PyQGIS Documentation

Release 1.4

Martin Dobias

December 06, 2010

CONTENTS

Contents:

CHAPTER

ONE

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:

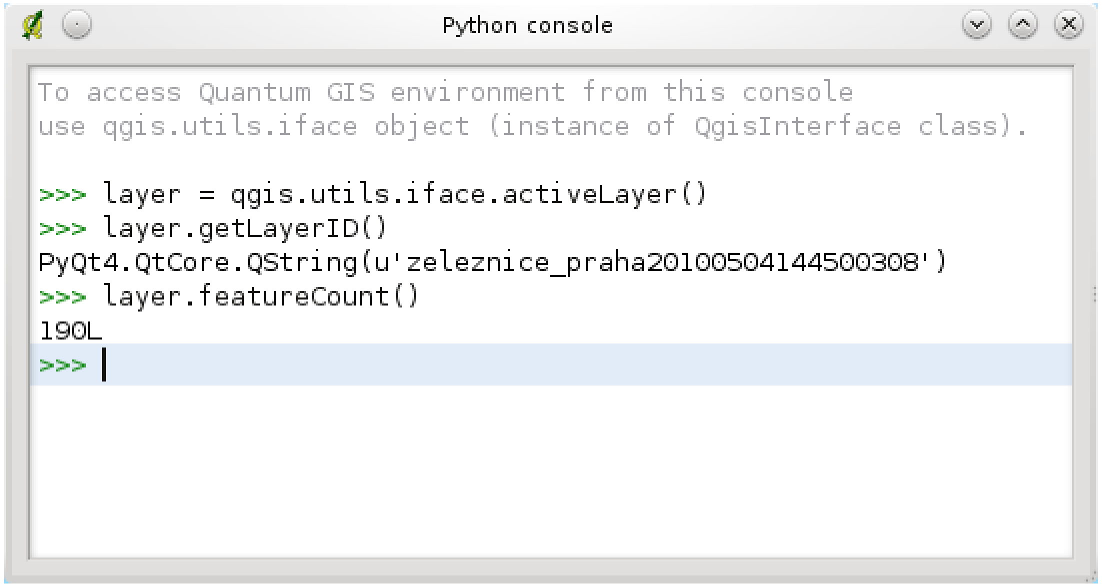
- issue commands in Python console within QGIS
- create and use plugins in Python
- create custom applications based on QGIS API

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.

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:



```
To access Quantum GIS environment from this console
use qgis.utils.iface object (instance of QgisInterface class).

>>> layer = qgis.utils.iface.activeLayer()
>>> layer.getLayerID()
PyQt4.QtCore.QString(u'zeleznice_praha20100504144500308')
>>> layer.featureCount()
190L
>>> 
```

The screenshot above illustrates how to get the layer currntly 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 `qgis.utils.iface` variable, which is instance of `QgisInterface`. 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...*)

1.2 Python Plugins

Quantum GIS 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. Great advantages over C++ plugins is the 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. Plugin installer allows users to easily fetch, upgrade and remove Python plugins. See 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

TWO

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")
  ```

- 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')
  uri.setDataSource('','Towns', 'Geometry')

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

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!"
```

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:

```
url = 'http://wms.jpl.nasa.gov/wms.cgi'
layers = [ 'global_mosaic' ]
styles = [ 'pseudo' ]
format = 'image/jpeg'
crs = 'EPSG:4326'
rlayer = QgsRasterLayer(0, url, 'some layer name', 'wms', layers, styles, format, crs)
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

THREE

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().toString()
PyQt4.QtCore.QString(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()
PyQt4.QtCore.QString(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	PalettedSingleBandPseudoColor	“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	MultiBandSingleBandPseudoColor	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 *Refresing Layers*.

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

3.3 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(25
>>> fcn = rlayer.rasterShader().rasterShaderFunction()
>>> fcn.setColorRampType(QgsColorRampShader.INTERPOLATED)
>>> fcn.setColorRampItemList(lst)
```

To return back to default gray levels, use:

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

3.4 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
```

3.5 Refresing 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.

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)
```

3.6 Query Values

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

```
ident = rlayer.identify(QgsPoint(15.30,40.98))
for (k,v) in ident.iteritems():
  print str(k),":",str(v)
```

The identify function returns a dictionary - keys are band names, values are the values at chosen point. Both key and value are QString instances so to see actual value you'll need to convert them to python strings (as shown in code snippet).

