowest Biomass'))  
i.append(qri(236, QColor(255,178,74,255), 'Lower Biomass'))  
i.append(qri(258, QColor(255,237,166,255), 'Low Biomass'))  
i.append(qri(280, QColor(173,232,94,255), 'Moderate Biomass'))  
i.append(qri(303, QColor(135,181,64,255), 'High Biomass'))  
i.append(qri(325, QColor(3,156,0,255), 'Higher Biomass'))  
i.append(qri(400, QColor(1,100,0,255), 'Highest Biomass'))
```

15. Now, we can add the entries to the shader and apply it to the image:

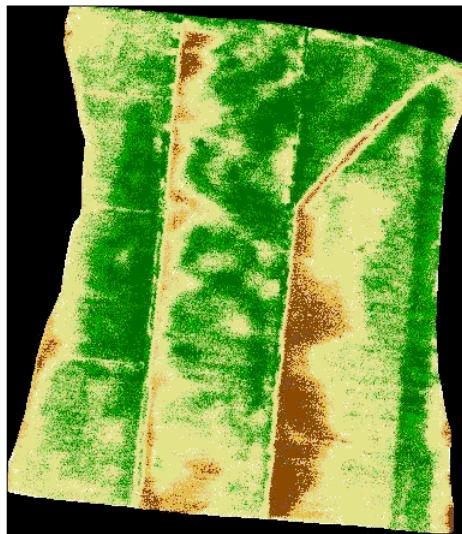
```
c.setColorRampItemList(i)  
s.setRasterShaderFunction(c)  
ps = QgsSingleBandPseudoColorRenderer(lyr.dataProvider(), 1,  
s)  
lyr.setRenderer(ps)
```

16. Finally, add the classified NDVI image to the map in order to visualize it:

```
QgsMapLayerRegistry.instance().addMapLayer(lyr)
```

How it works...

The QGIS raster calculator is exactly what its name implies. It allows you to perform array math on images. Both the QGIS raster menu and the Processing Toolbox have several raster processing tools, but the raster calculator can perform custom analysis that can be defined in a single mathematical equation. The NDVI algorithm is the infrared band minus the red band divided by the infrared band plus the red band, or $(IR-R)/(IR+R)$. In our calculator expression, we multiply each side of the equation by 1 . 0 to avoid division-by-zero errors. Your output should look similar to the following image if you load the result into QGIS. In this screenshot, NODATA values are represented as black; however, your QGIS installation may default to using white:



Geocoding addresses

Geocoding is the process of turning an address into earth coordinates. Geocoding requires a comprehensive dataset that ties zip codes, cities, streets, and street numbers (or street number ranges) to the coordinates. In order to have a geocoder that works for any address in the world with reasonable accuracy, you need to use a cloud service because geocoding datasets are very dense and can be quite large. Creating a geocoding dataset for any area beyond a few square miles requires a significant amount of resources. There are several services available, including Google and MapQuest. In QGIS, the easiest way to access these services is through the QGIS Python GeoCoding plugin. In this recipe, we'll use this plugin to programmatically geocode an address.

Getting ready

You will need to install the QGIS Python GeoCoding plugin by Alessandro Pasotti for this exercise, as follows:

1. From the QGIS **Plugins** menu, select **Manage and Install Plugins....**
2. In the **Plugins** dialog search box, search for Geocoding.
3. Select **GeoCoding** plugin and click on the **Install plugin** button.

How to do it...

In this recipe, we will access the GeoCoding plugin methods using Python, feed the plugin an address, and print the resulting coordinates. To do this, we need to perform the following steps:

1. In the QGIS **Python Console**, import the OpenStreetMap geoCoding object using the following code:

```
from GeoCoding.geopy.geocoders import Nominatim
```

2. Next, we'll create our geocoder:

```
geocoder = Nominatim()
```

3. Then, using the following code, we'll geocode an address:

```
location = geocoder.geocode("The Ugly Pirate, Bay Saint Louis,  
MS 39520")
```

4. Finally, we'll print the results to see the coordinates:

```
print location
```

5. Check whether you have received the following output printed to the console:

```
(u'The Ugly Pirate, 144, Demontluzin Street, Bay St. Louis,  
Hancock County, Mississippi, 39520, United States of America',  
(30.3124059, -89.3281418))
```

How it works...

The **GeoCoding** plugin is designed to be used with the QGIS GUI interface. However, like most QGIS plugins, it is written in Python and we can access it through the Python console.



This trick doesn't work with every plugin. Sometimes, the user interface is too intertwined with the plugin's GUI that you can't programmatically use the plugin's methods without triggering the GUI.

However, in most cases, you can use the plugins to not only extend QGIS but also for its powerful Python API. If you write a plugin yourself, consider making it accessible to the QGIS Python console in order to make it even more useful.

There's more...

The GeoCoding plugin also provides the Google geocoding engine as a service. Note that the Google mapping API, including geocoding, comes with some limitations that can be found at <https://developers.google.com/maps-engine/documentation/limits>.

Creating raster footprints

A common way to catalog raster datasets that consist of a large number of files is by creating a vector dataset with polygon footprints of the extent of each raster file. The vector footprint files can be easily loaded in QGIS or served over the Web. This recipe demonstrates a method to create a footprint vector from a directory full of raster files. We will build this program as a Processing Toolbox script, which is easier to build than a QGIS plugin and provides both a GUI and a clean programming API.

Getting ready

Download the sample raster image scenes from <https://geospatialpython.googlecode.com/svn/scenes.zip>. Unzip the `scenes` directory into a directory named `rasters` in your `qgis_data` directory.

For this recipe, we will create a new Processing Toolbox script using the following steps:

1. In the QGIS Processing Toolbox, expand the **Scripts** tree menu.
2. Next, expand the **Tools** tree menu.
3. Finally, double-click on the **Create new script** item to bring up the processing script editor.

How to do it...

First, we will use the Processing Toolbox header naming conventions ,which will simultaneously define our GUI and the input and output variables. Then, we'll create the logic, which processes a raster directory and calculates the image extents, and finally we'll create the vector file. To do this, we need to perform the following steps:

1. First, we define our input variables using comments to tell the Processing Toolbox to add these to the GUI when the script is invoked by a user. The first item defines the script's group menu to place our script in the toolbox, the second item defines the directory containing the rasters, and the third item is the output name of our shapefile. The script must start with these comments. Each item also declares a type allowed by the Processing Toolbox API. The names of the variables in these comments become available to the script:

```
##Vector=group  
##Input_Raster_Directory=folder  
##Output_Footprints_Vector=output vector
```

2. Next, we import the Python libraries we will need, using the following commands:

```
import os  
from qgis.core import *
```

3. Now, we get a list of files in the raster directory. The following script makes no attempt to filter the files by type. If there are other types of data in the directory that are not raster files, they will be included as well:

```
files = os.listdir(Input_Raster_Directory)
```

4. Then, we declare a couple of variables, which will hold our raster extents and the coordinate reference string, as shown here:

```
footprints = []  
crs = ""
```

5. Now, we loop through the rasters, load them as a raster layer to grab their extents, store them as point data in Python dictionaries, and add them to our list of footprints for temporary storage. If the raster can't be processed, a warning is issued using the Processing Toolbox progress object:

```
for f in files:  
    try:  
        fn = os.path.join(Input_Raster_Directory, f)  
        lyr = QgsRasterLayer(fn, "Input Raster")
```

```
    crs = lyr.crs()
    e = lyr.extent()
    ulx = e.xMinimum()
    uly = e.yMaximum()
    lrx = e.xMaximum()
    lry = e.yMinimum()
    ul = (ulx, uly)
    ur = (lrx, uly)
    lr = (lrx, lry)
    ll = (ulx, lry)
    fp = {}
    points = []
    points.append(QgsPoint(*ul))
    points.append(QgsPoint(*ur))
    points.append(QgsPoint(*lr))
    points.append(QgsPoint(*ll))
    points.append(QgsPoint(*ul))
    fp["points"] = points
    fp["raster"] = fn
    footprints.append(fp)
except:
    progress.setInfo("Warning: The file %s does not appear to
be a \
valid raster file." % f)
```

6. Using the following code, we will create a memory vector layer to build the footprint vector before writing it to a shapefile:

```
vectorLyr =
QgsVectorLayer("Polygon?crs=%s&field=raster:string(100)" \
% crs, "Footprints" , "memory")
vpr = vectorLyr.dataProvider()
```

7. Now, we'll turn our list of extents into features:

```
features = []
for fp in footprints:
    poly = QgsGeometry.fromPolygon([fp["points"]])
```

```
f = QgsFeature()
f.setGeometry(poly)
f.setAttributes([fp["raster"]])
features.append(f)
vpr.addFeatures(features)
vectorLyr.updateExtents()
```

8. We'll then set up the file driver and the CRS for the shapefile:

```
driver = "Esri Shapefile"
epsg = crs.postgisSrid()
srs = "EPSG:%s" % epsg
```

9. Finally, we'll write the selected output file, specifying the layer we are saving to disk; the name of the output file; the file encoding, which might change depending on the input; the coordinate reference system; and the driver for the output file type, which in this case is a shapefile:

```
error = QgsVectorFileWriter.writeAsVectorFormat\
(vectorLyr, Output_Footprints_Vector, \"utf-8\", srs, driver)
if error == QgsVectorFileWriter.NoError:
    pass
else:
    progress.setInfo("Unable to output footprints.")
```

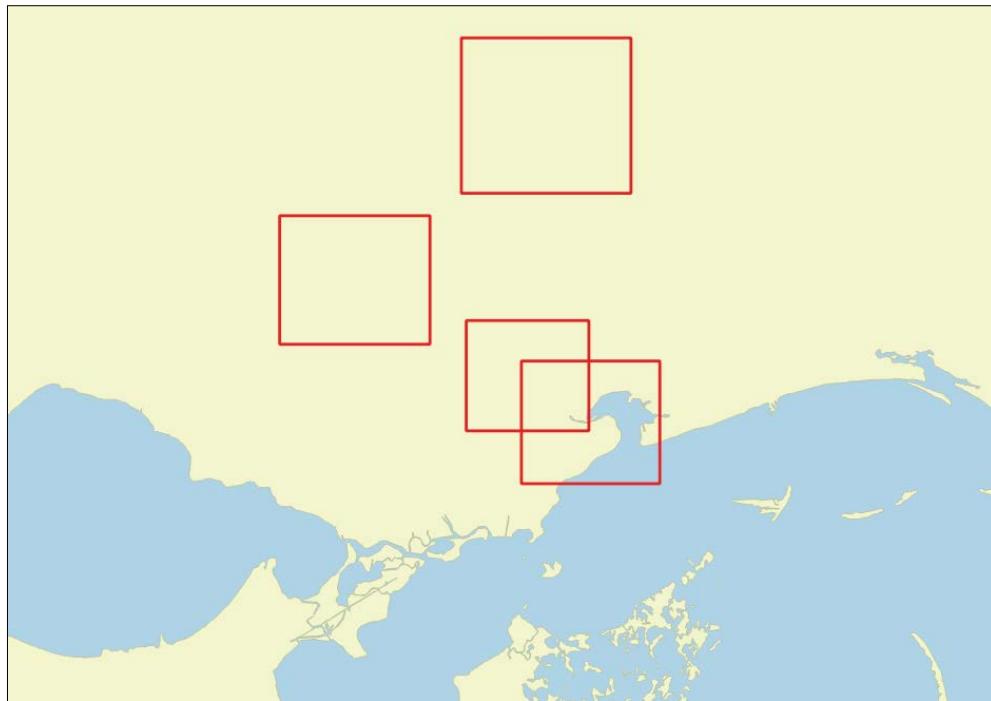
How it works...

It is important to remember that a Processing Toolbox script can be run in several different contexts: as a GUI process such as a plugin, as a programmatic script from the Python console, a Python plugin, or the Graphical Modeler framework. Therefore, it is important to follow the documented Processing Toolbox API so that it can work as expected in all of these contexts. This includes defining clear inputs and outputs and using the progress object. The progress object is the proper way to provide feedback to the user for both progress bars and messages. Although the API allows you to define outputs that let the user select different OGR and GDAL outputs, only shapefiles and GeoTiffs seem to be supported currently.

There's more...

The Graphical Modeler tool within the Processing Toolbox lets you visually chain different processing algorithms together to create complex workflows. Another interesting plugin is the Processing Workflows plugin, which not only allows you to chain algorithms together but also provides a nice tabbed interface with instructions for the end user to help beginners through complicated geospatial workflows.

The following screenshot shows the raster footprints over an OpenStreetMap basemap:



Performing network analysis

Network analysis allows you to find the most efficient route between two points along a defined network of connected lines. These lines might represent streets, pipes in a water system, the Internet, or any number of connected systems. Network analysis abstracts this common problem so that the same techniques and algorithms can be applied across a wide variety of applications. In this recipe, we'll use a generic line network to perform analysis using the Dijkstra algorithm, which is one of the oldest algorithms used to find the shortest path. QGIS has all of this functionality built in.

Getting ready

First, download the vector dataset from the following link, which includes two shapefiles, and unzip it to a directory named `shapes` in your `qgis_data` directory:

<https://geospatialpython.googlecode.com/svn/network.zip>

How to do it...

We will create a network graph by defining the beginning and end of our network of lines, and then use this graph to determine the shortest route along the line network between our two points. To do this, we need to perform the following steps:

1. In the QGIS **Python Console**, we'll first import the libraries we'll need, including the QGIS Network Analyzer:

```
from qgis.core import *
from qgis.gui import *
from qgis.networkanalysis import *
from PyQt4.QtCore import *
```

2. Next, we'll load our line network shapefile and the shapefile containing the points along the network we want the Network Analyzer to consider when selecting a route:

```
network =
QgsVectorLayer("/Users/joellawhead/qgis_data/shapes/\Network.shp",
                "Network Layer", "ogr")

waypoints =
QgsVectorLayer("/Users/joellawhead/qgis_data/shapes/\NetworkPoints.shp",
                "Waypoints", "ogr")
```

3. Now, we will create a graph director to define the properties of the graph. The `director` object accepts our line shapfile, a field ID for direction information, and some other documented integer codes involving direction properties in the network. In our example, we're going to tell the director to ignore directions. The `properter` object is a basic algorithm for a routing strategy that gets added to the network graph and considers line length:

```
director = QgsLineVectorLayerDirector(network, -1, '', '',
                                         3)

properter = QgsDistanceArcProperter()
director.addProperter(properter)

crs = network.crs()
```

4. Now, we create the `GraphBuilder` object to actually convert the line network into a graph:

```
builder = QgsGraphBuilder(crs)
```

5. We define the two points that are the start and end of our route:

```
ptStart = QgsPoint(-0.8095638694, -0.1578175511)
ptStop = QgsPoint(0.8907435677, 0.4430834924)
```

6. Then, we tell the director to turn our point layer into tie points in our network, which define the waypoints along our network and can also optionally provide resistance values:

```
tiePoints = director.makeGraph(builder, [ptStart, ptStop])
```

7. Now, we can use the following code to build the graph:

```
graph = builder.graph()
```

8. We now locate our start and end points as tie points in the graph:

```
tStart = tiePoints[0]
tStop = tiePoints[1]
idStart = graph.findVertex(tStart)
idStop = graph.findVertex(tStop)
```

9. Then, we can tell the Analyzer to use our start point in order to find the shortest route through the network:

```
(tree, cost) = QgsGraphAnalyzer.dijkstra(graph, idStart, 0)
```

10. Next, we loop through the resulting tree and grab the points along the output route:

```
p = []
curPos = idStop
while curPos != idStart:
    p.append(graph.vertex(graph.arc(tree[curPos]).inVertex()).point())
    curPos = graph.arc(tree[curPos]).outVertex()
p.append(tStart)
```

11. Now, we'll load our two input shapefiles onto the map and create a rubber band in order to visualize the route:

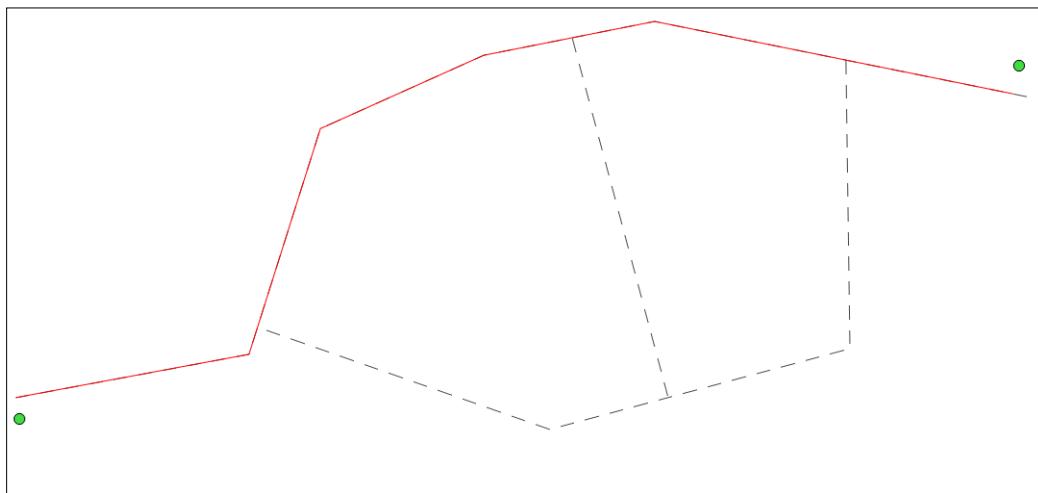
```
QgsMapLayerRegistry.instance().addMapLayers([network, waypoints])
rb = QgsRubberBand(iface.mapCanvas())
rb.setColor(Qt.red)
```

12. Finally, we'll add the route points to the rubber band in order to see the output of the Network Analyzer:

```
for pnt in p:
    rb.addPoint(pnt)
```

How it works...

This recipe is an extremely simple example to be used as a starting point for the investigation of a very complex and powerful tool. The line network shapefiles can have a field defining each line as one-way in a certain direction or bi-directional. The point shapefile provides waypoints along the network, as well as resistance values, which might represent elevation, traffic density, or other factors that will make a route less desirable. The output will look similar to the following image:



More information and examples of the network analysis tool are available in the QGIS documentation at http://docs.qgis.org/testing/en/docs/pyqgis_developer_cookbook/network_analysis.html.

Routing along streets

Sometimes, you may want to find the best driving route between two addresses. Street routing has now become so commonplace that we take it for granted. However, if you explore the recipes on geocoding and network analysis in this book, you will begin to see what a complex challenge street routing truly is. To perform routing operations in QGIS, we'll use the QGIS GeoSearch plugin, which is written in Python, so that we can access it from the console.

Getting ready

You will need to install the QGIS Python GeoSearch plugin for this exercise in order to do the routing, as well as the QGIS OpenLayers Plugin to overlay the result on a Google map, as follows:

1. From the QGIS **Plugins** menu, select **Manage and Install Plugins....**
2. If you have the QGIS GeoCoding Plugin installed, then you must uninstall it, as sometimes it conflicts with the GeoSearch plugin. So, select this in the plugin list and click on the **Uninstall plugin** button.
3. In the **Plugins** dialog search box, search for **GeoSearch**.
4. Select the **GeoSearch plugin** and click on the **Install plugin** button.
5. Next, in the **Plugins** search dialog, search for **OpenLayers**.
6. Select the **OpenLayers plugin** and click on the **Install plugin** button.

How to do it...

We will invoke the GeoSearch plugin's routing function, which uses Google's routing engine, and display the result over a Google map from the OpenLayers plugin. To do this, we need to perform the following steps:

1. In the **QGIS Python Console**, we first import the **QGIS utils library** as well as the required portions of the **GeoSearch** plugin:

