PyQGIS developer cookbook

Release 2.0

QGIS Project

March 17, 2014

Contents

CHAPTER 1

Introduction

This document is intended to work both as a tutorial and a reference guide. While it does not list all possible use cases, it should give a good overview of the principal functionality.

Starting from 0.9 release, QGIS has optional scripting support using Python language. We've decided for Python as it's one of the most favourite languages for scripting. PyQGIS bindings depend on SIP and PyQt4. The reason for using SIP instead of more widely used SWIG is that the whole QGIS code depends on Qt libraries. Python bindings for Qt (PyQt) are done also using SIP and this allows seamless integration of PyQGIS with PyQt.

TODO: Getting PyQGIS to work (Manual compilation, Troubleshooting)

There are several ways how to use QGIS python bindings, they are covered in detail in the following sections:

- issue commands in Python console within QGIS
- create and use plugins in Python
- create custom applications based on QGIS API

There is a complete QGIS API reference that documents the classes from the QGIS libraries. Pythonic QGIS API is nearly identical to the API in C++.

There are some resources about programming with PyQGIS on QGIS blog. See QGIS tutorial ported to Python for some examples of simple 3rd party apps. A good resource when dealing with plugins is to download some plugins from plugin repository and examine their code. Also, the `python/plugins/` folder in your QGIS installation contains some plugin that you can use to learn how to develop such plugin and how to perform some of the most common tasks

1.1 Python Console

For scripting, it is possible to take advantage of integrated Python console. It can be opened from menu: *Plugins → Python Console*. The console opens as a non-modal utility window:

The screenshot above illustrates how to get the layer currently selected in the layer list, show its ID and optionally, if it is a vector layer, show the feature count. For interaction with QGIS environment, there is a `iface` variable, which is an instance of `QgsInterface`. This interface allows access to the map canvas, menus, toolbars and other parts of the QGIS application.

For convenience of the user, the following statements are executed when the console is started (in future it will be possible to set further initial commands):

```
from qgis.core import *
import qgis.utils
```

For those which use the console often, it may be useful to set a shortcut for triggering the console (within menu *Settings → Configure shortcuts...*)

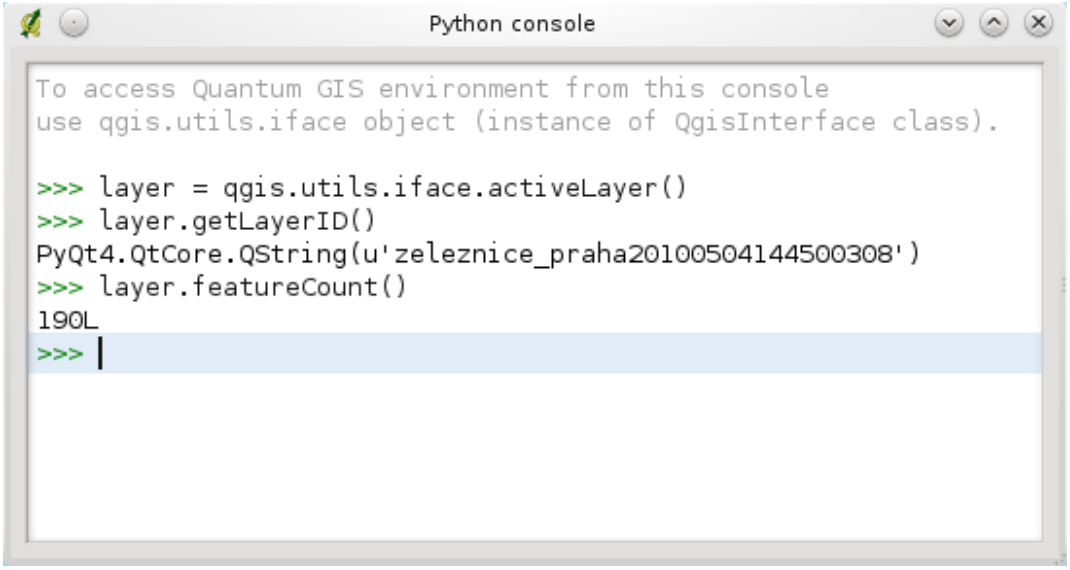

Figure 1.1: QGIS Python console

1.2 Python Plugins

QGIS allows enhancement of its functionality using plugins. This was originally possible only with C++ language. With the addition of Python support to QGIS, it is also possible to use plugins written in Python. The main advantage over C++ plugins is its simplicity of distribution (no compiling for each platform needed) and easier development.

Many plugins covering various functionality have been written since the introduction of Python support. The plugin installer allows users to easily fetch, upgrade and remove Python plugins. See the Python Plugin Repositories page for various sources of plugins.

Creating plugins in Python is simple, see *Developing Python Plugins* for detailed instructions.

1.3 Python Applications

Often when processing some GIS data, it is handy to create some scripts for automating the process instead of doing the same task again and again. With PyQGIS, this is perfectly possible — import the `qgis.core` module, initialize it and you are ready for the processing.

Or you may want to create an interactive application that uses some GIS functionality — measure some data, export a map in PDF or any other functionality. The `qgis.gui` module additionally brings various GUI components, most notably the map canvas widget that can be very easily incorporated into the application with support for zooming, panning and/or any further custom map tools.

1.3.1 Using PyQGIS in custom application

Note: do *not* use `qgis.py` as a name for your test script — Python will not be able to import the bindings as the script's name will shadow them.

First of all you have to import qgis module, set QGIS path where to search for resources — database of projections, providers etc. When you set prefix path with second argument set as `True`, QGIS will initialize all paths with standard dir under the prefix directory. Calling `initQgis()` function is important to let QGIS search for the available providers.

```
from qgis.core import *
```

```
# supply path to where is your qgis installed
QgsApplication.setPrefixPath("/path/to/qgis/installation", True)

# load providers
QgsApplication.initQgis()
```

Now you can work with QGIS API — load layers and do some processing or fire up a GUI with a map canvas. The possibilities are endless :-)

When you are done with using QGIS library, call `exitQgis()` to make sure that everything is cleaned up (e.g. clear map layer registry and delete layers):

```
QgsApplication.exitQgis()
```

1.3.2 Running Custom Applications

You will need to tell your system where to search for QGIS libraries and appropriate Python modules if they are not in a well-known location — otherwise Python will complain:

```
>>> import qgis.core
ImportError: No module named qgis.core
```

This can be fixed by setting the `PYTHONPATH` environment variable. In the following commands, `qgispath` should be replaced with your actual QGIS installation path:

- on Linux: **export PYTHONPATH=/qgispath/share/qgis/python**
- on Windows: **set PYTHONPATH=c:\qgispath\python**

The path to the PyQGIS modules is now known, however they depend on `qgis_core` and `qgis_gui` libraries (the Python modules serve only as wrappers). Path to these libraries is typically unknown for the operating system, so you get an import error again (the message might vary depending on the system):

```
>>> import qgis.core
ImportError: libqgis_core.so.1.5.0: cannot open shared object file: No such file or directory
```

Fix this by adding the directories where the QGIS libraries reside to search path of the dynamic linker:

- on Linux: **export LD_LIBRARY_PATH=/qgispath/lib**
- on Windows: **set PATH=C:\qgispath;%PATH%**

These commands can be put into a bootstrap script that will take care of the startup. When deploying custom applications using PyQGIS, there are usually two possibilities:

- require user to install QGIS on his platform prior to installing your application. The application installer should look for default locations of QGIS libraries and allow user to set the path if not found. This approach has the advantage of being simpler, however it requires user to do more steps.
- package QGIS together with your application. Releasing the application may be more challenging and the package will be larger, but the user will be saved from the burden of downloading and installing additional pieces of software.

The two deployment models can be mixed - deploy standalone application on Windows and Mac OS X, for Linux leave the installation of QGIS up to user and his package manager.

CHAPTER 2

Loading Layers

Let's open some layers with data. QGIS recognizes vector and raster layers. Additionally, custom layer types are available, but we are not going to discuss them here.

2.1 Vector Layers

To load a vector layer, specify layer's data source identifier, name for the layer and provider's name:

```
layer = QgsVectorLayer(data_source, layer_name, provider_name)
if not layer.isValid():
  print "Layer failed to load!"
```

The data source identifier is a string and it is specific to each vector data provider. Layer's name is used in the layer list widget. It is important to check whether the layer has been loaded successfully. If it was not, an invalid layer instance is returned.

The following list shows how to access various data sources using vector data providers:

- OGR library (shapefiles and many other file formats) — data source is the path to the file

  ```
  vlayer = QgsVectorLayer("/path/to/shapefile/file.shp", \
    "layer_name_you_like", "ogr")
  ```

- PostGIS database — data source is a string with all information needed to create a connection to PostgreSQL database. `QgsDataSourceURI` class can generate this string for you. Note that QGIS has to be compiled with Postgres support, otherwise this provider isn't available.

  ```
  uri = QgsDataSourceURI()
  # set host name, port, database name, username and password
  uri.setConnection("localhost", "5432", "dbname", "johny", "xxx")
  # set database schema, table name, geometry column and optionaly
  # subset (WHERE clause)
  uri.setDataSource("public", "roads", "the_geom", "cityid = 2643")

  vlayer = QgsVectorLayer(uri.uri(), "layer_name_you_like", "postgres")
  ```

- CSV or other delimited text files — to open a file with a semicolon as a delimiter, with field "x" for x-coordinate and field "y" with y-coordinate you would use something like this

  ```
  uri = "/some/path/file.csv?delimiter=%s&xField=%s&yField=%s" % (";", "x", "y")
  vlayer = QgsVectorLayer(uri, "layer_name_you_like", "delimitedtext")
  ```

  Note: from QGIS version 1.7 the provider string is structured as a URL, so the path must be prefixed with *file://*. Also it allows WKT (well known text) formatted geomtries as an alternative to "x" and "y" fields, and allows the coordinate reference system to be specified. For example

```
uri = "file:///some/path/file.csv?delimiter=%s&crs=epsg:4723&wktField=%s" \
  % (";", "shape")
```

- GPX files — the "gpx" data provider reads tracks, routes and waypoints from gpx files. To open a file, the type (track/route/waypoint) needs to be specified as part of the url

```
uri = "path/to/gpx/file.gpx?type=track"
vlayer = QgsVectorLayer(uri, "layer_name_you_like", "gpx")
```

- SpatiaLite database — supported from QGIS v1.1. Similarly to PostGIS databases, `QgsDataSourceURI` can be used for generation of data source identifier

```
uri = QgsDataSourceURI()
uri.setDatabase('/home/martin/test-2.3.sqlite')
schema = ''
table = 'Towns'
geom_column = 'Geometry'
uri.setDataSource(schema, table, geom_colum)

display_name = 'Towns'
vlayer = QgsVectorLayer(uri.uri(), display_name, 'spatialite')
```

- MySQL WKB-based geometries, through OGR — data source is the connection string to the table

```
uri = "MySQL:dbname,host=localhost,port=3306,user=root,password=xxx|\
  layername=my_table"
vlayer = QgsVectorLayer( uri, "my_table", "ogr" )
```

- WFS connection:. the connection is defined with a URI and using the `WFS` provider

```
uri = "http://localhost:8080/geoserver/wfs?srsname=EPSG:23030&typename=\
  union&version=1.0.0&request=GetFeature&service=WFS",
vlayer = QgsVectorLayer("my_wfs_layer", "WFS")
```

  The uri can be created using the standard `urllib` library.

```
params = {
    'service': 'WFS',
    'version': '1.0.0',
    'request': 'GetFeature',
    'typename': 'union',
    'srsname': "EPSG:23030"
}
uri = 'http://localhost:8080/geoserver/wfs?' + \
  urllib.unquote(urllib.urlencode(params))
```

  And you can also use the

2.2 Raster Layers

For accessing raster files, GDAL library is used. It supports a wide range of file formats. In case you have troubles with opening some files, check whether your GDAL has support for the particular format (not all formats are available by default). To load a raster from a file, specify its file name and base name

```
fileName = "/path/to/raster/file.tif"
fileInfo = QFileInfo(fileName)
baseName = fileInfo.baseName()
rlayer = QgsRasterLayer(fileName, baseName)
if not rlayer.isValid():
  print "Layer failed to load!"
```

Raster layers can also be created from a WCS service.

```
layer_name = 'elevation'
uri = QgsDataSourceURI()
uri.setParam ('url', 'http://localhost:8080/geoserver/wcs')
uri.setParam ( "identifier", layer_name)
rlayer = QgsRasterLayer(uri, 'my_wcs_layer', 'wcs')
```

Alternatively you can load a raster layer from WMS server. However currently it's not possible to access GetCapabilities response from API — you have to know what layers you want

```
urlWithParams = 'url=http://wms.jpl.nasa.gov/wms.cgi&layers=global_mosaic&\
  styles=pseudo&format=image/jpeg&crs=EPSG:4326'
rlayer = QgsRasterLayer(urlWithParams, 'some layer name', 'wms')
if not rlayer.isValid():
  print "Layer failed to load!"
```

2.3 Map Layer Registry

If you would like to use the opened layers for rendering, do not forget to add them to map layer registry. The map layer registry takes ownership of layers and they can be later accessed from any part of the application by their unique ID. When the layer is removed from map layer registry, it gets deleted, too.

Adding a layer to the registry:

```
QgsMapLayerRegistry.instance().addMapLayer(layer)
```

Layers are destroyed automatically on exit, however if you want to delete the layer explicitly, use:

```
QgsMapLayerRegistry.instance().removeMapLayer(layer_id)
```

TODO: More about map layer registry?

CHAPTER 3

Using Raster Layers

This sections lists various operations you can do with raster layers.

3.1 Layer Details

A raster layer consists of one or more raster bands - it is referred to as either single band or multi band raster. One band represents a matrix of values. Usual color image (e.g. aerial photo) is a raster consisting of red, blue and green band. Single band layers typically represent either continuous variables (e.g. elevation) or discrete variables (e.g. land use). In some cases, a raster layer comes with a palette and raster values refer to colors stored in the palette.

```
>>> rlayer.width(), rlayer.height()
(812, 301)
>>> rlayer.extent()
u'12.095833,48.552777 : 18.863888,51.056944'
>>> rlayer.rasterType()
2  # 0 = GrayOrUndefined (single band), 1 = Palette (single band), 2 = Multiband
>>> rlayer.bandCount()
3
>>> rlayer.metadata()
u'<p class="glossy">Driver:</p>...'
>>> rlayer.hasPyramids()
False
```

3.2 Drawing Style

When a raster layer is loaded, it gets a default drawing style based on its type. It can be altered either in raster layer properties or programmatically. The following drawing styles exist:

Index	Constant: QgsRasterLater.X	Comment
1	SingleBandGray	Single band image drawn as a range of gray colors
2	SingleBandPseudoColor	Single band image drawn using a pseudocolor algorithm
3	PalettedColor	"Palette" image drawn using color table
4	PalettedSingleBandGray	"Palette" layer drawn in gray scale
5	PalettedSingleBandPseudo-Color	"Palette" layerdrawn using a pseudocolor algorithm
7	MultiBandSingleBandGray	Layer containing 2 or more bands, but a single band drawn as a range of gray colors
8	MultiBandSingle-BandPseudoColor	Layer containing 2 or more bands, but a single band drawn using a pseudocolor algorithm
9	MultiBandColor	Layer containing 2 or more bands, mapped to RGB color space.

To query the current drawing style:

```
>>> rlayer.drawingStyle()
9
```

Single band raster layers can be drawn either in gray colors (low values = black, high values = white) or with a pseudocolor algorithm that assigns colors for values from the single band. Single band rasters with a palette can be additionally drawn using their palette. Multiband layers are typically drawn by mapping the bands to RGB colors. Other possibility is to use just one band for gray or pseudocolor drawing.

The following sections explain how to query and modify the layer drawing style. After doing the changes, you might want to force update of map canvas, see *Refreshing Layers*.

TODO: contrast enhancements, transparency (no data), user defined min/max, band statistics

3.2.1 Single Band Rasters

They are rendered in gray colors by default. To change the drawing style to pseudocolor:

```
>>> rlayer.setDrawingStyle(QgsRasterLayer.SingleBandPseudoColor)
>>> rlayer.setColorShadingAlgorithm(QgsRasterLayer.PseudoColorShader)
```

The `PseudoColorShader` is a basic shader that highlighs low values in blue and high values in red. Another, `FreakOutShader` uses more fancy colors and according to the documentation, it will frighten your granny and make your dogs howl.