CHAPTER

FOUR

USING VECTOR LAYERS

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

TODO: Editing, Layer vs. Data provider, ...

4.1 Iterating over Vector Layer

Below is an example how to go through the features of the layer. To read features from layer, initialize the retieval with `select()` and then use `nextFeature()` calls:

```
provider = vlayer.dataProvider()

feat = QgsFeature()
allAttrs = provider.attributeIndexes()

# start data retreival: fetch geometry and all attributes for each feature
provider.select(allAttrs)

# retreive every feature with its geometry and attributes
while provider.nextFeature(feat):

  # fetch geometry
  geom = feat.geometry()
  print "Feature ID %d: " % feat.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 map of attributes
  attrs = feat.attributeMap()

  # attrs is a dictionary: key = field index, value = QgsFeatureAttribute
  # show all attributes and their values
  for (k,attr) in attrs.iteritems():
    print "%d: %s" % (k, attr.toString())
```

`select()` gives you flexibility in what data will be fetched. It can get 4 arguments, all of them are optional:

1. **fetchAttributes** List of attributes which should be fetched. Default: empty list
2. **rect** Spatial filter. If empty rect is given (`QgsRect()`), all features are fetched. Default: empty rect
3. **fetchGeometry** Whether geometry of the feature should be fetched. Default: `True`
4. **useIntersect** When using spatial filter, this argument says whether accurate test for intersection should be done or whether test on bounding box suffices. This is needed e.g. for feature identification or selection. Default: `False`

Some examples:

```
# fetch features with geometry and only first two fields
provider.select([0,1])

# fetch features with geometry which are in specified rect, attributes won't be retreived
provider.select([], QgsRectangle(23.5, -10, 24.2, -7))

# fetch features without geometry, with all attributes
allAtt = provider.attributeIndexes()
provider.select(allAtt, QgsRectangle(), False)
```

To obtain field index from its name, use provider's `fieldNameIndex()` function:

```
fldDesc = provider.fieldNameIndex("DESCRIPTION")
if fldDesc == -1:
  print "Field not found!"
```

4.2 Using Spatial Index

TODO: Intro to spatial indexing

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(QgsRect(22.5, 15.3, 23.1, 17.2))
```

4.3 Writing Shapefiles

You can write shapefiles using `QgsVectorFileWriter` class. Besides shapefiles, it supports any kind of vector file that OGR supports.

There are two possibilities how to export a shapefile:

- from an instance of `QgsVectorLayer`:

```
error = QgsVectorFileWriter.writeAsShapefile(layer, "my_shapes.shp", "CP1250")

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

- directly from features:

```
# define fields for feature attributes
fields = { 0 : QgsField("first", QVariant.Int),
           1 : QgsField("second", QVariant.String) }

# create an instance of vector file writer, it will create the shapefile. Arguments:
# 1. path to new shapefile (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
writer = QgsVectorFileWriter("my_shapes.shp", "CP1250", fields, QGis.WKBPoint, None)

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

# add some features
fet = QgsFeature()
fet.setGeometry(QgsGeometry.fromPoint(QgsPoint(10,10)))
fet.addAttribute(0, QVariant(1))
fet.addAttribute(1, QVariant("text"))
writer.addFeature(fet)

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

4.4 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 (until now one had to use e.g. OGR provider). You can use it by passing `"memory"` provider string to vector layer constructor.

Following URIs are allowed: “Point” / “LineString” / “Polygon” / “MultiPoint” / “MultiLineString” / “MultiPolygon” - for different types of data stored in the layer.

The provider supports string, int and double fields.

Following example should be self-explaining:

```
# 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.setAttributeMap( { 0 : QVariant("Johny"),
                       1 : QVariant(20),
                       2 : QVariant(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:", pr.fieldCount()
print "features:", pr.featureCount()
e = pr.extent()
print "extent:", e.xMin(),e.yMin(),e.xMax(),e.yMax()

# iterate over features
f = QgsFeature()
pr.select()
while pr.nextFeature(f):
  print "F:",f.id(), f.attributeMap(), f.geometry().asPoint()
```

Memory provider also supports spatial indexing. This means you can call provider's `createSpatialIndex()` function. Once the spatial index is created (using `QgsSpatialIndex` class), 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).

4.5 Controlling Symbology of Vector Layers

When a vector layer is being rendered, the appearance of the data is given by one or more symbols. A symbol determines color, size and other properties of the feature. Renderer associated with the layer decides what symbol will be used for particular feature. There are four available renderers:

- single symbol renderer (`QgsSingleSymbolRenderer`) — all features are rendererd with the same symbol.
- unique value renderer (`QgsUniqueValueRenderer`) — symbol for each feature is choosen from attribute value.
- graduated symbol renderer (`QgsGraduatedSymbolRenderer`)
- continuous color renderer (`QgsContinuousSymbolRenderer`)

How to create a point symbol:

```
sym = QgsSymbol(QGis.Point)
sym.setColor(Qt.black)
sym.setFillColor(Qt.green)
sym.setFillStyle(Qt.SolidPattern)
sym.setLineWidth(0.3)
sym.setPointSize(3)
sym.setNamedPointSymbol("hard:triangle")
```

The `setNamedPointSymbol()` method determines the shape of the symbol. There are two classes: hardcoded symbols (prefixed `hard:`) and SVG symbols (prefixed `svg:`). The following hardcoded symbols are available: `circle`, `rectangle`, `diamond`, `pentagon`, `cross`, `cross2`, `triangle`, `equilateral_triangle`, `star`, `regular_star`, `arrow`.

How to create an SVG symbol:

```
sym = QgsSymbol(QGis.Point)
sym.setNamedPointSymbol("svg:Star1.svg")
sym.setPointSize(3)
```

SVG symbols do not support setting colors, fill and line styles.

How to create a line symbol:

```
TODO
```

How to create a fill symbol:

```
TODO
```

Create a single symbol renderer:

```
sr = QgsSingleSymbolRenderer(QGis.Point)
sr.addSymbol(sym)
```

Assign the renderer to a layer:

```
layer.setRenderer(sr)
```

Create unique value renderer:

```
TODO
```

Create graduated symbol renderer:

```
TODO
```

CHAPTER

FIVE

GEOMETRY HANDLING

Points, linestrings, polygons (and more complex shapes) are commonly referred to as geometries. They are represented using `QgsGeometry` class.

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

`asPoint()` returns `QgsPoint`

`asPolyline()` returns a list of `QgsPoint` items

`asPolygon()` returns a list of rings, every ring consists of a list of `QgsPoint` items. First ring is outer, subsequent rings are holes.

You can use `QgsGeometry.isMultipart()` to find out whether the feature is multipart. For multipart features there are similar accessor functions: `asMultiPoint()`, `asMultiPolyline()`, `asMultiPolygon()`.

There are several options how to create 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) ] ] )
```

- from well-known text (WKT):

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

- from well-known binary (WKB):

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

CHAPTER

SIX

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",0.017453292519943295],\
         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

SEVEN

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

7.4 Writing Custom Map Tools

TODO: how to create a map tool

7.5 Writing Custom Map Canvas Items

TODO: how to create a map canvas item

CHAPTER

EIGHT

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 = QgsMapRender()

# 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

The following piece of code renders layers from map canvas with the current extent into a PNG file. The default settings for composition are page size A4 and resolution 300 DPI (it's possible to change them).

