QGIS Python Programming Cookbook

QGIS Python Programming will teach you how to write Python code that works with spatial data to automate geoprocessing tasks in QGIS. It will cover topics such as querying and editing vector data and using raster data. You will also learn to create, edit, and optimize a vector layer for faster queries, reproject a vector layer, reduce the number of vertices in a vector layer without losing critical data, and convert a raster to a vector. Following this, you will come across recipes that will help you to compose static maps, create heavily customized maps, and add specialized labels and annotations. Apart from this, the book will also share a few tips and tricks based on different aspects of QGIS.

What this book will do for you...

- Build a library of reusable scripts with ScriptRunner
- Create, import, and edit geospatial data on disk or in memory
- Get to know more about dynamic mapping
- Create and add features to static maps
- Create a mapbook
- Reproject a vector layer
- Geolocate photos on a map
- Combine multiple rasters into one image

Inside the Cookbook...

- A straightforward and easy-to-follow format
- A selection of the most important tasks and problems
- Carefully organized instructions for solving the problem efficiently
- Clear explanations of what you did
- Apply the solution to other situations

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

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 9 'Other Tips and Tricks'
- A synopsis of the book’s content
- More information on QGIS Python Programming Cookbook

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.
QGIS Python Programming Cookbook

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.
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:
   ```python
   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:

   ```python
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:

   ```python
def done():
    print "FINISHED!!"
```

4. Now, we set the path to the shapefile and the tile's output direction:

   ```python
shp = "/qgis_data/shapes/countries.shp"
dir = "/qgis_data/tilecache"
```

5. Then, we load the shapefile:

   ```python
layer = QgsVectorLayer(shp, "Countries", "ogr")
```

6. After that, we define the field that is used to color the countries:

   ```python
field = 'CNTRY_NAME'
```

7. Now, we need to get all the features so that we can loop through them:

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

8. We'll build our color renderer:

   ```python
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:

```python
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:

```python
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:

```python
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:

```python
tt.processFinished.connect(done)
```

13. Finally, we start the tiling process:

```python
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](http://geospatialpython.github.io) web directory at [http://geospatialpython.github.io/qgis/tiles/countries.html](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:
   ```python
   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:
   ```python
   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:
   ```python
   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:
   
   ```python
   crs = QgsCoordinateReferenceSystem(4326,
   QgsCoordinateReferenceSystem.PostgisCrsId)
   ```

5. Next, we can write out the layer as GeoJSON:
   
   ```python
   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:
   
   ```python
   if error != QgsVectorFileWriter.NoError:
       print "Unable to write geoJSON!"
   ```

7. Now, we can read the GeoJSON content:
   
   ```python
   with open(str(tmpfile), 'r') as f:
       contents = f.read()
   ```

8. We then need to remove the temporary file:
   
   ```python
   os.remove(tmpfile)
   ```

9. We are ready to upload our GeoJSON to geojson.io using the `qgisio` module:
   
   ```python
   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:

    ```python
    QDesktopServices.openUrl(QUrl(url))
    ```

**How it works...**

This recipe actually uses two cloud services. The GeoJSON data is stored on a [https://github.com](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 qgis_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:
   ```python
   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:
   ```python
   prefix = "~/Users/joellawhead/qgis_data/mst/"
   ```

3. Now, we can load our railroad layer:
   ```python
   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:
   ```python
   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:
   ```python
   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:
   ```python
   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:

```python
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:

```python
root_rule.removeChildAt(0)
```

9. Now, we apply the renderer to our rails layer:

```python
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:

```python
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:

```python
ms = QgsVectorLayer(prefix + "mississippi.shp", "Mississippi", "ogr")
ms_style = {}...
```
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:
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:
   ```python
   from PyQt4.QtCore import *
   from PyQt4.QtGui import *
   ```

2. Next, we'll load our two raster layers:
   ```python
   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:
   ```python
   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:
   ```python
   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:
   
   ```python
   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:
   
   ```python
   value = f["SAMPLE"]
   ```

3. Now, we'll check the NULL value's type:
   
   ```python
   print "Check python value type:"
   print type(value)
   ```

4. Then, we'll see whether the value is the Python None type:
   
   ```python
   print "Check if value is None:"
   print value is None
   ```

5. Now, we'll see whether it is equivalent to None:

   ```python
   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:
   ```python
   from query import query
   ```

2. Then, we set up the path to our shapefile and load it as a vector layer:
   ```python
   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:
   ```python
   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:
   ```python
   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 PyQt4 GUI library for color management, and the MMQGIS plugin:
   
   ```python
   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:

   ```python
   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:

   ```python
   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:

   ```python
   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](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:
   ```python
   import json
   ```

2. Next, we'll create our function for the features that take a feature as input and return a GeoJSON-like object:
   ```python
   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:

```python
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:

```python
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 point in polygon 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:

   ```python
   import processing
   ```

2. Then, we'll load the line layer onto the map:

   ```python
   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:

   ```python
   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:

   ```python
   processing.runandload("qgis:randompointsinextent", "435727.015026,458285.819185,5566442.32879,5591754.78979", 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](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)
   ```
Other Tips and Tricks

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:
   ```python
   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:
   ```python
   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:

```python
def dynamicForm(dialog, lyrId, featId):
    global dynamicDialog
    dynamicDialog = dialog
    global popFld = dialog.findChild(QLineEdit, "POP")
    buttonBox =
    dialog.findChild(QDialogButtonBox, "buttonBox")
```

4. We must disconnect the dialog from the action that controls the form acceptance:

```python
buttonBox.accepted.disconnect(dynamicDialog.accept)
```

5. Next, we reconnect the dialogs, actions to our custom actions:

```python
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:

```python
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:

```python
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:
   ```python
   pth = "/Users/joellawhead/qgis_data/ms/ms_rails_mstm.shp"
   ```
2. Then, we'll load the layer:
   ```python
   lyr = QgsVectorLayer(pth, "Railroads", "ogr")
   ```
3. Next, we need a variable to total the line lengths:
   ```python
   total = 0
   ```
4. Now, we loop through the layer, getting the length of each line:
   ```python
   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:
   ```python
   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:
   ```python
   from PyQt4.QtCore import *
   ```

2. Then, we will set up the path to the shapefile and load it as a layer:
   ```python
   pth = "/qgis_data/ms/MSCities_Geo_Pts.shp"
   lyr = QgsVectorLayer(pth, "Cities", "ogr")
   ```

3. Now, we add the layer to the map:
   ```python
   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:
   ```python
   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:
   ```python
   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**:

   ```python
   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:

   ```python
   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:
   ```python
   from qgis.utils import qgsfunction
   from qgis.core import QGis
   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:
   ```python
   @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:
   ```python
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:

```python
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:

```python
lyr.setDisplayField('<img src="[% $googleStreetView %]\"/>')
```

6. Finally, we add it to the map:

```python
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:
Other Tips and Tricks

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:

```python
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:
   ```python
   import processing
   ```

2. Next, we need to load the base layer:
   ```python
   lyr = QgsVectorLayer("/qgis_data/ms/MSCoast_geo.shp", "MS Coast", "ogr")
   ```

3. Then, we'll need the layer's extents for the processing algorithms:
   ```python
   ext = lyr.extent()
   w = ext.xMinimum()
   s = ext.yMinimum()
   e = ext.xMaximum()
   n = ext.yMaximum()
   ```
Next, we create the query:

```python
factory = processing.runalg("quickosm:queryfactory", 
"tourism","","%s,%s,%s,%s" % (w,e,s,n),"",25)
q = factory["OUTPUT_QUERY"]
```

The Quick OSM algorithm has a bug in its output, so we'll create a properly formatted XML tag and perform a string replace:

```python
bbox_query = """"<bbox-query e="%s" n="%s" s="%s" \ w="%s"/>"""
% (e,n,s,w)
bbox_query = """"<bbox-query %s,%s,%s,%s/>"""
% (w,e,s,n)
good_query = q.replace(bad_xml, bbox_query)
```

Now, we download the OSM data using our query:

```python
results = processing.runalg("quickosm:queryoverpassapiwithastring"
,\"http://overpass-api.de/api\\","good_query,\"0,0,0,0\\","",None)
osm = results["OUTPUT_FILE"]
```

We define the names of the shapefiles we will create from the OSM output:

```python
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"
```

Now, we convert the OSM data to shapefiles:

```python
processing.runalg("quickosm:ogrdefault",osm,"","","","",poly,multiline,line,points)
```

We place the points as a layer:

```python
tourism_points = QgsVectorLayer(points, "Points of Interest", "ogr")
```

Finally, we can add them to a map:

```python
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:
   ```python
   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:
   ```python
   iface.mapCanvas().setCrsTransformEnabled(False)
   iface.mapCanvas().setMapUnits(0)
   ```

3. Now, we can load our raster layers:
   ```python
   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:

```python
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.setColorRampFunction(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:

```python
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:

```python
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:

```python
props = [None,
        None,
        {u'spinBox_Roughening': 4,
```
8. Now, we will apply these properties to the plugin:

```python
    d.properties = props
```

9. We must set the output file for the HTML page:

```python
    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:

```python
    def sp(a,b):
        return
    
    d.saveProperties = sp
```

11. Now, we are ready to run the plugin:

```python
    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:
   ```python
   from mmqgis import mmqgis_library as mmqgis
   import webbrowser
   import os
   ```

2. Now, we'll load the layer:
   ```python
   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:
   ```python
   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:
   ```python
   nameAttr = "FIPS_CNTRY"
   desc = ["COUNTRY_NAME"],
   sep = "Paragraph"
   ```

5. Now, we can use the plugin to create the KML:
   ```python
   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:
   ```python
   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.
Where to buy this book

You can buy QGIS Python Programming Cookbook from the Packt Publishing website. Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers. Click here for ordering and shipping details.