There is also `ColorRampShader` which maps the colors as specified by its color map. It has three modes of interpolation of values:

- linear (`INTERPOLATED`): resulting color is linearly interpolated from the color map entries above and below the actual pixel value
- discrete (`DISCRETE`): color is used from the color map entry with equal or higher value
- exact (`EXACT`): color is not interpolated, only the pixels with value equal to color map entries are drawn

To set an interpolated color ramp shader ranging from green to yellow color (for pixel values from 0 to 255):

```
>>> rlayer.setColorShadingAlgorithm(QgsRasterLayer.ColorRampShader)
>>> lst = [ QgsColorRampShader.ColorRampItem(0, QColor(0,255,0)), \
    QgsColorRampShader.ColorRampItem(255, QColor(255,255,0)) ]
>>> fcn = rlayer.rasterShader().rasterShaderFunction()
>>> fcn.setColorRampType(QgsColorRampShader.INTERPOLATED)
>>> fcn.setColorRampItemList(lst)
```

To return back to default gray levels, use:

```
>>> rlayer.setDrawingStyle(QgsRasterLayer.SingleBandGray)
```

3.2.2 Multi Band Rasters

By default, QGIS maps the first three bands to red, green and blue values to create a color image (this is the `MultiBandColor` drawing style. In some cases you might want to override these setting. The following code interchanges red band (1) and green band (2):

```
>>> rlayer.setGreenBandName(rlayer.bandName(1))
>>> rlayer.setRedBandName(rlayer.bandName(2))
```

In case only one band is necessary for visualization of the raster, single band drawing can be chosen — either gray levels or pseudocolor, see previous section:

```
>>> rlayer.setDrawingStyle(QgsRasterLayer.MultiBandSingleBandPseudoColor)
>>> rlayer.setGrayBandName(rlayer.bandName(1))
>>> rlayer.setColorShadingAlgorithm(QgsRasterLayer.PseudoColorShader)
>>> # now set the shader
```

3.3 Refreshing Layers

If you do change layer symbology and would like ensure that the changes are immediately visible to the user, call these methods:

```
if hasattr(layer, "setCacheImage"): layer.setCacheImage(None)
layer.triggerRepaint()
```

The first call will ensure that the cached image of rendered layer is erased in case render caching is turned on. This functionality is available from QGIS 1.4, in previous versions this function does not exist — to make sure that the code works with all versions of QGIS, we first check whether the method exists.

The second call emits signal that will force any map canvas containing the layer to issue a refresh.

With WMS raster layers, these commands do not work. In this case, you have to do it explicitily:

```
layer.dataProvider().reloadData()
layer.triggerRepaint()
```

In case you have changed layer symbology (see sections about raster and vector layers on how to do that), you might want to force QGIS to update the layer symbology in the layer list (legend) widget. This can be done as follows (`iface` is an instance of QgisInterface):

```
iface.legendInterface().refreshLayerSymbology(layer)
```

3.4 Query Values

To do a query on value of bands of raster layer at some specified point:

```
ident = rlayer.dataProvider().identify(QgsPoint(15.30,40.98), \
  QgsRaster.IdentifyFormatValue)
if ident.isValid():
  print ident.results()
```

The `results` method in this case returs a dictionary, with band indices as keys, and band values as values.

```
{1: 17, 2: 220}
```

CHAPTER 4

Using Vector Layers

This section summarizes various actions that can be done with vector layers.

4.1 Iterating over Vector Layer

Iterating over the features in a vector layer is one of the most common tasks. Below is an example of the simple basic code to perform this task and showing some information about each feature. the `layer` variable is assumed to have a QgsVectorLayer object

```
iter = layer.getFeatures()
for feature in iter:
  # retreive every feature with its geometry and attributes
    # fetch geometry
    geom = feature.geometry()
    print "Feature ID %d: " % feature.id()

    # show some information about the feature
    if geom.vectorType() == QGis.Point:
      x = geom.asPoint()
      print "Point: " + str(x)
    elif geom.vectorType() == QGis.Line:
      x = geom.asPolyline()
      print "Line: %d points" % len(x)
    elif geom.vectorType() == QGis.Polygon:
      x = geom.asPolygon()
      numPts = 0
      for ring in x:
        numPts += len(ring)
      print "Polygon: %d rings with %d points" % (len(x), numPts)
    else:
      print "Unknown"

    # fetch attributes
    attrs = feature.attributes()

    # attrs is a list. It contains all the attribute values of this feature
    print attrs
```

Attributes can be refered by index.

```
idx = layer.fieldNameIndex('name')
print feature.attributes()[idx]
```

4.1.1 Iterating over selected features

4.1.2 Convenience methods

For the above cases, and in case you need to consider selection in a vector layer in case it exist, you can use the `getfeatures()` method from the buil-in processing plugin, as follows:

```
import processing
features = processing.getfeatures(layer)
for feature in features:
  #Do whatever you need with the feature
```

This will iterate over all the features in the layer, in case there is no selection, or over the selected features otherwise.

4.1.3 Iterating over a subset of features

If you want to iterate over a given subset of features in a layer, such as those within a given area, you have to add a `QgsFeatureRequest` object to the `getFeatures()` call. Here's an example

```
request=QgsFeatureRequest()
request.setFilterRect(areaOfInterest)
  for f in layer.getFeatures(request):
    ...
```

The request can be used to define the data retrieved for each feature, so the iterator returns all features, but return partial data for each of them.

```
request.setSubsetOfFields([0,2])                    # Only return selected fields
request.setSubsetOfFields(['name','id'],layer.fields())  # More user friendly version
request.setFlags( QgsFeatureRequest.NoGeometry )  # Don't return geometry objects
```

4.2 Modifying Vector Layers

Most vector data providers support editing of layer data. Sometimes they support just a subset of possible editing actions. Use the `capabilities()` function to find out what set of functionality is supported:

```
caps = layer.dataProvider().capabilities()
```

By using any of following methods for vector layer editing, the changes are directly committed to the underlying data store (a file, database etc). In case you would like to do only temporary changes, skip to the next section that explains how to do *modifications with editing buffer*.

4.2.1 Add Features

Create some `QgsFeature` instances and pass a list of them to provider's `addFeatures()` method. It will return two values: result (true/false) and list of added features (their ID is set by the data store):

```
if caps & QgsVectorDataProvider.AddFeatures:
  feat = QgsFeature()
  feat.addAttribute(0,"hello")
  feat.setGeometry(QgsGeometry.fromPoint(QgsPoint(123,456)))
  (res, outFeats) = layer.dataProvider().addFeatures( [ feat ] )
```

4.2.2 Delete Features

To delete some features, just provide a list of their feature IDs:

```
if caps & QgsVectorDataProvider.DeleteFeatures:
  res = layer.dataProvider().deleteFeatures([ 5, 10 ])
```

4.2.3 Modify Features

It is possible to either change feature's geometry or to change some attributes. The following example first changes values of attributes with index 0 and 1, then it changes the feature's geometry:

```
fid = 100   # ID of the feature we will modify

if caps & QgsVectorDataProvider.ChangeAttributeValues:
  attrs = { 0 : "hello", 1 : 123 }
  layer.dataProvider().changeAttributeValues({ fid : attrs })

if caps & QgsVectorDataProvider.ChangeGeometries:
  geom = QgsGeometry.fromPoint(QgsPoint(111,222))
  layer.dataProvider().changeGeometryValues({ fid : geom })
```

4.2.4 Adding and Removing Fields

To add fields (attributes), you need to specify a list of field defnitions. For deletion of fields just provide a list of field indexes.

```
if caps & QgsVectorDataProvider.AddAttributes:
  res = layer.dataProvider().addAttributes( [ QgsField("mytext", \
    QVariant.String), QgsField("myint", QVariant.Int) ] )

if caps & QgsVectorDataProvider.DeleteAttributes:
  res = layer.dataProvider().deleteAttributes( [ 0 ] )
```

After adding or removing fields in the data provider the layer's fields need to be updated because the changes are not automatically propagated.

```
layer.updateFields()
```

4.3 Modifying Vector Layers with an Editing Buffer

When editing vectors within QGIS application, you have to first start editing mode for a particular layer, then do some modifications and finally commit (or rollback) the changes. All the changes you do are not written until you commit them — they stay in layer's in-memory editing buffer. It is possible to use this functionality also programmatically — it is just another method for vector layer editing that complements the direct usage of data providers. Use this option when providing some GUI tools for vector layer editing, since this will allow user to decide whether to commit/rollback and allows the usage of undo/redo. When committing changes, all changes from the editing buffer are saved to data provider.

To find out whether a layer is in editing mode, use `isEditing()` — the editing functions work only when the editing mode is turned on. Usage of editing functions:

```
# add two features (QgsFeature instances)
layer.addFeatures([feat1,feat2])
# delete a feature with specified ID
layer.deleteFeature(fid)

# set new geometry (QgsGeometry instance) for a feature
```

```
layer.changeGeometry(fid, geometry)
# update an attribute with given field index (int) to given value (QVariant)
layer.changeAttributeValue(fid, fieldIndex, value)

# add new field
layer.addAttribute(QgsField("mytext", QVariant.String))
# remove a field
layer.deleteAttribute(fieldIndex)
```

In order to make undo/redo work properly, the above mentioned calls have to be wrapped into undo commands. (If you do not care about undo/redo and want to have the changes stored immediately, then you will have easier work by *editing with data provider*.) How to use the undo functionality

```
layer.beginEditCommand("Feature triangulation")

# ... call layer's editing methods ...

if problem_occurred:
  layer.destroyEditCommand()
  return

# ... more editing ...

layer.endEditCommand()
```

The `beginEndCommand()` will create an internal "active" command and will record subsequent changes in vector layer. With the call to `endEditCommand()` the command is pushed onto the undo stack and the user will be able to undo/redo it from GUI. In case something went wrong while doing the changes, the `destroyEditCommand()` method will remove the command and rollback all changes done while this command was active.

To start editing mode, there is `startEditing()` method, to stop editing there are `commitChanges()` and `rollback()` — however normally you should not need these methods and leave this functionality to be triggered by the user.

4.4 Using Spatial Index

Spatial indexes can dramatically improve the performance of your code if you need to do frequent queries to a vector layer. Imagin, for instance, that you are writing an interpolation algorithm, and that for a given location you need to know the 10 closest point from a points layer,, in order to use those point for calculating the interpolated value. Without a spatial index, the only way for QGIS to find those 10 points is to compute the distance from each and every point to the specified location and then compare those distances. This can be a very time consuming task, specilly if it needs to be repeated fro several locations. If a spatial index exists for the layer, the operation is much more effective.

Think of a layer withou a spatial index as a telephone book in which telephone number are not orderer or indexed. The only way to find the telephone number of a given person is to read from the beginning until you find it.

Spatial indexes are not created by default for a QGIS vector layer, but you can create them easily. This is what you have to do.

1. create spatial index — the following code creates an empty index:

   ```
   index = QgsSpatialIndex()
   ```

2. add features to index — index takes `QgsFeature` object and adds it to the internal data structure. You can create the object manually or use one from previous call to provider's `nextFeature()`

   ```
   index.insertFeature(feat)
   ```

3. once spatial index is filled with some values, you can do some queries:

```
# returns array of feature IDs of five nearest features
nearest = index.nearestNeighbor(QgsPoint(25.4, 12.7), 5)

# returns array of IDs of features which intersect the rectangle
intersect = index.intersects(QgsRectangle(22.5, 15.3, 23.1, 17.2))
```

4.5 Writing Vector Layers

You can write vector layer files using `QgsVectorFileWriter` class. It supports any other kind of vector file that OGR supports (shapefiles, GeoJSON, KML and others).

There are two possibilities how to export a vector layer:

- from an instance of `QgsVectorLayer`:

```
error = QgsVectorFileWriter.writeAsVectorFormat(layer, "my_shapes.shp", \
  "CP1250", None, "ESRI Shapefile")

if error == QgsVectorFileWriter.NoError:
  print "success!"

error = QgsVectorFileWriter.writeAsVectorFormat(layer, "my_json.json", \
  "utf-8", None, "GeoJSON")
if error == QgsVectorFileWriter.NoError:
  print "success again!"
```

  The third parameter specifies output text encoding. Only some drivers need this for correct operation - shapefiles are one of those — however in case you are not using international characters you do not have to care much about the encoding. The fourth parameter that we left as `None` may specify destination CRS — if a valid instance of `QgsCoordinateReferenceSystem` is passed, the layer is transformed to that CRS.

  For valid driver names please consult the supported formats by OGR — you should pass the value in 'the "Code" column as the driver name. Optionally you can set whether to export only selected features, pass further driver-specific options for creation or tell the writer not to create attributes — look into the documentation for full syntax.

- directly from features:

```
# define fields for feature attributes. A list of QgsField objects is needed
fields = [QgsField("first", QVariant.Int),
          QgsField("second", QVariant.String) ]

# create an instance of vector file writer, which will create the vector file.
# Arguments:
# 1. path to new file (will fail if exists already)
# 2. encoding of the attributes
# 3. field map
# 4. geometry type - from WKBTYPE enum
# 5. layer's spatial reference (instance of
#    QgsCoordinateReferenceSystem) - optional
# 6. driver name for the output file
writer = QgsVectorFileWriter("my_shapes.shp", "CP1250", fields, \
  QGis.WKBPoint, None, "ESRI Shapefile")

if writer.hasError() != QgsVectorFileWriter.NoError:
  print "Error when creating shapefile: ", writer.hasError()

# add a feature
fet = QgsFeature()
fet.setGeometry(QgsGeometry.fromPoint(QgsPoint(10,10)))
fet.setAttributes([1, "text"])
```

```
    writer.addFeature(fet)

    # delete the writer to flush features to disk (optional)
    del writer
```

4.6 Memory Provider

Memory provider is intended to be used mainly by plugin or 3rd party app developers. It does not store data on disk, allowing developers to use it as a fast backend for some temporary layers.

The provider supports string, int and double fields.

The memory provider also supports spatial indexing, which is enabled by calling the provider's `createSpatialIndex()` function. Once the spatial index is created you will be able to iterate over features within smaller regions faster (since it's not necessary to traverse all the features, only those in specified rectangle).

A memory provider is created by passing `"memory"` as the provider string to the `QgsVectorLayer` constructor.

The constructor also takes a URI defining the geometry type of the layer, one of: `"Point"`, `"LineString"`, `"Polygon"`, `"MultiPoint"`, `"MultiLineString"`, or `"MultiPolygon"`.

The URI can also specify the coordinate reference system, fields, and indexing of the memory provider in the URI. The syntax is:

crs=definition Specifies the coordinate reference system, where definition may be any of the forms accepted by `QgsCoordinateReferenceSystem.createFromString()`

index=yes Specifies that the provider will use a spatial index

field=name:type(length,precision) Specifies an attribute of the layer. The attribute has a name, and optionally a type (integer, double, or string), length, and precision. There may be multiple field definitions.

The following example of a URI incorporates all these options:

```
"Point?crs=epsg:4326&field=id:integer&field=name:string(20)&index=yes"
```

The following example code illustrates creating and populating a memory provider:

```
# create layer
vl = QgsVectorLayer("Point", "temporary_points", "memory")
pr = vl.dataProvider()

# add fields
pr.addAttributes( [ QgsField("name", QVariant.String),
                    QgsField("age",  QVariant.Int),
                    QgsField("size", QVariant.Double) ] )

# add a feature
fet = QgsFeature()
fet.setGeometry( QgsGeometry.fromPoint(QgsPoint(10,10)) )
fet.setAttributes(["Johny", 2, 0.3])
pr.addFeatures([fet])

# update layer's extent when new features have been added
# because change of extent in provider is not propagated to the layer
vl.updateExtents()
```

Finally, let's check whether everything went well:

```
# show some stats
print "fields:", len(pr.fields())
print "features:", pr.featureCount()
```

```
e = layer.extent()
print "extent:", e.xMin(),e.yMin(),e.xMax(),e.yMax()

# iterate over features
f = QgsFeature()
features = vl.getFeatures()
for f in features:
  print "F:",f.id(), f.attributes(), f.geometry().asPoint()
```

4.7 Appearance (Symbology) of Vector Layers

When a vector layer is being rendered, the appearance of the data is given by **renderer** and **symbols** associated with the layer. Symbols are classes which take care of drawing of visual representation of features, while renderers determine what symbol will be used for a particular feature.

The renderer for a given layer can obtained as shown below:

```
renderer = layer.rendererV2()
```

And with that reference, let us explore it a bit:

```
print "Type:", rendererV2.type()
```

There are several known renderer types available in QGIS core library:

Type	Class	Description
singleSymbol	`QgsSingleSymbolRendererV2`	Renders all features with the same symbol
categorizedSymbol	`QgsCategorizedSymbolRendererV2`	Renders features using a different symbol for each category
graduatedSymbol	`QgsGraduatedSymbolRendererV2`	Renders features using a different symbol for each range of values

There might be also some custom renderer types, so never make an assumption there are just these types. You can query `QgsRendererV2Registry` singleton to find out currently available renderers.

It is possible to obtain a dump of a renderer contents in text form — can be useful for debugging:

```
print rendererV2.dump()
```

You can get the symbol used for rendering by calling `symbol()` method and change it with `setSymbol()` method (note for C++ devs: the renderer takes ownership of the symbol.)

You can query and set attribute name which is used for classification: use `classAttribute()` and `setClassAttribute()` methods.

To get a list of categories:

```
for cat in rendererV2.categories():
  print "%s: %s :: %s" % (cat.value().toString(), cat.label(), str(cat.symbol()))
```

Where `value()` is the value used for discrimination between categories, `label()` is a text used for category description and `symbol()` method returns assigned symbol.

The renderer usually stores also original symbol and color ramp which were used for the classification: `sourceColorRamp()` and `sourceSymbol()` methods.

This renderer is very similar to the categorized symbol renderer described above, but instead of one attribute value per class it works with ranges of values and thus can be used only with numerical attributes.

