PyQGIS developer cookbook

Versão 2.18

QGIS Project

25/05/2018

Contents

CHAPTER 1

Introdução

Este documento é um tutorial e um guia de referência. Ela não lista todos os possíveis casos de uso, mas dá uma boa visão geral dos principais recursos.

Desde a versão 0.9, o QGIS tem suporte opcional à scripts usando a linguagem Python. Nós decidimos pelo Python por ser ela uma das linguagens favoritas para scripts. As ligações PyQGIS dependem da SIP e da PyQt4. A razão para usar SIP ao invés da mais usada SWIG é que todo o código QGIS depende das bibliotecas Qt. As ligações Python para o Qt (PyQt) são feitas usando SIP também e isso permite uma integração perfeita do PyQGIS com PyQt.

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

- executar automaticamente código em Phyton quando o QGIS é iniciado
- emitir comandos no console do Python no QGIS
- criar e utilizar plugins em Python
- criar aplicativos personalizados com base na API do QGIS

Python bindings are also available for QGIS Server:

- starting from 2.8 release, Python plugins are also available on QGIS Server (see *Server Python Plugins*)
- starting from 2.11 version (Master at 2015-08-11), QGIS Server library has Python bindings that can be used to embed QGIS Server into a Python application.

Existe uma referência completa à API QGIS que documenta as classes das bibliotecas QGIS. A API Python é quase identica à API em C++.

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 Executa o código em Phyton quando o QGIS é iniciado

Existem dois métodos distintos de executar o código em Phyton toda vez que o QGIS é iniciado.

1.1.1 PYQGIS_STARTUP environment variable

You can run Python code just before QGIS initialization completes by setting the `PYQGIS_STARTUP` environment variable to the path of an existing Python file.

This method is something you will probably rarely need, but worth mentioning here because it is one of the several ways to run Python code within QGIS and because this code will run before QGIS initialization is complete. This method is very useful for cleaning sys.path, which may have undesireable paths, or for isolating/loading the initial environ without requiring a virt env, e.g. homebrew or MacPorts installs on Mac.

1.1.2 The `startup.py` file

Every time QGIS starts, the user's Python home directory (usually: `.qgis2/python`) is searched for a file named `startup.py`, if that file exists, it is executed by the embedded Python interpreter.

1.2 Terminal Python

Para usar scripts é possível tirar proveito do terminal Python integrado. Ele pode ser aberto a partir do menu: *Complementos* → *Terminal Python*. O terminal abre como uma janela não modal:

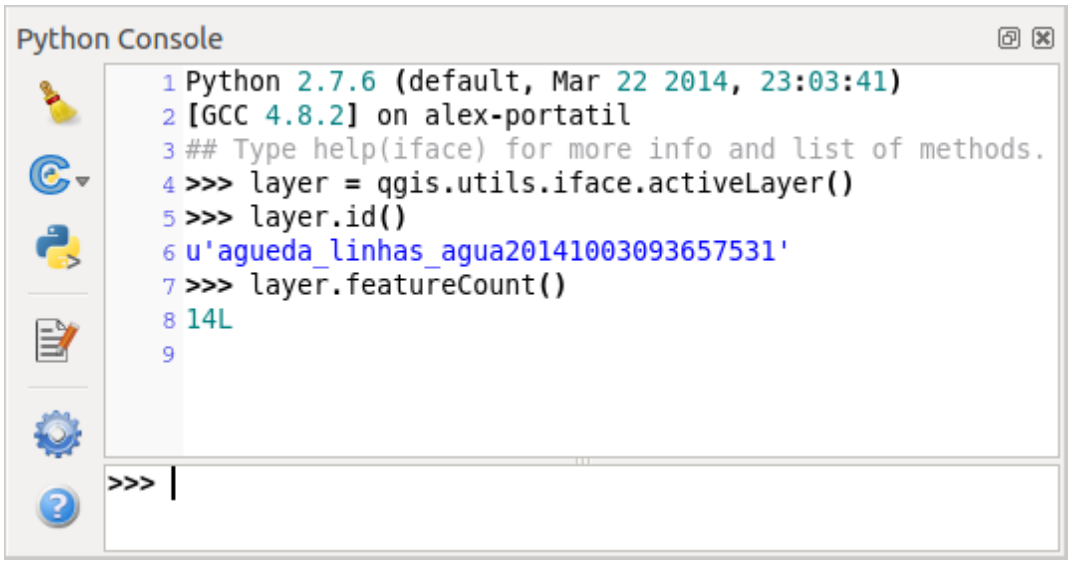


Figure 1.1: Terminal Python QGIS

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...*)

1.3 Complementos Python

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 *Desenvolvimento de Complementos Python* for detailed instructions.

Nota: Python plugins are also available in QGIS server (*label_qgisserver*), see *Complementos de servidores Python do QGIS* for further details.

1.4 Aplicações Python

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.

PyQGIS custom applications or standalone scripts must be configured to locate the QGIS resources such as projection information, providers for reading vector and raster layers, etc. QGIS Resources are initialized by adding a few lines to the beginning of your application or script. The code to initialize QGIS for custom applications and standalone scripts is similar, but examples of each are provided below.

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.

1.4.1 Using PyQGIS in standalone scripts

To start a standalone script, initialize the QGIS resources at the beginning of the script similar to the following code:

```
from qgis.core import *

# supply path to qgis install location
QgsApplication.setPrefixPath("/path/to/qgis/installation", True)

# create a reference to the QgsApplication, setting the
# second argument to False disables the GUI
qgs = QgsApplication([], False)

# load providers
qgs.initQgis()

# Write your code here to load some layers, use processing algorithms, etc.
```

```
# When your script is complete, call exitQgis() to remove the provider and
# layer registries from memory
qgs.exitQgis()
```

We begin by importing the `qgis.core` module and then configuring the prefix path. The prefix path is the location where QGIS is installed on your system. It is configured in the script by calling the `setPrefixPath` method. The second argument of `setPrefixPath` is set to `True`, which controls whether the default paths are used.

The QGIS install path varies by platform; the easiest way to find it for your your system is to use the *Terminal Python* from within QGIS and look at the output from running `QgsApplication.prefixPath()`.

After the prefix path is configured, we save a reference to `QgsApplication` in the variable `qgs`. The second argument is set to `False`, which indicates that we do not plan to use the GUI since we are writing a standalone script. With the `QgsApplication` configured, we load the QGIS data providers and layer registry by calling the `qgs.initQgis()` method. With QGIS initialized, we are ready to write the rest of the script. Finally, we wrap up by calling `qgs.exitQgis()` to remove the data providers and layer registry from memory.