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

# set up composition
mapRenderer = iface.mapCanvas().mapRenderer()
c = QgsComposition(mapRenderer)
c.setPlotStyle(QgsComposition.Print)

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

# add a map to the composition
x, y = 0, 0
w, h = c.paperWidth(), c.paperHeight()
composerMap = QgsComposerMap(c, x,y,w,h)
c.addItem(composerMap)

# 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")
```

TODO: Output to PDF, Loading/saving compositions, More composer items (north arrow, scale, ...)

CHAPTER

NINE

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 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`

Compatibility note: mathematical, geometry, conversion functions and power operator ^ are available from QGIS 1.4.

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

TODO: parsing, error handling

```
>>> s = QgsSearchString()
>>> s.setString("1 + 1 = 2")
True
>>> s.setString("1 + 1 =")
False
>>> s.parserErrorMsg()
PyQt4.QtCore.QString(u'syntax error, unexpected $end')
```

TODO: working with the tree, evaluation as a predicate, as a function, error handling

9.2 Evaluating Expressions

```
st = ss.tree()
if not st:
  raise ValueError, "empty expression was used"

print st.makeSearchString()

res = st.checkAgainst(fields, feature.attributeMap())

res, value = st.getValue(st, fields, feature.attributeMap(), feature.geometry())

print st.errorMsg()
```

CHAPTER

TEN

MEASURING

To compute distances or areas, use `QgsDistanceArea` class. If projections are turned off, calculations will be planar, otherwise they'll be done on ellipsoid. When ellipsoid is not set explicitly it uses WGS84 parameters for calculations.

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

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

TODO: area, planar vs. ellipsoid

CHAPTER

ELEVEN

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, ...

11.1 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

TWELVE

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__`). 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. Ready to create a plugin but no idea what to do? Python Plugin Ideas wiki page lists wishes from the community!

12.2 Creating necessary files

Here's the directory structure of our example plugin:

```
PYTHON_PLUGINS_PATH/
  testplug/
    __init__.py
    plugin.py
    resources.qrc
    resources.py
    form.ui
    form.py
```

What is the meaning of the files:

- __init__.py = The starting point of the plugin. Contains general info, version, name and main class.
- 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.

Here and there are two automated ways of creating the basic files (skeleton) of a typical QGIS Python plugin. Useful to help you start with a typical plugin.

12.3 Writing code

12.3.1 __init__.py

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

```
def name():
  return "My testing plugin"

def description():
  return "This plugin has no real use."

def version():
  return "Version 0.1"

def qgisMinimumVersion():
  return "1.0"

def authorName():
  return "Developer"

def classFactory(iface):
  # load TestPlugin class from file testplugin.py
  from testplugin import TestPlugin
  return TestPlugin(iface)
```

12.3.2 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!"
```

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

12.3.3 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 plugin manager and see a message in console when toolbar icon or appopriate 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.

12.4 Documentation

This documentation method requires Qgis version 1.5.

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 users 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.

12.5 Code Snippets

This section features code snippets to facilitate plugin development.

12.5.1 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 is triggered by the 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")
```

12.5.2 How to toggle Layers (work around)

Note: from QGIS 1.5 there is `QgsLegendInterface` class that allows some manipulation with list of layers within legend.

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)
```

12.5.3 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
  else:
    QMessageBox.critical(self.iface.mainWindow(),"Error","Please select a layer")
```

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

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

12.5.4 How to debug a plugin using PDB

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!

12.6 Testing

12.7 Releasing the plugin

Once your plugin is ready and you think the plugin could be helpful for some people, do not hesitate to upload it to PyQGIS plugin repository. On that page you can find also packaging guidelines 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.

12.8 Remark: 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`.

I will illustrate how to set up the Pyscripter IDE. Other IDE's might require a slightly different approach:

- 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, in version 1.3 of qgis this is python 2.5.
- 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 and it will start pyscripter.

CHAPTER

THIRTEEN

INDICES AND TABLES

- *Index*
- *Module Index*
- *Search Page*