To find out more about ranges used in the renderer:

```
for ran in rendererV2.ranges():
  print "%f - %f: %s %s" % (
      ran.lowerValue(),
```

```
      ran.upperValue(),
      ran.label(),
      str(ran.symbol())
      )
```

you can again use `classAttribute()` to find out classification attribute name, `sourceSymbol()` and `sourceColorRamp()` methods. Additionally there is `mode()` method which determines how the ranges were created: using equal intervals, quantiles or some other method.

If you wish to create your own graduated symbol renderer you can do so as illustrated in the example snippet below (which creates a simple two class arrangement):

```
from qgis.core import  (QgsVectorLayer,
                        QgsMapLayerRegistry,
                        QgsGraduatedSymbolRendererV2,
                        QgsSymbolV2,
                        QgsRendererRangeV2)

myVectorLayer = QgsVectorLayer(myVectorPath, myName, 'ogr')
myTargetField = 'target_field'
myRangeList = []
myOpacity = 1
# Make our first symbol and range...
myMin = 0.0
myMax = 50.0
myLabel = 'Group 1'
myColour = QtGui.QColor('#ffee00')
mySymbol1 = QgsSymbolV2.defaultSymbol(
           myVectorLayer.geometryType())
mySymbol1.setColor(myColour)
mySymbol1.setAlpha(myOpacity)
myRange1 = QgsRendererRangeV2(
                myMin,
                myMax,
                mySymbol1,
                myLabel)
myRangeList.append(myRange1)
#now make another symbol and range...
myMin = 50.1
myMax = 100
myLabel = 'Group 2'
myColour = QtGui.QColor('#00eeff')
mySymbol2 = QgsSymbolV2.defaultSymbol(
           myVectorLayer.geometryType())
mySymbol2.setColor(myColour)
mySymbol2.setAlpha(myOpacity)
myRange2 = QgsRendererRangeV2(
                myMin,
                myMax,
                mySymbol2
                myLabel)
myRangeList.append(myRange2)
myRenderer = QgsGraduatedSymbolRendererV2(
                '', myRangeList)
myRenderer.setMode(
        QgsGraduatedSymbolRendererV2.EqualInterval)
myRenderer.setClassAttribute(myTargetField)

myVectorLayer.setRendererV2(myRenderer)
QgsMapLayerRegistry.instance().addMapLayer(myVectorLayer)
```

For representation of symbols, there is `QgsSymbolV2` base class with three derived classes:

- `QgsMarkerSymbolV2` - for point features

- `QgsLineSymbolV2` - for line features
- `QgsFillSymbolV2` - for polygon features

Every symbol consists of one or more symbol layers (classes derived from `QgsSymbolLayerV2`). The symbol layers do the actual rendering, the symbol class itself serves only as a container for the symbol layers.

Having an instance of a symbol (e.g. from a renderer), it is possible to explore it: `type()` method says whether it is a marker, line or fill symbol. There is a `dump()` method which returns a brief description of the symbol. To get a list of symbol layers:

```
for i in xrange(symbol.symbolLayerCount()):
  lyr = symbol.symbolLayer(i)
  print "%d: %s" % (i, lyr.layerType())
```

To find out symbol's color use `color()` method and `setColor()` to change its color. With marker symbols additionally you can query for the symbol size and rotation with `size()` and `angle()` methods, for line symbols there is `width()` method returning line width.

Size and width are in millimeters by default, angles are in degrees.

As said before, symbol layers (subclasses of `QgsSymbolLayerV2`) determine the appearance of the features. There are several basic symbol layer classes for general use. It is possible to implement new symbol layer types and thus arbitrarily customize how features will be rendered. The `layerType()` method uniquely identifies the symbol layer class — the basic and default ones are SimpleMarker, SimpleLine and SimpleFill symbol layers types.

You can get a complete list of the types of symbol layers you can create for a given symbol layer class like this:

```
from qgis.core import QgsSymbolLayerV2Registry
myRegistry = QgsSymbolLayerV2Registry.instance()
myMetadata = myRegistry.symbolLayerMetadata("SimpleFill")
for item in myRegistry.symbolLayersForType(QgsSymbolV2.Marker):
  print item
```

Output:

```
EllipseMarker
FontMarker
SimpleMarker
SvgMarker
VectorField
```

`QgsSymbolLayerV2Registry` class manages a database of all available symbol layer types.

To access symbol layer data, use its `properties()` method that returns a key-value dictionary of properties which determine the appearance. Each symbol layer type has a specific set of properties that it uses. Additionally, there are generic methods `color()`, `size()`, `angle()`, `width()` with their setter counterparts. Of course size and angle is available only for marker symbol layers and width for line symbol layers.

Imagine you would like to customize the way how the data gets rendered. You can create your own symbol layer class that will draw the features exactly as you wish. Here is an example of a marker that draws red circles with specified radius:

```
class FooSymbolLayer(QgsMarkerSymbolLayerV2):

  def __init__(self, radius=4.0):
    QgsMarkerSymbolLayerV2.__init__(self)
    self.radius = radius
    self.color = QColor(255,0,0)

  def layerType(self):
    return "FooMarker"

  def properties(self):
    return { "radius" : str(self.radius) }
```

```
  def startRender(self, context):
    pass

  def stopRender(self, context):
    pass

  def renderPoint(self, point, context):
    # Rendering depends on whether the symbol is selected (Qgis >= 1.5)
    color = context.selectionColor() if context.selected() else self.color
    p = context.renderContext().painter()
    p.setPen(color)
    p.drawEllipse(point, self.radius, self.radius)

  def clone(self):
    return FooSymbolLayer(self.radius)
```

The `layerType()` method determines the name of the symbol layer, it has to be unique among all symbol layers. Properties are used for persistence of attributes. `clone()` method must return a copy of the symbol layer with all attributes being exactly the same. Finally there are rendering methods: `startRender()` is called before rendering first feature, `stopRender()` when rendering is done. And `renderPoint()` method which does the rendering. The coordinates of the point(s) are already transformed to the output coordinates.

For polylines and polygons the only difference would be in the rendering method: you would use `renderPolyline()` which receives a list of lines, resp. `renderPolygon()` which receives list of points on outer ring as a first parameter and a list of inner rings (or None) as a second parameter.

Usually it is convenient to add a GUI for setting attributes of the symbol layer type to allow users to customize the appearance: in case of our example above we can let user set circle radius. The following code implements such widget:

```
class FooSymbolLayerWidget(QgsSymbolLayerV2Widget):
  def __init__(self, parent=None):
    QgsSymbolLayerV2Widget.__init__(self, parent)

    self.layer = None

    # setup a simple UI
    self.label = QLabel("Radius:")
    self.spinRadius = QDoubleSpinBox()
    self.hbox = QHBoxLayout()
    self.hbox.addWidget(self.label)
    self.hbox.addWidget(self.spinRadius)
    self.setLayout(self.hbox)
    self.connect( self.spinRadius, SIGNAL("valueChanged(double)"), \
      self.radiusChanged)

  def setSymbolLayer(self, layer):
    if layer.layerType() != "FooMarker":
      return
    self.layer = layer
    self.spinRadius.setValue(layer.radius)

  def symbolLayer(self):
    return self.layer

  def radiusChanged(self, value):
    self.layer.radius = value
    self.emit(SIGNAL("changed()"))
```

This widget can be embedded into the symbol properties dialog. When the symbol layer type is selected in symbol properties dialog, it creates an instance of the symbol layer and an instance of the symbol layer widget. Then it calls `setSymbolLayer()` method to assign the symbol layer to the widget. In that method the widget should

update the UI to reflect the attributes of the symbol layer. `symbolLayer()` function is used to retrieve the symbol layer again by the properties dialog to use it for the symbol.

On every change of attributes, the widget should emit `changed()` signal to let the properties dialog update the symbol preview.

Now we are missing only the final glue: to make QGIS aware of these new classes. This is done by adding the symbol layer to registry. It is possible to use the symbol layer also without adding it to the registry, but some functionality will not work: e.g. loading of project files with the custom symbol layers or inability to edit the layer's attributes in GUI.

We will have to create metadata for the symbol layer:

```
class FooSymbolLayerMetadata(QgsSymbolLayerV2AbstractMetadata):

  def __init__(self):
    QgsSymbolLayerV2AbstractMetadata.__init__(self, "FooMarker", QgsSymbolV2.Marker)

  def createSymbolLayer(self, props):
    radius = float(props[QString("radius")]) if QString("radius") in props else 4.0
    return FooSymbolLayer(radius)

  def createSymbolLayerWidget(self):
    return FooSymbolLayerWidget()

QgsSymbolLayerV2Registry.instance().addSymbolLayerType( FooSymbolLayerMetadata() )
```

You should pass layer type (the same as returned by the layer) and symbol type (marker/line/fill) to the constructor of parent class. `createSymbolLayer()` takes care of creating an instance of symbol layer with attributes specified in the *props* dictionary. (Beware, the keys are QString instances, not "str" objects). And there is `createSymbolLayerWidget()` method which returns settings widget for this symbol layer type.

The last step is to add this symbol layer to the registry — and we are done.

It might be useful to create a new renderer implementation if you would like to customize the rules how to select symbols for rendering of features. Some use cases where you would want to do it: symbol is determined from a combination of fields, size of symbols changes depending on current scale etc.

The following code shows a simple custom renderer that creates two marker symbols and chooses randomly one of them for every feature:

```
import random

class RandomRenderer(QgsFeatureRendererV2):
  def __init__(self, syms=None):
    QgsFeatureRendererV2.__init__(self, "RandomRenderer")
    self.syms = syms if syms else [ QgsSymbolV2.defaultSymbol(QGis.Point), \
      QgsSymbolV2.defaultSymbol(QGis.Point) ]

  def symbolForFeature(self, feature):
    return random.choice(self.syms)

  def startRender(self, context, vlayer):
    for s in self.syms:
      s.startRender(context)

  def stopRender(self, context):
    for s in self.syms:
      s.stopRender(context)

  def usedAttributes(self):
    return []

  def clone(self):
    return RandomRenderer(self.syms)
```

The constructor of parent `QgsFeatureRendererV2` class needs renderer name (has to be unique among renderers). `symbolForFeature()` method is the one that decides what symbol will be used for a particular feature. `startRender()` and `stopRender()` take care of initialization/finalization of symbol rendering. `usedAttributes()` method can return a list of field names that renderer expects to be present. Finally `clone()` function should return a copy of the renderer.

Like with symbol layers, it is possible to attach a GUI for configuration of the renderer. It has to be derived from `QgsRendererV2Widget`. The following sample code creates a button that allows user to set symbol of the first symbol:

```
class RandomRendererWidget(QgsRendererV2Widget):
  def __init__(self, layer, style, renderer):
    QgsRendererV2Widget.__init__(self, layer, style)
    if renderer is None or renderer.type() != "RandomRenderer":
      self.r = RandomRenderer()
    else:
      self.r = renderer
    # setup UI
    self.btn1 = QgsColorButtonV2("Color 1")
    self.btn1.setColor(self.r.syms[0].color())
    self.vbox = QVBoxLayout()
    self.vbox.addWidget(self.btn1)
    self.setLayout(self.vbox)
    self.connect(self.btn1, SIGNAL("clicked()"), self.setColor1)

  def setColor1(self):
    color = QColorDialog.getColor( self.r.syms[0].color(), self)
    if not color.isValid(): return
    self.r.syms[0].setColor( color );
    self.btn1.setColor(self.r.syms[0].color())

  def renderer(self):
    return self.r
```

The constructor receives instances of the active layer (`QgsVectorLayer`), the global style (`QgsStyleV2`) and current renderer. If there is no renderer or the renderer has different type, it will be replaced with our new renderer, otherwise we will use the current renderer (which has already the type we need). The widget contents should be updated to show current state of the renderer. When the renderer dialog is accepted, widget's `renderer()` method is called to get the current renderer — it will be assigned to the layer.

The last missing bit is the renderer metadata and registration in registry, otherwise loading of layers with the renderer will not work and user will not be able to select it from the list of renderers. Let us finish our RandomRenderer example:

```
class RandomRendererMetadata(QgsRendererV2AbstractMetadata):
  def __init__(self):
    QgsRendererV2AbstractMetadata.__init__(self, "RandomRenderer", "Random renderer")

  def createRenderer(self, element):
    return RandomRenderer()
  def createRendererWidget(self, layer, style, renderer):
    return RandomRendererWidget(layer, style, renderer)

QgsRendererV2Registry.instance().addRenderer(RandomRendererMetadata())
```

Similarly as with symbol layers, abstract metadata constructor awaits renderer name, name visible for users and optionally name of renderer's icon. `createRenderer()` method passes `QDomElement` instance that can be used to restore renderer's state from DOM tree. `createRendererWidget()` method creates the configuration widget. It does not have to be present or can return *None* if the renderer does not come with GUI.

To associate an icon with the renderer you can assign it in `QgsRendererV2AbstractMetadata` constructor as a third (optional) argument — the base class constructor in the RandomRendererMetadata `__init__()` function becomes:

```
QgsRendererV2AbstractMetadata.__init__(self,
    "RandomRenderer",
    "Random renderer",
    QIcon(QPixmap("RandomRendererIcon.png", "png")) )
```

The icon can be associated also at any later time using `setIcon()` method of the metadata class. The icon can be loaded from a file (as shown above) or can be loaded from a Qt resource (PyQt4 includes .qrc compiler for Python).

TODO:

- creating/modifying symbols
- working with style (`QgsStyleV2`)
- working with color ramps (`QgsVectorColorRampV2`)
- rule-based renderer
- exploring symbol layer and renderer registries

CHAPTER 5

Geometry Handling

Points, linestrings and polygons that represent a spatial feature are commonly referred to as geometries. In QGIS they are represented with the `QgsGeometry` class. All possible geometry types are nicely shown in JTS discussion page.

Sometimes one geometry is actually a collection of simple (single-part) geometries. Such a geometry is called a multi-part geometry. If it contains just one type of simple geometry, we call it multi-point, multi-linestring or multi-polygon. For example, a country consisting of multiple islands can be represented as a multi-polygon.

The coordinates of geometries can be in any coordinate reference system (CRS). When fetching features from a layer, associated geometries will have coordinates in CRS of the layer.

5.1 Geometry Construction

There are several options for creating a geometry:

- from coordinates:

```
gPnt = QgsGeometry.fromPoint(QgsPoint(1,1))
gLine = QgsGeometry.fromPolyline( [ QgsPoint(1,1), QgsPoint(2,2) ] )
gPolygon = QgsGeometry.fromPolygon( [ [ QgsPoint(1,1), QgsPoint(2,2), \
  QgsPoint(2,1) ] ] )
```

  Coordinates are given using `QgsPoint` class.

  Polyline (Linestring) is represented by a list of points. Polygon is represented by a list of linear rings (i.e. closed linestrings). First ring is outer ring (boundary), optional subsequent rings are holes in the polygon.

  Multi-part geometries go one level further: multi-point is a list of points, multi-linestring is a list of linestrings and multi-polygon is a list of polygons.

- from well-known text (WKT):

```
gem = QgsGeometry.fromWkt("POINT (3 4)")
```

- from well-known binary (WKB):

```
g = QgsGeometry()
g.setWkbAndOwnership(wkb, len(wkb))
```

5.2 Access to Geometry

First, you should find out geometry type, `wkbType()` method is the one to use — it returns a value from QGis.WkbType enumeration:

```
>>> gPnt.wkbType() == QGis.WKBPoint
True
>>> gLine.wkbType() == QGis.WKBLineString
True
>>> gPolygon.wkbType() == QGis.WKBPolygon
True
>>> gPolygon.wkbType() == QGis.WKBMultiPolygon
False
```

As an alternative, one can use `type()` method which returns a value from QGis.GeometryType enumeration. There is also a helper function `isMultipart()` to find out whether a geometry is multipart or not.

To extract information from geometry there are accessor functions for every vector type. How to use accessors:

```
>>> gPnt.asPoint()
(1,1)
>>> gLine.asPolyline()
[(1,1), (2,2)]
>>> gPolygon.asPolygon()
[[(1,1), (2,2), (2,1), (1,1)]]
```

Note: the tuples (x,y) are not real tuples, they are `QgsPoint` objects, the values are accessible with `x()` and `y()` methods.

For multipart geometries there are similar accessor functions: `asMultiPoint()`, `asMultiPolyline()`, `asMultiPolygon()`.

5.3 Geometry Predicates and Operations

QGIS uses GEOS library for advanced geometry operations such as geometry predicates (`contains()`, `intersects()`, ...) and set operations (`union()`, `difference()`, ...). It can also compute geometric properties of geometries, such as area (in the case of polygons) or lengths (for polygons and lines)

Here you have a small example that combines iterating over the features in a given layer and perfoming some geometric computations based on their geometries.

```
#we assume that 'layer' is a polygon layer
features = layer.getFeatures()
for f in features:
  geom = f.geometry()
  print "Area:", geom.area()
  print "Perimeter:", geom.length()
```

Areas and perimeters don't take CRS into account when computed using these methods from the `QgsGeometry` class. For a more powerful area and distance calculation, the `QgsDistanceArea` class can be used. If projections are turned off, calculations will be planar, otherwise they'll be done on the ellipsoid. When an ellipsoid is not set explicitly, WGS84 parameters are used for calculations.

```
d = QgsDistanceArea()
d.setProjectionsEnabled(True)

print "distance in meters: ", d.measureLine(QgsPoint(10,10),QgsPoint(11,11))
```

You can find many example of algorithms that are included in QGIS and use these methods to analyze and transform vector data. Here are some links to the code of a few of them.