1.4.2 Using PyQGIS in custom applications

The only difference between *Using PyQGIS in standalone scripts* and a custom PyQGIS application is the second argument when instantiating the `QgsApplication`. Pass `True` instead of `False` to indicate that we plan to use a GUI.

```
from qgis.core import *

# supply path to qgis install location
QgsApplication.setPrefixPath("/path/to/qgis/installation", True)

# create a reference to the QgsApplication
# setting the second argument to True enables the GUI, which we need to do
# since this is a custom application
qgs = QgsApplication([], True)

# load providers
qgs.initQgis()

# Write your code here to load some layers, use processing algorithms, etc.

# When your script is complete, call exitQgis() to remove the provider and
# layer registries from memory
qgs.exitQgis()
```

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 :-)

1.4.3 Executando aplicativos personalizados

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:

- no Linux: **export PYTHONPATH=/qgispath/share/qgis/python**
- no 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:

- no Linux: **export LD_LIBRARY_PATH=/qgispath/lib**
- no 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 macOS, for Linux leave the installation of QGIS up to user and his package manager.

CHAPTER 2

Carregando projetos

Sometimes you need to load an existing project from a plugin or (more often) when developing a stand-alone QGIS Python application (see: *Aplicações Python*).

To load a project into the current QGIS application you need a `QgsProject instance()` object and call its `read()` method passing to it a `QFileInfo` object that contains the path from where the project will be loaded:

```
# If you are not inside a QGIS console you first need to import
# qgis and PyQt4 classes you will use in this script as shown below:
from qgis.core import QgsProject
from PyQt4.QtCore import QFileInfo
# Get the project instance
project = QgsProject.instance()
# Print the current project file name (might be empty in case no projects have been loaded)
print project.fileName()
u'/home/user/projects/my_qgis_project.qgs'
# Load another project
project.read(QFileInfo('/home/user/projects/my_other_qgis_project.qgs'))
print project.fileName()
u'/home/user/projects/my_other_qgis_project.qgs'
```

In case you need to make some modifications to the project (for example add or remove some layers) and save your changes, you can call the `write()` method of your project instance. The `write()` method also accepts an optional `QFileInfo` that allows you to specify a path where the project will be saved:

```
# Save the project to the same
project.write()
# ... or to a new file
project.write(QFileInfo('/home/user/projects/my_new_qgis_project.qgs'))
```

Both `read()` and `write()` funtions return a boolean value that you can use to check if the operation was successful.

Nota: If you are writing a QGIS standalone application, in order to synchronise the loaded project with the canvas you need to instanciate a `QgsLayerTreeMapCanvasBridge` as in the example below:

```
bridge = QgsLayerTreeMapCanvasBridge( \
         QgsProject.instance().layerTreeRoot(), canvas)
# Now you can safely load your project and see it in the canvas
project.read(QFileInfo('/home/user/projects/my_other_qgis_project.qgs'))
```

CHAPTER 3

Carregando Camadas

Vamos abrir algumas camadas com dados. QGIS reconhece camadas vetoriais e matriciais. Adicionalmente, camadas personalizadas estão disponíveis, mas não discutiremos este tipo de camadas aqui.

3.1 Camadas Vetoriais

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!"
```

O identificador da fonte de dados é uma string e é especifico para cada provedor de dados vetoriais. O nome da camada é usado no painel de lista de camadas. É importante verificar se a camada foi carregada com sucesso. Se não for, uma instância de camada inválida é retornada.

The quickest way to open and display a vector layer in QGIS is the addVectorLayer function of the `QgisInterface`:

```
layer = iface.addVectorLayer("/path/to/shapefile/file.shp", "layer name you like", "ogr")
if not layer:
  print "Layer failed to load!"
```

This creates a new layer and adds it to the map layer registry (making it appear in the layer list) in one step. The function returns the layer instance or *None* if the layer couldn't be loaded.

A lista a seguir mostra como acessar várias fontes de dados usando provedores de dados vetoriais:

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

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

  - for dxf (note the internal options in data source uri):

    ```
    uri = "/path/to/dxffile/file.dxf|layername=entities|geometrytype=Point"
    vlayer = QgsVectorLayer(uri, "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 optionally
# subset (WHERE clause)
uri.setDataSource("public", "roads", "the_geom", "cityid = 2643")

vlayer = QgsVectorLayer(uri.uri(False), "layer name you like", "postgres")
```

Nota: The `False` argument passed to `uri.uri(False)` prevents the expansion of the authentication configuration parameters, if you are not using any authentication configuration this argument does not make any difference.

- 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" for 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")
```

Nota: The provider string is structured as a URL, so the path must be prefixed with `file://`. Also it allows WKT (well known text) formatted geometries 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 — 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_column)

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&re
vlayer = QgsVectorLayer(uri, "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))
```

Nota: You can change the data source of an existing layer by calling `setDataSource()` on a `QgsVectorLayer` instance, as in the following example:

```
# layer is a vector layer, uri is a QgsDataSourceURI instance
layer.setDataSource(uri.uri(), "layer name you like", "postgres")
```

3.2 Camadas Matriciais

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!"
```

Similarly to vector layers, raster layers can be loaded using the addRasterLayer function of the `QgisInterface`:

```
iface.addRasterLayer("/path/to/raster/file.tif", "layer name you like")
```

Isto cria uma nova camada e adiciona ao registro de camadas do mapa (fazendo ele aparecer na lista de camadas) em um passo.

Raster layers can also be created from a WCS service:

```
layer_name = 'modis'
uri = QgsDataSourceURI()
uri.setParam('url', 'http://demo.mapserver.org/cgi-bin/wcs')
uri.setParam("identifier", layer_name)
rlayer = QgsRasterLayer(str(uri.encodedUri()), 'my wcs layer', 'wcs')
```

detailed URI settings can be found in provider documentation

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://irs.gis-lab.info/?layers=landsat&styles=&format=image/jpeg&crs=EPSG:4
rlayer = QgsRasterLayer(urlWithParams, 'some layer name', 'wms')
if not rlayer.isValid():
  print "Layer failed to load!"
```

3.3 Registro de Camada de Mapa

Se você gostaria de usar as camadas abertas para renderização, não se esqueça de adicioná-los ao registro da camada de mapa. O registro camada de mapa apropria-se das camadas e podem ser mais tarde acessadas a partir de qualquer parte do aplicativo pelo seu ID único. Quando a camada é removida do registro camada do mapa, ele é excluído, também.