```
import qgis.utils  
from GeoSearch import geosearchdialog, GoogleMapsApi
```

2. Next, we'll use the QGIS utils library to access the **OpenLayers plugin**:

```
openLyrs = qgis.utils.plugins['openlayers_plugin']
```

3. The GeoSearch plugin isn't really designed for programmatic use, so in order to invoke this plugin, we must invoke it through the GUI interface, but then we need to pass blank values so that it doesn't trigger the GUI plugin interface:

```
g = geosearchdialog.GeoSearchDialog(iface)  
g.SearchRoute([])
```

4. Now, using the following code, we can safely create our routing engine object:

```
d = GoogleMapsApi.directions.Directions()
```

5. Next, we create our origin and destination addresses:

```
origin = "Boston, MA"  
dest = "2517 Main Rd, Dedham, ME 04429"
```

6. Then, we can calculate the route using the simplest possible options, as shown here:

```
route = d.GetDirections(origin, dest, mode = "driving", \
    waypoints=None, avoid=None, units="imperial")
```

7. Now, we use the **OpenLayers** plugin to add the Google Maps base map to the QGIS map:

```
layerType = openLyrs._olLayerTypeRegistry.getById(4)
openLyrs.addLayer(layerType)
```

8. Finally, we use the **GeoSearch plugin** to create a QGIS layer on top of the base map for our route:

```
g.CreateVectorLayerGeoSearch_Route(route)
```

How it works...

Even though they are built in Python, neither the GeoSearch nor OpenLayers plugins are designed to be used with Python by a programmer. However, we are still able to use the tools in a script without much trouble. To take advantage of some of the routing options available with the GeoSearch plugin, you can use its GUI to see what is available and then add those options to your script. Beware that most plugins don't have a true API, so a slight change to the plugin in a future version can break your script.

Tracking a GPS

QGIS has the ability to connect to a GPS that uses the NMEA standard. QGIS can use a serial connection to the GPS or communicate with it through the open source software called gpsd using the QGIS GPS information panel. The location information from the GPS can be displayed on the QGIS map, and QGIS can even automatically pan the map to follow the GPS point. In this recipe, we'll use the QGIS API to process NMEA sentences and update a point on a global map. The information needed to connect to different GPS units can vary widely, so we'll use an online NMEA sentence generator to get some simulated GPS information.

Getting ready

This recipe doesn't require any preparation.

How to do it...

We'll grab a batch of NMEA GPS sentences from a free online generator, create a worldwide basemap using online geojson data, create a vector point layer to represent the GPS, and finally loop through the sentences and make our track point move around the map.

To do this, we need to perform the following steps:

1. First, we need to import some standard Python libraries using the QGIS

Python Console:

```
import urllib
import urllib2
import time
```

2. Next, we'll connect to the online NMEA generator, download a batch of sentences, and turn them into a list, as follows:

```
url = 'http://freenmea.net/api/emitnmea'
values = {'types' : 'default'}
data = urllib.urlencode(values)
req = urllib2.Request(url, data)
response = urllib2.urlopen(req)
results = response.read().split("\n")
```

3. Next, we can add our world countries basemap using a geojson service:

```
wb =
"https://raw.githubusercontent.com/johan/world.geo.json/master/
countries.geo.json"
basemap = QgsVectorLayer(wb, "Countries", "ogr")
qmr = QgsMapLayerRegistry.instance()
qmr.addMapLayer(basemap)
```

4. Now, we can create our GPS point layer and access its data provider:

```
vectorLyr = QgsVectorLayer('Point?crs=epsg:4326', 'GPS Point',
, "memory")
vpr = vectorLyr.dataProvider()
```

5. Then, we need some variables to hold the current coordinates as we loop through the locations, and we'll also access the mapCanvas object:

```
cLat = None
cLon = None
canvas = iface.mapCanvas()
```

6. Next, we'll create a GPS connection object for data processing. If we are using a live GPS object, we will use this line to enter the device's information:

```
c = QgsNMEAConnection(None)
```

-
7. Now, we set up a flag to determine whether we are processing the first point or not:

```
firstPt = True
```

8. We can loop through the NMEA sentences now, but we must check the sentence type to see which type of information we are using. In a live GPS connection, QGIS handles this part automatically and this part of the code will be unnecessary:

```
for r in results:  
    l = len(r)  
    if "GGA" in r:  
        c.processGGASentence(r,l)  
    elif "RMC" in r:  
        c.processRMCSentence(r,l)  
    elif "GSV" in r:  
        c.processGSVSentence(r,l)  
    elif "VTG" in r:  
        c.processVTGSentence(r,l)  
    elif "GSA" in r:  
        c.processGSASentence(r,l)
```

9. Then, we can get the current GPS information:

```
i=c.currentGPSInformation()
```

10. Now, we will check this information to make sure that the GPS location has actually changed since the previous loop before we try to update the map:

```
if i.latitude and i.longitude:  
    lat = i.latitude  
    lon = i.longitude  
    if lat==cLat and lon==cLon:  
        continue  
    cLat = lat  
    cLon = lon  
    pnt = QgsGeometry.fromPoint(QgsPoint(lon,lat))
```

11. Now that we have a new point, we check whether this is the first point and add the whole layer to the map if it is. Otherwise, we edit the layer and add a new feature, as follows:

```
if firstPt:  
    firstPt = False  
    f = QgsFeature()  
    f.setGeometry(pnt)  
    vpr.addFeatures([f])  
    qmr.addMapLayer(vectorLyr)  
  
else:  
    print lon, lat  
    vectorLyr.startEditing()  
    vectorLyr.changeGeometry(1,pnt)  
    vectorLyr.commitChanges()
```

12. Finally, we refresh the map and watch the tracking point jump to a new location:

```
vectorLyr.setCacheImage(None)  
vectorLyr.updateExtents()  
vectorLyr.triggerRepaint()  
time.sleep(1)
```

How it works...

A live GPS will move in a linear, incremental path across the map. In this recipe, we used randomly-generated points that leap around the world, but the concept is the same. To connect a live GPS, you will need to use QGIS's GPS information GUI first to establish a connection, or at least get the correct connection information, and then use Python to automate things from there. Once you have the location information, you can easily manipulate the QGIS map using Python.

There's more...

The NMEA standard is old and widely used, but it is a poorly-designed protocol by modern standards. Nearly every smartphone has a GPS now, but they do not use the NMEA protocol. There are, however, several apps available for nearly every smartphone platform that will output the phone's GPS as NMEA sentences, which can be used by QGIS. Later in this chapter, in the *Collecting field data* recipe, we'll demonstrate another method for tracking a cell phone, GPS, or even estimated locations for digital devices, which is much simpler and much more modern.

Creating a mapbook

A mapbook is an automatically-generated document, which can also be called an **atlas**. A mapbook takes a dataset and breaks it down into smaller, detailed maps based on a coverage layer that zooms the larger map to each feature in the coverage in order to make a page of the mapbook. The coverage layer may or may not be the same as the map layer featured on each page of the mapbook. In this recipe, we'll create a mapbook that features all the countries in the world.

Getting ready

For this recipe, you need to download the world countries dataset from <https://geospatialpython.googlecode.com/svn/countries.zip> and put it in a directory named `shapes` within your `qgis_data` directory.

Next, you'll need to install the `PyPDF2` library. On Linux or OS X, just open a console and run the following command:

```
sudo easy_install PyPDF2
```

On Windows, open the OSGEO4W console from your start menu and run this:

```
easy_install PyPDF2
```

Finally, in your `qgis_data` directory, create a folder called `atlas` to store the mapbook's output.

How to do it...

We will build a QGIS composition and set it to atlas mode. Then, we'll add a composer map, where each country will be featured, and an overview map. Next, we'll run the atlas process to produce each page of the mapbook as separate PDF files. Finally, we'll combine the individual PDFs into a single PDF file. To do this, we need to perform the following steps:

1. First, import all the libraries that are needed:

```
from PyQt4.QtCore import *
from PyQt4.QtGui import *
from qgis.core import *
import PyPDF2
import os
```

2. Next, create variables related to the output files, including the mapbook's name, the coverage layer, and the naming pattern for the individual PDF files:

```
filenames = []
mapbook = "/Users/joellawhead/qgis_data/atlas/mapbook.pdf"
coverage = "/Users/joellawhead/qgis_data/shapes/countries.shp"
atlasPattern = "/Users/joellawhead/qgis_data/atlas/output_"
```

3. Now, add the coverage layer to the map using the following code:

```
vlyr = QgsVectorLayer(coverage, "Countries", "ogr")
QgsMapLayerRegistry.instance().addMapLayer(vlyr)
```

4. Next, establish the map renderer:

```
mr = QgsMapRenderer()
mr.setLayerSet([vlyr.id()])
mr.setProjectionsEnabled(True)
mr.setMapUnits(QGis.DecimalDegrees)
crs = QgsCoordinateReferenceSystem()
crs.createFromSrid(4326)
mr.setDestinationCrs(crs)
```

5. Then, set up the composition:

```
c = QgsComposition(mr)
c.setPaperSize(297, 210)
```

6. Create a symbol for the coverage layer:

```
gray = {"color": "155,155,155"}
mapSym = QgsFillSymbolV2.createSimple(gray)
renderer = QgsSingleSymbolRendererV2(mapSym)
vlyr.setRendererV2(renderer)
```

7. Now, add the first composer map to the composition, as shown here:

```
atlasMap = QgsComposerMap(c, 20, 20, 130, 130)
atlasMap.setFrameEnabled(True)
c.addComposerMap(atlasMap)
```

8. Then, create the atlas framework:

```
atlas = c.atlasComposition()
atlas.setCoverageLayer(vlyr)
atlas.setHideCoverage(False)
atlas.setEnabled(True)
c.setAtlasMode(QgsComposition.ExportAtlas)
```

9. Next, establish the overview map:

```
ov = QgsComposerMap(c, 180, 20, 50, 50)
ov.setFrameEnabled(True)
ov.setOverviewFrameMap(atlasMap.id())
c.addComposerMap(ov)
rect = QgsRectangle(vlyr.extent())
ov.setNewExtent(rect)
```

10. Then, create the overview map symbol:

```
yellow = {"color": "255,255,0,255"}
ovSym = QgsFillSymbolV2.createSimple(yellow)
ov.setOverviewFrameSymbol(ovSym)
```

11. Next, you need to label each page with the name of the country, which is stored in the CNTRY_NAME field of the shapefile:

```
lbl = QgsComposerLabel(c)
c.addComposerLabel(lbl)
lbl.setText('[% "CNTRY_NAME" %]')
lbl.setFont(QgsFontUtils.getStandardTestFont())
lbl.adjustSizeToText()
lbl.setSceneRect(QRectF(150, 5, 60, 15))
```

12. Now, we'll tell the atlas to use automatic scaling for each country in order to best fit each map in the window:

```
atlasMap.setAtlasDriven(True)
atlasMap.setAtlasScalingMode(QgsComposerMap.Auto)
atlasMap.setAtlasMargin(0.10)
```

13. Now we tell the atlas to loop through all the features and create PDF maps, as follows:

```
atlas.setFilenamePattern("%s || $feature" % atlasPattern)
atlas.beginRender()

for i in range(0, atlas.numFeatures()):
    atlas.prepareForFeature(i)
    filename = atlas.currentFilename() + ".pdf"
    print "Writing file %s" % filename
    filenames.append(filename)
    c.exportAsPDF(filename)

atlas.endRender()
```

14. Finally, we will use the PyPDF2 library to combine the individual PDF files into a single PDF file, as shown here:

```
output = PyPDF2.PdfFileWriter()
for f in filenames:
    pdf = open(f, "rb")
    page = PyPDF2.PdfFileReader(pdf)
    output.addPage(page.getPage(0))
    os.remove(f)
print "Writing final mapbook..."
book = open(mapbook, "wb")
output.write(book)
with open(mapbook, 'wb') as book:
    output.write(book)
```

How it works...

You can customize the template that creates the individual pages as much as you want. The GUI atlas tool can export the atlas to a single file, but this functionality is not available in PyQGIS, so we use the pure Python PyPDF2 library. You can also create a template in the GUI, save it, and load it with Python, but it is often easier to make changes if you have the layout available in the code. You should also know that the PDF pages are just images. The maps are exported as rasters, so the mapbook will not be searchable and the file size can be large.

Finding the least cost path

Least cost path (LCP) analysis is the raster equivalent of network analysis, which is used to find the optimal path between two points in a raster. In this recipe, we'll perform LCP analysis on a digital elevation model (DEM).

Getting ready

You need to download the following DEM and extract the ZIP file to your `qgis_data/rasters` directory: <https://geospatialpython.googlecode.com/svn/lcp.zip>

How to do it...

We will load our DEM and two shapefiles consisting of start and end points. Then, we'll use GRASS through the Processing Toolbox to create a cumulative cost layer that assigns a cost to each cell in a raster based on its elevation, the value of the other cells around it, and its distance to and from the end points.

Then, we'll use a SAGA processing algorithm to find the least cost path between two points. Finally, we'll load the output onto the map. To do this, we need to perform the following steps:

1. First, we'll import the QGIS processing Python library:

```
import processing
```

2. Now, we'll set the paths to the layers, as follows:

```
path = "/Users/joellawhead/qgis_data/rasters"/  
dem = path + "dem.asc"  
start = path + "start-point.shp"  
finish = path + "end-point.shp"
```

3. We need the DEM's extent as a string for the algorithms:

```
demLyr = QgsRasterLayer(dem, "DEM")  
ext = demLyr.extent()  
xmin = ext.xMinimum()  
ymin = ext.yMinimum()  
xmax = ext.xMaximum()  
ymax = ext.yMaximum()  
box = "%s,%s,%s,%s" % (xmin,xmax,ymin,ymax)
```

4. Using the following code, we will establish the end points as layers:

```
a = QgsVectorLayer(start, "Start", "ogr")  
b = QgsVectorLayer(finish, "End", "ogr")
```

5. Then, we'll create the cumulative cost raster, specifying the algorithm name, cost layer (DEM), start point layer, end point layer, speed or accuracy option, keep null values option, extent of interest, cell size (0 for default), and some additional defaults:

```
tmpCost = processing.runalg("grass:r.cost",dem,a,b,\  
False,False,box,0,-1,0.0001,None)  
cost = tmpCost["output"]
```

6. We also need to combine the points into a single layer for the SAGA algorithm:

```
tmpMerge =  
processing.runalg("saga:mergeshapeslayers",\start,finish,None)  
merge = tmpMerge["OUT"]
```

7. Next, we set up the inputs and outputs for the LCP algorithm:

```
vLyr = QgsVectorLayer(merge, "Destination Points", "ogr")
rLyr = QgsRasterLayer(cost, "Accumulated Cost")
line = path + "path.shp"
```

8. Then, we run the LCP analysis using the following code:

```
results =
processing.runalg("saga:leastcostpaths", \lyr, rLyr, demLyr, None,
line)
```

9. Finally, we can load the path to view it:

```
path = QgsVectorLayer(line, "Least Cost Path", "ogr")
QgsMapLayerRegistry.instance().addMapLayers([demLyr, \ vLyr,
path])
```

How it works...

GRASS has an LCP algorithm too, but the SAGA algorithm is easier to use. GRASS does a great job of creating the cost grid. Processing Toolbox algorithms allow you to create temporary files that are deleted when QGIS closes. So, we use temporary files for the intermediate products, including the cost grid and the merged shapefile.

Performing nearest neighbor analysis

Nearest neighbor analysis relates one point to the nearest point in one or more datasets. In this recipe, we'll relate one set of points to the closest point from another dataset. In this case, we'll find the closest major city for each entry in a catalog of unidentified flying object (UFO) sightings from the National UFO reporting center. This analysis will tell you which major cities have the most UFO activity. The UFO catalog data just contains latitude and longitude points, so we'll use nearest neighbor analysis to assign names to places.

Getting ready

Download the following ZIP file and extract it to a directory named `ufo` in your `qgis_data` directory:

<https://geospatialpython.googlecode.com/svn/ufo.zip>

You will also need the MMQGIS plugin:

1. From the QGIS **Plugins** menu, select **Manage and Install Plugins....**
2. In the **Plugins** dialog search box, search for `mmqgis`.
3. Select the **MMQGIS plugin** and click on the **Install plugin** button.

How to do it...

This recipe is simple. Here, we will load the layers and run the nearest neighbor algorithm within the MMQGIS plugin, as follows:

1. First, we'll import the MMQGIS plugin:

```
from mmqgis import mmqgis_library as mmqgis
```

2. Next, as shown here, we'll load all our datasets:

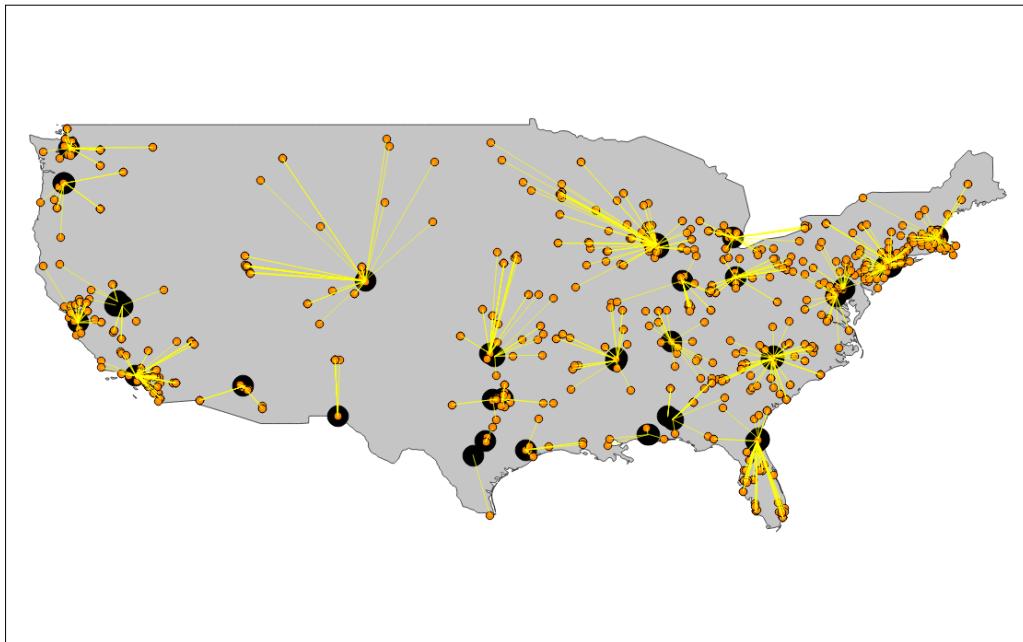
```
srcPath = "/qgis_data/ufo/ufo-sightings.shp"
dstPath = "/qgis_data/ufo/major-cities.shp"
usPth = "/qgis_data/ufo/continental-us.shp"
output = "/qgis_data/ufo/alien_invasion.shp"
srcName = "UFO Sightings"
dstName = "Major Cities"
usName = "Continental US"
source = QgsVector(srcPath, srcName, "ogr")
dest = QgsVector(dstPath, dstName, "ogr")
us = QgsVector(usPth, usName, "ogr")
```

3. Finally, we'll run and load the algorithm, which will draw lines from each UFO sighting point to the nearest city:

```
mmqgis.mmqgis_hub_distance(iface, srcName, dstName, \"NAME\",
" Miles", True, output, True)
```

How it works...

There are a couple of different nearest neighbor algorithms in QGIS, but the MMQGIS version is an excellent implementation and has the best visualization. Like the other recipes in this chapter, the plugin doesn't have an intentional Python API, so a good way to explore its functionality is to use the GUI interface before taking a look at the Python code. The following image shows the output, with UFO sightings represented by smaller points and hub lines leading to the cities, which are represented by larger, darker points.



Creating a heat map

A **heat map** is used to show the geographic clustering of data using a raster image that shows density. The clustering can also be weighed using a field in the data to not only show geographic density but also an intensity factor. In this recipe, we'll use earthquake point data to create a heat map of the impact of an earthquake and weigh the clustering by the earthquake's magnitude.

Getting ready

This recipe requires no preparation.

How to do it...

We will build a map with a worldwide base layer of countries and earthquake locations, both in GeoJSON. Next, we'll run the SAGA kernel density estimation algorithm to produce the heat map image. We'll create a layer from the output, add a color shader to it, and add it to the map.

To do this, we need to perform the following steps:

1. First, we'll import the Python libraries that we'll need in the Python console:

```
from PyQt4.QtCore import *
from PyQt4.QtGui import *
import processing
```

2. Next, using the following code, we'll define our map layers and the output raster name:

```
countries =
"https://raw.githubusercontent.com/johan/world.geo.json/master/countries.geo.json"

quakes =
"https://geospatialpython.googlecode.com/svn/quakes2014.geojson"

output = "/Users/joellawhead/qgis_data/rasters/heat.tif"
```

3. Now we'll add the layers to the map:

```
basemap = QgsVectorLayer(countries, "World", "ogr")
quakeLyr = QgsVectorLayer(quakes, "Earthquakes", "ogr")
QgsMapLayerRegistry.instance().addMapLayers([quakeLyr,
basemap])
```

4. We need to get the extent of the earthquake layer for the Processing Toolbox algorithm to use:

```
ext = quakeLyr.extent()
xmin = ext.xMinimum()
ymin = ext.yMinimum()
xmax = ext.xMaximum()
ymax = ext.yMaximum()
box = "%s,%s,%s,%s" % (xmin,xmax,ymin,ymax)
```

5. Now, we can run the kernel density estimation algorithm by specifying the mag or magnitude field as our weighting factor:

```
processing.runalg("saga:kerneldensityestimation",quakeLyr,"mag",
",10,0,0,box,1,output)
```

6. Next, we load the output as a layer:

```
heat = QgsRasterLayer(output, "Earthquake Heatmap")
```

7. Then, we create the color ramp shader and apply it to the layer:

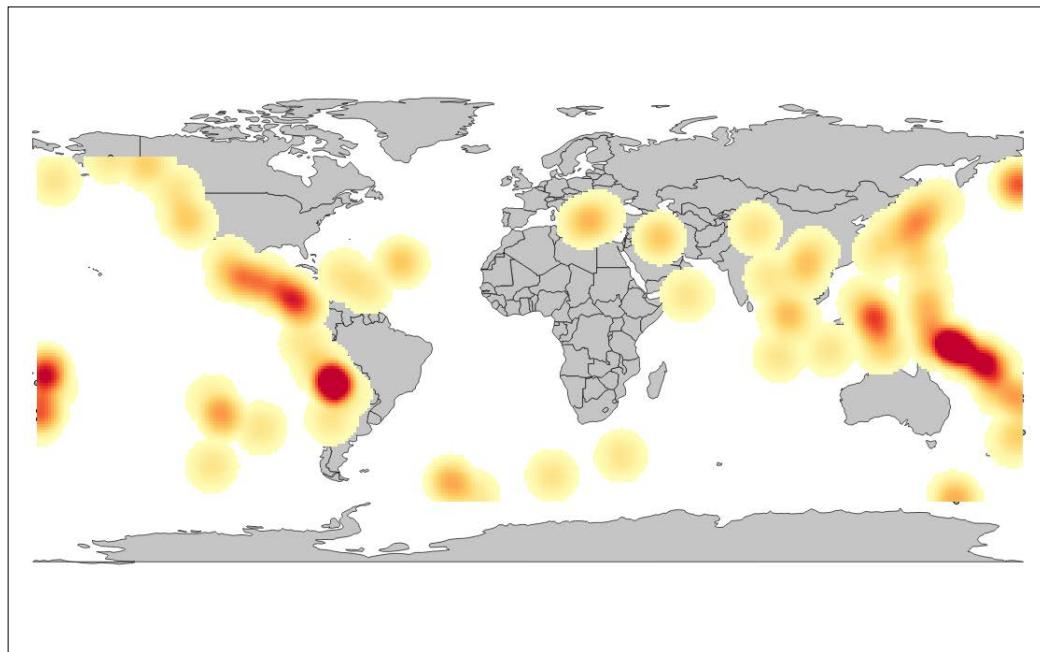
```
algorithm = QgsContrastEnhancement.StretchToMinimumMaximum
limits = QgsRaster.ContrastEnhancementMinMax
heat.setContrastEnhancement(algorithm, limits)
s = QgsRasterShader()
c = QgsColorRampShader()
c.setColorRampType(QgsColorRampShader.INTERPOLATED)
i = []
qri = QgsColorRampShader.ColorRampItem
i.append(qri(0, QColor(255,255,178,255), \
'Lowest Earthquake Impact'))
i.append(qri(0.106023, QColor(254,204,92,255), \
'Lower Earthquake Impact'))
i.append(qri(0.212045, QColor(253,141,60,255), \
'Moderate Earthquake Impact'))
i.append(qri(0.318068, QColor(240,59,32,255), \
'Higher Earthquake Impact'))
i.append(qri(0.42409, QColor(189,0,38,255), \
'Highest Earthquake Impact'))
c.setColorRampItemList(i)
s.setRasterShaderFunction(c)
ps = QgsSingleBandPseudoColorRenderer(heat.dataProvider(), \ 1,
s)
heat.setRenderer(ps)
```

8. Finally, we add the Heatmap to our map:

```
QgsMapLayerRegistry.instance().addMapLayers([heat])
```

How it works...

The kernel density estimation algorithm looks at the point dataset and forms clusters. The higher the value, the denser is the cluster. The algorithm then increases values based on the weighting factor, which is the earthquake's magnitude. The output image is, of course, a grayscale geotiff, but we use the color ramp shader to make the visualization easier to understand. The following screenshot shows the expected output:



There's more...

QGIS has a fantastic plugin available, called heat map, that works well on a wide variety of data automatically. However, it is written in C++ and does not have a Python API.

Creating a dot density map

A dot density map uses point density to illustrate a field value within a polygon. We'll use this technique to illustrate population density in some US census bureau tracts.

Getting ready

You will need to download the census tract layer and extract it to a directory named `census` in your `qgis_data` directory from https://geospatialpython.googlecode.com/files/GIS_CensusTract.zip.

How to do it...

We will load the census layer, create a memory layer, loop through the features in the census layer, calculate a random point within the feature for every 100 people, and finally add the point to the memory layer. To do this, we need to perform the following steps:

1. In the QGIS **Python Console**, we'll import the `random` module:

```
import random
```

2. Next, we'll load the census layer:

```
src = "/Users/joellawhead/qgis_data/census/\\
GIS_CensusTract_poly.shp"
tractLyr = QgsVectorLayer(src, "Census Tracts", "ogr")
```

3. Then, we'll create our memory layer:

```
popLyr = QgsVectorLayer('Point?crs=epsg:4326', "Population" ,
"memory")
```

4. We need the index for the population value:

```
i = tractLyr.fieldNameIndex('POPULAT11')
```

5. Now, we get our census layer's features as an iterator:

```
features = tractLyr.getFeatures()
```

6. We need a data provider for the memory layer so that we can edit it:

```
vpr = popLyr.dataProvider()
```

-
7. We'll create a list to store our random points:

```
dotFeatures = []
```

8. Then, we can loop through the features and calculate the density points:

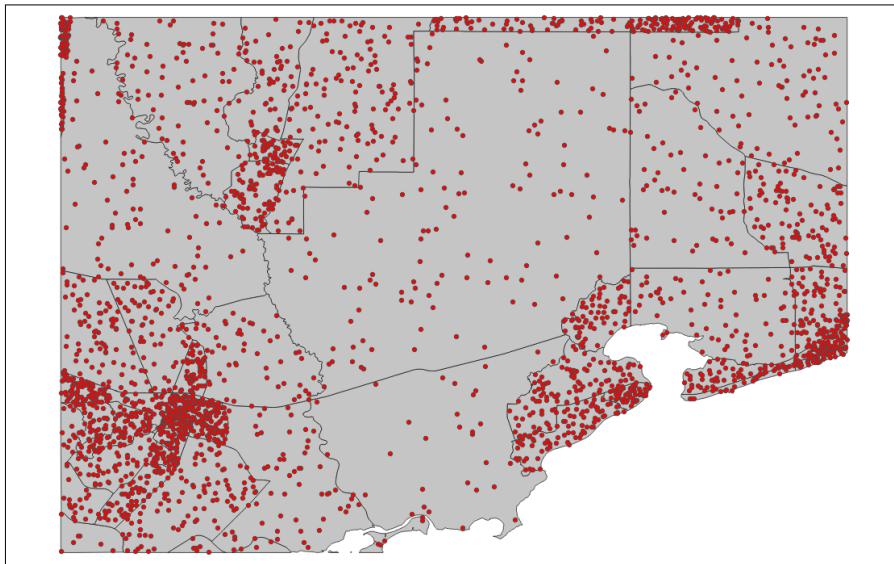
```
for feature in features:  
    pop = feature.attributes()[i]  
    density = pop / 100  
    found = 0  
    dots = []  
    g = feature.geometry()  
    minx = g.boundingBox().xMinimum()  
    miny = g.boundingbox().yMinimum()  
    maxx = g.boundingbox().xMaximum()  
    maxy = g.boundingbox().yMaximum()  
    while found < density:  
        x = random.uniform(minx,maxx)  
        y = random.uniform(miny,maxy)  
        pnt = QgsPoint(x,y)  
        if g.contains(pnt):  
            dots.append(pnt)  
            found += 1  
    geom = QgsGeometry.fromMultiPoint(dots)  
    f = QgsFeature()  
    f.setGeometry(geom)  
    dotFeatures.append(f)
```

9. Now, we can add our features to the memory layer using the following code and add them to the map in order to see the result:

```
vpr.addFeatures(dotFeatures)  
popLyr.updateExtents()  
QgsMapLayerRegistry.instance().addMapLayers(\  
[popLyr,tractLyr])
```

How it works...

This approach is slightly inefficient; it uses a brute-force approach that can place randomly generated points outside irregular polygons. We use the feature's extents to contain the random points as close as possible and then use the geometry contains method to verify that the point is inside the polygon. The following screenshot shows a sample of the output:



Collecting field data

For decades, collecting field observation data from the field into a GIS required hours of manual data entry or, at best, loading data after the trip. Smartphones and laptops with cellular connections have revolutionized this process. In this recipe, we'll use a simple but interesting geojson-based framework to enter information and a map location from any Internet-connected device with a web browser and update a map in QGIS.

Getting ready

There is no preparation required for this recipe.

How to do it...

We will load a world boundaries layer and the field data layer onto a QGIS map, go to the field data mobile website and create an entry, and then refresh the QGIS map to see the update. To do this, we need to perform the following steps:

1. In the QGIS **Python Console**, add the following geojson layers:

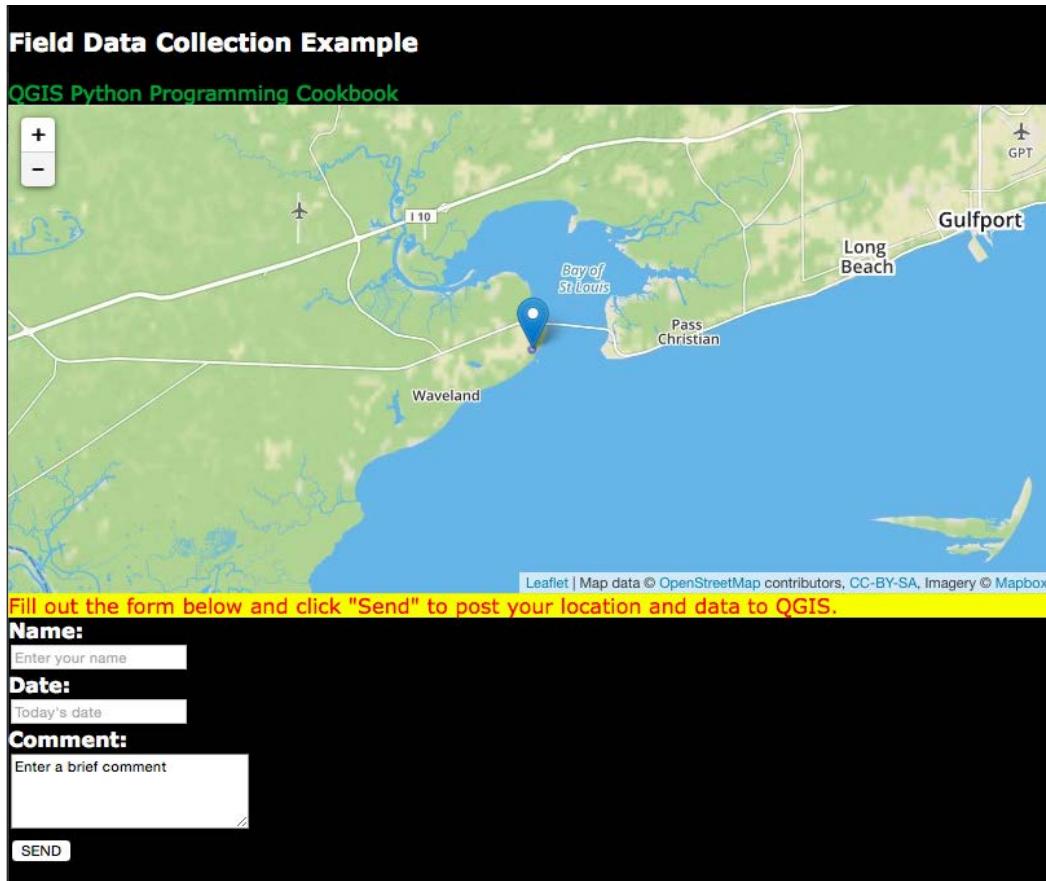
```
wb = "https://raw.githubusercontent.com/johan/\nworld.geo.json/master/countries.geo.json"\n\nbasemap = QgsVectorLayer(wb, "Countries", "ogr")\nobservations = \
QgsVectorLayer("http://bit.ly/QGISFieldApp", \
"Field Observations", "ogr")\n\nQgsMapLayerRegistry.instance().addMapLayers(\n[basemap, observations])
```

2. Now, in a browser on your computer, or preferably on a mobile device with a data connection, go to <http://geospatialpython.github.io/qgis/fieldwork.html>. The application will ask you for permission to use your location, which you should temporarily allow for the program to work.
3. Enter information in the form and click on the **Send** button.
4. Verify that you can see the geojson data, including your submission, at <https://api.myjson.com/bins/3ztvz>.
5. Finally, update the map in QGIS by zooming or panning and locate your record.

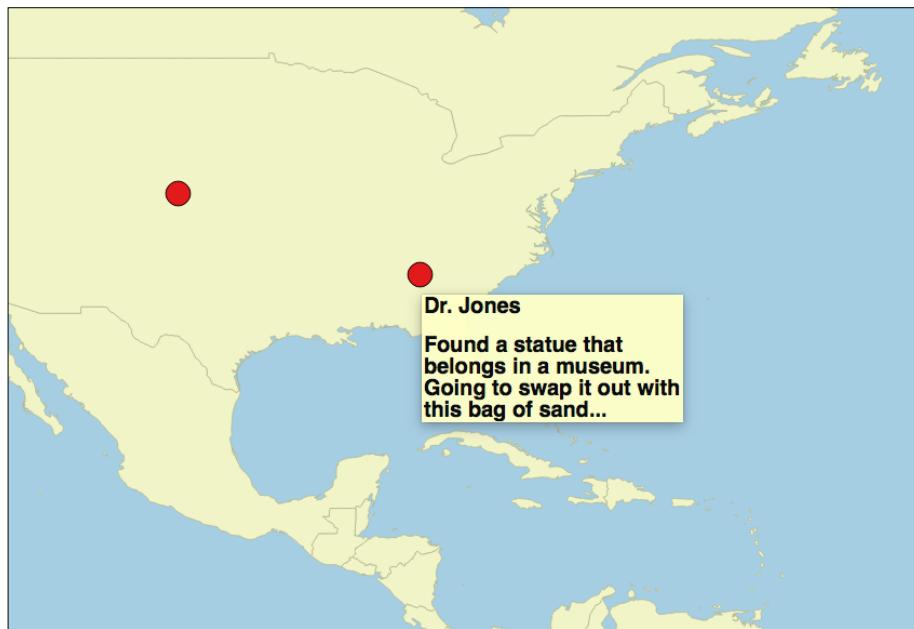
How it works...

The simple mobile-friendly web page uses the Leaflet.js library for mapping and HTML5 for the form submission. The data is stored as a snippet on the MyJSON.com service. This approach serves our examples and demonstrates the client-server model. However, it is not very robust because users working concurrently can easily overwrite each other's data. So, if you don't see your update, try it again once or twice and it will probably work. Sample observations are reset from time to time in order to keep the site lightweight. Note that it's important to refresh the map either manually or programmatically to force QGIS to refresh the network link. You can get the source code for the mobile page on GitHub.com (<https://github.com/GeospatialPython/qgis>).

The following image shows the mobile field application on an iPhone:



This image shows how the corresponding data looks in QGIS:



Computing road slope using elevation data

A common geospatial workflow is to assign raster values to a coincident vector layer so that you can style or perform further analysis on the vector layer. This recipe will use this concept to illustrate the steepness of a road using color by mapping values to the road vector from a slope raster.

Getting ready

You will need to download a zipped directory from <https://geospatialpython.googlecode.com/svn/road.zip> and place the directory, named `road`, in your `qgis_data` directory.

How to do it...

We'll start with a DEM and compute its slope. Then, we'll load a road vector layer and break it into interval lengths of 500 meters. Next, we'll load the layer and style it using green, yellow, and red values for each segment to show the range of steepness. We'll overlay this layer on a hillshade of the DEM for a nice visualization. To do this, we need to perform the following steps:

1. First, we need to import the QGIS processing module, the QGIS constants module, the Qt GUI module, and the os module in the QGIS **Python Console**:

```
from qgis.core import *
from PyQt4.QtGui import *
import processing
```

2. Now, we need to set the coordinate reference system (CRS) of our project to that of our digital elevation model (DEM), which is EPSG code 26910, so we can work with the data in meters instead of decimal degrees:

```
myCrs = QgsCoordinateReferenceSystem(26910,
QgsCoordinateReferenceSystem.EpsgCrsId)

iface.mapCanvas().mapRenderer().setDestinationCrs(myCrs)

iface.mapCanvas().setMapUnits(QGis.Meters)

iface.mapCanvas().refresh()
```

3. Now, we'll set the paths of all the layers. For this, we'll use intermediate layers that we create so that we can change them in one place, if needed:

```
src_dir = "/Users/joellawhead/qgis_data/road/"

dem = os.path.join(src_dir, "dem.asc")
road = os.path.join(src_dir, "road.shp")
slope = os.path.join(src_dir, "slope.tif")
segRoad = os.path.join(src_dir, "segRoad.shp")
steepness = os.path.join(src_dir, "steepness.shp")
hillshade = os.path.join(src_dir, "hillshade.tif")
```

4. We will load the DEM and road layer so that we can get the extents for the processing algorithms:

```
demLyr = QgsRasterLayer(dem, "DEM")
roadLyr = QgsVectorLayer(road, "Road", "ogr")
```

-
5. Now, build a string with the DEM extent using the following code:

```
ext = demLyr.extent()
xmin = ext.xMinimum()
ymin = ext.yMinimum()
xmax = ext.xMaximum()
ymax = ext.yMaximum()
demBox = "%s,%s,%s,%s" % (xmin,xmax,ymin,ymax)
```

6. Next, compute the slope grid:

```
processing.runalg("grass:r.slope",dem,0,0,1,0,True, \
demBox,0,slope)
```

7. Then, we can get the extent of the road layer as a string:

```
ext = roadLyr.extent()
xmin = ext.xMinimum()
ymin = ext.yMinimum()
xmax = ext.xMaximum()
ymax = ext.yMaximum()
roadBox = "%s,%s,%s,%s" % (xmin,xmax,ymin,ymax)
```

8. Now, we'll break the road layer into segments of 500 meters to have a meaningful length for the slope valuation:

```
processing.runalg("grass:v.split.length",road,500, \
roadBox,-1,0.0001,0,segRoad)
```

9. Next, we'll add the slope and segmented layer to the map interface for the next algorithm, but we'll keep them hidden from view using the boolean `False` option in the `addMapLayers` method:

```
slopeLyr = QgsRasterLayer(slope, "Slope")
segRoadLyr = QgsVectorLayer(segRoad, \
"Segmented Road", "ogr")
QgsMapLayerRegistry.instance().addMapLayers([ \
segRoadLyr,slopeLyr], False)
```

10. Now, we can transfer the slope values to the segmented road layer in order to create the steepness layer:

```
processing.runalg("saga:addgridvaluestoshapes", \
segRoad,slope,0,steepness)
```

11. Now, we can load the steepness layer:

```
steepLyr = QgsVectorLayer(steepness, \ "Road Gradient", "ogr")
```

12. We'll style the steepness layer to use the stoplight red, yellow, and green values, with red being the steepest:

```
roadGrade = (
("Rolling Hill", 0.0, 20.0, "green"),
("Steep", 20.0, 40.0, "yellow"),
("Very Steep", 40.0, 90.0, "red"))

ranges = []

for label, lower, upper, color in roadGrade:
    sym = QgsSymbolV2.defaultSymbol(steepLyr.geometryType())
    sym.setColor(QColor(color))
    sym.setWidth(3.0)
    rng = QgsRendererRangeV2(lower, upper, sym, label)
    ranges.append(rng)

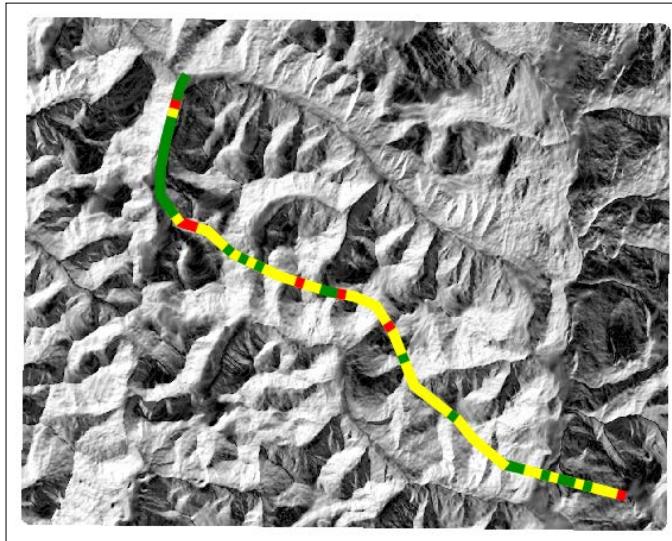
field = "slope"
renderer = QgsGraduatedSymbolRendererV2(field, ranges)
steepLyr.setRendererV2(renderer)
```

13. Next, we'll create a hillshade from the DEM for visualization and load everything onto the map:

```
processing.runalg("saga:analyticalhillshading",dem, \
0,315,45,4,hillshade)
hs = QgsRasterLayer(hillshade, "Terrain")
QgsMapLayerRegistry.instance().addMapLayers([steepLyr, hs])
```

How it works...

For each of our 500-meter line segments, the algorithm averages the underlying slope values. This workflow is fairly simple and also provides all the building blocks you need for a more complex version. While performing calculations that involve measurements over a relatively small area, using projected data is the best option. The following image shows how the output looks:



Geolocating photos on the map

Photos taken with GPS-enabled cameras, including smartphones, store location information in the header of the file, in a format called EXIF tags. These tags are largely based on the same header tags used by the TIFF image standard. In this recipe, we'll use these tags to create locations on a map for some photos and provide links to open them.

Getting ready

You will need to download some sample geotagged photos from <https://github.com/GeospatialPython/qgis/blob/gh-pages/photos.zip?raw=true> and place them in a directory named `photos` in your `qgis_data` directory.

How to do it...

QGIS requires the **Python Imaging Library (PIL)**, which should already be included with your installation. PIL can parse EXIF tags. We will gather the filenames of the photos, parse the location information, convert it to decimal degrees, create the point vector layer, add the photo locations, and add an action link to the attributes. To do this, we need to perform the following steps:

1. In the QGIS **Python Console**, import the libraries that we'll need, including `k` for parsing image data and the `glob` module for doing wildcard file searches:

```
import glob
import Image
from ExifTags import TAGS
```

2. Next, we'll create a function that can parse the header data:

```
def exif(img):
    exif_data = {}
    try:
        i = Image.open(img)
        tags = i._getexif()
        for tag, value in tags.items():
            decoded = TAGS.get(tag, tag)
            exif_data[decoded] = value
    except:
        pass
    return exif_data
```

3. Now, we'll create a function that can convert degrees-minute-seconds to decimal degrees, which is how coordinates are stored in JPEG images:

```
def dms2dd(d, m, s, i):
    sec = float((m * 60) + s)
    dec = float(sec / 3600)
    deg = float(d + dec)
    if i.upper() == 'W':
        deg = deg * -1
    elif i.upper() == 'S':
        deg = deg * -1
    return float(deg)
```

4. Next, we'll define a function to parse the location data from the header data:

```
def gps(exif):
    lat = None
```

```
lon = None
if exif['GPSInfo']:
    # Lat
    coords = exif['GPSInfo']
    i = coords[1]
    d = coords[2][0][0]
    m = coords[2][1][0]
    s = coords[2][2][0]
    lat = dms2dd(d, m, s, i)
    # Lon
    i = coords[3]
    d = coords[4][0][0]
    m = coords[4][1][0]
    s = coords[4][2][0]
    lon = dms2dd(d, m, s, i)
return lat, lon
```

5. Next, we'll loop through the photos directory, get the filenames, parse the location information, and build a simple dictionary to store the information, as follows:

```
photos = []
photo_dir = "/Users/joellawhead/qgis_data/photos/"
files = glob.glob(photo_dir + "*.jpg")
for f in files:
    e = exif(f)
    lat, lon = gps(e)
    photos[f] = [lon, lat]
```

6. Now, we'll set up the vector layer for editing:

```
lyr_info = "Point?crs=epsg:4326&field=photo:string(75)"
vectorLyr = QgsVectorLayer(lyr_info, \"Geotagged Photos\" ,
"memory")
vpr = vectorLyr.dataProvider()
```

7. We'll add the photo details to the vector layer:

```
features = []
for pth, p in photos.items():
    lon, lat = p
    pnt = QgsGeometry.fromPoint(QgsPoint(lon, lat))
    f = QgsFeature()
    f.setGeometry(pnt)
    f.setAttributes([pth])
    features.append(f)
```

```
vpr.addFeatures(features)
vectorLyr.updateExtents()
```

8. Now, we can add the layer to the map and make the active layer:

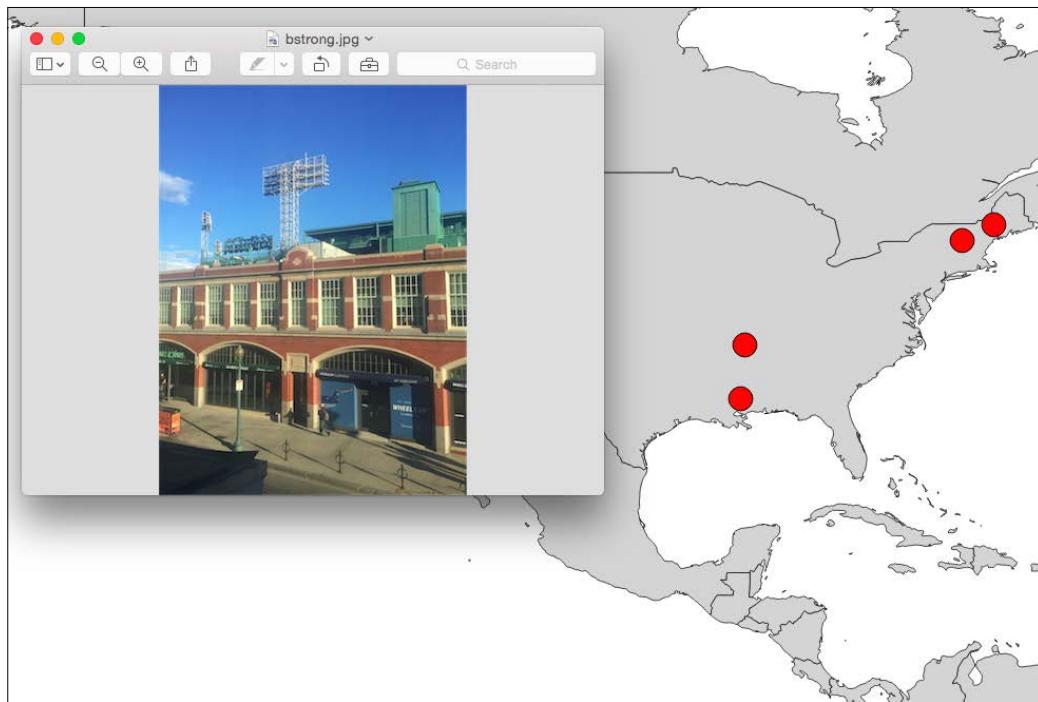
```
QgsMapLayerRegistry.instance().addMapLayer(vectorLyr)
iface.setActiveLayer(vectorLyr)
activeLyr = iface.activeLayer()
```

9. Finally, we'll add an action that allows you to click on it and open the photo:

```
actions = activeLyr.actions()
actions.addAction(QgsAction.OpenUrl, "Photos", \'[% "photo"
%] \' )
```

How it works...

Using the included PIL EXIF parser, getting location information and adding it to a vector layer is relatively straightforward. The interesting part of this recipe is the QGIS action to open the photo. This action is a default option for opening a URL. However, you can also use Python expressions as actions to perform a variety of tasks. The following screenshot shows an example of the data visualization and photo popup:



There's more...

Another plugin called Photo2Shape is available, but it requires you to install an external EXIF tag parser.

Image change detection

Change detection allows you to automatically highlight the differences between two images in the same area if they are properly orthorectified. In this recipe, we'll do a simple difference change detection on two images, which are several years apart, to see the differences in urban development and the natural environment.

Getting ready

You can download the two images for this recipe from <https://github.com/GeospatialPython/qgis/blob/gh-pages/change-detection.zip?raw=true> and put them in a directory named `change-detection` in the `rasters` directory of your `qgis_data` directory. Note that the file is 55 megabytes, so it may take several minutes to download.

How to do it...

We'll use the QGIS raster calculator to subtract the images in order to get the difference, which will highlight significant changes. We'll also add a color ramp shader to the output in order to visualize the changes. To do this, we need to perform the following steps:

1. First, we need to import the libraries that we need in to the QGIS console:

```
from PyQt4.QtGui import *
from PyQt4.QtCore import *
from qgis.analysis import *
```

2. Now, we'll set up the path names and raster names for our images:

```
before = "/Users/joellawhead/qgis_data/rasters/change-
detection/before.tif"
after = "/Users/joellawhead/qgis_data/rasters/change-
detection/after.tif"
beforeName = "Before"
afterName = "After"
```

3. Next, we'll establish our images as raster layers:

```
beforeRaster = QgsRasterLayer(before, beforeName)
afterRaster = QgsRasterLayer(after, afterName)
```

4. Then, we can build the calculator entries:

```
beforeEntry = QgsRasterCalculatorEntry()
afterEntry = QgsRasterCalculatorEntry()
beforeEntry.raster = beforeRaster
afterEntry.raster = afterRaster
beforeEntry.bandNumber = 1
afterEntry.bandNumber = 2
beforeEntry.ref = beforeName + "@1"
afterEntry.ref = afterName + "@2"
entries = [afterEntry, beforeEntry]
```

5. Now, we'll set up the simple expression that does the math for remote sensing:

```
exp = "%s - %s" % (afterEntry.ref, beforeEntry.ref)
```

6. Then, we can set up the output file path, the raster extent, and pixel width and height:

```
output = "/Users/joellawhead/qgis_data/rasters/change-
detection/change.tif"
e = beforeRaster.extent()
w = beforeRaster.width()
h = beforeRaster.height()
```

7. Now, we perform the calculation:

```
change = QgsRasterCalculator(exp, output, "GTiff", e, w, h,
entries)
change.processCalculation()
```

8. Finally, we'll load the output as a layer, create the color ramp shader, apply it to the layer, and add it to the map, as shown here:

```
lyr = QgsRasterLayer(output, "Change")
algorithm = QgsContrastEnhancement.StretchToMinimumMaximum
limits = QgsRaster.ContrastEnhancementMinMax
```

```
lyr.setContrastEnhancement(algorithm, limits)
s = QgsRasterShader()
c = QgsColorRampShader()
c.setColorRampType(QgsColorRampShader.INTERPOLATED)
i = []
qri = QgsColorRampShader.ColorRampItem
i.append(qri(0, QColor(0,0,0,0), 'NODATA'))
i.append(qri(-101, QColor(123,50,148,255), 'Significant Intensity Decrease'))
i.append(qri(-42.2395, QColor(194,165,207,255), 'Minor Intensity Decrease'))
i.append(qri(16.649, QColor(247,247,247,0), 'No Change'))
i.append(qri(75.5375, QColor(166,219,160,255), 'Minor Intensity Increase'))
i.append(qri(135, QColor(0,136,55,255), 'Significant Intensity Increase'))
c.setColorRampItemList(i)
s.setRasterShaderFunction(c)
ps = QgsSingleBandPseudoColorRenderer(lyr.dataProvider(), 1, s)
lyr.setRenderer(ps)
QgsMapLayerRegistry.instance().addMapLayer(lyr)
```

How it works...

The concept is simple. We subtract the older image data from the new image data. Concentrating on urban areas tends to be highly reflective and results in higher image pixel values. If a building is added in the new image, it will be brighter than its surroundings. If a building is removed, the new image will be darker in that area. The same holds true for vegetation, to some extent.

9

Other Tips and Tricks

In this chapter, we will cover the following recipes:

- ▶ Creating tiles from a QGIS map
- ▶ Adding a layer to geojson.io
- ▶ Rendering map layers based on rules
- ▶ Creating a layer style file
- ▶ Using NULL values in PyQGIS
- ▶ Using generators for layer queries
- ▶ Using alpha values to show data density
- ▶ Using the `__geo_interface__` protocol
- ▶ Generating points along a line
- ▶ Using expression-based labels
- ▶ Creating dynamic forms in QGIS
- ▶ Calculating length for all the selected lines
- ▶ Using a different status bar CRS than the map
- ▶ Creating HTML labels in QGIS
- ▶ Using OpenStreetMap's points of interest in QGIS
- ▶ Visualizing data in 3D with WebGL
- ▶ Visualizing data on a globe

Introduction

This chapter provides interesting and valuable QGIS Python tricks that didn't fit into any topics in other chapters. Each recipe has a specific purpose, but in many cases, a recipe may demonstrate multiple concepts that you'll find useful in other programs. All the recipes in this chapter run in the QGIS Python console.

Creating tiles from a QGIS map

This recipe creates a set of Internet web map tiles from your QGIS map. What's interesting about this recipe is that once the static map tiles are generated, you can serve them up locally or from any web-accessible directory using the client-side browser's JavaScript without the need of a map server, or you can serve them (for example, distribute them on a portable USB drive).

Getting ready

You will need to download the zipped shapefile from <https://geospatialpython.googlecode.com/svn/countries.zip>.

Unzip the shapefile to a directory named `shapes` in your `qgis_data` directory. Next, create a directory called `tilecache` in your `qgis_data` directory. You will also need to install the **QTiles** plugin using the **QGIS Plugin Manager**. This plugin is experimental, so make sure that the **Show also experimental plugins** checkbox is checked in the QGIS Plugin Manager's **Settings** tab.

How to do it...

We will load the shapefile and randomly color each country. We'll then manipulate the **QTiles** plugin using Python to generate map tiles for 5 zoom levels' worth of tiles. To do this, we need to perform the following steps:

1. First, we need to import all the necessary Python libraries, including the QTiles plugin:

```
from PyQt4.QtCore import *
from PyQt4.QtGui import *
import qtiles
import random
```

2. Now, we create a color function that can produce random colors. This function accepts a mixed color, which defaults to white, to change the overall tone of the color palette:

```
def randomColor(mix=(255,255,255)):  
    red = random.randrange(0,256)  
    green = random.randrange(0,256)  
    blue = random.randrange(0,256)  
    r,g,b = mix  
    red = (red + r) / 2  
    green = (green + g) / 2  
    blue = (blue + b) / 2  
    return (red, green, blue)
```

3. Next, we'll create a simple callback function for notification of when the tile generation is done. This function will normally be used to create a message bar or other notification, but we'll keep things simple here:

```
def done():  
    print "FINISHED!!"
```

4. Now, we set the path to the shapefile and the tile's output direction:

```
shp = "/qgis_data/shapes/countries.shp"  
dir = "/qgis_data/tilecache"
```

5. Then, we load the shapefile:

```
layer = QgsVectorLayer(shp, "Countries", "ogr")
```

6. After that, we define the field that is used to color the countries:

```
field = 'CNTRY_NAME'
```

7. Now, we need to get all the features so that we can loop through them:

```
features = layer.getFeatures()
```

8. We'll build our color renderer:

```
categories = []  
for feature in features:  
    country = feature[field]  
    sym = QgsSymbolV2.defaultSymbol(layer.geometryType())  
    r,g,b = randomColor()  
    sym.setColor(QColor(r,g,b,255))  
    category = QgsRendererCategoryV2(country, sym, country)  
    categories.append(category)
```

9. Then, we'll set the layer renderer and add it to the map:

```
renderer = QgsCategorizedSymbolRendererV2(field, categories)
layer.setRendererV2(renderer)
QgsMapLayerRegistry.instance().addMapLayer(layer)
```

10. Now, we'll set all the properties we need for the image tiles, including the map elements and image properties:

```
canvas = iface.mapCanvas()
layers = canvas.mapSettings().layers()
extent = canvas.extent()
minZoom = 0
maxZoom = 5
width = 256
height = 256
transp = 255
quality = 70
format = "PNG"
outputPath = QFileInfo(dir)
rootDir = "countries"
antialiasing = False
tmsConvention = True
mapUrl = False
viewer = True
```

11. We are ready to generate the tiles using the efficient threading system of the QTiles plugin. We'll create a thread object and pass it all of the tile settings previously mentioned:

```
tt = qtiles.tilingthread.TilingThread(layers, extent, minZoom,
maxZoom, width, height, transp,
quality, format, outputPath, rootDir, antialiasing, tmsConvention,
mapUrl, viewer)
```

12. Then, we can connect the finish signal to our simple callback function:

```
tt.processFinished.connect(done)
```

13. Finally, we start the tiling process:

```
tt.start()
```

14. Once you receive the completion message, check the output directory and verify that there is an HTML file named `countries.html` and a directory named `countries`.
15. Double-click on the `countries.html` page to open it in a browser.
16. Once the map loads, click on the plus symbol (+) in the upper-left corner twice to zoom the map.
17. Next, pan around to see the tiled version of your map load.

How it works...

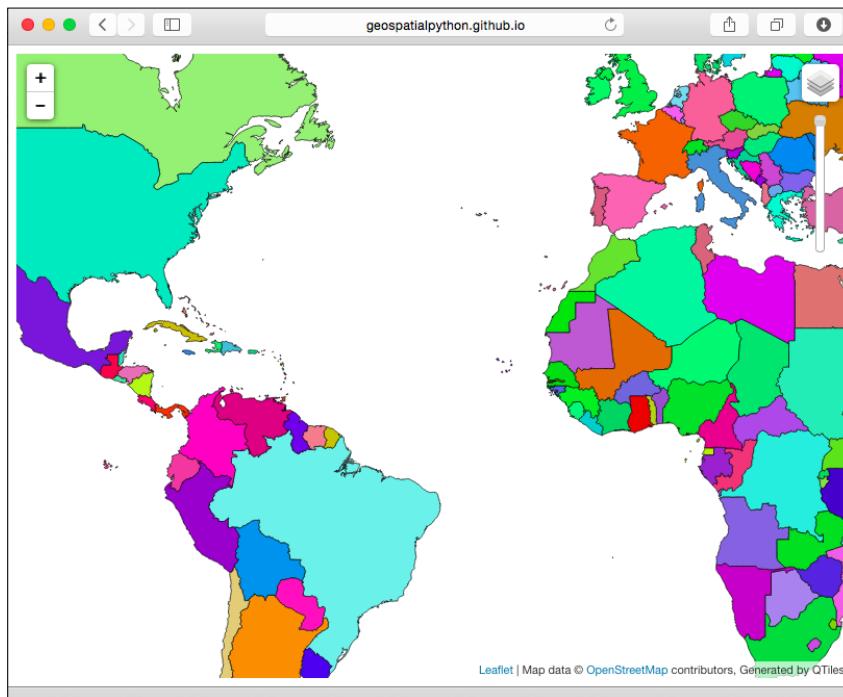
You can generate up to 16 zoom levels with this plugin. After eight zoom levels, the tile generation process takes a long time and the tile set becomes quite large on the filesystem, totaling hundreds of megabytes. One way to avoid creating a lot of files is to use the **mbtiles** format, which stores all the data in a single file. However, you need a web application using GDAL to access it.



You can see a working example of the output recipe stored in a `github.io` web directory at <http://geospatialpython.github.io/qgis/tiles/countries.html>.



The following image shows the output in a browser:



Adding a layer to geojson.io

Cloud services have become common and geospatial maps are no exception. This recipe uses a service named geojson.io, which serves vector layers online, which you can upload from QGIS using Python.

Getting ready

For this recipe, you will need to install the **qgisio** plugin using the **QGIS Plugin Manager**.

You will also need a shapefile in a geodetic coordinate system (WGS84) from <https://geospatialpython.googlecode.com/svn/union.zip>.

Decompress the ZIP file and place it in your `qgis_data` directory named `shapes`.

How to do it...

We will convert our shapefile to GeoJSON using a temporary file. We'll then use Python to call the **qgisio** plugin in order to upload the data to be displayed online. To do this, we need to perform the following steps:

1. First, we need to import all the relevant Python libraries:

```
from PyQt4.QtCore import *
from PyQt4.QtGui import *
from qgis.core import *
from tempfile import mkstemp
import os
from qgisio import geojsonio
```

2. Now, we set up the layer and get the layer's name:

```
layer = QgsVectorLayer("/qgis_data/shapes/building.shp",
"Building", "ogr")
name = layer.name()
```

3. Next, we establish a temporary file using the Python tempfile module for the GeoJSON conversion:

```
handle, tmpfile = mkstemp(suffix=".geojson")
os.close(handle)
```

4. Now, we'll establish the coordinate reference system needed for the conversion, which must be WGS84 Geographic, to work with the cloud service:

```
crs = QgsCoordinateReferenceSystem(4326,  
QgsCoordinateReferenceSystem.PostgisCrsId)
```

5. Next, we can write out the layer as GeoJSON:

```
error = QgsVectorFileWriter.writeAsVectorFormat(layer, tmpfile,  
"utf-8", crs, "GeoJSON", onlySelected=False)
```

6. Then, we can make sure that the conversion didn't have any problems:

```
if error != QgsVectorFileWriter.NoError:  
    print "Unable to write geoJSON!"
```

7. Now, we can read the GeoJSON content:

```
with open(str(tmpfile), 'r') as f:  
    contents = f.read()
```

8. We then need to remove the temporary file:

```
os.remove(tmpfile)
```

9. We are ready to upload our GeoJSON to geojson.io using the `qgisio` module:

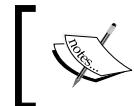
```
url = geojsonio._create_gist(contents, "Layer exported from QGIS",  
name + ".geojson")
```

10. We can then use the Qt library to open the map in a browser:

```
QDesktopServices.openUrl(QUrl(url))
```

How it works...

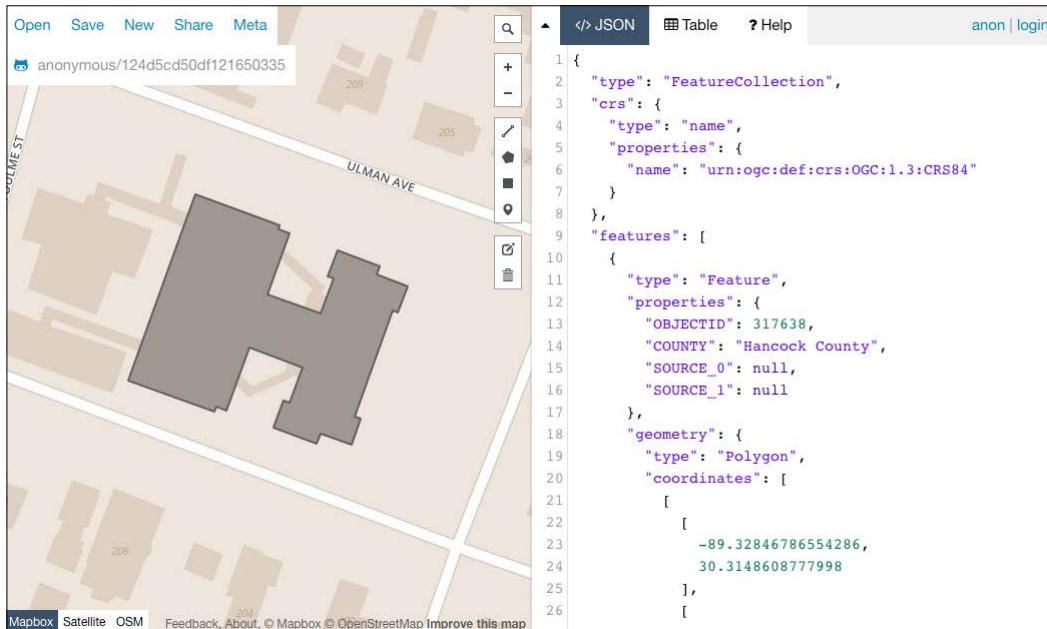
This recipe actually uses two cloud services. The GeoJSON data is stored on a <https://github.com> service named Gist that allows you to store code snippets such as JSON. The geojson.io service can read data from Gist.



Note that sometimes it can take several seconds to several minutes for the generated URL to become available online.

Other Tips and Tricks —

This screenshot shows the building layer on an OSM map on geojson.io, with the GeoJSON displayed next to the map:



There's more...

There are additional advanced services that can serve QGIS maps, including www.QGISCloud.com and www.CartoDB.com, which can also display raster maps. Both of these services have free options and QGIS plugins. However, they are far more difficult to script from Python if you are trying to automate publishing maps to the Web as part of a workflow.

Rendering map layers based on rules

Rendering rules provide a powerful way to control how and when a layer is displayed relative to other layers or to the properties of the layer itself. Using a rule-based renderer, this recipe demonstrates how to color code a layer based on an attribute.

Getting ready

You will need to download a zipped shapefile from https://geospatialpython.googlecode.com/svn/ms_rails_mstm.zip.

Unzip it and place it in the directory named `ms` in your `ggis_data` directory.

In this same directory, download and unzip the following shapefile:

<https://geospatialpython.googlecode.com/files/Mississippi.zip>

Finally, add this shapefile to the directory as well:

<https://geospatialpython.googlecode.com/svn/jackson.zip>

How to do it...

We will set up a railroad layer, then we'll set up our rules as Python tuples to color code it based on the frequency of use. Finally, we'll add some other layers to the map for reference. To do this, we need to perform the following steps:

1. First, we need to import the QTGui library to work with colors:

```
from PyQt4.QtGui import *
```

2. Next, we'll set up our data path to avoid typing it repeatedly. Replace this string with the path to your qgis_data directory:

```
prefix = "/Users/joellawhead/qgis_data/ms/"
```

3. Now, we can load our railroad layer:

```
rails = QgsVectorLayer(prefix + "ms_rails_mstm.shp", "Railways",
"ogr")
```

4. Then, we can define our rules as a set of tuples. Each rule defines a label and an expression, detailing which attribute values make up that rule, a color name, and the minimum/maximum map scale values at which the described features are visible:

```
rules = (
    ('Heavily Used', '"DEN09CODE" > 3', 'red', (0,
6000000)),
    ('Moderately Used', '"DEN09CODE" < 4 AND "DEN09CODE" >
1', 'orange', (0, 1500000)),
    ('Lightly Used', '"DEN09CODE" < 2', 'grey', (0,
250000)),
)
```

5. Next, we create a rule-based renderer and a base symbol to begin applying our rules:

```
sym_rails = QgsSymbolV2.defaultSymbol(rails.geometryType())
rend_rails = QgsRuleBasedRendererV2(sym_rails)
```

6. The rules are a hierarchy based on a root rule, so we must access the root first:

```
root_rule = rend_rails.rootRule()
```

7. Now, we will loop through our rules, clone the default rule, and append our custom rule to the tree:

```
for label, exp, color, scale in rules:  
    # create a clone (i.e. a copy) of the default rule  
    rule = root_rule.children()[0].clone()  
    # set the label, exp and color  
    rule.setLabel(label)  
    rule.setFilterExpression(exp)  
    rule.symbol().setColor(QColor(color))  
    # set the scale limits if they have been specified  
    if scale is not None:  
        rule.setScaleMinDenom(scale[0])  
        rule.setScaleMaxDenom(scale[1])  
    # append the rule to the list of rules  
    root_rule.appendChild(rule)
```

8. We can now delete the default rule, which isn't part of our rendering scheme:

```
root_rule.removeChildAt(0)
```

9. Now, we apply the renderer to our rails layer:

```
rails.setRendererV2(rend_rails)
```

10. We'll establish and style a city layer, which will provide a focal point to zoom into so that we can easily see the scale-based rendering effect:

```
jax = QgsVectorLayer(prefix + "jackson.shp", "Jackson", "ogr")  
jax_style = {}  
jax_style['color'] = "#ffff00"  
jax_style['name'] = 'regular_star'  
jax_style['outline'] = '#000000'  
jax_style['outline-width'] = '1'  
jax_style['size'] = '8'  
sym_jax = QgsSimpleMarkerSymbolLayerV2.create(jax_style)  
jax.rendererV2().symbols()[0].changeSymbolLayer(0, sym_jax)
```

11. Then, we'll set up and style a border layer around both the datasets:

```
ms = QgsVectorLayer(prefix + "mississippi.shp", "Mississippi",  
"ogr")  
ms_style = {}yea
```

```
ms_style['color'] = "#F7F5EB"
sym_ms = QgsSimpleFillSymbolLayerV2.create(ms_style)
ms.rendererV2().symbols()[0].changeSymbolLayer(0, sym_ms)
```

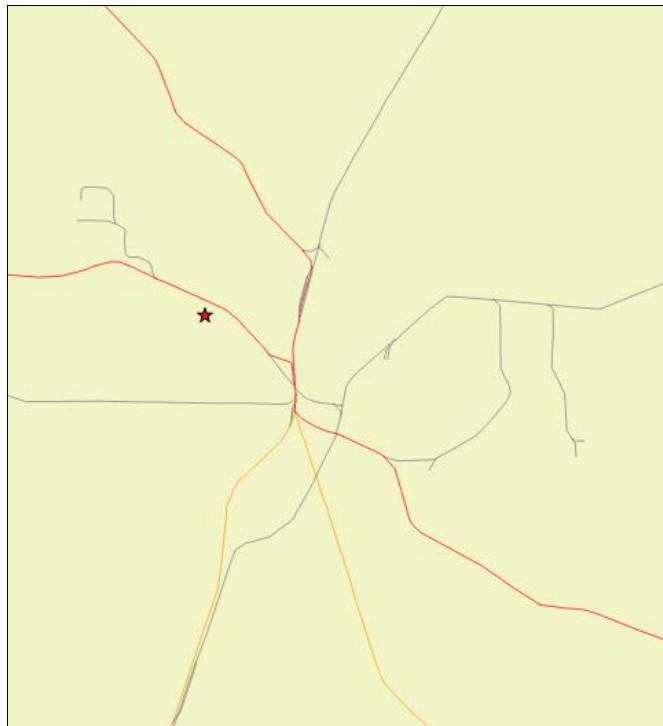
12. Finally, we'll add everything to the map:

```
QgsMapLayerRegistry.instance().addMapLayers([jax, rails, ms])
```

How it works...

Rules are a hierarchical collection of symbols and expressions. Symbols are collections of symbol layers. This recipe is relatively simple but contains over 50 lines of code. Rendering is one of the most complex features to code in QGIS. However, rules also have their own sets of properties, separate from layers and symbols. Notice that in this recipe, we are able to set labels and filters for the rules, properties that are normally relegated to layers. One way to think of rules is as separate layers. We can do the same thing by loading our railroad layer as a new layer for each rule. Rules are a more compact way to break up the rendering for a single layer.

This image shows the rendering at a scale where all the rule outputs are visible:



Creating a layer style file

Layer styling is one of the most complex aspects of the QGIS Python API. Once you've developed the style for a layer, it is often useful to save the styling to the **QGIS Markup Language (QML)** in the XML format.

Getting ready

You will need to download the zipped directory named `saveqml` and decompress it to your `qgis_data/rasters` directory from <https://geospatialpython.googlecode.com/svn/saveqml.zip>.

How to do it...

We will create a color ramp for a DEM and make it semi transparent to overlay a hillshaded tiff of the DEM. We'll save the style we create to a QML file. To do this, we need to perform the following steps:

1. First, we'll need the following Python Qt libraries:

```
from PyQt4.QtCore import *
from PyQt4.QtGui import *
```

2. Next, we'll load our two raster layers:

```
hs = QgsRasterLayer("/qgis_data/saveqml/hillshade.tif",
"hillshade")
dem = QgsRasterLayer("/qgis_data/saveqml/dem.asc", "DEM")
```

3. Next, we'll perform a histogram stretch on our DEM for better visualization:

```
algorithm = QgsContrastEnhancement.StretchToMinimumMaximum
limits = QgsRaster.ContrastEnhancementMinMax
dem.setContrastEnhancement(algorithm, limits)
```

4. Now, we'll create a visually pleasing color ramp based on the elevation values of the DEM as a renderer and apply it to the layer:

```
s = QgsRasterShader()
c = QgsColorRampShader()
```

```
c.setColorRampType(QgsColorRampShader.INTERPOLATED)
i = []
qri = QgsColorRampShader.ColorRampItem
i.append(qri(356.334, QColor(63,159,152,255), '356.334'))
i.append(qri(649.292, QColor(96,235,155,255), '649.292'))
i.append(qri(942.25, QColor(100,246,174,255), '942.25'))
i.append(qri(1235.21, QColor(248,251,155,255), '1235.21'))
i.append(qri(1528.17, QColor(246,190,39,255), '1528.17'))
i.append(qri(1821.13, QColor(242,155,39,255), '1821.13'))
i.append(qri(2114.08, QColor(165,84,26,255), '2114.08'))
i.append(qri(2300, QColor(236,119,83,255), '2300'))
i.append(qri(2700, QColor(203,203,203,255), '2700'))
c.setColorRampItemList(i)
s.setRasterShaderFunction(c)
ps = QgsSingleBandPseudoColorRenderer(dem.dataProvider(), 1, s)
ps.setOpacity(0.5)
dem.setRenderer(ps)
```

5. Now, we can add the layers to the map:

```
QgsMapLayerRegistry.instance().addMapLayers([dem, hs])
```

6. Finally, with this line, we can save the DEM's styling to a reusable QML file:

```
dem.saveNamedStyle("/qgis_data/saveqml/dem.qml")
```

How it works...

The QML format is easy to read and can be edited by hand. The `saveNamedStyle()` method works on vector layers in the exact same way. Instead of styling the preceding code, you can just reference the QML file using the `loadNamedStyle()` method:

```
dem.loadNamedStyle("/qgis_data/saveqml/dem.qml")
```

If you save the QML file along with a shapefile and use the same filename (with the `.qml` extension), then QGIS will load the style automatically when the shapefile is loaded.

Using NULL values in PyQGIS

QGIS can use NULL values as field values. Python has no concept of NULL values. The closest type it has is the `None` type. You must be aware of this fact when working with Python in QGIS. In this recipe, we'll explore the implications of QGIS's NULL values in Python. The computing of a NULL value involves a pointer that is an uninitialized, undefined, empty, or meaningless value.

Getting ready

In your `qgis_data/shapes` directory, download the shapefile from <https://geospatialpython.googlecode.com/svn/NullExample.zip>, which contains some NULL field values, and unzip it.

How to do it...

We will load the shapefile and grab its first feature. Then, we'll access one of its NULL field values. Next, we'll run through some tests that allow you to see how the NULL values behave in Python. To do this, we need to perform the following steps:

1. First, we'll load the shapefile and access its first feature:

```
lyrPth = "/qgis_data/shapes/NullExample.shp"
lyr = QgsVectorLayer(lyrPth, "Null Field Example", "ogr")
features = lyr.getFeatures()
f = features.next()
```

2. Next, we'll grab one of the NULL field values:

```
value = f["SAMPLE"]
```

3. Now, we'll check the NULL value's type:

```
print "Check python value type:"
print type(value)
```

4. Then, we'll see whether the value is the Python `None` type:

```
print "Check if value is None:"
print value is None
```

5. Now, we'll see whether it is equivalent to `None`:

```
print "Check if value == None:"
print value == None
```

6. Next, we'll see whether the value matches the QGIS NULL type:

```
print "Check if value == NULL:"  
print value == NULL
```

7. Then, we'll see whether it is actually NULL:

```
print "Check if value is NULL:"  
print value is NULL
```

8. Finally, we'll do a type match to the QGIS NULL:

```
print "Check type(value) is type(NULL) :"  
print type(value) is type(NULL)
```

How it works...

As you can see, the type of the NULL value is `PyQt4.QtCore.QPyNullVariant`. This class is a special type injected into the PyQt framework. It is important to note the cases where the comparison using the `is` operator returns a different value than the `==` operator comparison. You should be aware of the differences to avoid unexpected results in your code.

Using generators for layer queries

Python generators provide an efficient way to process large datasets. A QGIS developer named Nathan Woodrow has created a simple Python QGIS query engine that uses generators to easily fetch features from QGIS layers. We'll use this engine in this recipe to query a layer.

Getting ready

You need to install the query engine using `easy_install` or by downloading it and adding it to your QGIS Python installation. To use `easy_install`, run the following command from a console, which downloads a clone of the original code that includes a Python setup file:

```
easy_install  
https://github.com/GeospatialPython/qquery/archive/master.zip
```

You can also download the ZIP file from <https://github.com/NathanW2/qquery/archive/master.zip> and copy the contents to your working directory or the `site-packages` directory of your QGIS Python installation.

You will also need to download the zipped shapefile and decompress it to a directory named `ms` in your `qgis_data` directory from the following location:

```
https://geospatialpython.googlecode.com/files/MS\_UrbanAnC10.zip
```

How to do it...

We'll load a layer containing population data. Then, we'll use the query engine to perform a simple query for an urban area with less than 50,000 people. We'll filter the results to only give us three columns, place name, population level, and land area. To do this, we need to perform the following steps:

1. First, we import the query engine module:

```
from query import query
```

2. Then, we set up the path to our shapefile and load it as a vector layer:

```
pth = "/Users/joellawhead/qgis_data/ms/MS_UrbanAnC10.shp"
layer = QgsVectorLayer(pth, "Urban Areas", "ogr")
```

3. Now, we can run the query, which uses Python's dot notation to perform a `where` clause search and then filter using a `select` statement. This line will return a generator with the result:

```
q = (query(layer).where("POP > 50000").select('NAME10', "POP",
    "AREALAND", "POPDEN"))
```

4. Finally, we'll use the query's generator to iterate to the first result:

```
q().next()
```

How it works...

As you can see, this module is quite handy. To perform this same query using the default PyQGIS API, it would take nearly four times as much code.

Using alpha values to show data density

Thematic maps often use a color ramp based on a single color to show data density. Darker colors show a higher concentration of objects, while lighter colors show lower concentrations. You can use a transparency ramp instead of a color ramp to show density as well. This technique is useful if you want to overlay the density layer on imagery or other vector layers. In this recipe, we'll be using some bear-sighting data to show the concentration of bears over an area. We'll use alpha values to show the density. We'll use an unusual hexagonal grid to divide the area and a rule-based renderer to build the display.

Getting ready

You will need to install the MMQGIS plugin, which is used to build the hexagonal grid using the QGIS **Plugin Manager**.

You also need to download the bear data from <https://geospatialpython.googlecode.com/svn/bear-data.zip>, unzip the shapefile, and put it in the `ms` directory of your `qgis_data` directory.

How to do it...

We will load the bear data. Then, we will use the MMQGIS plugin to generate the hexagonal grid. Then, we'll use the Processing Toolbox to clip the hexagon to the bear shapefile, and join the shapefile attribute data to the hexagon grid. Finally, we'll use a rule-based renderer to apply alpha values based on bear-sighting density and add the result to the map. To do this, we need to perform the following steps:

1. First, we import all the libraries we'll need, including the processing engine, the PyQt GUI library for color management, and the MMQGIS plugin:

```
import processing
from PyQt4.QtGui import *
from mmqgis import mmqgis_library as mmqgis
```

2. Next, we'll set up the paths for all of our input and output shapefiles:

```
dir = "/qgis_data/ms/"
source = dir + "bear-data.shp"
grid = dir + "grid.shp"
clipped_grid = dir + "clipped_grid.shp"
output = dir + "ms-bear-sightings.shp"
```

3. Now, we can set up the input shapefile as a layer:

```
layer = QgsVectorLayer(source, "bear data", "ogr")
```

4. We'll need the extent of the shapefile to create the grid as well as the width and height, in map units:

```
e = layer.extent()
llx = e.xMinimum()
lly = e.yMinimum()
w = e.width()
h = e.height()
```

5. Now, we can use the MMQGIS plugin to generate the grid over the entire shapefile's extent. We'll use a grid cell size of one-tenth of a degree (approximately 6 miles):

```
mmqgis.mmqgis_grid(iface, grid, .1, .1, w, h, llx, lly, "Hexagon  
(polygon)", False)
```

6. Then, we can clip the grid to the shape of our source data using the Processing Toolbox:

```
processing.runalg("qgis:clip",grid,source,clipped_grid)
```

7. Next, we need to do a spatial join in order to match the source data's attributes based on counties to each grid cell:

```
processing.runalg("qgis:joinbylocation",source,clipped_grid,0,  
"sum,mean,min,max,median",0,0,output)
```

8. Now, we can add this output as a layer:

```
bears = QgsVectorLayer(output, "Bear Sightings", "ogr")
```

9. Next, we create our rendering rule set as Python tuples, specifying a label, value expression, color, and alpha level for the symbols between 0 and 1:

```
rules = (  
    ('RARE', '"BEARS" < 5', (227,26,28,255), .2),  
    ('UNCOMMON', '"BEARS" > 5 AND "BEARS" < 15', (227,26,28,255),  
.4),  
    ('OCCASIONAL', '"BEARS" > 14 AND "BEARS" < 50',  
(227,26,28,255), .6),  
    ('FREQUENT', '"BEARS" > 50', (227,26,28,255), 1),  
)
```

10. We then create the default symbol rule renderer and add the rules to the renderer:

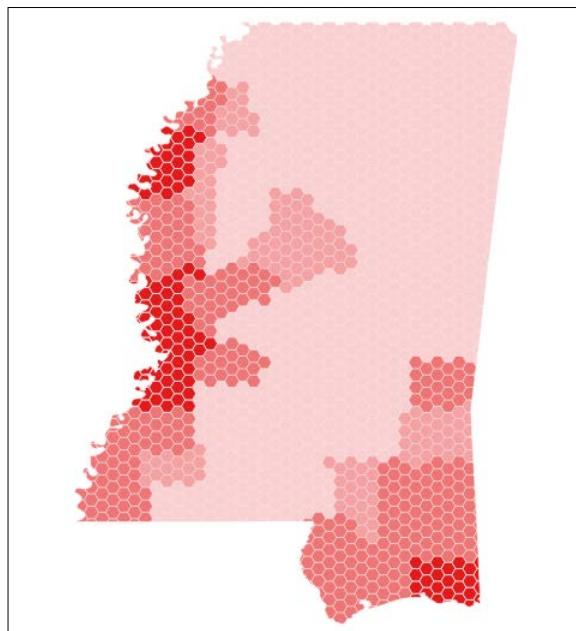
```
sym_bears = QgsFillSymbolV2.createSimple({"outline_  
color":"white","outline_width":.26})  
rend_bears = QgsRuleBasedRendererV2(sym_bears)  
root_rule = rend_bears.rootRule()  
for label, exp, color, alpha in rules:  
    # create a clone (i.e. a copy) of the default rule  
    rule = root_rule.children()[0].clone()  
    # set the label, exp and color  
    rule.setLabel(label)  
    rule.setFilterExpression(exp)  
    r,g,b,a = color  
    rule.symbol().setColor(QColor(r,g,b,a))
```

```
# set the transparency level  
rule.symbol().setAlpha(alpha)  
# append the rule to the list of rules  
root_rule.appendChild(rule)  
  
11. We remove the default rule:  
root_rule.removeChildAt(0)  
  
12. We apply the renderer to the layer:  
bears.setRendererV2(rend_bears)  
  
13. Finally, we add the finished density layer to the map:  
QgsMapLayerRegistry.instance().addMapLayer(bears)
```

How it works...

The rule-based renderer forms the core of this recipe. However, the hexagonal grid provides a more interesting way to visualize statistical data. Like a dot-based density map, hexagons are not entirely spatially accurate or precise but make it very easy to understand the overall trend of the data. The interesting feature of hexagons is their centroid, which is equidistant to each of their neighbors, whereas with a square grid, the diagonal neighbors are further away.

This image shows how the resulting map will look:



Using the `__geo_interface__` protocol

The `__geo_interface__` protocol is a new protocol created by Sean Gillies and is targeted mainly at Python to provide a string representation of geographic data following Python's built-in protocols. The string representation for geographic data is basically GeoJSON.



You can read more about this protocol at <https://gist.github.com/sgillies/2217756>.



Two developers, Nathan Woodrow and Martin Laloux, refined a version of this protocol for QGIS Python data objects. This recipe borrows from their examples to provide a code snippet that you can put at the beginning of your Python scripts to retrofit QGIS features and geometry objects with a `__geo_interface__` method.

Getting ready

This recipe requires no preparation.

How to do it...

We will create two functions: one for features and one for geometry. We'll then use Python's dynamic capability to patch the QGIS objects with a `__geo_interface__` built-in method. To do this, we need to perform the following steps:

1. First, we'll need the Python `json` module:

```
import json
```

2. Next, we'll create our function for the features that take a feature as input and return a GeoJSON-like object:

```
def mapping_feature(feature):
    geom = feature.geometry()
    properties = {}
    fields = [field.name() for field in feature.fields()]
    properties = dict(zip(fields, feature.attributes()))
    return { 'type' : 'Feature',
            'properties' : properties,
            'geometry' : geom.__geo_interface__}
```

3. Now, we'll create the `geometry` function:

```
def mapping_geometry(geometry):  
    geo = geometry.exportToGeoJSON()  
    return json.loads(geo)
```

4. Finally, we'll patch the QGIS feature and geometry objects with our custom built-in to call our functions when the built-in is accessed:

```
QgsFeature.__geo_interface__ = property(lambda self:  
    mapping_feature(self))  
  
QgsGeometry.__geo_interface__ = property(lambda self:  
    mapping_geometry(self))
```

How it works...

This recipe is surprisingly simple but exploits some of Python's most interesting features. First, note that the `feature` function actually calls the `geometry` function as part of its output. Also, note that adding the `__geo_interface__` built-in function is as simple as using the double-underscore naming convention and Python's built-in `property` method to declare lambda functions as internal to the objects. Another interesting Python feature is that the QGIS objects are able to pass themselves to our custom functions using the `self` keyword.

Generating points along a line

You can generate points within a polygon in a fairly simple way by using the `pointInPolygon` method. However, sometimes you may want to generate points along a line. You can randomly place points inside the polygon's extent — which is essentially just a rectangular polygon — or you can place points at random locations along the line at random distances. In this recipe, we'll demonstrate both of these methods.

Getting ready

You will need to download the zipped shapefile and place it in a directory named `shapes` in your `qgis_data` directory from the following:

<https://geospatialpython.googlecode.com/svn/path.zip>

How to do it...

First, we will generate random points along a line using a `grass()` function in the Processing Toolbox. Then, we'll generate points within the line's extent using a native QGIS processing function. To do this, we need to perform the following steps:

1. First, we need to import the processing module:

```
import processing
```

2. Then, we'll load the line layer onto the map:

```
line = QgsVectorLayer("/qgis_data/shapes/path.shp", "Line",
"ogr")
QgsMapLayerRegistry.instance().addMapLayer(line)
```

3. Next, we'll generate points along the line by specifying the path to the shapefile, a maximum distance between the points in map units (meters), the type of feature we want to output (vertices), extent, snap tolerance option, minimum distance between the points, output type, and output name. We won't specify the name and tell QGIS to load the output automatically:

```
processing.runandload("grass:v.to.points",line,"1000",False,
False,True,"435727.015026,458285.819185,5566442.32879,5591754.
78979",-1,0.0001,0,None)
```

4. Finally, we'll create some points within the lines' extent and load them as well:

```
processing.runandload("qgis:randompointsinextent","435727.0150
26,458285.819185,5566442.32879,5591754.78979",100,100,None)
```

How it works...

The first algorithm puts the points on the line. The second places them within the vicinity. Both approaches have different use cases.

There's more...

Another option will be to create a buffer around the line at a specified distance and clip the output of the second algorithm so that the points aren't near the corners of the line extent. The `QgsGeometry` class also has an `interpolate` which allows you to create a point on a line at a specified distance from its origin. This is documented at <http://qgis.org/api/classQgsGeometry.html#a8c3bb1b01d941219f2321e6c6c3db7e1>.

Using expression-based labels

Expressions are a kind of mini-programming language or SQL-like language found throughout different QGIS functions to select features. One important use of expressions is to control labels. Maps easily become cluttered if you label every single feature. Expressions make it easy to limit labels to important features. You can filter labels using expressions from within Python, as we will do in this recipe.

Getting ready

You will need to download the zipped shapefile and decompress it to a directory named `ms` in your `qgis_data` directory from the following:

https://geospatialpython.googlecode.com/files/MS_UrbanAnC10.zip

How to do it...

We'll use the QGIS PAL labeling engine to filter labels based on a field name. After loading the layer, we'll create our PAL settings and write them to the layer. Finally, we'll add the layer to the map. To do this, we need to perform the following steps:

1. First, we'll set up the path to our shapefile:

```
pth = "/Users/joellawhead/qgis_data/ms/MS_UrbanAnC10.shp"
```

2. Next, we'll load our layer:

```
lyr = QgsVectorLayer(pth, "Urban Areas", "ogr")
```

3. Now, we create a labeling object and read the layer's current labeling settings:

```
palyr = QgsPalLayerSettings()
palyr.readFromLayer(lyr)
```

4. We create our expression to only label the features whose population field is greater than 50,000:

```
palyr.fieldName = 'CASE WHEN "POP" > 50000 THEN NAME10 END'
```

5. Then, we enable these settings:

```
palyr.enabled = True
```

6. Finally, we apply the labeling filter to the layer and add it to the map:

```
palyr.writeToLayer(lyr)
QgsMapLayerRegistry.instance().addMapLayer(lyr)
```

How it works...

While labels are a function of the layer, the settings for the labeling engine are controlled by an external object and then applied to the layer.

Creating dynamic forms in QGIS

When you edit the fields of a layer in QGIS, you have the option of using a spreadsheet-like table view or you can use a database-style form view. Forms are useful because you can change the design of the form and add interactive features that react to user input in order to better control data editing. In this recipe, we'll add some custom validation to a form that checks user input for valid values.

Getting ready

You will need to download the zipped shapefile and decompress it to a directory named `ms` in your `qgis_data` directory from the following:

https://geospatialpython.googlecode.com/files/MS_UrbanAnC10.zip

You'll also need to create a blank Python file called `validate.py`, which you'll edit as shown in the following steps. Put the `validate.py` file in the `ms` directory of your `qgis_data` directory with the shapefile.

How to do it...

We'll create the two functions we need for our validation engine. Then, we'll use the QGIS interface to attach the action to the layer. Make sure that you add the following code to the `validate.py` file in the same directory as the shapefile, as follows:

1. First, we'll import the Qt libraries:

```
from PyQt4.QtCore import *
from PyQt4.QtGui import *
```

2. Next, we'll create some global variables for the attribute we'll be validating and the form dialog:

```
popFld = None
dynamicDialog = None
```

3. Now, we'll begin building the function that changes the behavior of the dialog and create variables for the field we want to validate and the submit button:

```
def dynamicForm(dialog, lyrId, featId):  
    globaldynamicDialog  
    dynamicDialog = dialog  
    globalpopFld = dialog.findChild(QLineEdit, "POP")  
    buttonBox=\  
        dialog.findChild(QDialogButtonBox, "buttonBox")
```

4. We must disconnect the dialog from the action that controls the form acceptance:

```
buttonBox.accepted.disconnect(dynamicDialog.accept)
```

5. Next, we reconnect the dialogs, actions to our custom actions:

```
buttonBox.accepted.connect(validate)  
buttonBox.rejected.connect(dynamicDialog.reject)
```

6. Now, we'll create the validation function that will reject the form if the population field has a value less than 1:

```
def validate():  
    if not float(popFld.text()) > 0:  
        msg = QMessageBox(f)  
        msg.setText("Population must be \  
greater than zero.")  
        msg.exec_()  
    else:  
        dynamicDialog.accept()
```

7. Next, open QGIS and drag and drop the shapefile from your filesystem onto the map canvas.

8. Save the project and give it a name in the same directory as the validate.py file.

9. In the QGIS legend, double-click on the layer name.

10. Select the **Fields** tab on the left-hand side of the **Layer Properties** dialog.

11. In the **Fields** tab at the top-right of the screen, enter the following line into the **PythonInit Function** field:

```
validate.dynamicForm
```

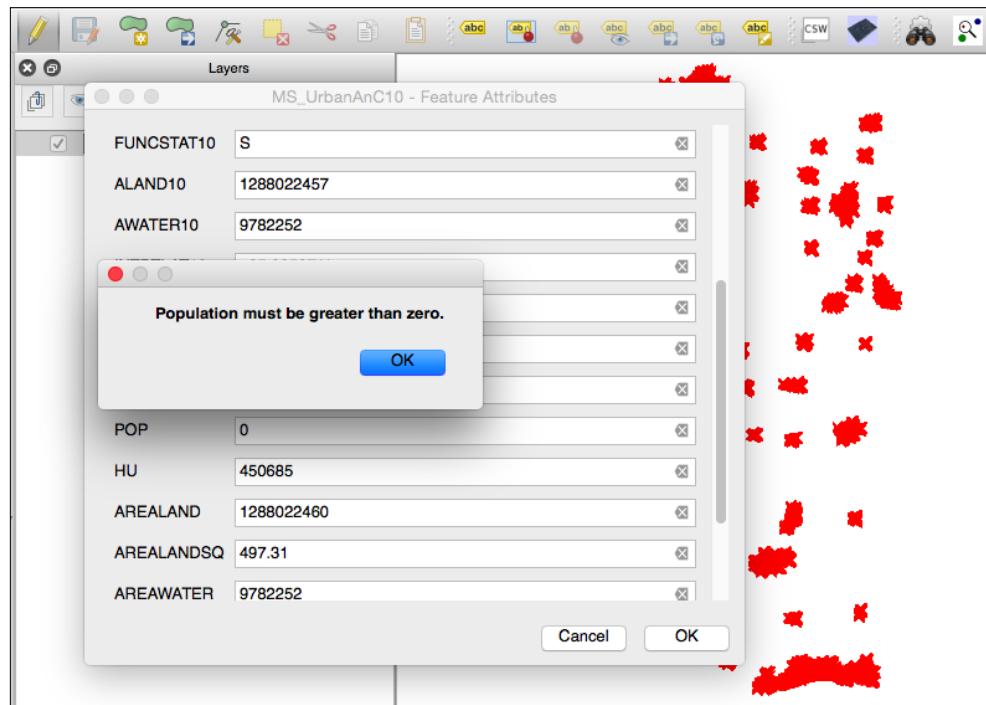
12. Click on the **OK** button, in the bottom-right of the **Layer Properties** dialog.

13. Now, use the identify tool to select a feature.
14. In the **Feature Properties** dialog, click on the form icon in the top-left of the image.
15. Once the feature form is open, switch back to the **QGIS Legend**, right-click on the layer name, and select **Toggle Editing**.
16. Switch back to the feature form, scroll down to the **POP** field, and change the value to 0.
17. Now, click on the **OK** button and verify that you've received the warning dialog, which requires the value to be greater than 0.

How it works...

The validate.py file must be in your Python path. Putting this file in the same directory as the project makes the functions available. Validation is one of the simplest functions you can implement.

This screenshot shows the rejection message when the population is set to 0:



Calculating length for all selected lines

If you need to calculate the total of a given dataset property, such as length, the easiest thing to do is use Python. In this recipe, we'll total the length of the railways in a dataset.

Getting ready

You will need to download a zipped shapefile from https://geospatialpython.googlecode.com/svn/ms_rails_mstm.zip.

Unzip it and place it in directory named `ms` in your `qgis_data` directory.

How to do it...

We will load the layer, loop through the features while keeping a running total of line lengths, and finally convert the result to kilometers. To do this, we need to perform the following steps:

1. First, we'll set up the path to our shapefile:

```
pth = "/Users/joellawhead/qgis_data/ms/ms_rails_mstm.shp"
```

2. Then, we'll load the layer:

```
lyr = QgsVectorLayer(pth, "Railroads", "ogr")
```

3. Next, we need a variable to total the line lengths:

```
total = 0
```

4. Now, we loop through the layer, getting the length of each line:

```
for f in lyr.getFeatures():
    geom = f.geometry()
    total += geom.length()
```

5. Finally, we print the total length converted to kilometers and format the string to only show two decimal places:

```
print "%0.2f total kilometers of rails." % (total / 1000)
```

How it works...

This function is simple, but it's not directly available in the QGIS API. You can use a similar technique to total up the area of a set of polygons or perform conditional counting.

Using a different status bar CRS than the map

Sometimes, you may want to display a different coordinate system for the mouse coordinates in the status bar than what the source data is. With this recipe, you can set a different coordinate system without changing the data coordinate reference system or the CRS for the map.

Getting ready

Download the zipped shapefile and unzip it to your `qgis_data/ms` directory from the following:

https://geospatialpython.googlecode.com/files/MSCities_Geo.zip

How to do it...

We will load our layer, establish a message in the status bar, create a special event listener to transform the map coordinates at the mouse's location to our alternate CRS, and then connect the map signal for the mouse's map coordinates to our listener function. To do this, we need to perform the following steps:

1. First, we need to import the Qt core library:

```
from PyQt4.QtCore import *
```

2. Then, we will set up the path to the shapefile and load it as a layer:

```
pth = "/qgis_data/ms/MSCities_Geo_Pts.shp"
lyr = QgsVectorLayer(pth, "Cities", "ogr")
```

3. Now, we add the layer to the map:

```
QgsMapLayerRegistry.instance().addMapLayer(lyr)
```

4. Next, we create a default message that will be displayed in the status bar and will be replaced by the alternate coordinates later, when the event listener is active:

```
msg = "Alternate CRS ( x: %s, y: %s )"
```

5. Then, we display our default message in the left-hand side of the status bar as a placeholder:

```
i iface mainWindow().statusBar().showMessage(msg % ("--", "--"))
```

6. Now, we create our custom event-listener function to transform the mouse's map location to our custom CRS, which in this case is **EPSG 3815**:

```
def listen_xyCoordinates(point):
    crsSrc = iface.mapCanvas().mapRenderer().destinationCrs()
    crsDest = QgsCoordinateReferenceSystem(3815)
    xform = QgsCoordinateTransform(crsSrc, crsDest)
    xpoint = xform.transform(point)
    iface.mainWindow().statusBar().showMessage(msg %
(xpoint.x(), xpoint.y()))
```

7. Next, we connect the map canvas signal that is emitted when the mouse coordinates are updated to our custom event listener:

```
QObject.connect(iface.mapCanvas(), SIGNAL("xyCoordinates(const
QgsPoint &)"), listen_xyCoordinates)
```

8. Finally, verify that when you move the mouse around the map, the status bar is updated with the transformed coordinates.

How it works...

The coordinate transformation engine in QGIS is very fast. Normally, QGIS tries to transform everything to WGS84 Geographic, but sometimes you need to view coordinates in a different reference system.

Creating HTML labels in QGIS

QGIS map tips allow you to hover the mouse cursor over a feature in order to create a popup that displays information. This information is normally a data field, but you can also display other types of information using a subset of HTML tags. In this recipe, we'll create an HTML map tip that displays a Google Street View image at the feature's location.

Getting ready

In your `qgis_data` directory, create a directory named `tmp`.

You will also need to download the following zipped shapefile and place it in your `qgis_data/nyc` directory:

https://geospatialpython.googlecode.com/files/NYC_MUSEUMS_GEO.zip

How to do it...

We will create a function to process the Google data and register it as a QGIS function. Then, we'll load the layer and set its map tip display field. To do this, we need to perform the following steps:

1. First, we need to import the Python libraries we'll need:

```
from qgis.utils import QgsFunction  
from qgis.core import Qgs  
import urllib
```

2. Next, we'll set a special QGIS Python decorator that registers our function as a QGIS function. The first argument, 0, means that the function won't accept any arguments itself. The second argument, Python, defines the group in which the function will appear when you use the expression builder:

```
@qgsfunction(0, "Python")
```

3. We'll create a function that accepts a feature and uses its geometry to pull down a Google Street View image. We must cache the images locally because the Qt widget that displays the map tips only allows you to use local images:

```
def googleStreetView(values, feature, parent):  
    x,y = feature.geometry().asPoint()  
    baseurl = "https://maps.googleapis.com/maps/api/streetview?"  
    w = 400  
    h = 400  
    fov = 90  
    heading = 235  
    pitch = 10  
    params = "size=%sx%s&" % (w,h)  
    params += "location=%s,%s&" % (y,x)  
    params += "fov=%s&heading=%s&pitch=%s" % (fov, heading, pitch)  
    url = baseurl + params  
    tmpdir = "/qgis_data/tmp/"  
    img = tmpdir + str(feature.id()) + ".jpg"  
    urllib.urlretrieve(url, img)  
    return img
```

4. Now, we can load the layer:

```
pth = "/qgis_data/nyc/nyc_museums_geo.shp"
lyr = QgsVectorLayer(pth, "New York City Museums", "ogr")
```

5. Next, we can set the display field using a special QGIS tag with the name of our function:

```
lyr.setDisplayField('')
```

6. Finally, we add it to the map:

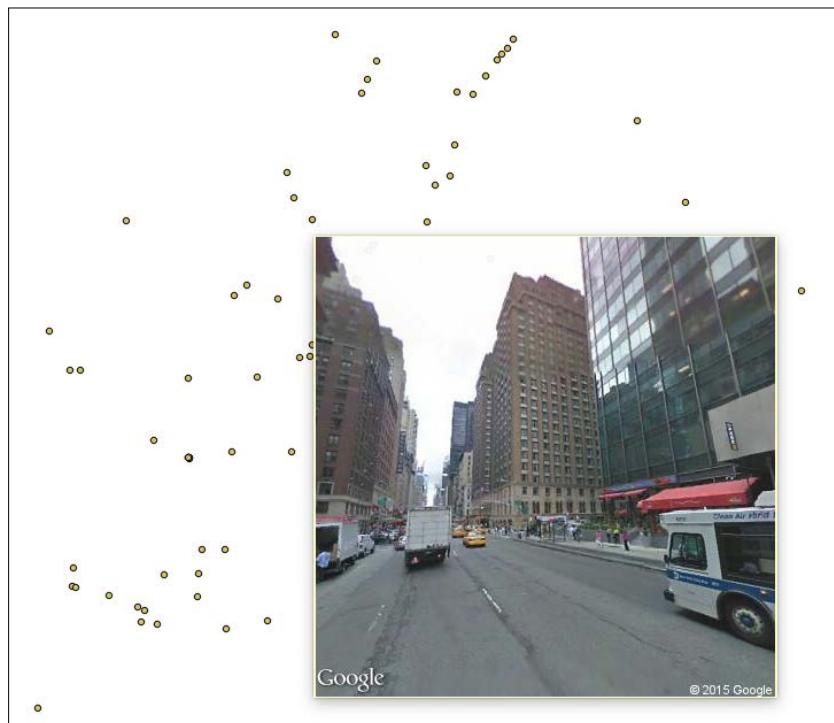
```
QgsMapLayerRegistry.instance().addMapLayer(lyr)
```

7. Select the map tips tool and hover over the different points to see the Google Street View images.

How it works...

The key to this recipe is the `@qgsfunction` decorator. When you register the function in this way, it shows up in the menus for Python functions in expressions. The function must also have the parent and value parameters, but we didn't need them in this case.

The following screenshot shows a Google Street View map tip:



There's more...

If you don't need the function any more, you must unregister it for the function to go away. The `unregister` command uses the following convention, referencing the function name with a dollar sign:

```
QgsExpression.unregisterFunction("$googleStreetView")
```

Using OpenStreetMap's points of interest in QGIS

OpenStreetMap has an API called Overpass that lets you access OSM data dynamically. In this recipe, we'll add some OSM tourism points of interest to a map.

Getting ready

You will need to use the QGIS **Plugin Manager** to install the **Quick OSM** plugin.

You will also need to download the following shapefile and unzip it to your `qgis_data/ms` directory:

```
https://geospatialpython.googlecode.com/svn/MSCoast\_geo.zip
```

How to do it...

We will load our base layer that defines the area of interest. Then, we'll use the Processing Toolbox to build a query for OSM, download the data, and add it to the map. To do this, we need to perform the following steps:

1. First, we need to import the processing module:

```
import processing
```

2. Next, we need to load the base layer:

```
lyr = QgsVectorLayer("/qgis_data/ms/MSCoast_geo.shp", "MS  
Coast", "ogr")
```

3. Then, we'll need the layer's extents for the processing algorithms:

```
ext = lyr.extent()  
w = ext.xMinimum()  
s = ext.yMinimum()  
e = ext.xMaximum()  
n = ext.yMaximum()
```

4. Next, we create the query:

```
factory = processing.runalg("quickosm:queryfactory", \
"tourism","","%s,%s,%s,%s" % (w,e,s,n),"",25)
q = factory["OUTPUT_QUERY"]
```

5. The Quick OSM algorithm has a bug in its output, so we'll create a properly formatted XML tag and perform a string replace:

```
bbox_query = """<bbox-query e="%s" n="%s" s="%s" \ w="%s"/>"""
% (e,n,s,w)
bad_xml = """<bbox-query %s,%s,%s,%s/>""" % (w,e,s,n)
good_query = q.replace(bad_xml, bbox_query)
```

6. Now, we download the OSM data using our query:

```
results = processing.runalg("quickosm:queryoverpassapiwithastrings",
,"htt
p://overpass-api.de/api/",good_query,"0,0,0,0","",None)
osm = results["OUTPUT_FILE"]
```

7. We define the names of the shapefiles we will create from the OSM output:

```
poly = "/qgis_data/ms/tourism_poly.shp"
multiline = "/qgis_data/ms/tourism_multil.shp"
line = "/qgis_data/ms/tourism_lines.shp"
points = "/qgis_data/ms/tourism_points.shp"
```

8. Now, we convert the OSM data to shapefiles:

```
processing.runalg("quickosm:ogrdefault",osm,"","",","",poly,m
ultiline,line,points)
```

9. We place the points as a layer:

```
tourism_points = QgsVectorLayer(points, "Points of Interest",
"ogr")
```

10. Finally, we can add them to a map:

```
QgsMapLayerRegistry.instance().addMapLayers([tourism_points,
lyr])
```

How it works...

The Quick OSM plugin manages the Overpass API. What's interesting about this plugin is that it provides processing algorithms in addition to a GUI interface. The processing algorithm that creates the query unfortunately formats the bbox-query tag improperly, so we need to work around this issue with the string replace. The API returns an OSM XML file that we must convert to shapefiles for use in QGIS.

Visualizing data in 3D with WebGL

QGIS displays data in a two-dimensions even if the data is three-dimensional. However, most modern browsers can display 3D data using the WebGL standard. In this recipe, we'll use the **Qgis2threejs** plugin to display QGIS data in 3D in a browser.

Getting ready

You will need to download some raster elevation data in the zipped directory and place it in your `qgis_data` directory from the following:

<https://geospatialpython.googlecode.com/svn/saveqml.zip>

You will also need to install the **Qgis2threejs** plugin using the QGIS **Plugin Manager**.

How to do it...

We will set up a color ramp for a DEM draped over a hillshade image and use the plugin to create a WebGL page in order to display the data. To do this, we need to perform the following steps:

1. First, we will need to import the relevant libraries and the **Qgis2threejs** plugin:

```
from PyQt4.QtCore import *
from PyQt4.QtGui import *
import Qgis2threejs as q23js
```

2. Next, we'll disable QGIS automatic reprojection to keep the data display in meters:

```
iface.mapCanvas().setCrsTransformEnabled(False)
iface.mapCanvas().setMapUnits(0)
```

3. Now, we can load our raster layers:

```
demPth = "/Users/joellawhead/qgis_data/saveqml/dem.asc"
hillshadePth =
"/Users/joellawhead/qgis_data/saveqml/hillshade.tif"
dem = QgsRasterLayer(demPth, "DEM")
hillshade = QgsRasterLayer(hillshadePth, "Hillshade")
```

4. Then, we can create the color ramp renderer for the DEM layer:

```
algorithm = QgsContrastEnhancement.StretchToMinimumMaximum
limits = QgsRaster.ContrastEnhancementMinMax
dem.setContrastEnhancement(algorithm, limits)
s = QgsRasterShader()
c = QgsColorRampShader()
c.setColorRampType(QgsColorRampShader.INTERPOLATED)
i = []
qri = QgsColorRampShader.ColorRampItem
i.append(qri(356.334, QColor(63,159,152,255), '356.334'))
i.append(qri(649.292, QColor(96,235,155,255), '649.292'))
i.append(qri(942.25, QColor(100,246,174,255), '942.25'))
i.append(qri(1235.21, QColor(248,251,155,255), '1235.21'))
i.append(qri(1528.17, QColor(246,190,39,255), '1528.17'))
i.append(qri(1821.13, QColor(242,155,39,255), '1821.13'))
i.append(qri(2114.08, QColor(165,84,26,255), '2114.08'))
i.append(qri(2300, QColor(236,119,83,255), '2300'))
i.append(qri(2700, QColor(203,203,203,255), '2700'))
c.setColorRampItemList(i)
s.setRasterShaderFunction(c)
ps = QgsSingleBandPseudoColorRenderer(dem.dataProvider(), 1, s)
ps.setOpacity(0.5)
dem.setRenderer(ps)
```

5. Now, we're ready to add the raster layers to the map:

```
QgsMapLayerRegistry.instance().addMapLayers([dem, hillshade])
```

6. To create the WebGL interface, we need to take control of the plugin's GUI dialog, but we will keep it hidden:

```
d = q23js.qgis2threejsdialog.Qgis2threejsDialog(iface)
```

7. Next, we must create a dictionary of the properties required by the plugin. The most important is the layer ID of the DEM layer:

```
props = [None,
         None,
         {u'spinBox_Roughening': 4,
```

```
u'checkBox_Surroundings': False,
u'horizontalSlider_Resolution': 2,
u'lineEdit_Color': u'',
'visible': False,
'dem_Height': 163,
u'checkBox_Frame': False,
u'lineEdit_ImageFile': u'',
u'spinBox_Size': 5,
u'spinBox_sidetransp': 0,
u'lineEdit_xmax': u'',
u'radioButton_MapCanvas': True,
'dem_Width': 173,
u'radioButton_Simple': True,
u'lineEdit_xmin': u'',
u'checkBox_Sides': True,
u'comboBox_DEMLayer': dem.id(),
u'spinBox_demtransp': 0,
u'checkBox_Shading': False,
u'lineEdit_ymax': u'',
u'lineEdit_ymin': u'',
u'spinBox_Height': {5}, {}, {}, {}, {}]
```

8. Now, we will apply these properties to the plugin:

```
d.properties = props
```

9. We must set the output file for the HTML page:

```
d.ui.lineEdit_OutputFilename.setText('/qgis_data/3D/3d.html')
```

10. In the next step, we must override the method that saves the properties, otherwise it overwrites the properties we set:

```
def sp(a,b):
    return
d.saveProperties = sp
```

11. Now, we are ready to run the plugin:

```
d.run()
```

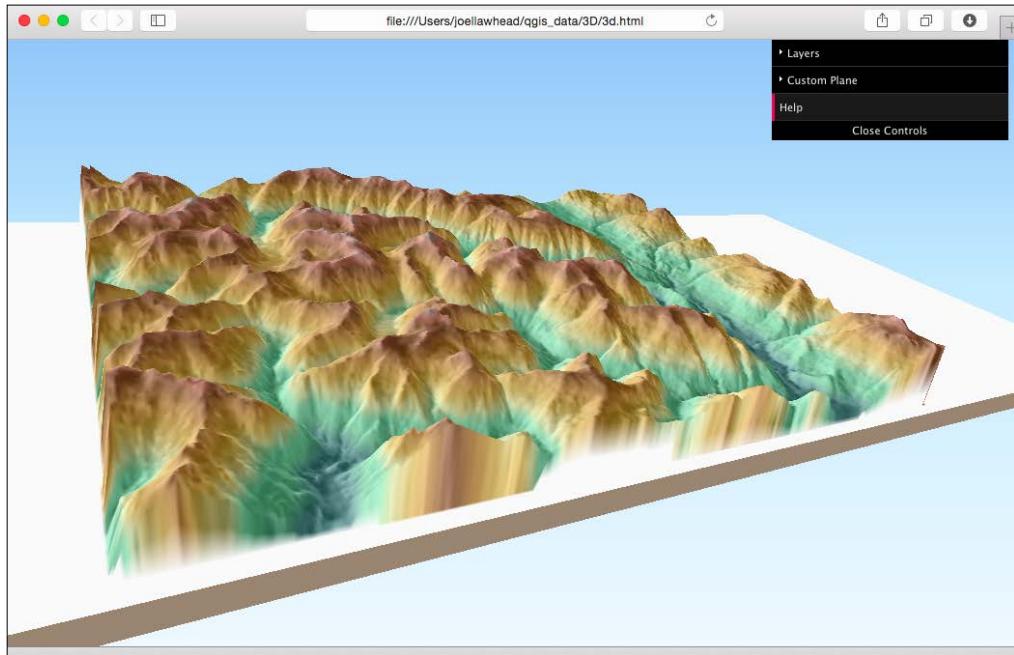
12. On your filesystem, navigate to the HTML output page and open it in a browser.

13. Follow the help instructions to move the 3D elevation display around.

How it works...

This plugin is absolutely not designed for script-level access. However, Python is so flexible that we can even script the plugin at the GUI level and avoid displaying the GUI, so it is seamless to the user. The only glitch in this approach is that the save method overwrites the properties we set, so we must insert a dummy function that prevents this overwrite.

The following image shows the WebGL viewer in action:



Visualizing data on a globe

Ever since the release of Google Earth, *spinning globe* applications have become a useful and popular method of geographic exploration. QGIS has an experimental plugin called **QGIS Globe**, which is similar to Google Earth; however, it is extremely unstable. In this recipe, we'll display a layer in Google Earth.

Getting ready

You will need to use the QGIS **Plugin Manager** to install the **MMQGIS** plugin.

Make sure you have Google Earth installed from <https://www.google.com/earth/>.

You will also need the following dataset from a previous recipe. It is a zipped directory called `ufo` which you should uncompress to your `qgis_data` directory:

<https://geospatialpython.googlecode.com/svn/ufo.zip>

How to do it...

We will load our layer and set up the attribute we want to use for the Google Earth KML output as the descriptor. We'll use the MMQIGS plugin to output our layer to KML. Finally, we'll use a cross-platform technique to open the file, which will trigger it to open in Google Earth. To do this, we need to perform the following steps:

1. First, we will import the relevant Python libraries including the plugin. We will use the Python `webbrowser` module to launch Google Earth:

```
from mmqgis import mmqgis_library as mmqgis
import webbrowser
import os
```

2. Now, we'll load the layer:

```
pth = "/Users/joellawhead/qgis_data/continental-us"
lyrName = "continental-us"
lyr = QgsVectorLayer(pth, lyrName, "ogr")
```

3. Next, we'll set the output path for the KML:

```
output = "/Users/joellawhead/qgis_data/us.kml"
```

4. Then, we'll set up the variables needed by the plugin for the KML output which make up the layer identifier:

```
nameAttr = "FIPS_CNTRY"
desc = ["CNTRY_NAME",]
sep = "Paragraph"
```

5. Now, we can use the plugin to create the KML:

```
mmqgis.mmqgis_kml_export(iface, lyrName, nameAttr, desc, \
sep, output, False)
```

6. Finally, we'll use the `webbrowser` module to open the KML file, which will default to opening in Google Earth. We need to add the `file` protocol at the beginning of our output for the `webbrowser` module to work:

```
webbrowser.open("file://" + output)
```

How it works...

The MMQGIS plugin does a good job with custom scripts and has easy-to-use functions. While our method for automatically launching Google Earth may not work in every possible case, it is almost perfect.

Index

Symbol

__geo_interface__ protocol

about 288
using 288, 289

A

addresses

geocoding 227-229

alpha values

used, for displaying data density 284-287

application parameter 28

area of polygon

calculating 47, 48

atlas 242

attributes, vector layer

examining 37
feature, modifying 68, 69

autocomplete 12

B

bearing of end points, line

calculating 50, 51

binary large objects (BLOBs) 86

binary packages, for Linux

URL 106

Bing aerial image service

using 155, 156

buffer() method 44

C

categorized vector layer symbol, dynamic maps

about 142
creating 142-144

cell size, raster

querying 89

changeGeometryValues() method 68

checkboxes

about 214
creating 214, 215

color ramp 135

combobox

about 211
creating 211, 212

common extent

creating, for raster 106-108

complex vector layer symbol, dynamic maps

creating 137-139

Coordinate Reference System (CRS) 5

CSV File

shapefile attribute table, joining to 65-67

custom selection tool, dynamic maps

building 168-170

custom shape

adding, to map 189-192

D

data

loading, from spreadsheet 51-53
visualizing in 3D, with WebGL 302-305
visualizing, on globe 305, 306

- data density**
displaying, alpha values used 284-287
- dataset, raster**
sampling, regular grid used 100-103
- data value, raster**
querying, at specified point 93, 94
- Debian package manager**
used, for installing PyQGIS 2
- delimitedtext 53**
- different status bar CRS**
using 296
- distance**
measuring, along sample line 45-48
measuring, between two points 44, 45
- dot density map**
about 253
creating 253-255
- downsampling 108**
- dynamic forms**
creating, in QGIS 292, 293
- dynamic maps**
Bing aerial image service, using 155, 156
canvas, accessing 130, 131
categorized vector layer symbol,
 creating 142, 143
complex vector layer symbol,
 creating 137-139
creating 130
custom selection tool, building 168-170
graduated vector layer symbol renderer,
 creating 141, 142
icons, using as vector layer symbols 139, 140
label features 158, 159
layers, iterating over 132, 133
map bookmark, creating 144-146
map bookmark, navigating to 146, 147
map layer transparency, modifying 159
map tool, used for drawing points on
 canvas 163-165
map tool, used for drawing polygons or lines
 on canvas 165-167
mouse coordinate tracking tool,
 creating 171, 172
- OpenStreetMap service, using 154, 155
pie charts, using for symbols 150-153
real-time weather data, adding from
 OpenWeatherMap 157
scale-based visibility, setting for
 layer 147, 148
single band raster, rendering with color
 ramp algorithm 135, 136
standard map tools, adding to
 canvas 160-163
SVG, using for layer symbols 148, 149
units, changing 131, 132
vector layer, symbolizing 133, 134
- E**
- editing buffer 59**
- elevation data**
adding, to line vertices with DEM 104, 105
used, for computing road slope 258-262
- elevation hillshade**
creating 96, 97
- EmittingPoint 167**
- environment variables**
setting, on Linux 3
setting, on Windows 3
- error dialog**
creating 205, 206
- expression-based labels**
using 291, 292
- F**
- false color image**
URL 92
- features, vector layer**
examining 36, 37
- field**
adding, to vector layer 63, 64
- field data**
collecting 255-258
- file input dialog**
creating 209, 210

G

gdalDEM documentation

URL 97

generators

using, for layer queries 283, 284

geocoding

about 227

addresses 228

geojson.io

layer, adding to 274, 275

Geospatial Data Abstraction

Library (GDAL) 154

github.io web directory

URL 273

global preferences

reading 27, 28

storing 27

GoogleEarth 117

GPS

tracking 238-241

graduated vector layer symbol renderer, dynamic maps

about 141

creating 141, 142

Graphical Modeler tool 232

grid

adding, to static map 193-195

ground control points (GCP) 124

H

hancock

URL 174

heat map

about 249

creating 249-251

HelloWorld plugin 16, 17

hillshade 96

HTML labels

creating, in QGIS 297-300

I

icons, dynamic maps

using, as vector layer symbols 139, 140

image change detection

about 266

performing 266-268

Integrated Development Environment (IDE) 1

J

JPEG image

TIFF image, converting to 112

K

kernel density estimation algorithm 252

K-means clustering 121

KML

about 120

shapefile, converting to 73

KML image overlay, for raster

creating 117-120

L

labels

adding, to static map 179-181

labeling features, dynamic maps

exploring 158, 159

layer

filtering, by attributes 40, 41

filtering, by geometry 38-40

layer style file

creating 280, 281

least cost path (LCP)

about 245

searching 245-247

legend

adding, to static map 188, 189

length, for selected lines

calculating 295

line feature

adding, to vector layer 59, 60

Linux

environment variables, setting 3

log files

about 202

using 202, 203

logo

adding, to static map 186, 187

M

map
photos, geolocating on 262-266
mapbook
about 242
creating 242-245
map bookmark, dynamic maps
about 144
creating 145, 146
navigating to 146, 147
map canvas, dynamic maps
accessing 130, 131
map composer
using 176-179
map coordinates, raster
converting, to pixel location 116, 117
map layers
iterating over 132, 133
rendering, based on rules 276-279
transparency, modifying 159, 160
URL 132
map tool, dynamic maps
used, for drawing points on canvas 163-165
used, for drawing polygons or lines on
canvas 165-167
map units, dynamic maps
changing 131, 132
mbtiles format 273
memory
vector layer, creating in 56, 57
merge processing algorithm 112
message dialog
creating 203, 204
creating, hint used 221, 222
**mouse coordinate tracking tool,
dynamic maps**
creating 171

N

nearest neighbor analysis
about 247
performing 247, 248
network analysis
about 233
performing 233-236

network analysis tool

URL 236
**Normalized Difference Vegetation
Index (NDVI)**
creating 224-226
north arrow
about 183
adding, to static map 183-186
NULL values
using, in PyQGIS 282, 283

O

Open Geospatial Consortium (OGC) 73, 117
Open GIS Consortium 157
OpenStreetMap service
points of interest, using in QGIS 300, 301
using 154, 155
Orfeo Toolbox 120
organization parameter 28
overlapping images
URL 107

P

Photo2Shape 266
photos
geolocating, on map 262-265
pie charts, dynamic maps
using, for symbols 150-153
pixel locations, raster
converting, to map coordinates 114-116
Plugin Reloader 13
point feature
adding, to vector layer 57-59
point in point feature
buffering 42-44
point layer
URL 38
points
generating, along time 289, 290
polygon feature
adding, to vector layer 60, 61
polygonize 122
PostGIS layer
loading, into QGIS map 34

Processing Toolbox
about 56
using 230

progress bar
about 206
displaying 207, 208

project
static map, loading from 200
static map, saving 199

project preferences
reading 28, 29
storing 28, 29

py2exe
URL 27

PyInstaller
URL 27

PyQGIS
about 59
NULL values, using 282

PyQGIS 2.6 API
URL 18

PyQGIS API
adding, to QGIS IDE 12
categories 18
Core module 18
navigating 17-19
URL 18

PyQGIS path
searching, on Windows 3

pyramids, raster
creating 113, 114

Python Imaging Library (PIL) 263

Python-ordered dictionary
used, for building string 57

Q

QGIS
debugging, URL 17
environment variables, setting up 3
installing, for system development 2, 3
installing, with Debian package manager 2
installing, with RPM package manager 2

QGIS Cloud 34

QGIS Globe
about 305
data, visualizing on 305, 306

QGIS IDE
environment variables, adding 13
PyQGIS API, adding 12
PyQGIS module paths, adding 10, 11
QGIS Python interpreter, adding
on Windows 8, 9
setting up 8
URL 8
working 13

QGIS map
tiles, creating from 270-273

QGIS Map Composer 176

QGIS Markup Language (QML) 280

qgis:mergevectorlayers module 75

QGIS plugin
creating 19-22
distributing 22-24
URL 22

QGIS Python
location, searching on other platforms 4

QGIS Python console
using, for interactive control 5

QGIS Python interpreter
adding, on Windows 8-10
PyQGIS module paths, adding to 10, 11

QGIS Python ScriptRunner plugin
using 6
working 7

QGIS Python scripts
debugger, testing 16
debugging 13, 14
Eclipse, configuring 15
working 17

QGIS raster calculator
controlling 224
working 227

QGIS tutorial
URL, for map projections 72

QgsComposition object 179

QgsDistanceArea.convertMeasurement()
method 45

QgsMultiBandColorRenderer 93

QgsRasterLayer object 88

QgsVectorLayer object

about 56
arguments 57

Qt

about 202
URL, for documentation 202

Qt Creator

about 22
URL 22

R**radio buttons**

about 212
creating 212, 213

raster

classifying 121, 122
clipping, shapefile used 126, 127
common extent, creating 106-108
converting, to vector 122-124
dataset, sampling with regular grid 100-103
data value, querying at specified point 93, 94
georeferencing, from control points 124, 125
KML image overlay, creating 117-120
mosaicing 111
pyramids, creating 113, 114
reprojecting 94, 95
resolution, resampling 108, 109
unique values, counting 110, 111

raster bands

counting 91
swapping 92

raster footprints

creating 229-232

raster layer

cell size, querying 89
loading 87, 88
width and height, obtaining 90

real-time weather data, dynamic maps

adding, from OpenWeatherMap 157

Red Hat Package Manager (RPM) 2**Remote Debug plugin 13****road slope**

computing, elevation data used 258-262

RPM package manager

used, for installing PyQGIS 2

S**SatImage raster**

URL 89

Scalable Vector Graphics (SVG)

about 148
using, for dynamic map layer
symbols 148, 149

scale bar

about 181
adding, to static map 181, 182

scale-based visibility, dynamic maps

setting, for layer 147, 148

script path

accessing, from within QGIS script 30

semi-automatic classification

URL 122

set of attributes

adding, to vector layer 62, 63

shapefile

converting, to KML 73
merging 74, 75
splitting 75, 76
used, for clipping raster 126, 127

shapefile attribute table

joining, to CSV File 65-67

simplest map renderer

creating 174, 175

single band raster, dynamic maps

rendering, with color ramp
algorithm 135, 136

spatial database

vector layer, loading from 34, 35

spatial index

creating 48, 49

splash plugin 29**spreadsheet**

data, loading from 51-53

standalone application

creating 25-27

standard map tools, dynamic maps

adding, to canvas 160-163

static map

custom shape, adding 189-192

grid, adding 193, 194

labels, adding 179-181

legend, adding 188, 189
loading, from project 200
logo, adding 186, 187
north arrow, adding 183-186
saving, to project 199
scale bar, adding 181, 182
table, adding 195, 196
world file, adding 197, 198

street routing

about 236
performing 237, 238

string

building, Python-ordered dictionary used 57

T

table

adding, to map 195-197

tabs

about 216
creating 217, 218

textboxes 216

text input dialog

creating 208, 209

three-page wizard

creating 218-221

TIFF image

converting, to JPEG image 112

tiles

creating, from QGIS map 270-273

traditional log file 203

U

union

performing, on vector shapes 80, 81

unique values, raster

counting 110, 111

V

vector

raster, converting to 122-124

vector contours

creating, from elevation data 98, 99

vector layer

attribute, deleting 71

attribute of feature, modifying 68, 69

attributes, examining 37, 38
creating, in memory 56, 57
feature, deleting 70
features, examining 36
field, adding to 63, 64
generalizing 76-78
geometry, moving 67, 68
line feature, adding to 59, 60
loading, from file sample 32, 33
loading, from spatial database 34, 35
point feature, adding to 57, 59
polygon feature, adding to 60, 61
rasterizing 82
reprojecting 72
set of attributes, adding to 62, 63
symbolizing 133, 134

vector shapes

dissolving 78-80
union, performing on 80, 81

virtualenv tool 13

W

warning dialog

creating 204, 205

WebGL

used, for visualizing data in 3D 302-304

Web Map Service (WMS) 154

Well-Known Text (WKT) 27

Windows

environment variables, setting 3

PyQGIS path, searching 3

QGIS Python interpreter, adding 8-10

wizard 218

world file

about 197, 198

adding, to map image 197, 198

Z

zip tool

URL 23



Thank you for buying **QGIS Python Programming Cookbook**

About Packt Publishing

Packt, pronounced 'packed', published its first book, *Mastering phpMyAdmin for Effective MySQL Management*, in April 2004, and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern yet unique publishing company that focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website at www.packtpub.com.

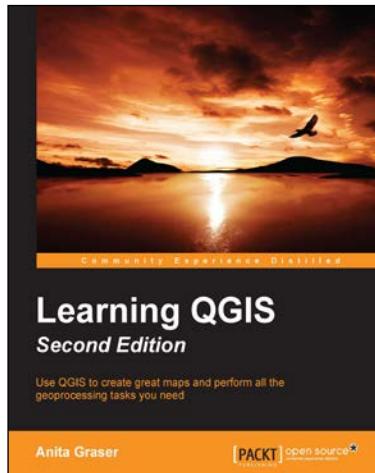
About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt open source brand, home to books published on software built around open source licenses, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's open source Royalty Scheme, by which Packt gives a royalty to each open source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, then please contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.



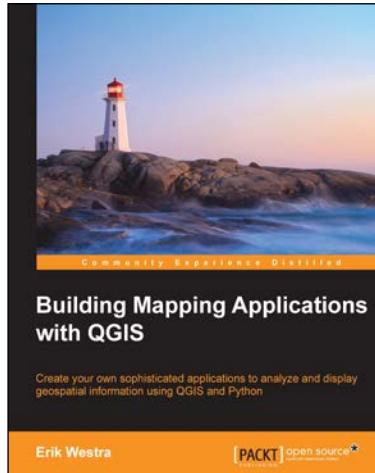
Learning QGIS

Second Edition

ISBN: 978-1-78439-203-1 Paperback: 150 pages

Use QGIS to create great maps and perform all the geoprocessing tasks you need

1. Load, visualize, and edit vector and raster data.
2. Create professional maps and applications to present geospatial data.
3. A concise guide, packed with detailed real-world examples to get you started with QGIS.



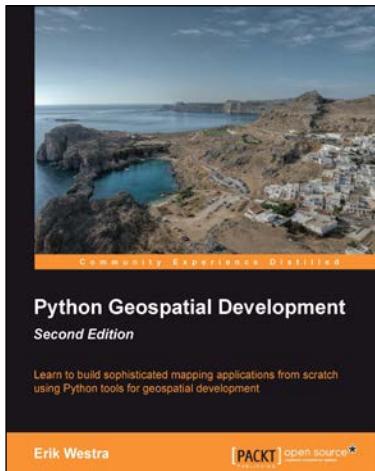
Building Mapping Applications with QGIS

ISBN: 978-1-78398-466-4 Paperback: 264 pages

Create your own sophisticated applications to analyze and display geospatial information using QGIS and Python

1. Make use of the geospatial capabilities of QGIS within your Python programs.
2. Build complete standalone mapping applications based on QGIS and Python.
3. Use QGIS as a Python geospatial development environment.

Please check www.PacktPub.com for information on our titles



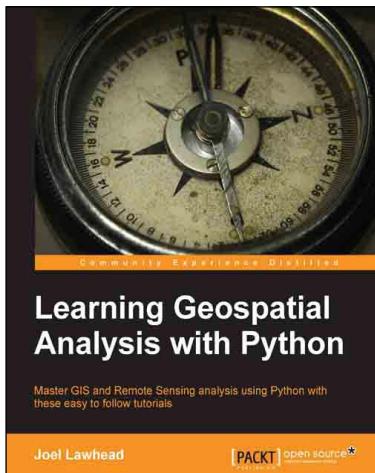
Python Geospatial Development

Second Edition

ISBN: 978-1-78216-152-3 Paperback: 508 pages

Learn to build sophisticated mapping applications from scratch using Python tools for geospatial development

1. Build your own complete and sophisticated mapping applications in Python.
2. Walks you through the process of building your own online system for viewing and editing geospatial data
3. Practical, hands-on tutorial that teaches you all about geospatial development in Python



Learning Geospatial Analysis with Python

ISBN: 978-1-78328-113-8 Paperback: 364 pages

Master GIS and Remote Sensing analysis using Python with these easy to follow tutorials

1. Construct applications for GIS development by exploiting Python
2. Focuses on built-in Python modules and libraries compatible with the Python Packaging Index distribution system – no compiling of C libraries necessary
3. This is a practical, hands-on tutorial that teaches you all about Geospatial analysis in Python

Please check www.PacktPub.com for information on our titles