- Geometry transformation: Reproject algorithm
- Distance and area using the `QgsDistanceArea` class: Distance matrix algorithm
- Multi-part to single-part algorithm

CHAPTER 6

Projections Support

6.1 Coordinate reference systems

Coordinate reference systems (CRS) are encapsulated by `QgsCoordinateReferenceSystem` class. Instances of this class can be created by several different ways:

- specify CRS by its ID:

  ```
  # PostGIS SRID 4326 is allocated for WGS84
  crs = QgsCoordinateReferenceSystem(4326, \
    QgsCoordinateReferenceSystem.PostgisCrsId)
  ```

  QGIS uses three different IDs for every reference system:

  - `PostgisCrsId` - IDs used within PostGIS databases.
  - `InternalCrsId` - IDs internally used in QGIS database.
  - `EpsgCrsId` - IDs assigned by the EPSG organization

  If not specified otherwise in second parameter, PostGIS SRID is used by default.

- specify CRS by its well-known text (WKT):

  ```
  wkt = 'GEOGCS["WGS84", DATUM["WGS84", SPHEROID["WGS84", 6378137.0, \
         298.257223563]],\
         PRIMEM["Greenwich", 0.0], UNIT["degree",0.0174532925199433295],\
         AXIS["Longitude",EAST], AXIS["Latitude",NORTH]]'
  crs = QgsCoordinateReferenceSystem(wkt)
  ```

- create invalid CRS and then use one of the `create*()` functions to initialize it. In following example we use Proj4 string to initialize the projection:

  ```
  crs = QgsCoordinateReferenceSystem()
  crs.createFromProj4("+proj=longlat +ellps=WGS84 +datum=WGS84 +no_defs")
  ```

It's wise to check whether creation (i.e. lookup in the database) of the CRS has been successful: `isValid()` must return `True`.

Note that for initialization of spatial reference systems QGIS needs to lookup appropriate values in its internal database `srs.db`. Thus in case you create an independent application you need to set paths correctly with `QgsApplication.setPrefixPath()` otherwise it will fail to find the database. If you are running the commands from QGIS python console or developing a plugin you do not care: everything is already set up for you.

Accessing spatial reference system information:

```
print "QGIS CRS ID:", crs.srsid()
print "PostGIS SRID:", crs.srid()
print "EPSG ID:", crs.epsg()
```

```
print "Description:", crs.description()
print "Projection Acronym:", crs.projectionAcronym()
print "Ellipsoid Acronym:", crs.ellipsoidAcronym()
print "Proj4 String:", crs.proj4String()
# check whether it's geographic or projected coordinate system
print "Is geographic:", crs.geographicFlag()
# check type of map units in this CRS (values defined in QGis::units enum)
print "Map units:", crs.mapUnits()
```

6.2 Projections

You can do transformation between different spatial reference systems by using `QgsCoordinateTransform` class. The easiest way to use it is to create source and destination CRS and construct `QgsCoordinateTransform` instance with them. Then just repeatedly call `transform()` function to do the transformation. By default it does forward transformation, but it is capable to do also inverse transformation:

```
crsSrc = QgsCoordinateReferenceSystem(4326)    # WGS 84
crsDest = QgsCoordinateReferenceSystem(32633)  # WGS 84 / UTM zone 33N
xform = QgsCoordinateTransform(crsSrc, crsDest)

# forward transformation: src -> dest
pt1 = xform.transform(QgsPoint(18,5))
print "Transformed point:", pt1

# inverse transformation: dest -> src
pt2 = xform.transform(pt1, QgsCoordinateTransform.ReverseTransform)
print "Transformed back:", pt2
```

CHAPTER 7

Using Map Canvas

The Map canvas widget is probably the most important widget within QGIS because it shows the map composed from overlaid map layers and allows interaction with the map and layers. The canvas shows always a part of the map defined by the current canvas extent. The interaction is done through the use of **map tools**: there are tools for panning, zooming, identifying layers, measuring, vector editing and others. Similar to other graphics programs, there is always one tool active and the user can switch between the available tools.

Map canvas is implemented as `QgsMapCanvas` class in `qgis.gui` module. The implementation is based on the Qt Graphics View framework. This framework generally provides a surface and a view where custom graphics items are placed and user can interact with them. We will assume that you are familiar enough with Qt to understand the concepts of the graphics scene, view and items. If not, please make sure to read the overview of the framework.

Whenever the map has been panned, zoomed in/out (or some other action triggers a refresh), the map is rendered again within the current extent. The layers are rendered to an image (using `QgsMapRenderer` class) and that image is then displayed in the canvas. The graphics item (in terms of the Qt graphics view framework) responsible for showing the map is `QgsMapCanvasMap` class. This class also controls refreshing of the rendered map. Besides this item which acts as a background, there may be more **map canvas items**. Typical map canvas items are rubber bands (used for measuring, vector editing etc.) or vertex markers. The canvas items are usually used to give some visual feedback for map tools, for example, when creating a new polygon, the map tool creates a rubber band canvas item that shows the current shape of the polygon. All map canvas items are subclasses of `QgsMapCanvasItem` which adds some more functionality to the basic `QGraphicsItem` objects.

To summarize, the map canvas architecture consists of three concepts:

- map canvas — for viewing of the map
- map canvas items — additional items that can be displayed in map canvas
- map tools — for interaction with map canvas

7.1 Embedding Map Canvas

Map canvas is a widget like any other Qt widget, so using it is as simple as creating and showing it:

```
canvas = QgsMapCanvas()
canvas.show()
```

This produces a standalone window with map canvas. It can be also embedded into an existing widget or window. When using .ui files and Qt Designer, place a `QWidget` on the form and promote it to a new class: set `QgsMapCanvas` as class name and set `qgis.gui` as header file. The `pyuic4` utility will take care of it. This is a very convenient way of embedding the canvas. The other possibility is to manually write the code to construct map canvas and other widgets (as children of a main window or dialog) and create a layout.

By default, map canvas has black background and does not use anti-aliasing. To set white background and enable anti-aliasing for smooth rendering:

```
canvas.setCanvasColor(Qt.white)
canvas.enableAntiAliasing(True)
```

(In case you are wondering, `Qt` comes from `PyQt4.QtCore` module and `Qt.white` is one of the predefined `QColor` instances.)

Now it is time to add some map layers. We will first open a layer and add it to the map layer registry. Then we will set the canvas extent and set the list of layers for canvas:

```
layer = QgsVectorLayer(path, name, provider)
if not layer.isValid():
  raise IOError, "Failed to open the layer"

# add layer to the registry
QgsMapLayerRegistry.instance().addMapLayer(layer)

# set extent to the extent of our layer
canvas.setExtent(layer.extent())

# set the map canvas layer set
canvas.setLayerSet( [ QgsMapCanvasLayer(layer) ] )
```

After executing these commands, the canvas should show the layer you have loaded.

7.2 Using Map Tools with Canvas

The following example constructs a window that contains a map canvas and basic map tools for map panning and zooming. Actions are created for activation of each tool: panning is done with `QgsMapToolPan`, zooming in/out with a pair of `QgsMapToolZoom` instances. The actions are set as checkable and later assigned to the tools to allow automatic handling of checked/unchecked state of the actions – when a map tool gets activated, its action is marked as selected and the action of the previous map tool is deselected. The map tools are activated using `setMapTool()` method.

```
from qgis.gui import *
from PyQt4.QtGui import QAction, QMainWindow
from PyQt4.QtCore import SIGNAL, Qt, QString

class MyWnd(QMainWindow):
  def __init__(self, layer):
    QMainWindow.__init__(self)

    self.canvas = QgsMapCanvas()
    self.canvas.setCanvasColor(Qt.white)

    self.canvas.setExtent(layer.extent())
    self.canvas.setLayerSet( [ QgsMapCanvasLayer(layer) ] )

    self.setCentralWidget(self.canvas)

    actionZoomIn = QAction(QString("Zoom in"), self)
    actionZoomOut = QAction(QString("Zoom out"), self)
    actionPan = QAction(QString("Pan"), self)

    actionZoomIn.setCheckable(True)
    actionZoomOut.setCheckable(True)
    actionPan.setCheckable(True)

    self.connect(actionZoomIn, SIGNAL("triggered()"), self.zoomIn)
    self.connect(actionZoomOut, SIGNAL("triggered()"), self.zoomOut)
    self.connect(actionPan, SIGNAL("triggered()"), self.pan)
```

```
    self.toolbar = self.addToolBar("Canvas actions")
    self.toolbar.addAction(actionZoomIn)
    self.toolbar.addAction(actionZoomOut)
    self.toolbar.addAction(actionPan)

    # create the map tools
    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)

    self.pan()

  def zoomIn(self):
    self.canvas.setMapTool(self.toolZoomIn)

  def zoomOut(self):
    self.canvas.setMapTool(self.toolZoomOut)

  def pan(self):
    self.canvas.setMapTool(self.toolPan)
```

You can put the above code to a file, e.g. `mywnd.py` and try it out in Python console within QGIS. This code will put the currently selected layer into newly created canvas:

```
import mywnd
w = mywnd.MyWnd(qgis.utils.iface.activeLayer())
w.show()
```

Just make sure that the `mywnd.py` file is located within Python search path (`sys.path`). If it isn't, you can simply add it: `sys.path.insert(0, '/my/path')` — otherwise the import statement will fail, not finding the module.

7.3 Rubber Bands and Vertex Markers

To show some additional data on top of the map in canvas, use map canvas items. It is possible to create custom canvas item classes (covered below), however there are two useful canvas item classes for convenience: `QgsRubberBand` for drawing polylines or polygons, and `QgsVertexMarker` for drawing points. They both work with map coordinates, so the shape is moved/scaled automatically when the canvas is being panned or zoomed.

To show a polyline:

```
r = QgsRubberBand(canvas, False)  # False = not a polygon
points = [ QgsPoint(-1,-1), QgsPoint(0,1), QgsPoint(1,-1) ]
r.setToGeometry(QgsGeometry.fromPolyline(points), None)
```

To show a polygon:

```
r = QgsRubberBand(canvas, True)  # True = a polygon
points = [ [ QgsPoint(-1,-1), QgsPoint(0,1), QgsPoint(1,-1) ] ]
r.setToGeometry(QgsGeometry.fromPolygon(points), None)
```

Note that points for polygon is not a plain list: in fact, it is a list of rings containing linear rings of the polygon: first ring is the outer border, further (optional) rings correspond to holes in the polygon.

Rubber bands allow some customization, namely to change their color and line width:

```
r.setColor(QColor(0,0,255))
r.setWidth(3)
```

The canvas items are bound to the canvas scene. To temporarily hide them (and show again, use the `hide()` and `show()` combo. To completely remove the item, you have to remove it from the scene of the canvas:

```
canvas.scene().removeItem(r)
```

(in C++ it's possible to just delete the item, however in Python `del r` would just delete the reference and the object will still exist as it is owned by the canvas)

Rubber band can be also used for drawing points, however `QgsVertexMarker` class is better suited for this (`QgsRubberBand` would only draw a rectangle around the desired point). How to use the vertex marker:

```
m = QgsVertexMarker(canvas)
m.setCenter(QgsPoint(0,0))
```

This will draw a red cross on position [0,0]. It is possible to customize the icon type, size, color and pen width:

```
m.setColor(QColor(0,255,0))
m.setIconSize(5)
m.setIconType(QgsVertexMarker.ICON_BOX) # or ICON_CROSS, ICON_X
m.setPenWidth(3)
```

For temporary hiding of vertex markers and removing them from canvas, the same applies as for the rubber bands.

7.4 Writing Custom Map Tools

You can write your custom tools, to implement a custom behaviour to actions perfored by users on the canvas.

Map tools should inherit from the `QgsMapTool` class or any derived class, and selected as active tools in the canvas using the `setMapTool()` method as we have already seen.

Here is an example of a map tool that allows to define a rectangular extent by clicking and draggin on the canvas. When the rectangle is defined, it prints its boundary coordinates in the console. It uses the rubber band elements described before to show the selected rectangle as it is being defined.

```
class RectangleMapTool(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.reset()

  def reset(self):
      self.startPoint = self.endPoint = None
      self.isEmittingPoint = False
      self.rubberBand.reset(QGis.Polygon)

  def canvasPressEvent(self, e):
      self.startPoint = self.toMapCoordinates(e.pos())
      self.endPoint = self.startPoint
      self.isEmittingPoint = True
      self.showRect(self.startPoint, self.endPoint)

  def canvasReleaseEvent(self, e):
      self.isEmittingPoint = False
      r = self.rectangle()
      if r is not None:
        print "Rectangle:", r.xMin(), r.yMin(), r.xMax(), r.yMax()

  def canvasMoveEvent(self, e):
      if not self.isEmittingPoint:
        return
```

```
        self.endPoint = self.toMapCoordinates( e.pos() )
        self.showRect(self.startPoint, self.endPoint)

    def showRect(self, startPoint, endPoint):
        self.rubberBand.reset(QGis.Polygon)
        if startPoint.x() == endPoint.x() or startPoint.y() == endPoint.y():
          return

        point1 = QgsPoint(startPoint.x(), startPoint.y())
        point2 = QgsPoint(startPoint.x(), endPoint.y())
        point3 = QgsPoint(endPoint.x(), endPoint.y())
        point4 = QgsPoint(endPoint.x(), startPoint.y())

        self.rubberBand.addPoint( point1, False )
        self.rubberBand.addPoint( point2, False )
        self.rubberBand.addPoint( point3, False )
        self.rubberBand.addPoint( point4, True )    # true to update canvas
        self.rubberBand.show()

    def rectangle(self):
        if self.startPoint is None or self.endPoint is None:
          return None
        elif self.startPoint.x() == self.endPoint.x() or self.startPoint.y() == \
          self.endPoint.y():
          return None

        return QgsRectangle(self.startPoint, self.endPoint)

    def deactivate(self):
        QgsMapTool.deactivate(self)
        self.emit(SIGNAL("deactivated()"))
```

7.5 Writing Custom Map Canvas Items

TODO: how to create a map canvas item

CHAPTER 8

Map Rendering and Printing

There are generally two approaches when input data should be rendered as a map: either do it quick way using `QgsMapRenderer` or produce more fine-tuned output by composing the map with `QgsComposition` class and friends.

8.1 Simple Rendering

Render some layers using `QgsMapRenderer` - create destination paint device (`QImage`, `QPainter` etc.), set up layer set, extent, output size and do the rendering:

```
# create image
img = QImage(QSize(800,600), QImage.Format_ARGB32_Premultiplied)

# set image's background color
color = QColor(255,255,255)
img.fill(color.rgb())

# create painter
p = QPainter()
p.begin(img)
p.setRenderHint(QPainter.Antialiasing)

render = QgsMapRenderer()

# set layer set
lst = [ layer.getLayerID() ]  # add ID of every layer
render.setLayerSet(lst)

# set extent
rect = QgsRect(render.fullExtent())
rect.scale(1.1)
render.setExtent(rect)

# set output size
render.setOutputSize(img.size(), img.logicalDpiX())

# do the rendering
render.render(p)

p.end()

# save image
img.save("render.png","png")
```

8.2 Output using Map Composer

Map composer is a very handy tool if you would like to do a more sophisticated output than the simple rendering shown above. Using the composer it is possible to create complex map layouts consisting of map views, labels, legend, tables and other elements that are usually present on paper maps. The layouts can be then exported to PDF, raster images or directly printed on a printer.

The composer consists of a bunch of classes. They all belong to the core library. QGIS application has a convenient GUI for placement of the elements, though it is not available in the gui library. If you are not familiar with Qt Graphics View framework, then you are encouraged to check the documentation now, because the composer is based on it.

The central class of the composer is `QgsComposition` which is derived from `QGraphicsScene`. Let us create one:

```
mapRenderer = iface.mapCanvas().mapRenderer()
c = QgsComposition(mapRenderer)
c.setPlotStyle(QgsComposition.Print)
```

Note that the composition takes an instance of `QgsMapRenderer`. In the code we expect we are running within QGIS application and thus use the map renderer from map canvas. The composition uses various parameters from the map renderer, most importantly the default set of map layers and the current extent. When using composer in a standalone application, you can create your own map renderer instance the same way as shown in the section above and pass it to the composition.

It is possible to add various elements (map, label, ...) to the composition — these elements have to be descendants of `QgsComposerItem` class. Currently supported items are:

- map — this item tells the libraries where to put the map itself. Here we create a map and stretch it over the whole paper size:

  ```
  x, y = 0, 0
  w, h = c.paperWidth(), c.paperHeight()
  composerMap = QgsComposerMap(c, x,y,w,h)
  c.addItem(composerMap)
  ```

- label — allows displaying labels. It is possible to modify its font, color, alignment and margin:

  ```
  composerLabel = QgsComposerLabel(c)
  composerLabel.setText("Hello world")
  composerLabel.adjustSizeToText()
  c.addItem(composerLabel)
  ```

- legend

  ```
  legend = QgsComposerLegend(c)
  legend.model().setLayerSet(mapRenderer.layerSet())
  c.addItem(legend)
  ```

- scale bar

  ```
  item = QgsComposerScaleBar(c)
  item.setStyle('Numeric') # optionally modify the style
  item.setComposerMap(composerMap)
  item.applyDefaultSize()
  c.addItem(item)
  ```

- arrow
- picture
- shape
- table

By default the newly created composer items have zero position (top left corner of the page) and zero size. The position and size are always measured in millimeters:

```
# set label 1cm from the top and 2cm from the left of the page
composerLabel.setItemPosition(20,10)
# set both label's position and size (width 10cm, height 3cm)
composerLabel.setItemPosition(20,10, 100, 30)
```

A frame is drawn around each item by default. How to remove the frame:

```
composerLabel.setFrame(False)
```

Besides creating the composer items by hand, QGIS has support for composer templates which are essentially compositions with all their items saved to a .qpt file (with XML syntax). Unfortunately this functionality is not yet available in the API.