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)
```

For a list of loaded layers and layer ids, use:

```
QgsMapLayerRegistry.instance().mapLayers()
```

CHAPTER 4

Usando Camadas Raster

Esta seção lista várias operações que você pode fazer com camadas raster.

4.1 Detalhes da Camada

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()
<qgis._core.QgsRectangle object at 0x000000000F8A2048>
rlayer.extent().toString()
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
```

4.2 Renderer

When a raster layer is loaded, it gets a default renderer based on its type. It can be altered either in raster layer properties or programmatically.

To query the current renderer:

```
>>> rlayer.renderer()
<qgis._core.QgsSingleBandPseudoColorRenderer object at 0x7f471c1da8a0>
>>> rlayer.renderer().type()
u'singlebandpseudocolor'
```

To set a renderer use `setRenderer()` method of `QgsRasterLayer`. There are several available renderer classes (derived from `QgsRasterRenderer`):

- QgsMultiBandColorRenderer
- QgsPalettedRasterRenderer
- QgsSingleBandColorDataRenderer
- QgsSingleBandGrayRenderer
- QgsSingleBandPseudoColorRenderer

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 *Atualizando camadas*.

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

4.2.1 Rasters única banda

Let's say we want to render our raster layer (assuming one band only) with colors ranging from green to yellow (for pixel values from 0 to 255). In the first stage we will prepare `QgsRasterShader` object and configure its shader function:

```
>>> fcn = QgsColorRampShader()
>>> fcn.setColorRampType(QgsColorRampShader.INTERPOLATED)
>>> lst = [ QgsColorRampShader.ColorRampItem(0, QColor(0,255,0)), \
    QgsColorRampShader.ColorRampItem(255, QColor(255,255,0)) ]
>>> fcn.setColorRampItemList(lst)
>>> shader = QgsRasterShader()
>>> shader.setRasterShaderFunction(fcn)
```

The shader maps the colors as specified by its color map. The color map is provided as a list of items with pixel value and its associated color. There are 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

In the second step we will associate this shader with the raster layer:

```
>>> renderer = QgsSingleBandPseudoColorRenderer(layer.dataProvider(), 1, shader)
>>> layer.setRenderer(renderer)
```

The number 1 in the code above is band number (raster bands are indexed from one).

4.2.2 Rasters Multi Banda

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.renderer().setGreenBand(1)
rlayer.renderer().setRedBand(2)
```

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

4.3 Atualizando camadas

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.

Nota: This method is deprecated as of QGIS 2.18.0 and will produce a warning. Simply calling `triggerRepaint()` is sufficient.

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 explicitly

```
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)
```

4.4 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 returns a dictionary, with band indices as keys, and band values as values.

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

CHAPTER 5

Usando Camadas Vetor

Esta seção lista várias operações que podem ser realizadas com camadas vetoriais.

5.1 Retrieving information about attributes

You can retrieve information about the fields associated with a vector layer by calling `pendingFields()` on a `QgsVectorLayer` instance:

```
# "layer" is a QgsVectorLayer instance
for field in layer.pendingFields():
    print field.name(), field.typeName()
```

Nota: Starting from QGIS 2.12 there is also a `fields()` in `QgsVectorLayer` which is an alias to `pendingFields()`.

5.2 Selecionando características

In QGIS desktop, features can be selected in different ways, the user can click on a feature, draw a rectangle on the map canvas or use an expression filter. Selected features are normally highlighted in a different color (default is yellow) to draw user's attention on the selection. Sometimes can be useful to programmatically select features or to change the default color.

To change the selection color you can use `setSelectionColor()` method of `QgsMapCanvas` as shown in the following example:

```
iface.mapCanvas().setSelectionColor( QColor("red") )
```

To add add features to the selected features list for a given layer, you can call `setSelectedFeatures()` passing to it the list of features IDs:

```
# Get the active layer (must be a vector layer)
layer = iface.activeLayer()
# Get the first feature from the layer
feature = layer.getFeatures().next()
# Add this features to the selected list
layer.setSelectedFeatures([feature.id()])
```

To clear the selection, just pass an empty list:

```
layer.setSelectedFeatures([])
```

5.3 Interagindo sobre camada vetor

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:
    # retrieve 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.type() == QGis.Point:
        x = geom.asPoint()
        print "Point: " + str(x)
    elif geom.type() == QGis.Line:
        x = geom.asPolyline()
        print "Line: %d points" % len(x)
    elif geom.type() == 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
```

5.3.1 Acessando atributos

Atributos podem ser refenciados pelo nome

```
print feature['name']
```

Alternatively, attributes can be referred to by index. This is will be a bit faster than using the name. For example, to get the first attribute:

```
print feature[0]
```

5.3.2 Iteração sobre os feições selecionadas

if you only need selected features, you can use the `selectedFeatures()` method from vector layer:

```
selection = layer.selectedFeatures()
print len(selection)
for feature in selection:
    # do whatever you need with the feature
```

Outra opção é o metódo de processamento `features()`

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

By default, this will iterate over all the features in the layer, in case there is no selection, or over the selected features otherwise. Note that this behavior can be changed in the Processing options to ignore selections.

5.3.3 Iterando sobre um subconjunto de feições

Se você quer iterar sobre um conjunto de feições em uma camada, como por exemplo em uma determinada área, você precisa adicionar um objeto `QgsFeatureRequest` para a função `getFeatures()`. Segue um exemplo

```
request = QgsFeatureRequest()
request.setFilterRect(areaOfInterest)
for feature in layer.getFeatures(request):
    # do whatever you need with the feature
```

With `setLimit()` you can limit the number of requested features. Here's an example

```
request = QgsFeatureRequest()
request.setLimit(2)
for feature in layer.getFeatures(request):
    # loop through only 2 features
```

If you need an attribute-based filter instead (or in addition) of a spatial one like shown in the examples above, you can build an `QgsExpression` object and pass it to the `QgsFeatureRequest` constructor. Here's an example

```
# The expression will filter the features where the field "location_name"
# contains the word "Lake" (case insensitive)
exp = QgsExpression('location_name ILIKE \'%Lake%\'')
request = QgsFeatureRequest(exp)
```

See *Expressões, filtragem e cálculo dos valores* for the details about the syntax supported by `QgsExpression`.

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

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

Dica: Speed features request

If you only need a subset of the attributes or you don't need the geometry information, you can significantly increase the **speed** of the features request by using `QgsFeatureRequest.NoGeometry` flag or specifying a subset of attributes (possibly empty) like shown in the example above.