Once the composition is ready (the composer items have been created and added to the composition), we can proceed to produce a raster and/or vector output.

The default output settings for composition are page size A4 and resolution 300 DPI. You can change them if necessary. The paper size is specified in millimeters:

```
c.setPaperSize(width, height)
c.setPrintResolution(dpi)
```

8.2.1 Output to a raster image

The following code fragment shows how to render a composition to a raster image:

```
dpi = c.printResolution()
dpmm = dpi / 25.4
width = int(dpmm * c.paperWidth())
height = int(dpmm * c.paperHeight())

# create output image and initialize it
image = QImage(QSize(width, height), QImage.Format_ARGB32)
image.setDotsPerMeterX(dpmm * 1000)
image.setDotsPerMeterY(dpmm * 1000)
image.fill(0)

# 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()

image.save("out.png", "png")
```

8.2.2 Output to PDF

The following code fragment renders a composition to a PDF file:

```
printer = QPrinter()
printer.setOutputFormat(QPrinter.PdfFormat)
printer.setOutputFileName("out.pdf")
printer.setPaperSize(QSizeF(c.paperWidth(), c.paperHeight()), QPrinter.Millimeter)
printer.setFullPage(True)
printer.setColorMode(QPrinter.Color)
printer.setResolution(c.printResolution())

pdfPainter = QPainter(printer)
```

```
paperRectMM = printer.pageRect(QPrinter.Millimeter)
paperRectPixel = printer.pageRect(QPrinter.DevicePixel)
c.render(pdfPainter, paperRectPixel, paperRectMM)
pdfPainter.end()
```

CHAPTER 9

Expressions, Filtering and Calculating Values

QGIS has some support for parsing of SQL-like expressions. Only a small subset of SQL syntax is supported. The expressions can be evaluated either as boolean predicates (returning True or False) or as functions (returning a scalar value).

Three basic types are supported:

- number — both whole numbers and decimal numbers, e.g. `123`, `3.14`
- string — they have to be enclosed in single quotes: `'hello world'`
- column reference — when evaluating, the reference is substituted with the actual value of the field. The names are not escaped.

The following operations are available:

- arithmetic operators: `+`, `-`, `*`, `/`, `^`
- parentheses: for enforcing the operator precedence: `(1 + 1) * 3`
- unary plus and minus: `-12`, `+5`
- mathematical functions: `sqrt`, `sin`, `cos`, `tan`, `asin`, `acos`, `atan`
- geometry functions: `$area`, `$length`
- conversion functions: `to int`, `to real`, `to string`

And the following predicates are supported:

- comparison: `=`, `!=`, `>`, `>=`, `<`, `<=`
- pattern matching: `LIKE` (using % and _), ~ (regular expressions)
- logical predicates: `AND`, `OR`, `NOT`
- NULL value checking: `IS NULL`, `IS NOT NULL`

Examples of predicates:

- `1 + 2 = 3`
- `sin(angle) > 0`
- `'Hello' LIKE 'He%'`
- `(x > 10 AND y > 10) OR z = 0`

Examples of scalar expressions:

- `2 ^ 10`
- `sqrt(val)`
- `$length + 1`

9.1 Parsing Expressions

```
>>> exp = QgsExpression('1 + 1 = 2')
>>> exp.hasParserError()
False
>>> exp = QgsExpression('1 + 1 = ')
>>> exp.hasParserError()
True
>>> exp.parserErrorString()
PyQt4.QtCore.QString(u'syntax error, unexpected $end')
```

9.2 Evaluating Expressions

9.2.1 Basic Expressions

```
>>> exp = QgsExpression('1 + 1 = 2')
>>> value = exp.evaluate()
>>> value
1
```

9.2.2 Expressions with features

The following example will evaluate the given expression against a feature. "Column" is the name of the field in the layer.

```
>>> exp = QgsExpression('Column = 99')
>>> value = exp.evaluate(feature, layer.pendingFields())
>>> bool(value)
True
```

You can also use `QgsExpression.prepare()` if you need check more than one feature. Using `QgsExpression.prepare()` will increase the speed that evaluate takes to run.

```
>>> exp = QgsExpression('Column = 99')
>>> exp.prepare(layer.pendingFields())
>>> value = exp.evaluate(feature)
>>> bool(value)
True
```

9.2.3 Handling errors

```
exp = QgsExpression("1 + 1 = 2 ")
if exp.hasParserError():
  raise Expection(exp.parserErrorString())

value = exp.evaluate()
if exp.hasEvalError():
  raise ValueError(exp.evalErrorString())

print value
```

9.3 Examples

The following example can be used to filter a layer and return any feature that matches a predicate.

```
def where(layer, exp):
  print "Where"
  exp = QgsExpression(exp)
  if exp.hasParserError():
    raise Expection(exp.parserErrorString())
  exp.prepare(layer.pendingFields())
  for feature in layer.getFeatures():
    value = exp.evaluate(feature)
    if exp.hasEvalError():
      raise ValueError(exp.evalErrorString())
    if bool(value):
      yield feature

layer = qgis.utils.iface.activeLayer()
for f in where(layer, 'Test > 1.0'):
  print f + " Matches expression"
```

CHAPTER 10

Reading And Storing Settings

Many times it is useful for a plugin to save some variables so that the user does not have to enter or select them again next time the plugin is run.

These variables can be saved a retrieved with help of Qt and QGIS API. For each variable, you should pick a key that will be used to access the variable — for user's favourite color you could use key "favourite_color" or any other meaningful string. It is recommended to give some structure to naming of keys.

We can make difference between several types of settings:

- **global settings** — they are bound to the user at particular machine. QGIS itself stores a lot of global settings, for example, main window size or default snapping tolerance. This functionality is provided directly by Qt framework by the means of QSettings class. By default, this class stores settings in system's "native" way of storing settings, that is — registry (on Windows), .plist file (on Mac OS X) or .ini file (on Unix). The QSettings documentation is comprehensive, so we will provide just a simple example:

```
def store():
  s = QSettings()
  s.setValue("myplugin/mytext", "hello world")
  s.setValue("myplugin/myint",  10)
  s.setValue("myplugin/myreal", 3.14)

def read():
  s = QSettings()
  mytext = s.value("myplugin/mytext", "default text")
  myint  = s.value("myplugin/myint", 123)
  myreal = s.value("myplugin/myreal", 2.71)
```

The second parameter of the `value()` method is optional and specifies the default value if there is no previous value set for the passed setting name.

- **project settings** — vary between different projects and therefore they are connected with a project file. Map canvas background color or destination coordinate reference system (CRS) are examples — white background and WGS84 might be suitable for one project, while yellow background and UTM projection are better for another one. An example of usage follows:

```
proj = QgsProject.instance()

# store values
proj.writeEntry("myplugin", "mytext", "hello world")
proj.writeEntry("myplugin", "myint", 10)
proj.writeEntry("myplugin", "mydouble", 0.01)
proj.writeEntry("myplugin", "mybool", True)

# read values
mytext = proj.readEntry("myplugin", "mytext", "default text")[0]
myint = proj.readNumEntry("myplugin", "myint", 123)[0]
```

As you can see, the `writeEntry()` method is used for all data types, but several methods exist for reading the setting value back, and the corresponding one has to be selected for each data type.

- **map layer settings** — these settings are related to a particular instance of a map layer with a project. They are *not* connected with underlying data source of a layer, so if you create two map layer instances of one shapefile, they will not share the settings. The settings are stored in project file, so if the user opens the project again, the layer-related settings will be there again. This functionality has been added in QGIS v1.4. The API is similar to QSettings — it takes and returns QVariant instances:

```
# save a value
layer.setCustomProperty("mytext", "hello world")

# read the value again
mytext = layer.customProperty("mytext", "default text")
```

CHAPTER 11

Communicating with the user

This section shows some methods and elements that should be used to communicate with the user, in order to keep consistency in the User Interface.

11.1 Showing messages. The QgsMessageBar class.

Using mesages boxes can be a bad idea from a user experience point of view. For showing a small info line or a warning/error messages, the QGIS message bar is usually a better option

Using the reference to the QGIS interface object, you can show a message in the message bar with the following code.

```
iface.messageBar().pushMessage("Error", "I'm sorry Dave, I'm afraid I can't \
  do that", level=QgsMessageBar.CRITICAL)
```

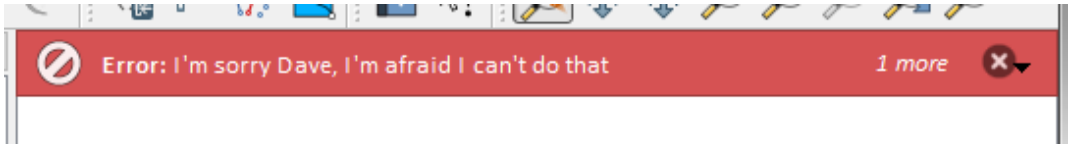

Figure 11.1: QGIS Message bar

You can set a duration to show it for a limited time.

```
iface.messageBar().pushMessage("Error", ""Ooops, the plugin is not working as \
  it should", level=QgsMessageBar.CRITICAL, duration=3)
```

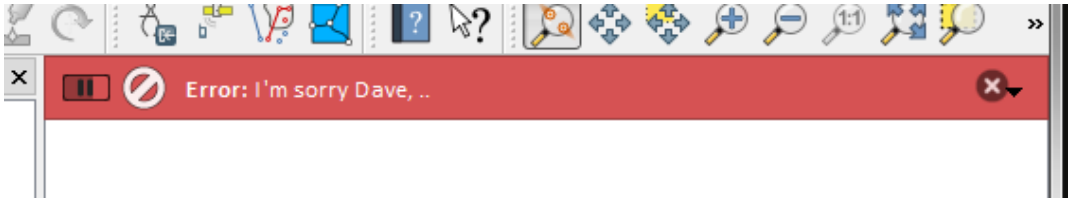

Figure 11.2: QGIS Message bar with timer

The examples above show an error bar, but the `level` parameter can be used to creating warning messages or info messages, using the `QgsMessageBar.WARNING` and `QgsMessageBar.INFO` constants repectively.

Widgets can be added to the message bar, like for instance a button to show more info

```
def showError():
pass
```

static/pyqgis_developer_cookbook/warningbar.png

Figure 11.3: QGIS Message bar (warning)

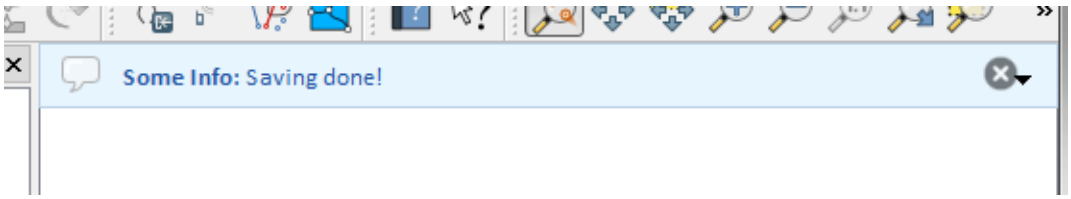

Figure 11.4: QGIS Message bar (info)

```
widget = iface.messageBar().createMessage("Missing Layers", "Show Me")
button = QPushButton(widget)
button.setText("Show Me")
button.pressed.connect(showError)
widget.layout().addWidget(button)
iface.messageBar().pushWidget(widget, QgsMessageBar.WARNING)
```

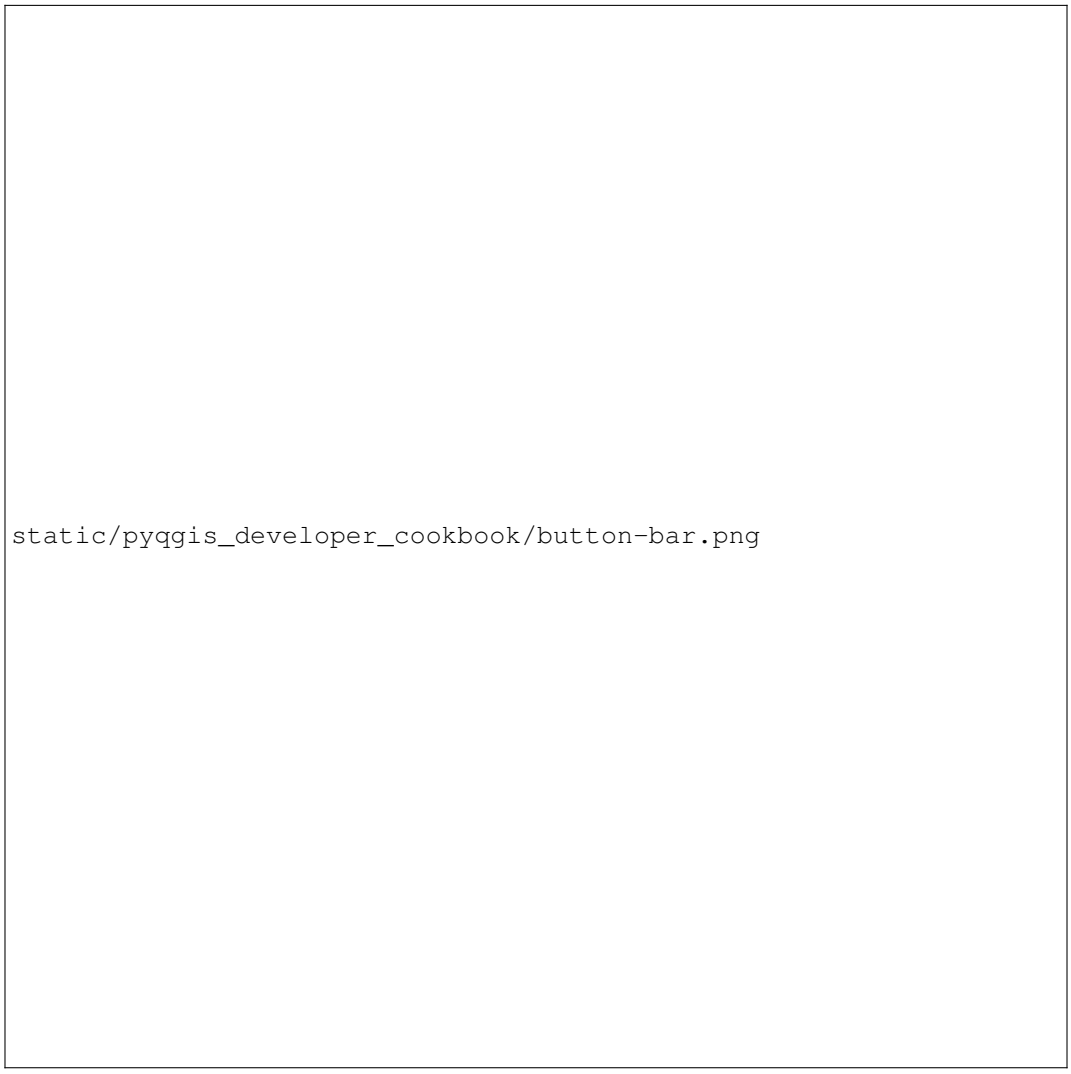

Figure 11.5: QGIS Message bar with a button

You can even use a message bar in your own dialog so you don't have to show a message box, or if it doesn't make sense to show it in the main QGIS window.

```
class MyDialog(QDialog):
    def __init__(self):
        QDialog.__init__(self)
        self.bar = QgsMessageBar()
        self.bar.setSizePolicy( QSizePolicy.Minimum, QSizePolicy.Fixed )
        self.setLayout(QGridLayout())
        self.layout().setContentsMargins(0,0,0,0)
        self.buttonbox = QDialogButtonBox(QDialogButtonBox.Ok)
        self.buttonbox.accepted.connect(self.run)
        self.layout().addWidget(self.buttonbox , 0,0,2,1)
        self.layout().addWidget(self.bar, 0,0,1,1)

    def run(self):
```

```
        self.bar.pushMessage("Hello", "World", level=QgsMessageBar.INFO)
```

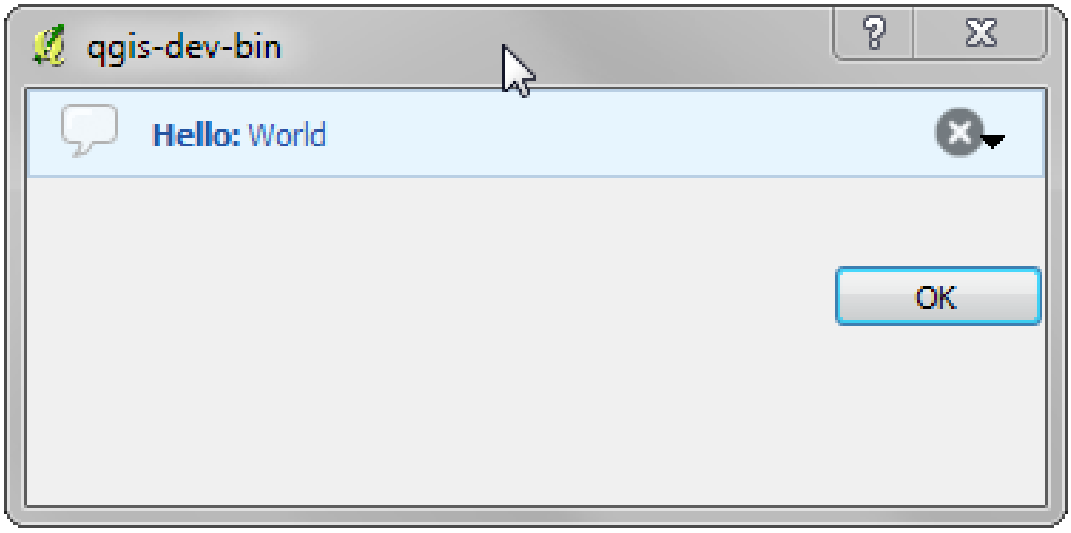

Figure 11.6: QGIS Message bar in custom dialog

11.2 Showing progress

Progress bars can also be put in the QGIS message bar, since, as we have seen, it accepts widgets. Here is an example that you can try in the console.

```
import time
from PyQt4.QtGui import QProgressBar
from PyQt4.QtCore import *
progressMessageBar = iface.messageBar().createMessage("Doing something boring...")
progress = QProgressBar()
progress.setMaximum(10)
progress.setAlignment(Qt.AlignLeft|Qt.AlignVCenter)
progressMessageBar.layout().addWidget(progress)
iface.messageBar().pushWidget(progressMessageBar, iface.messageBar().INFO)
for i in range(10):
        time.sleep(1)
        progress.setValue(i + 1)
iface.messageBar().clearWidgets()
```

Also, you can use the built-in status bar to report progress, as in the next example.