5.4 Modificando Camadas Vetoriais

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()
# Check if a particular capability is supported:
caps & QgsVectorDataProvider.DeleteFeatures
# Print 2 if DeleteFeatures is supported
```

For a list of all available capabilities, please refer to the API Documentation of QgsVectorDataProvider

To print layer's capabilities textual description in a comma separated list you can use `capabilitiesString()` as in the following example:

```
caps_string = layer.dataProvider().capabilitiesString()
# Print:
# u'Add Features, Delete Features, Change Attribute Values,
# Add Attributes, Delete Attributes, Create Spatial Index,
# Fast Access to Features at ID, Change Geometries,
# Simplify Geometries with topological validation'
```

By using any of the 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*.

Nota: If you are working inside QGIS (either from the console or from a plugin), it might be necessary to force a redraw of the map canvas in order to see the changes you've done to the geometry, to the style or to the attributes:

```
# If caching is enabled, a simple canvas refresh might not be sufficient
# to trigger a redraw and you must clear the cached image for the layer
if iface.mapCanvas().isCachingEnabled():
    layer.setCacheImage(None)
else:
    iface.mapCanvas().refresh()
```

5.4.1 Adicionar feições

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).

To set up the attributes you can either initialize the feature passing a `QgsFields` instance or call `initAttributes()` passing the number of fields you want to be added.

```
if caps & QgsVectorDataProvider.AddFeatures:
    feat = QgsFeature(layer.pendingFields())
    feat.setAttributes([0, 'hello'])
    # Or set a single attribute by key or by index:
    feat.setAttribute('name', 'hello')
    feat.setAttribute(0, 'hello')
    feat.setGeometry(QgsGeometry.fromPoint(QgsPoint(123, 456)))
    (res, outFeats) = layer.dataProvider().addFeatures([feat])
```

5.4.2 Excluir feições

Para excluir algumas características, apenas providencie a lista com a identificação das características

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

5.4.3 Modificar Feições

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 })
```

Dica: Favor QgsVectorLayerEditUtils class for geometry-only edits

If you only need to change geometries, you might consider using the `QgsVectorLayerEditUtils` which provides some of useful methods to edit geometries (translate, insert or move vertex etc.).

Dica: Directly save changes using `with` based command

Using `with edit(layer):` the changes will be commited automatically calling `commitChanges()` at the end. If any exception occurs, it will `rollBack()` all the changes. See *Modificando Camadas Vetoriais com um Buffer*.

5.4.4 Adicionando e Removendo Campos

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

```
from PyQt4.QtCore import QVariant

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()
```

5.5 Modificando Camadas Vetoriais com um Buffer

Durante a edição de vetores com o QGIS, você precisa primeiramente colocar a camada alvo em modo de edição, então faça algumas modificações e finalmente envie (ou desfaça) as mudanças. Todas as alterações realizadas até antes do envio, permanecerão num buffer de edição em memória. É possível usar esta funcionalidade programaticamente, isso é apenas um outro métodos de edição de vetores que complementam o uso direto de provedores de dados. Use esta opção quando desenvolvendo alguma ferramenta de interface (GUI) para edição de camadas vetoriais, desde que você permita ao usuário decidir enviar/desfazer e desfazer/refazer. Quando enviar as alterações, toda edição no buffer de edição em memória será salvo em uma provedor de dados.

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

```
from PyQt4.QtCore import QVariant

# 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 `beginEditCommand()` 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.

You can also use the `with edit(layer)`-statement to wrap commit and rollback into a more semantic code block as shown in the example below:

```
with edit(layer):
  feat = layer.getFeatures().next()
  feat[0] = 5
  layer.updateFeature(feat)
```

This will automatically call `commitChanges()` in the end. If any exception occurs, it will `rollBack()` all the changes. In case a problem is encountered within `commitChanges()` (when the method returns False) a `QgsEditError` exception will be raised.

5.6 Utilizando Índices Espaciais

Spatial indexes can dramatically improve the performance of your code if you need to do frequent queries to a vector layer. Imagine, for instance, that you are writing an interpolation algorithm, and that for a given location you need to know the 10 closest points 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, especially if it needs to be repeated for several locations. If a spatial index exists for the layer, the operation is much more effective.

Think of a layer without a spatial index as a telephone book in which telephone numbers are not ordered 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:

- criar um índice espacial — o seguinte código cria um índice vazio.

  ```
  index = QgsSpatialIndex()
  ```

- 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)
  ```

- alternatively, you can load all features of a layer at once using bulk loading

  ```
  index = QgsSpatialIndex(layer.getFeatures())
  ```

- uma vez que o índice espacial é preenchido com alguns valores, você pode fazer algumas consultas

  ```
  # 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))
  ```

5.7 Escrevendo Camadas Vetoriais

Você pode escrever um arquivo de camada vetorial utilizando a classe `QgsVectorFileWriter`. Ela atende a qualquer outro tipo de arquivo vetorial com suporte OGR (shapefiles, GeoJSON, KML e outros).

Há duas possibilidades de exportação de camadas vetoriais:

- from an instance of `QgsVectorLayer`

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

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

error = QgsVectorFileWriter.writeAsVectorFormat(layer, "my_json.json", "utf-8", None, "GeoJSO
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.

- diretamente das características

```
from PyQt4.QtCore import QVariant

# define fields for feature attributes. A QgsFields object is needed
fields = QgsFields()
fields.append(QgsField("first", QVariant.Int))
fields.append(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 Sh

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

# 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
del writer
```

5.8 provedor de Memória

O Gerenciador de memória foi desenvolvido para ser utilizado principalmente por plugins ou aplicativos desenvolvidos por terceiros. Estes não são armazenados no disco, permitindo aos desenvolvedores utilizá-los como um rápido backend para algumas camadas temporárias.

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

index=yes Especifica que o provedor irá usar o index espacial

field=name:type(tamanho,precisão) 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.

O exemplo seguinte de URL incorpora todas estas opções

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

The following example code illustrates creating and populating a memory provider

```
from PyQt4.QtCore import QVariant

# 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)])
vl.updateFields() # tell the vector layer to fetch changes from the provider

# 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.xMiniminum(), e.yMinimum(), e.xMaximum(), e.yMaximum()

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

5.9 Aparencia (Simbologia) de Camadas de Vetor

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.

A renderização para uma dada camada pode ser obtida como mostrada abaixo

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

And with that reference, let us explore it a bit

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

Existem muitos tipos de renderização conhecidas disponíveis na biblioteca principal do QGIS

Tipo	Classes	Descrição
singleSymbol	`QgsSingleSymbolRendererV2`	Renderiza todas as características com o mesmo símbolo
categorizedSymbol	`QgsCategorizedSymbolRendererV2`	Renderiza características usando um símbolo diferente para cada categoria
graduatedSymbol	`QgsGraduatedSymbolRendererV2`	Renderiza caracter´sticas usando diferents símbolos para cada limite de valores

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:

```
print QgsRendererV2Registry.instance().renderersList()
# Print:
[u'singleSymbol',
u'categorizedSymbol',
u'graduatedSymbol',
u'RuleRenderer',
u'pointDisplacement',
u'invertedPolygonRenderer',
u'heatmapRenderer']
```

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

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

5.9.1 Single Symbol Renderer

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 change the symbol used by a particular vector layer by calling `setSymbol()` passing an instance of the appropriate symbol instance. Symbols for *point*, *line* and *polygon* layers can be created by calling the `createSimple()` function of the corresponding classes `QgsMarkerSymbolV2`, `QgsLineSymbolV2` and `QgsFillSymbolV2`.

The dictionary passed to `createSimple()` sets the style properties of the symbol.

For example you can replace the symbol used by a particular **point** layer by calling `setSymbol()` passing an instance of a `QgsMarkerSymbolV2` as in the following code example:

```
symbol = QgsMarkerSymbolV2.createSimple({'name': 'square', 'color': 'red'})
layer.rendererV2().setSymbol(symbol)
```

`name` indicates the shape of the marker, and can be any of the following:

- `circulo`
- `quadrado`

- cross
- *retângulo‘*
- diamante
- pentágono
- triângulo
- triângulo_equilateral
- estrela
- regular_star
- flecha
- filled_arrowhead
- x

To get the full list of properties for the first symbol layer of a simbol instance you can follow the example code:

```
print layer.rendererV2().symbol().symbolLayers()[0].properties()
# Prints
{u'angle': u'0',
u'color': u'0,128,0,255',
u'horizontal_anchor_point': u'1',
u'name': u'circle',
u'offset': u'0,0',
u'offset_map_unit_scale': u'0,0',
u'offset_unit': u'MM',
u'outline_color': u'0,0,0,255',
u'outline_style': u'solid',
u'outline_width': u'0',
u'outline_width_map_unit_scale': u'0,0',
u'outline_width_unit': u'MM',
u'scale_method': u'area',
u'size': u'2',
u'size_map_unit_scale': u'0,0',
u'size_unit': u'MM',
u'vertical_anchor_point': u'1'}
```

This can be useful if you want to alter some properties:

```
# You can alter a single property...
layer.rendererV2().symbol().symbolLayer(0).setName('square')
# ... but not all properties are accessible from methods,
# you can also replace the symbol completely:
props = layer.rendererV2().symbol().symbolLayer(0).properties()
props['color'] = 'yellow'
props['name'] = 'square'
layer.rendererV2().setSymbol(QgsMarkerSymbolV2.createSimple(props))
```

5.9.2 Categorized Symbol Renderer

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.

5.9.3 Graduated Symbol Renderer

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 *

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)
```

5.9.4 Trabalhando com Símbolos

Para representação de símbolos, existe `QgsSymbolV2` que é a classe básica com três classes derivadas.

- `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.

Por padrão, tamanho e largura são em milimetros e ângulos em graus.

Trabalhando com Camadas de Símbolos

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
```

Saída

```
EllipseMarker
FontMarker
SimpleMarker
SvgMarker
VectorField
```

A classe `QgsSymbolLayerV2Registry` gerencia um banco de dados com todos os tipo de símbolo de camadas.

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.

Creating Custom Symbol Layer Types

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.

Você terá que criar metadados para a camada de símbolos

```
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.

O último passo para adicionar este símbolo de camada para o registro — e estamos prontos.

5.9.5 Creating Custom Renderers

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.defaultSymbo

  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()
    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).

5.10 Outros Tópicos

TODO:

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

CHAPTER 6

Manipulação Geométrica

Pontos, cadeias lineares e polígonos que representam uma característica espacial são comumente referido como geometrias. Em QGIS eles são representados com a classe : class `QgsGeometry`. Todos os tipos possíveis de geometria são mostrados na página de discussão JTS discussion page.

Às vezes, uma geometria é realmente uma coleção dex simples geometrias (single-part). Tal geometria é chamada de geometria de várias partes. Se ele contém apenas um tipo de simples geometria, podemos chamar de multi-ponto, multi-cadeia linear ou multi-polígono. Por exemplo, um país que consiste de múltiplas ilhas pode ser representado como um sistema multi-polígono.

As coordenadas de geometrias podem estar em qualquer sistema de referência de coordenadas (SRC). Ao buscar feições a partir de uma camada, geometrias associadas terão coordenadas no SRC da camada.

6.1 Construção de Geométria

Existem várias opções para criar uma geometria:

- a partir das coordenadas

```
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)]])
```

  As coordenadas são dadas usando a classe `QgsPoint`.

  Polyline (Cadeia Linear) é representado por uma lista de pontos. Polígono é representado por uma lista de anéis lineares (ou seja, cadeias lineares fechada). Primeiro anel é o anel exterior (fronteira), anéis subsequentes opcionais são buracos no polígono.

  Geometrias multi-parte passam para um nível maior: multi-ponto é uma lista de pontos, multi-cadeia linear é uma lista de cadeias lineares e multi-polígono é uma lista de polígonos.

- a partir de textos conhecidos (WKT)

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

- a partir de binários conhecidos (WKB)

```
>>> g = QgsGeometry()
>>> wkb = '0101000000000000000000454000000000000001440'.decode('hex')
>>> g.fromWkb(wkb)
```

```
>>> g.exportToWkt()
'Point (42 5)'
```

6.2 Acesso a Geometria

Primeiro, você deve descobrir o tipo de geometria, `wkbType()` é o método único a usar — ele retorna um valor de enumeração do `QGis.WkbType`

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

Como alternativa, pode-se usar o método `type()` que retorna um valor de enumeração `QGis.GeometryType`. Há também uma função de ajuda : `isMultipart()` para descobrir se uma geometria é multi-parte ou não.

Para extrair informações de geometria existem funções de acesso para cada tipo de vetor. Como usar acessores

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

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

Para geometrias de multi-partes existem funções de acesso similares: `asMultiPoint()`, `asMultiPolyline()`, `asMultiPolygon()`.

6.3 Operações e Predicados Geométricos

QGIS usa biblioteca GEOS para operações avançadas de geometria como predicados geométrico (`contains()`, `intersects()`, ...) e operações de conjunto (`union()`, `difference()`, ...). Também podem calcular propriedades geométricas das geometrias, tais como a área (no caso de polígonos) ou comprimentos (para polígonos e linhas)

Aqui você tem um pequeno exemplo que combina a iteração sobre as caracteríticas em uma determinada camada e executando alguns cálculos geométricos com base em suas geometrias.

```
# 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.