:: count = layers.featureCount() for i, feature in enumerate(features):

#do something time-consuming here ... percent = i / float(count) * 100 iface.mainWindow().statusBar().showMessage("Processed {} %".format(int(percent)))

iface.mainWindow().statusBar().clearMessage()

11.3 Logging

You can use the QGIS logging system to log all the information that you want to save about the execution of your code.

```
QgsMessageLog.logMessage("Your plugin code has been executed correctly", \
  QgsMessageLog.INFO)
QgsMessageLog.logMessage("Your plugin code might have some problems", \
```

```
  QgsMessageLog.WARNING)
QgsMessageLog.logMessage("Your plugin code has crashed!", \
  QgsMessageLog.CRITICAL)
```

CHAPTER 12

Developing Python Plugins

It is possible to create plugins in Python programming language. In comparison with classical plugins written in C++ these should be easier to write, understand, maintain and distribute due the dynamic nature of the Python language.

Python plugins are listed together with C++ plugins in QGIS plugin manager. They're being searched for in these paths:

- UNIX/Mac: `~/.qgis/python/plugins` and `(qgis_prefix)/share/qgis/python/plugins`
- Windows: `~/.qgis/python/plugins` and `(qgis_prefix)/python/plugins`

Home directory (denoted by above `~`) on Windows is usually something like `C:\Documents and Settings\(user)`. Subdirectories of these paths are considered as Python packages that can be imported to QGIS as plugins.

Steps:

1. *Idea*: Have an idea about what you want to do with your new QGIS plugin. Why do you do it? What problem do you want to solve? Is there already another plugin for that problem?
2. *Create files*: Create the files described next. A starting point (`__init.py__`). Fill in the *Plugin metadata* (`metadata.txt`) A main python plugin body (`plugin.py`). A form in QT-Designer (`form.ui`), with its `resources.qrc`.
3. *Write code*: Write the code inside the `plugin.py`
4. *Test*: Close and re-open QGIS and import your plugin again. Check if everything is OK.
5. *Publish*: Publish your plugin in QGIS repository or make your own repository as an "arsenal" of personal "GIS weapons"

12.1 Writing a plugin

Since the introduction of python plugins in QGIS, a number of plugins have appeared - on Plugin Repositories wiki page you can find some of them, you can use their source to learn more about programming with PyQGIS or find out whether you are not duplicating development effort. QGIS team also maintains an *Official python plugin repository*. Ready to create a plugin but no idea what to do? Python Plugin Ideas wiki page lists wishes from the community!

12.1.1 Plugin files

Here's the directory structure of our example plugin:

```
PYTHON_PLUGINS_PATH/
  testplug/
    __init__.py
```

```
plugin.py
metadata.txt
resources.qrc
resources.py
form.ui
form.py
```

What is the meaning of the files:

- `__init__.py` = The starting point of the plugin. It is normally empty.
- `plugin.py` = The main working code of the plugin. Contains all the information about the actions of the plugin and the main code.
- `resources.qrc` = The .xml document created by QT-Designer. Contains relative paths to resources of the forms.
- `resources.py` = The translation of the .qrc file described above to Python.
- `form.ui` = The GUI created by QT-Designer.
- `form.py` = The translation of the form.ui described above to Python.
- `metadata.txt` = Required for QGIS >= 1.8.0. Containts general info, version, name and some other metadata used by plugins website and plugin infrastructure. Metadata in `metadata.txt` is preferred to the methods in `__init__.py`. If the text file is present, it is used to fetch the values. From QGIS 2.0 the metadata from `__init__.py` will not be accepted and the `metadata.txt` file will be required.

Here and there are two automated ways of creating the basic files (skeleton) of a typical QGIS Python plugin.

Also there is a QGIS plugin called *Plugin Builder* that creates plugin template from QGIS and doesn't require internet connection. This is the recommended option, as it produces 2.0 compatible sources.

Warning: If you plan to upload the plugin to the *Official python plugin repository* you must check that your plugin follows some additional rules, required for plugin *Validation*

12.2 Plugin content

Here you can find information and examples about what to add in each of the files in the file structure described above.

12.2.1 Plugin metadata

First, plugin manager needs to retrieve some basic information about the plugin such as its name, description etc. File `metadata.txt` is the right place where to put this information.

Important: All metadata must be in UTF-8 encoding.

Metadata name	Required	Notes
name	True	a short string containing the name of the plugin
qgisMinimumVersion	True	dotted notation of minimum QGIS version
qgisMaximumVersion	False	dotted notation of maximum QGIS version
description	True	longer text which describes the plugin, no HTML allowed
version	True	short string with the version dotted notation
author	True	author name
email	True	email of the author, will *not* be shown on the web site
changelog	False	string, can be multiline, no HTML allowed
experimental	False	boolean flag, *True* or *False*
deprecated	False	boolean flag, *True* or *False*, applies to the whole plugin and not just to the uploaded version
tags	False	comma separated list, spaces are allowe inside individual tags
homepage	False	a valid URL pointing to the homepage of your plugin
repository	False	a valid URL for the source code repository
tracker	False	a valid URL for tickets and bug reports
icon	False	a file name or a relative path (relative to the base folder of the plugin's compressed package)
category	False	one of *Raster*, *Vector*, *Database* and *Web*

By default, plugins are placed in the *Plugins* menu (we will see in the next section how to add a menu entry for your plugin) but they can also be placed the into *Raster*, *Vector*, *Database* and *Web* menus. A corresponding "category" metadata entry exists to specify that, so the plugin can be classified accordingly. This metadata entry is used as tip for users and tells them where (in which menu) the plugin can be found. Allowed values for "category" are: Vector, Raster, Database, Web and Layers. For example, if your plugin will be available from *Raster* menu, add this to `metadata.txt`:

```
category=Raster
```

Note: If *qgisMaximumVersion* is empty, it will be automatically set to the major version plus *.99* when uploaded to the *Official python plugin repository*.

An example for this metadata.txt:

```
; the next section is mandatory

[general]
name=HelloWorld
email=me@example.com
author=Just Me
qgisMinimumVersion=2.0
description=This is a plugin for greeting the
    (going multiline) world
version=version 1.2
; end of mandatory metadata

; start of optional metadata
category=Raster
changelog=this is a very
    very
    very
    very
    very
    very long multiline changelog

; tags are in comma separated value format, spaces are allowed
tags=wkt,raster,hello world
```

```
; these metadata can be empty
homepage=http://www.itopen.it
tracker=http://bugs.itopen.it
repository=http://www.itopen.it/repo
icon=icon.png

; experimental flag
experimental=True

; deprecated flag (applies to the whole plugin and not only to the uploaded version)
deprecated=False

; if empty, it will be automatically set to major version + .99
qgisMaximumVersion=2.0
```

12.2.2 plugin.py

One thing worth mentioning is `classFactory()` function which is called when the plugin gets loaded to QGIS. It receives reference to instance of `QgisInterface` and must return instance of your plugin - in our case it's called `TestPlugin`. This is how should this class look like (e.g. `testplugin.py`):

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

# initialize Qt resources from file resouces.py
import resources

class TestPlugin:

  def __init__(self, iface):
    # save reference to the QGIS interface
    self.iface = iface

  def initGui(self):
    # create action that will start plugin configuration
    self.action = QAction(QIcon(":/plugins/testplug/icon.png"), "Test plugin", \
      self.iface.mainWindow())
    self.action.setWhatsThis("Configuration for test plugin")
    self.action.setStatusTip("This is status tip")
    QObject.connect(self.action, SIGNAL("triggered()"), self.run)

    # add toolbar button and menu item
    self.iface.addToolBarIcon(self.action)
    self.iface.addPluginToMenu("&Test plugins", self.action)

    # connect to signal renderComplete which is emitted when canvas
    # rendering is done
    QObject.connect(self.iface.mapCanvas(), SIGNAL("renderComplete(QPainter *)"), \
      self.renderTest)

  def unload(self):
    # remove the plugin menu item and icon
    self.iface.removePluginMenu("&Test plugins",self.action)
    self.iface.removeToolBarIcon(self.action)

    # disconnect form signal of the canvas
    QObject.disconnect(self.iface.mapCanvas(), SIGNAL("renderComplete(QPainter *)"), \
      self.renderTest)

  def run(self):
```

```
    # create and show a configuration dialog or something similar
    print "TestPlugin: run called!"

  def renderTest(self, painter):
    # use painter for drawing to map canvas
    print "TestPlugin: renderTest called!"
```

The only plugin functions that must exist are `initGui()` and `unload()`. These functions are called when the plugin is loaded and unloaded.

You can see that in the above example, the `addPluginMenu()` is used. This will add the corresponding menu action to the *Plugins* menu. Alternative methods exist to add the action to a different menu. Here is a list of those methods:

- `addPluginToRasterMenu()`
- `addPluginToVectorMenu()`
- `addPluginToDatabaseMenu()`
- `addPluginToWebMenu()`

All of them have the same syntax as the `addPluginToMenu()` method.

Adding your plugin menu to one of those predefined method is recommended to keep consistency in how plugin entries are organized. However, you can add your custom menu group directly to the menu bar, as the next example demonstrates:

```
def initGui(self):
    self.menu = QMenu(self.iface.mainWindow())
    self.menu.setTitle("MyMenu")

    self.action = QAction(QIcon(":/plugins/testplug/icon.png"), "Test plugin", \
      self.iface.mainWindow())
    self.action.setWhatsThis("Configuration for test plugin")
    self.action.setStatusTip("This is status tip")
    QObject.connect(self.action, SIGNAL("triggered()"), self.run)
    self.menu.addAction(self.action)

    menuBar = self.iface.mainWindow().menuBar()
    menuBar.insertMenu(self.iface.firstRightStandardMenu().menuAction(), self.menu)

def unload(self):
    self.menu.deleteLater()
```

12.2.3 Resource File

You can see that in `initGui()` we've used an icon from the resource file (called `resources.qrc` in our case):

```
<RCC>
  <qresource prefix="/plugins/testplug" >
     <file>icon.png</file>
  </qresource>
</RCC>
```

It is good to use a prefix that will not collide with other plugins or any parts of QGIS, otherwise you might get resources you did not want. Now you just need to generate a Python file that will contain the resources. It's done with **pyrcc4** command:

```
pyrcc4 -o resources.py resources.qrc
```

And that's all... nothing complicated :) If you've done everything correctly you should be able to find and load your plugin in the plugin manager and see a message in console when toolbar icon or appropriate menu item is selected.

When working on a real plugin it's wise to write the plugin in another (working) directory and create a makefile which will generate UI + resource files and install the plugin to your QGIS installation.

12.3 Documentation

The documentation for the plugin can be written as HTML help files. The `qgis.utils` module provides a function, `showPluginHelp()` which will open the help file browser, in the same way as other QGIS help.

The `showPluginHelp`()` function looks for help files in the same directory as the calling module. It will look for, in turn, `index-ll_cc.html`, `index-ll.html`, `index-en.html`, `index-en_us.html` and `index.html`, displaying whichever it finds first. Here `ll_cc` is the QGIS locale. This allows multiple translations of the documentation to be included with the plugin.

The `showPluginHelp()` function can also take parameters packageName, which identifies a specific plugin for which the help will be displayed, filename, which can replace "index" in the names of files being searched, and section, which is the name of an html anchor tag in the document on which the browser will be positioned.

CHAPTER 13

IDE settings for writing and debugging plugins

Although each programmer has his prefered IDE/Text editor, here are some recommendations for setting up popular IDE's for writing and debugging QGIS Python plugins.

13.1 A note on configuring your IDE on Windows

On Linux there is no additional configuration needed to develop plug-ins. But on Windows you need to make sure you that you have the same environment settings and use the same libraries and interpreter as QGIS. The fastest way to do this, is to modify the startup batch file of QGIS.

If you used the OSGeo4W Installer, you can find this under the bin folder of your OSGoeW install. Look for something like `C:\OSGeo4W\bin\qgis-unstable.bat`.

For using Pyscripter IDE, here's what ou have to do:

- Make a copy of qgis-unstable.bat and rename it pyscripter.bat.
- Open it in an editor. And remove the last line, the one that starts qgis.
- Add a line that points to the your pyscripter executable and add the commandline argument that sets the version of python to be used (2.7 in the case of QGIs 2.0)
- Also add the argument that points to the folder where pyscripter can find the python dll used by qgis, you can find this under the bin folder of your OSGeoW install:

```
@echo off
SET OSGEO4W_ROOT=C:\OSGeo4W
call "%OSGEO4W_ROOT%"\bin\o4w_env.bat
call "%OSGEO4W_ROOT%"\bin\gdal16.bat
@echo off
path %PATH%;%GISBASE%\bin
Start C:\pyscripter\pyscripter.exe --python25 --pythondllpath=C:\OSGeo4W\bin
```

Now when you double click this batch file it will start pyscripter, with the correct path.

More popular that Pyscripter, Eclipse is a common choice among developers. In the following sections, we will be explaining how to configure it for depelopping and testing plugins. To prepare your environment for using Eclipse in windows, you should also create a batch file and use it to start Eclipse.

To create that batch file, follow these steps.

- Locate the folder where `qgis_core.dll` resides in. Normally this is `C:\OSGeo4W\apps\qgis\bin`, but if you compiled your own qgis application this is in your build folder in `output/bin/RelWithDebInfo`
- Locate your eclipse.exe executable.
- Create the following script and use this to start eclipse when developping QGIS plugins.