```
d = QgsDistanceArea()
d.setEllipsoid('WGS84')
```

```
d.setEllipsoidalMode(True)

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

Você pode encontrar muitos exemplo de algoritmos que estão incluídos no QGIS e usar esses métodos para analisar e transformar dados vetoriais. Aqui estão alguns links para o código de alguns deles.

Informação adicional pode ser encontada nas seguintes fontes:

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

CHAPTER 7

Suporte a projeções

7.1 Sistemas de Referencia de Coordenadas

Os Sistemas de Referencia de Coordenadas (SRC), estão incluídas na classe:*Sistema de ReferenciaCoordenadasQgs* . Instâncias desta classe podem ser criados de diferentes maneiras:

- especificar CRS pela sua identificação

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

  O QGIS usa três diferentes ID para cada sistema de referencia:

  - `PostgisCrsId` — Identificação usada nos bancos de dados PostGIS
  - `InternalCrsId` — Identificação usada internamente no banco de dados QGIS
  - `EpsgCrsId` — Identificação designada pela organização EPSG

  Se não é determinado uma coisa diferente no segundo parâmetro, o SRID PostGIS será usado por padrão..

- especificar CRS pelo seu texto conhecido (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)
  ```

- crie CRS inválidos e, em seguida, use uma das funções `create*()` para inicializa-la. No exemplo seguinte, nós usamos seqüência Proj4 para inicializar a projeção

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

É interessante checar se a criação do SRC foi bem sucedida, a função : `Évalido()` deve retornar o valor `Verdadeiro`.

Note que para a inicialização de sistemas de referência espacial QGIS precisa procurar os valores apropriados em seu banco de dados interno `srs.db`. Assim, no caso de você criar um aplicativo independente que você precisa para definir os caminhos corretamente com `QgsApplication.setPrefixPath()` caso contrário ele não vai encontrar o banco de dados. Se você estiver executando os comandos a partir do console QGIS python ou desenvolvendo um plugin você não precisa se preocupar: tudo já está configurado para você.

Acessando informações do sistema de referência espacial

```
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.toProj4()
# 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()
```

7.2 Projeções

Você pode fazer a transformação entre os diferentes sistemas de referência espacial usando a classe `QgsCoordinateTransform`. A maneira mais fácil de usá-lo é criar a origem e o destino CRS e construir a instância `QgsCoordinateTransform` com eles. Em seguida, chamar repetidamente apenas a função `transform()` para fazer a transformação. Por padrão, ele faz a transformação para a frente, mas é capaz de fazer também transformação inversa

```
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 8

Utilizando a Tela do Mapa

O Widget tela Mapa é provavelmente o widget mais importante dentro de QGIS porque mostra o mapa composto de camadas de mapas sobrepostos e permite a interação com o mapa e camadas. A tela mostra sempre uma parte do mapa definido pela extensão tela atual. A interação é feita através do uso de **ferramentas de mapa** existem ferramentas para panorâmica, zoom, identificando camadas, de medida, de edição de vetores e outros. Semelhante a outros programas gráficos, há sempre uma ferramenta ativa e que o usuário pode alternar entre as ferramentas disponíveis.

Mapa na tela é implementado como: class: *class QgsMapCanvas* no módulo: mod:‘ qgis.gui‘. A implementação é baseada no framework Qt Gráficos View. Este quadro geralmente fornece uma superfície e uma vista onde os itens gráficos personalizados são colocados e usuário pode interagir com eles. Vamos supor que você está bastante familiarizado com Qt para entender os conceitos de cena, vista e itens gráficos. Se não, por favor, certifique-se de ler a visão geral do quadro.

Sempre que o mapa foi muito deslocado, zoom in / out (ou alguma outra ação dispara uma atualização), o mapa é processado novamente dentro da extensão atual. As camadas são prestados a uma imagem (usando: class: *classe QgsMapRenderer*) e essa imagem é exibida na tela. O item gráfico (em termos de gráficos vista framework Qt) responsável por mostrar o mapa é: class: *classe QgsMapCanvasMap*. Esta classe também controla a atualização do mapa renderizado. Além disso este item, que atua como um fundo, pode haver mais itens ** mapa de itens na tela**. Mapa típica itens na tela são elásticos (usados para medir, edição vetor etc.) ou marcadores de vértice. Os itens de lona são normalmente utilizados para dar algum feedback visual para ferramentas de mapa, por exemplo, ao criar um novo polígono, a ferramenta de mapa cria um item de canvas elástico que mostra a forma atual do polígono. Todos os itens de mapa de lona são subclasses de: class: *QgsMapCanvasItem* que acrescenta mais algumas funcionalidades para os‘ *objetos básicos QGraphicsItem*‘.

Para resumir, a arquitetura do mapa na tela são constituídas por três conceitos:

- tela do mapa — para visualização do mapa
- itens na tela do mapa — itens adicionais que podem ser exibidos na tela do mapa
- ferramentas do mapa — para a interagir com o mapa na tela

8.1 Incorporar o Mapa da Tela

Mapa natela é um widget, como qualquer outro widget Qt, então usá-lo é tão simples como criar e mostrá-lo

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

Isso produz uma janela independente com o mapa de lona. Ela também pode ser incorporado em um widget ou janela existente. Ao usar .ui arquivos e Qt Designer, coloque um ‘‘ QWidget‘‘ no formulário e promovê-lo a uma nova classe: definir ‘‘ QgsMapCanvas‘‘ como nome da classe e definir ‘‘ qgis.gui‘‘ como arquivo de cabeçalho. O utilitário ‘‘ pyuic4‘‘ vai cuidar dele. Esta é uma maneira muito conveniente de incorporar a tela. A outra possibilidade é escrever manualmente o código para construir do mapa na tela e outros widgets (como filhos de uma janela principal ou diálogo) e criar um layout.

Por padrão, o mapa na tela tem fundo preto e não usa anti-aliasing. Para definir o fundo branco e permitir anti-aliasing para renderização suave

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

(Caso você esteja se perguntando, ‘‘ Qt‘‘ vem de ‘‘ módulo PyQt4.QtCore‘‘ e ‘‘ Qt.white‘‘ é um dos pré-definido ‘‘ QColor‘‘ instâncias.)

Agora é hora de adicionar algumas camadas do mapa. Vamos abrir primeiro uma camada e adicioná-lo ao registro camada do mapa. Então, vamos definir a extensão da tela e definir a lista de camadas para tela

```
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)])
```

Depois de executar esses comandos, a tela deve mostrar a camada que você carregou.