```
call "C:\OSGeo4W\bin\o4w_env.bat"
set PATH=%PATH%;C:\path\to\your\qgis_core.dll\parent\folder
C:\path\to\your\eclipse.exe
```

13.2 Debugging using Eclipse and PyDev

13.2.1 Installation

To use Eclipse, make sure you have installed the following

- Eclipse
- Aptana Eclipse Plugin or PyDev
- QGIS 2.0

13.2.2 Preparing QGIS

There is some preparation to be done on QGIS itself. Two plugins are of interest: *Remote Debug* and *Plugin reloader*.

- Go to *Plugins/Fetch python plugins*
- Search for Remote Debug (at the moment it's still experimental, so enable experimental plugins under the Options tab in case it does not show up). Install it.
- Search for *Plugin reloader* and install it as well. This will let you reload a plugin instead of having to close and restart QGIS to have the plugin reloaded.

13.2.3 Setting up Eclipse

In Eclipse, create a new project. You can select *General Project* and link your real sources later on, so it does not really mather where you place this project.

Now right click your new project and choose *New => Folder*.

Click *Advanced* and choose *Link to alternate location (Linked Folder)*. In case you already have sources you want to debug, choose these, in case you don't, create a folder as it was already explained

Now in the view *Project Explorer*, your source tree pops up and you can start working with the code. You already have syntax highlighting and all the other powerful IDE tools available.

13.2.4 Configuring the debugger

To get the debugger working, switch to the Debug perspective in eclipse (*Window=>Open Perspective=>Other=>Debug*).

Now start the PyDev debug server by choosing *PyDev=>Start Debug Server*.

Eclipse is now waiting for a connection from QGIS to its debug server and when QGIS connects to the debug server it will allow it to control the python scripts. That's exactly what we installed the Remote Debug plugin for. So start QGIS in case you did not already and click the bug symbol .

Now you can set a breakpoint and as soon as the code hits it, execution will stop and you can inspect the current state of your plugin. (The breakpoint is the green dot in the image below, set one by double clicking in the white space left to the line you want the breakpoint to be set)

A very interesting thing you can make use of now is the debug console. Make sure that the execution is currently stopped at a break point, before you proceed.

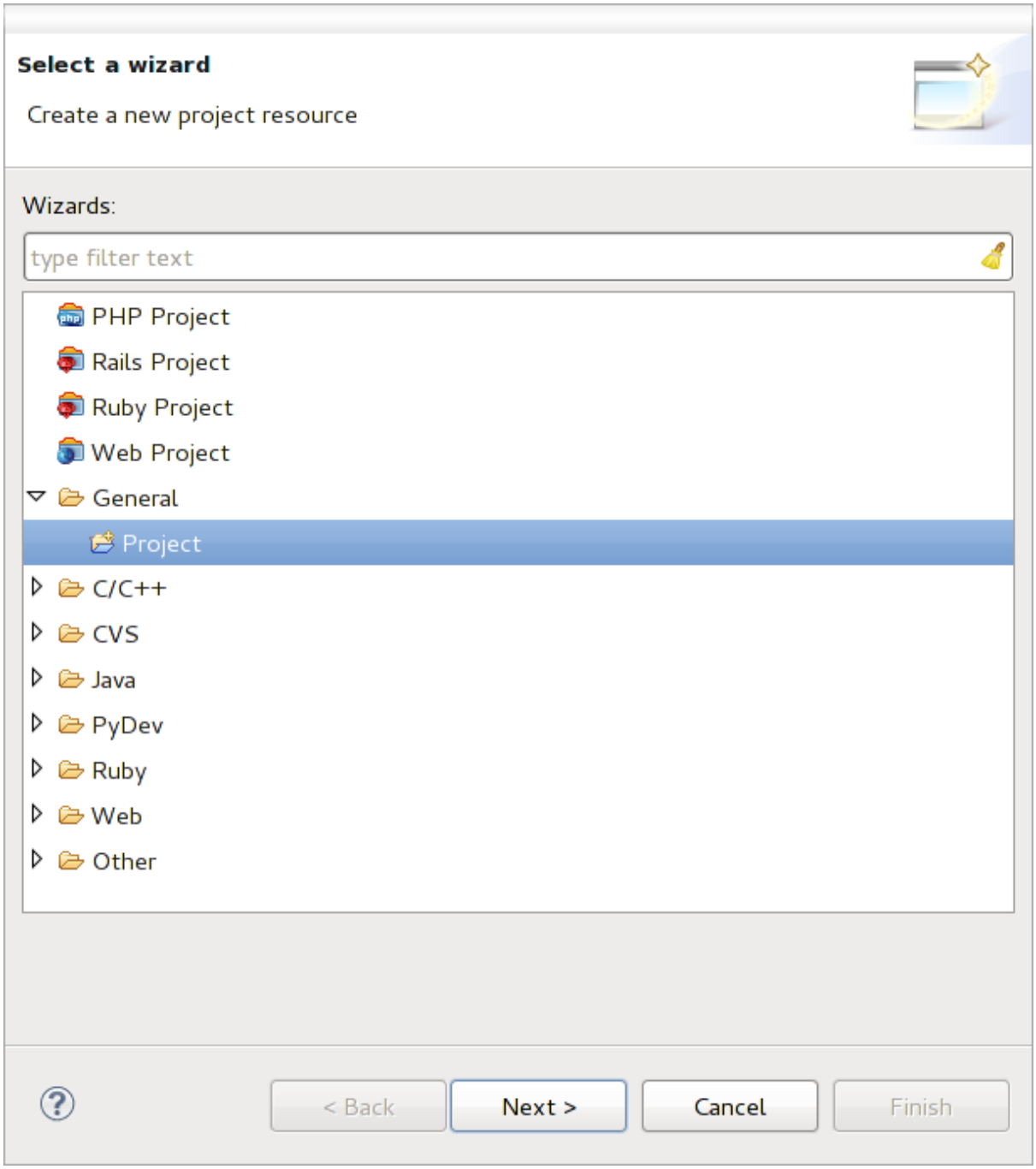

Figure 13.1: Eclipse project

```
88
89      def printProfile(self):
90          printer = QPrinter( QPrinter.HighResolution )
91          printer.setOutputFormat( QPrinter.PdfFormat )
92          printer.setPaperSize( QPrinter.A4 )
93          printer.setOrientation( QPrinter.Landscape )
94
95          printPreviewDlg = QPrintPreviewDialog( )
96          printPreviewDlg.paintRequested.connect( self.printRequested )
97
98          printPreviewDlg.exec_()
99
100     @pyqtSlot( QPrinter )
101     def printRequested( self, printer ):
102         self.webView.print_( printer )
```

Figure 13.2: Breakpoint

Open the Console view (*Window => Show view*). It will show the Debug Server console which is not very interesting. But there is a button *Open Console* which lets you change to a more interesting PyDev Debug Console. Click the arrow next to the Open Console button and choose *PyDev Console*. A window opens up to ask you which console you want to start. Choose *PyDev Debug Console*. In case its greyed out and tells you to Start the debugger and select the valid frame, make sure that you've got the remote debugger attached and are currently on a breakpoint.

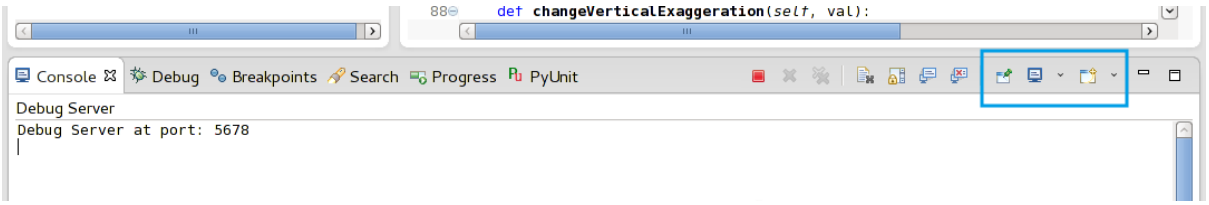

Figure 13.3: PyDev Debug Console

You have now an interactive console which let's you test any commands from within the current context. You can manipulate variables or make API calls or whatever you like.

A little bit annoying is, that everytime you enter a command, the console switches back to the Debug Server. To stop this behavior, you can click the *Pin Console* button when on the Debug Server page and it should remember this decision at least for the current debug session.

13.2.5 Making eclipse understand the API

A very handy feature is to have Eclipse actually know about the QGIS API. This enables it to check your code for typos. But not only this, it also enables Eclipse to help you with autocompletion from the imports to API calls.

To do this, Eclipse parses the QGIS library files and gets all the information out there. The only thing you have to do is to tell Eclipse where to find the libraries.

Click *Window=>Preferences=>PyDev=>Interpreter - Python*.

You will see your configured python interpreter in the upper part of the window (at the moment python2.7 for QGIS) and some tabs in the lower part. The interesting tabs for us are *Libraries* and *Forced Builtins*.

First open the Libraries tab. Add a New Folder and choose the python folder of your QGIS installation. If you do not know where this folder is (it's not the plugins folder) open QGIS, start a python console and simply enter `qgis` and press enter. It will show you which qgis module it uses and its path. Strip the trailing `/qgis/__init__.pyc` from this path and you've got the path you are looking for.

You should also add your plugins folder here (on linux its ~/.qgis/python/plugins).

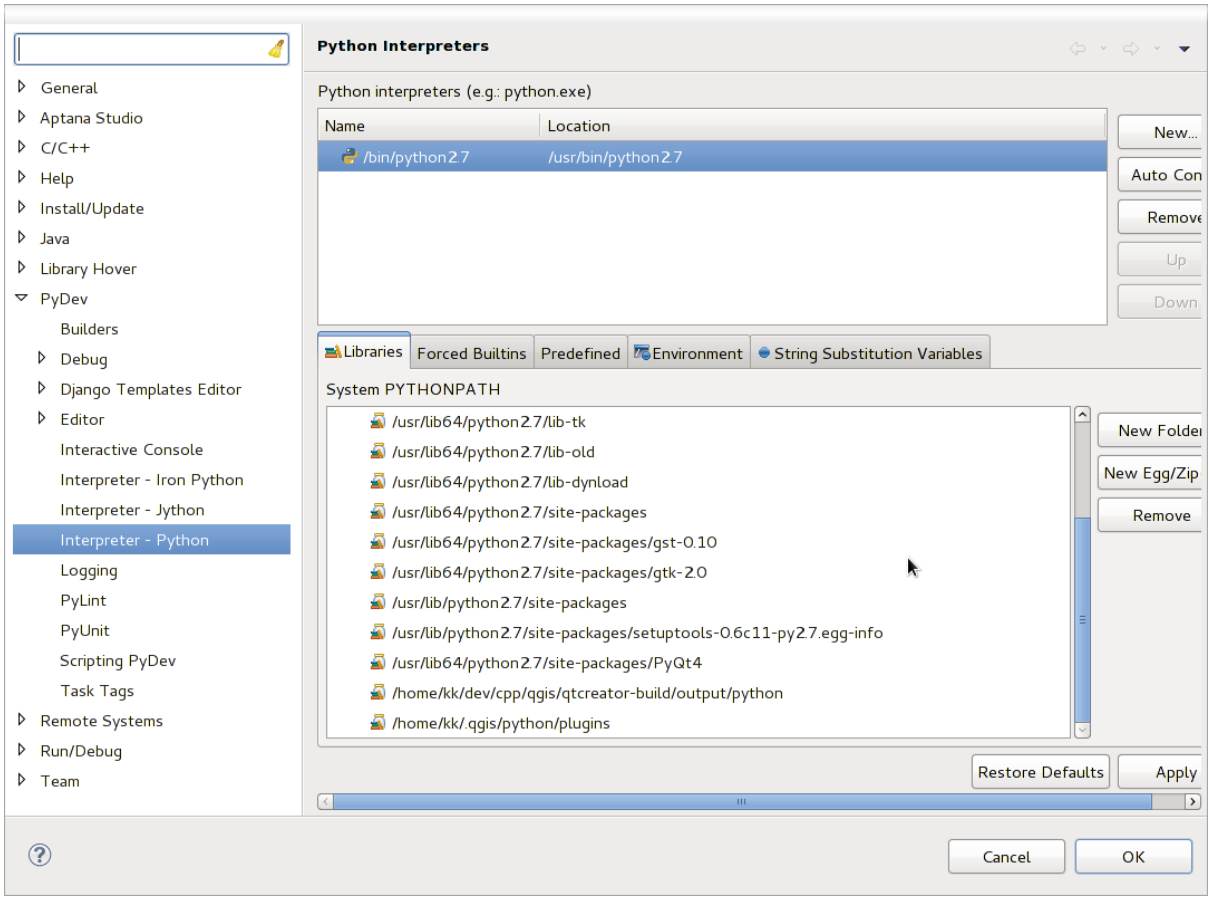

Figure 13.4: PyDev Debug Console

Next jump to the *Forced Builtins* tab, click on *New...* and enter `qgis`. This will make eclipse parse the QGIS API. You probably also want eclipse to know about the PyQt4 API. Therefore also add PyQt4 as forced builtin. That should probably already be present in your libraries tab

Click *OK* and you're done.

Note: everytime the QGIS API changes (e.g. if you're compiling QGIS master and the sip file changed), you should go back to this page and simply click *Apply*. This will let Eclipse parse all the libraries again.

For another possible setting of Eclipse to work with QGIS Python plugins, check this link

13.3 Debugging using PDB

If you do not use an IDE such as Eclipse, you can debug using PDB, following this steps.

First add this code in the spot where you would like to debug:

```
# Use pdb for debugging
import pdb
# These lines allow you to set a breakpoint in the app
pyqtRemoveInputHook()
pdb.set_trace()
```

Then run QGIS from the command line.

On Linux do:

$ ./Qgis

On Mac OS X do:

$ /Applications/Qgis.app/Contents/MacOS/Qgis

And when the application hits your breakpoint you can type in the console!

CHAPTER 14

Using Plugin Layers

If your plugin uses its own methods to render a map layer, writing your own layer type based on QgsPluginLayer might be the best way to implement that.

TODO: Check correctness and elaborate on good use cases for QgsPluginLayer, ...

14.1 Subclassing QgsPluginLayer

Below is an example of a minimal QgsPluginLayer implementation. It is an excerpt of the Watermark example plugin:

```
class WatermarkPluginLayer(QgsPluginLayer):

  LAYER_TYPE="watermark"

  def __init__(self):
    QgsPluginLayer.__init__(self, WatermarkPluginLayer.LAYER_TYPE, \
      "Watermark plugin layer")
    self.setValid(True)

  def draw(self, rendererContext):
    image = QImage("myimage.png")
    painter = rendererContext.painter()
    painter.save()
    painter.drawImage(10, 10, image)
    painter.restore()
    return True
```

Methods for reading and writing specific information to the project file can also be added:

```
def readXml(self, node):

def writeXml(self, node, doc):
```

When loading a project containing such a layer, a factory class is needed:

```
class WatermarkPluginLayerType(QgsPluginLayerType):

  def __init__(self):
    QgsPluginLayerType.__init__(self, WatermarkPluginLayer.LAYER_TYPE)

  def createLayer(self):
    return WatermarkPluginLayer()
```

You can also add code for displaying custom information in the layer properties:

```
def showLayerProperties(self, layer):
```

CHAPTER 15

Compatibility with older QGIS versions

15.1 Plugin menu

If you place your plugin menu entries into one of the new menus (*Raster*, *Vector*, *Database* or *Web*), you should modify the code of the `initGui()` and `unload()` functions. Since these new menus are available only in QGIS 2.0, the first step is to check that the running QGIS version has all necessary functions. If the new menus are available, we will place our plugin under this menu, otherwise we will use the old *Plugins* menu. Here is an example for *Raster* menu:

```
def initGui(self):
  # create action that will start plugin configuration
  self.action = QAction(QIcon(":/plugins/testplug/icon.png"), "Test plugin", \
    self.iface.mainWindow())
  self.action.setWhatsThis("Configuration for test plugin")
  self.action.setStatusTip("This is status tip")
  QObject.connect(self.action, SIGNAL("triggered()"), self.run)

  # check if Raster menu available
  if hasattr(self.iface, "addPluginToRasterMenu"):
    # Raster menu and toolbar available
    self.iface.addRasterToolBarIcon(self.action)
    self.iface.addPluginToRasterMenu("&Test plugins", self.action)
  else:
    # there is no Raster menu, place plugin under Plugins menu as usual
    self.iface.addToolBarIcon(self.action)
    self.iface.addPluginToMenu("&Test plugins", self.action)

  # connect to signal renderComplete which is emitted when canvas rendering is done
  QObject.connect(self.iface.mapCanvas(), SIGNAL("renderComplete(QPainter *)"), \
    self.renderTest)

def unload(self):
  # check if Raster menu available and remove our buttons from appropriate
  # menu and toolbar
  if hasattr(self.iface, "addPluginToRasterMenu"):
    self.iface.removePluginRasterMenu("&Test plugins",self.action)
    self.iface.removeRasterToolBarIcon(self.action)
  else:
    self.iface.removePluginMenu("&Test plugins",self.action)
    self.iface.removeToolBarIcon(self.action)

  # disconnect from signal of the canvas
  QObject.disconnect(self.iface.mapCanvas(), SIGNAL("renderComplete(QPainter *)"), \
    self.renderTest)
```

CHAPTER 16

Releasing your plugin

Once your plugin is ready and you think the plugin could be helpful for some people, do not hesitate to upload it to *Official python plugin repository*. On that page you can find also packaging guidelines about how to prepare the plugin to work well with the plugin installer. Or in case you would like to set up your own plugin repository, create a simple XML file that will list the plugins and their metadata, for examples see other plugin repositories.

16.1 Official python plugin repository

You can find the *official* python plugin repository at http://plugins.qgis.org/.

In order to use the official repository you must obtain an OSGEO ID from the OSGEO web portal.

Once you have uploaded your plugin it will be approved by a staff member and you will be notified.

16.1.1 Permissions

These rules have been implemented in the official plugin repository:

- every registered user can add a new plugin
- *staff* users can approve or disapprove all plugin versions
- users which have the special permission *plugins.can_approve* get the versions they upload automatically approved
- users which have the special permission *plugins.can_approve* can approve versions uploaded by others as long as they are in the list of the plugin *owners*
- a particular plugin can be deleted and edited only by *staff* users and plugin *owners*
- if a user without *plugins.can_approve* permission uploads a new version, the plugin version is automatically unapproved.

16.1.2 Trust management

Staff members can grant *trust* to selected plugin creators setting *plugins.can_approve* permission through the front-end application.

The plugin details view offers direct links to grant trust to the plugin creator or the plugin *owners*.

16.1.3 Validation

Plugin's metadata are automatically imported and validated from the compressed package when the plugin is uploaded.

Here are some validation rules that you should aware of when you want to upload a plugin on the official repository:

1. the name of the main folder containing your plugin must contain only contains ASCII characters (A-Z and a-z), digits and the characters underscore (_) and minus (-), also it cannot start with a digit
2. `metadata.txt` is required
3. all required metadata listed in *metadata table* must be present
4. the *version* metadata field must be unique

CHAPTER 17

Code Snippets

This section features code snippets to facilitate plugin development.

17.1 How to call a method by a key shortcut

In the plug-in add to the `initGui()`:

```
self.keyAction = QAction("Test Plugin", self.iface.mainWindow())
self.iface.registerMainWindowAction(self.keyAction, "F7") # action1 triggered by F7 key
self.iface.addPluginToMenu("&Test plugins", self.keyAction)
QObject.connect(self.keyAction, SIGNAL("triggered()"),self.keyActionF7)
```

To `unload()` add:

```
self.iface.unregisterMainWindowAction(self.keyAction)
```

The method that is called when F7 is pressed:

```
def keyActionF7(self):
  QMessageBox.information(self.iface.mainWindow(),"Ok", "You pressed F7")
```

17.2 How to toggle Layers (work around)

As there is currently no method to directly access the layers in the legend, here is a workaround how to toggle the layers using layer transparency:

```
def toggleLayer(self, lyrNr):
  lyr = self.iface.mapCanvas().layer(lyrNr)
  if lyr:
    cTran = lyr.getTransparency()
    lyr.setTransparency(0 if cTran > 100 else 255)
    self.iface.mapCanvas().refresh()
```

The method requires the layer number (0 being the top most) and can be called by:

```
self.toggleLayer(3)
```

17.3 How to access attribute table of selected features

```
def changeValue(self, value):
  layer = self.iface.activeLayer()
  if(layer):
```

```
    nF = layer.selectedFeatureCount()
    if (nF > 0):
    layer.startEditing()
    ob = layer.selectedFeaturesIds()
    b = QVariant(value)
    if (nF > 1):
      for i in ob:
      layer.changeAttributeValue(int(i),1,b) # 1 being the second column
    else:
      layer.changeAttributeValue(int(ob[0]),1,b) # 1 being the second column
    layer.commitChanges()
    else:
      QMessageBox.critical(self.iface.mainWindow(),"Error", "Please select at \
        least one feature from current layer")
  else:
    QMessageBox.critical(self.iface.mainWindow(),"Error","Please select a layer")
```

The method requires one parameter (the new value for the attribute field of the selected feature(s)) and can be called by:

```
self.changeValue(50)
```

CHAPTER 18

Network analysis library

Starting from revision ee19294562 (QGIS >= 1.8) the new network analysis library was added to the QGIS core analysis library. The library:

- creates mathematical graph from geographical data (polyline vector layers)
- implements basics method of the graph theory (currently only Dijkstra's algorithm)

Network analysis library was created by exporting basics functions from RoadGraph core plugin and now you can use it's methods in plugins or directly from Python console.

18.1 General information

Briefly typical use case can be described as:

1. create graph from geodata (usually polyline vector layer)
2. run graph analysis
3. use analysis results (for example, visualize them)

18.2 Building graph

The first thing you need to do — is to prepare input data, that is to convert vector layer into graph. All further actions will use this graph, not the layer.

As a source we can use any polyline vector layer. Nodes of the polylines become graph vertices, and segments of the polylines are graph edges. If several nodes have the same coordinates then they are the same graph vertex. So two lines that have a common node become connected to each other.

Additionally, during graph creation it is possible to "fix" ("tie") to the input vector layer any number of additional points. For each additional point a match will be found — closest graph vertex or closest graph edge. In the latter case the edge will be splitted and new vertex added.

As the properties of the edge a vector layer attributes can be used and length of the edge.

Converter from vector layer to graph is developed using Builder programming pattern. For graph construction response so-called Director. There is only one Director for now: QgsLineVectorLayerDirector. The director sets the basic settings that will be used to construct a graph from a line vector layer, used by the builder to create graph. Currently, as in the case with the director, only one builder exists: QgsGraphBuilder, that creates QgsGraph objects. You may want to implement your own builders that will build a graphs compatible with such libraries as BGL or NetworkX.

To calculate edge properties programming pattern strategy is used. For now only QgsDistanceArcProperter strategy is available, that takes into account the length of the route. You can implement your own strategy that will use all necessary parameters. For example, RoadGraph plugin uses strategy that compute travel time using edge length and speed value from attributes.

It's time to dive in the process.

First of all, to use this library we should import networkanalysis module:

```
from qgis.networkanalysis import *
```

Than create director:

```
# don't use information about road direction from layer attributes,
# all roads are treated as two-way
director = QgsLineVectorLayerDirector( vLayer, -1, '', '', '', 3 )

# use fied with index 5 as source of information about roads direction.
# unilateral roads with direct direction have attribute value "yes",
# unilateral roads with reverse direction - "1", and accordingly bilateral
# roads - "no". By default roads are treated as two-way. This
# scheme can be used with OpenStreetMap data
director = QgsLineVectorLayerDirector( vLayer, 5, 'yes', '1', 'no', 3 )
```

To construct a director we should pass vector layer, that will be used as source for graph and information about allowed movement on each road segment (unilateral or bilateral movement, direct or reverse direction). Here is full list of this parameters:

- vl — vector layer used to build graph
- directionFieldId — index of the attribute table field, where information about roads directions is stored. If -1, then don't use this info at all
- directDirectionValue — field value for roads with direct direction (moving from first line point to last one)
- reverseDirectionValue — field value for roads with reverse direction (moving from last line point to first one)
- bothDirectionValue — field value for bilateral roads (for such roads we can move from first point to last and from last to first)
- defaultDirection — default road direction. This value will be used for those roads where field directionFieldId is not set or have some value different from above.

It is necessary then to create strategy for calculating edge properties:

```
properter = QgsDistanceArcProperter()
```

And tell the director about this strategy:

```
director.addProperter( properter )
```

Now we can create builder, which will create graph. QgsGraphBuilder constructor takes several arguments:

- crs — coordinate reference system to use. Mandatory argument.
- otfEnabled — use "on the fly" reprojection or no. By default const:*True* (use OTF).
- topologyTolerance — topological tolerance. Default value is 0.
- ellipsoidID — ellipsoid to use. By default "WGS84".

```
# only CRS is set, all other values are defaults
builder = QgsGraphBuilder( myCRS )
```

Also we can set several points, which will be used in analysis. For example:

```
startPoint = QgsPoint( 82.7112, 55.1672 )
endPoint = QgsPoint( 83.1879, 54.7079 )
```

Now all is in place so we can build graph and "tie" points to it:

```
tiedPoints = director.makeGraph( builder, [ startPoint, endPoint ] )
```

Building graph can take some time (depends on number of features in a layer and layer size). tiedPoints is a list with coordinates of "tied" points. When build operation is finished we can get graph and use it for the analysis:

```
graph = builder.graph()
```

With the next code we can get indexes of our points:

```
startId = graph.findVertex( tiedPoints[ 0 ] )
endId = graph.findVertex( tiedPoints[ 1 ] )
```

18.3 Graph analysis

Networks analysis is used to find answers on two questions: which vertices are connected and how to find a shortest path. To solve this problems network analysis library provides Dijkstra's algorithm.

Dijkstra's algorithm finds the best route from one of the vertices of the graph to all the others and the values of the optimization parameters. The results can be represented as shortest path tree.

The shortest path tree is as oriented weighted graph (or more precisely — tree) with the following properties:

- only one vertex have no incoming edges — the root of the tree
- all other vertices have only one incoming edge
- if vertex B is reachable from vertex A, then path from A to B is single available path and it is optimal (shortest) on this graph

To get shortest path tree use methods Use methods `shortestTree()` and `dijkstra()` of QgsGraphAnalyzer class. It is recommended to use method `dijkstra()` because it works faster and uses memory more efficiently.

The `shortestTree()` method is useful when you want to walk around the shortest path tree. It always creates new graph object (QgsGraph) and accepts three variables:

- source — input graph
- startVertexIdx — index of the point on the tree (the root of the tree)
- criterionNum — number of edge property to use (started from 0).

```
tree = QgsGraphAnalyzer.shortestTree( graph, startId, 0 )
```

The `dijkstra()` method has the same arguments, but returns two arrays. In the first array element i contains index of the incoming edge or -1 if there are no incoming edges. In the second array element i contains distance from the root of the tree to vertex i or DOUBLE_MAX if vertex i is unreachable from the root.

```
(tree, cost) = QgsGraphAnalyzer.dijkstra( graph, startId, 0 )
```

Here is very simple code to display shortest path tree using graph created with `shortestTree()` method (select linestring layer in TOC and replace coordinates with yours one). **Warning**: use this code only as an example, it creates a lots of QgsRubberBand objects and may be slow on large datasets.

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

from qgis.core import *
from qgis.gui import *
from qgis.networkanalysis import *

vl = qgis.utils.iface.mapCanvas().currentLayer()
director = QgsLineVectorLayerDirector( vl, -1, '', '', '', 3 )
properter = QgsDistanceArcProperter()
director.addProperter( properter )
crs = qgis.utils.iface.mapCanvas().mapRenderer().destinationCrs()
builder = QgsGraphBuilder( crs )
```

```
pStart = QgsPoint( -0.743804, 0.22954 )
tiedPoint = director.makeGraph( builder, [ pStart ] )
pStart = tiedPoint[ 0 ]

graph = builder.graph()

idStart = graph.findVertex( pStart )

tree = QgsGraphAnalyzer.shortestTree( graph, idStart, 0 )

i = 0;
while ( i < tree.arcCount() ):
  rb = QgsRubberBand( qgis.utils.iface.mapCanvas() )
  rb.setColor ( Qt.red )
  rb.addPoint ( tree.vertex( tree.arc( i ).inVertex() ).point() )
  rb.addPoint ( tree.vertex( tree.arc( i ).outVertex() ).point() )
  i = i + 1
```

Same thing but using `dijkstra()` method:

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

from qgis.core import *
from qgis.gui import *
from qgis.networkanalysis import *

vl = qgis.utils.iface.mapCanvas().currentLayer()
director = QgsLineVectorLayerDirector( vl, -1, '', '', '', 3 )
properter = QgsDistanceArcProperter()
director.addProperter( properter )
crs = qgis.utils.iface.mapCanvas().mapRenderer().destinationCrs()
builder = QgsGraphBuilder( crs )

pStart = QgsPoint( -1.37144, 0.543836 )
tiedPoint = director.makeGraph( builder, [ pStart ] )
pStart = tiedPoint[ 0 ]

graph = builder.graph()

idStart = graph.findVertex( pStart )

( tree, costs ) = QgsGraphAnalyzer.dijkstra( graph, idStart, 0 )

for edgeId in tree:
  if edgeId == -1:
    continue
  rb = QgsRubberBand( qgis.utils.iface.mapCanvas() )
  rb.setColor ( Qt.red )
  rb.addPoint ( graph.vertex( graph.arc( edgeId ).inVertex() ).point() )
  rb.addPoint ( graph.vertex( graph.arc( edgeId ).outVertex() ).point() )
```

18.3.1 Finding shortest path

To find optimal path between two points the following approach is used. Both points (start A and end B) are "tied" to graph when it builds. Than using methods `shortestTree()` or `dijkstra()` we build shortest tree with root in the start point A. In the same tree we also found end point B and start to walk through tree from point B to point A. Whole algorithm can be written as:

```
assign  = B
while  != A
    add point  to path
```

```
    get incoming edge for point
    look for point , that is start point of this edge
    assign  =
add point  to path
```

At this point we have path, in the form of the inverted list of vertices (vertices are listed in reversed order from end point to start one) that will be visited during traveling by this path.

Here is the sample code for QGIS Python Console (you will need to select linestring layer in TOC and replace coordinates in the code with yours) that uses method `shortestTree()`:

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

from qgis.core import *
from qgis.gui import *
from qgis.networkanalysis import *

vl = qgis.utils.iface.mapCanvas().currentLayer()
director = QgsLineVectorLayerDirector( vl, -1, '', '', '', 3 )
properter = QgsDistanceArcProperter()
director.addProperter( properter )
crs = qgis.utils.iface.mapCanvas().mapRenderer().destinationCrs()
builder = QgsGraphBuilder( crs )

pStart = QgsPoint( -0.835953, 0.15679 )
pStop = QgsPoint( -1.1027, 0.699986 )

tiedPoints = director.makeGraph( builder, [ pStart, pStop ] )
graph = builder.graph()

tStart = tiedPoints[ 0 ]
tStop = tiedPoints[ 1 ]

idStart = graph.findVertex( tStart )
tree = QgsGraphAnalyzer.shortestTree( graph, idStart, 0 )

idStart = tree.findVertex( tStart )
idStop = tree.findVertex( tStop )

if idStop == -1:
  print "Path not found"
else:
  p = []
  while ( idStart != idStop ):
    l = tree.vertex( idStop ).inArc()
    if len( l ) == 0:
      break
    e = tree.arc( l[ 0 ] )
    p.insert( 0, tree.vertex( e.inVertex() ).point() )
    idStop = e.outVertex()

  p.insert( 0, tStart )
  rb = QgsRubberBand( qgis.utils.iface.mapCanvas() )
  rb.setColor( Qt.red )

  for pnt in p:
    rb.addPoint(pnt)
```

And here is the same sample but using `dikstra()` method:

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

```
from qgis.core import *
from qgis.gui import *
from qgis.networkanalysis import *

vl = qgis.utils.iface.mapCanvas().currentLayer()
director = QgsLineVectorLayerDirector( vl, -1, '', '', '', 3 )
properter = QgsDistanceArcProperter()
director.addProperter( properter )
crs = qgis.utils.iface.mapCanvas().mapRenderer().destinationCrs()
builder = QgsGraphBuilder( crs )

pStart = QgsPoint( -0.835953, 0.15679 )
pStop = QgsPoint( -1.1027, 0.699986 )

tiedPoints = director.makeGraph( builder, [ pStart, pStop ] )
graph = builder.graph()

tStart = tiedPoints[ 0 ]
tStop = tiedPoints[ 1 ]

idStart = graph.findVertex( tStart )
idStop = graph.findVertex( tStop )

( tree, cost ) = QgsGraphAnalyzer.dijkstra( graph, idStart, 0 )

if tree[ idStop ] == -1:
  print "Path not found"
else:
  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 )

  rb = QgsRubberBand( qgis.utils.iface.mapCanvas() )
  rb.setColor( Qt.red )

  for pnt in p:
    rb.addPoint(pnt)
```

18.3.2 Areas of the availability

Area of availability for vertex A is a subset of graph vertices, that are accessible from vertex A and cost of the path from A to this vertices are not greater that some value.

More clearly this can be shown with the following example: “There is a fire station. What part of city fire command can reach in 5 minutes? 10 minutes? 15 minutes?”. Answers on this questions are fire station’s areas of availability.

To find areas of availablity we can use method `dijksta()` of the `QgsGraphAnalyzer` class. It is enough to compare elements of cost array with predefined value. If cost[i] is less or equal than predefined value, than vertex i is inside area of availability, otherwise — outside.

More difficult it is to get borders of area of availablity. Bottom border — is a set of vertices that are still accessible, and top border — is a set of vertices which are not accesible. In fact this is simple: availability border passed on such edges of the shortest path tree for which start vertex is accessible and end vertex is not accessible.

Here is an example:

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

from qgis.core import *
from qgis.gui import *
from qgis.networkanalysis import *

vl = qgis.utils.iface.mapCanvas().currentLayer()
director = QgsLineVectorLayerDirector( vl, -1, '', '', '', 3 )
properter = QgsDistanceArcProperter()
director.addProperter( properter )
crs = qgis.utils.iface.mapCanvas().mapRenderer().destinationCrs()
builder = QgsGraphBuilder( crs )

pStart = QgsPoint( 65.5462, 57.1509 )
delta = qgis.utils.iface.mapCanvas().getCoordinateTransform().mapUnitsPerPixel() * 1

rb = QgsRubberBand( qgis.utils.iface.mapCanvas(), True )
rb.setColor( Qt.green )
rb.addPoint( QgsPoint( pStart.x() - delta, pStart.y() - delta ) )
rb.addPoint( QgsPoint( pStart.x() + delta, pStart.y() - delta ) )
rb.addPoint( QgsPoint( pStart.x() + delta, pStart.y() + delta ) )
rb.addPoint( QgsPoint( pStart.x() - delta, pStart.y() + delta ) )

tiedPoints = director.makeGraph( builder, [ pStart ] )
graph = builder.graph()
tStart = tiedPoints[ 0 ]

idStart = graph.findVertex( tStart )

( tree, cost ) = QgsGraphAnalyzer.dijkstra( graph, idStart, 0 )

upperBound = []
r = 2000.0
i = 0
while i < len(cost):
  if cost[ i ] > r and tree[ i ] != -1:
    outVertexId = graph.arc( tree [ i ] ).outVertex()
    if cost[ outVertexId ] < r:
      upperBound.append( i )
  i = i + 1

for i in upperBound:
  centerPoint = graph.vertex( i ).point()
  rb = QgsRubberBand( qgis.utils.iface.mapCanvas(), True )
  rb.setColor( Qt.red )
  rb.addPoint( QgsPoint( centerPoint.x() - delta, centerPoint.y() - delta ) )
  rb.addPoint( QgsPoint( centerPoint.x() + delta, centerPoint.y() - delta ) )
  rb.addPoint( QgsPoint( centerPoint.x() + delta, centerPoint.y() + delta ) )
  rb.addPoint( QgsPoint( centerPoint.x() - delta, centerPoint.y() + delta ) )
```