8.2 Usando Ferramentas de Mapas na Tela

O exemplo a seguir constrói uma janela que contém um mapa de tela e ferramentas básicas de mapa para mapa panorâmico e zoom. Ações são criadas para a ativação de cada ferramenta: panorâmica é feito com: class: *QgsMapToolPan*, zoom in / out com um par de instâncias: class:‘ QgsMapToolZoom‘. As ações são definidas como verificável e posteriormente atribuído às ferramentas para permitir o tratamento automático de do estado das ações verificado / não verificado - quando uma ferramenta de mapa é ativado, a sua ação está marcado como selecionado e a acção da ferramenta mapa anterior é desmarcada. As ferramentas de mapa são ativadas usando o metodo: func: *setMapTool*.

```
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)
```

Você pode colocar o código acima para um arquivo, por exemplo, : file: *mywnd.py* e experimentá-lo no console do Python dentro QGIS. Este código irá colocar a camada selecionada atualmente na tela recém-criada

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

Apenas certifique-se de que o arquivo: file: *mywnd.py* está localizado dentro do caminhoo de busca Python (‘ *sys.path*‘). Se não for, você pode simplesmente adicioná-lo: `sys.path.insert(0, '/my/path')` — caso contrário, a declaração de importação irá falhar, não encontrando o módulo.

8.3 Bandas raster e fazedor de vértices

Para mostrar alguns dados adicionais na parte superior do mapa na tela, use itens de mapa de tela. É possível criar classes de itens de tela (mostradas abaixo), no entanto, existem duas classes úteis de item de tela para sua conveniência:: class: *QgsRubberBand* para desenhar multilinhas ou polígonos, e: class:‘ QgsVertexMarker‘ para desenhar pontos. Ambos trabalham com as coordenadas do mapa, assim que a forma é movida/escalada automaticamente quando a tela está em panorâmica ou ampliada.

Para mostrar as polilinhas

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

Para mostrar o poligono

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

Note-se que aponta para polígono não é uma lista simples: na verdade, é uma lista de anéis contendo anéis lineares do polígono: primeiro anel é a borda externa, ainda mais (opcional) anéis correspondem aos buracos no polígono.

As Bandas Raster permitem alguma personalização, ou seja, para mudar sua cor e linha de largura

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

Os itens em tela são obrigados a cena na tela. Para escondê-los temporariamente (e mostrar mais uma vez, use a Combinação: `hide()` and `show()` Para remover completamente o item, você tem que removê-lo da cena na tela

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

(em C ++ é possível simplesmente apagar o item, no entanto, em Python ‘‘ del r‘‘ seria apenas suprimir a referência eo objeto continuará a existir, uma vez que é de propriedade da tela)

As bandas Rasters também podem serem usadas para desenhar pontos, no entanto a classe : class: *class QgsVertexMarker* é mais adequado para isso (`QgsRubberBand` só desenhar um retângulo ao redor do ponto desejado). Como usar o fazedor de Vértices

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

Isso vai chamar a uma cruz vermelha sobre a posição [0,0]. É possível personalizar o tipo de ícone, tamanho, cor e espessura da caneta

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

Para esconder temporáriamente os marcadores de vértice e removê-los da tela, o mesmo se aplica quanto para as bandas rasters.

8.4 Desenhar ferramenta de mapa personalizada

Você pode escrever suas ferramentas personalizadas, para implementar um comportamento personalizado às ações realizadas pelos usuários na tela.

Ferramentas de mapa deve herdar a classe: class: *QgsMapTool* ou qualquer outra classe derivada, e selecionado como ferramentas ativas na tela usando o método: func:*setMapTool* como já vimos.

Aqui está um exemplo de uma ferramenta de mapa que permite definir uma medida retangular, clicando e arrastando na tela. Quando o retângulo é definido, ele imprime coordena seu limite no console. Ele utiliza os elementos de banda de borracha descritos antes para mostrar o retângulo selecionado, uma vez que está a ser definida.

```
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.xMinimum(), r.yMinimum(), r.xMaximum(), r.yMaximum()

    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):
        super(RectangleMapTool, self).deactivate()
        self.emit(SIGNAL("deactivated()"))
```

8.5 Desenhar itens da tela do mapa

TODO: Como criar itens do mapa na tela

```
import sys
from qgis.core import QgsApplication
from qgis.gui import QgsMapCanvas

def init():
  a = QgsApplication(sys.argv, True)
  QgsApplication.setPrefixPath('/home/martin/qgis/inst', True)
  QgsApplication.initQgis()
  return a

def show_canvas(app):
  canvas = QgsMapCanvas()
  canvas.show()
  app.exec_()
app = init()
show_canvas(app)
```

CHAPTER 9

Renderização em impressão de mapas

Em geral, há duas abordagens quando os dados de entrada deve ser processado como um mapa: quer fazê-lo maneira rápida usando: class: *QgsMapRenderer* ou produzir saída mais afinadas, compondo o mapa na classe de amigos: class:‘ classe QgsComposition‘.

9.1 Renderização simples

Renderizar algumas camadas usando: class: *QgsMapRenderer* — cria um dispositivo de pintura de destino (‘ *QImage*‘,‘ *QPainter*‘ etc.), configurar conjunto de camadas, extensão, tamanho de saída e fazer a renderização

```
# 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 = QgsRectangle(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")
```

9.2 Renderizando camadas com CRS diferente

Se você tiver mais de uma camada e eles têm CRS diferente, o simples exemplo acima provavelmente não vai funcionar: para obter os valores corretos dos cálculos de extensão, você tem que definir explicitamente a CRS de destino e permitem reprojeção OTF como no exemplo abaixo (apenas a parte de configuração do renderizador é relatada)

```
...
# set layer set
layers = QgsMapLayerRegistry.instance().mapLayers()
lst = layers.keys()
render.setLayerSet(lst)

# Set destination CRS to match the CRS of the first layer
render.setDestinationCrs(layers.values()[0].crs())
# Enable OTF reprojection
render.setProjectionsEnabled(True)
...
```

9.3 Saída usando Compositor de Mapa

Compositor de Mapa é uma ferramenta muito útil se você gostaria de fazer uma produção mais sofisticada do que a prestação simples mostrado acima. Usando o compositor é possível criar layouts mapa complexas consistindo de visualizações de mapas, etiquetas, legenda, tabelas e outros elementos que estão normalmente presentes em mapas de papel. Os layouts podem ser depois exportados para PDF, imagens raster ou diretamente impresso em uma impressora.

O compositor consiste em um grupo de classes. Todas elas pertencem à biblioteca núcleo. A aplicação QGIS tem uma interface gráfica para colocação conveniente dos elementos, embora ele não estejam disponíveis na biblioteca GUI. Se você não estiver familiarizado Com Qt Gráficos Ver quadro, em seguida, um cheque que são incentivados a documentação agora, porque o compositor é baseado nele. Verifique também a documentação do Python da Implementação do QGraphicView.

A classe central de composição é: class: *QgsComposition* que é derivada de: class:‘ QGraphicsScene‘. Vamos criar um

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

Note-se que a composição leva uma instância de: class: *QgsMapRenderer*. No código esperamos que estamos executando dentro aplicativo QGIS e, assim, usar o renderizador mapa do mapa de lona. A composição utiliza vários parâmetros do renderizador mapa, o mais importante o conjunto padrão de camadas de mapas e da extensão atual. Ao utilizar compositor em um aplicativo independente, você pode criar sua própria instância mapa processador da mesma forma como mostrado na seção acima e passá-lo para a composição.

É possível adicionar vários elementos (mapa, etiqueta, ...) para a composição — estes elementos têm de ser descendentes de: class: *classe QgsComposerItem*. Itens atualmente suportados são:

- map — este item diz as bibliotecas onde colocar o próprio mapa. Aqui criamos um mapa e esticamos sobre o tamanho do papel inteiro

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

- label — permite exibir rótulos. É possível modificar a sua fonte, cor, alinhamento e margem

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

- legenda

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

- barra de escala

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

- Seta
- Imagem
- basic shape
- nodes based shape

```
polygon = QPolygonF()
polygon.append(QPointF(0.0, 0.0))
polygon.append(QPointF(100.0, 0.0))
polygon.append(QPointF(200.0, 100.0))
polygon.append(QPointF(100.0, 200.0))

composerPolygon = QgsComposerPolygon(polygon, c)
c.addItem(composerPolygon)

props = {}
props["color"] = "green"
props["style"] = "solid"
props["style_border"] = "solid"
props["color_border"] = "black"
props["width_border"] = "10.0"
props["joinstyle"] = "miter"

style = QgsFillSymbolV2.createSimple(props)
composerPolygon.setPolygonStyleSymbol(style)
```

- Tabela

Por padrão, os itens de composição recém-criados têm posição zero (canto superior esquerdo da página) e tamanho zero. A posição e tamanho são sempre medido em milímetros

```
# 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)
```

Um quadro é desenhado em torno de cada item por padrão. Como remover o quadro

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

Além de criar os itens compositor à mão, QGIS tem suporte para modelos compositor que são essencialmente composições com todos os seus itens salvos em um arquivo .qpt (com a sintaxe XML). Infelizmente essa funcionalidade ainda não está disponível na API.

Uma vez que a composição está pronta (os itens do compositor tenham sido criados e adicionados à composição), podemos prosseguir para produzir um raster e / ou um vetor de saída.

As definições de saída padrão para a composição são página de tamanho A4 e resolução de 300 DPI. Você pode alterá-los, se necessário. O tamanho do papel é especificada em milímetros

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

9.3.1 Exportando para um arquivo de imagem

O seguinte fragmento de código mostra como renderizar a composição para uma imagem raster

```
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)
c.renderPage( imagePainter, 0 )
imagePainter.end()

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

9.3.2 Exportando em PDF

O seguinte fragmento de código renderiza a composição para um arquivo PDF

```
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 10

Expressões, filtragem e cálculo dos valores

O QGIS possui alguns recursos para análise de expressões semelhantes à SQL. Apenas um pequeno subconjunto da sintaxe SQL é suportado. As expressões podem ser avaliadas tanto como predicados booleanos (retornando Verdadeiro ou Falso) ou como funções (retornando um valor escalar) . Veja *vector_expressions* no Manual do Usuário para uma lista completa das funções disponíveis.

Três tipos básicos são suportados:

- número — ambos os números inteiros e números decimais, por exemplo, `123`, `3.14`
- texto — eles devem estar entre aspas simples: *'Olá world''*
- coluna referência — ao avaliar, a referência é substituída com o valor real do campo. Os nomes não alteram.

As seguintes operações estão disponíveis:

- operadores aritméticos: `+`, `-`, `*`, `/`, `^`
- parênteses: para fazer cumprir a precedência do operador: `(1 + 1) * 3`
- Sinal mais e menos: `-12`, `+5`
- funções matemáticas: `sqrt`, `sen`, `cos`, `tan`, `asen`, `acos`, `atan`
- Funções de conversão: "to_int", "to_real", "to_string", "to_date"
- funções geométricas: `$area`, `$length`
- funções de manipulação de geometria: "$x", "$y", "$geometry", "num_geometries", "centroid"

E os seguintes predicados são suportados:

- comparação: `=`, `!=`, `>`, `>=`, `<`, `<=`
- correspondência padrão: `LIKE` (usando % e _), ~ (expressões regulares)
- predicados lógicos: `AND`, `OR`, `NOT`
- verificação de valor nulo: `IS NULL`, `IS NOT NULL`

Exemplos de predicados:

- `1 + 2 = 3`
- `sen(ângulo) > 0`

- 'Hello' LIKE 'He%'
- (x > 10 AND y > 10) OR z = 0

Exemplos de expressões escalares:

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

10.1 Expressões de Análise

```
>>> 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')
```

10.2 Expressões de Avaliação

10.2.1 Expressões Básicas

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

10.2.2 Expressões com características

O exemplo seguinte irá avaliar a expressão dada contra uma característica. "Column" é o nome do campo na camada.

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

Você também pode usar `QgsExpression.prepare()` se você precisar verificar mais de uma característica. Usando `QgsExpression.prepare()` irá aumentar a velocidade que a avaliação leva para executar.

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

10.2.3 Manipulação de erros

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

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

print value
```

10.3 Exemplos

O exemplo seguinte pode ser usado para filtrar uma camada e devolver qualquer característica que corresponda a um predicado.

```
def where(layer, exp):
  print "Where"
  exp = QgsExpression(exp)
  if exp.hasParserError():
    raise Exception(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 11

Leitura e Armazenamento de Configurações

Muitas vezes é útil para o plugin salvar algumas variáveis para que o utilizador não necessite introduzir ou selecionar outra vez numa próxima vez que o plugin for acionado.

Estas variáveis podem ser salvas e recuperadas com a ajuda do Qt e QGIS API. Para cada variável, você deve pegar a chave que será usada para acessar a variável — para cor favorita do usuário use a chave "favourite_color" ou alguma outra palavra com significado. É recomendado dar alguma estrutura para criação do nome das chaves.

Nós podemos diferenciar os diversos tipos de configurações:

- **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 macOS) 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)
  ```

  O segundo parâmetro do método `value()` é opcional e especifica o valor padrão se não for fornecido uma definição do valor prévio para o nome da configuração passada.

- **project settings** — variar entre diferentes projetos e, portanto, eles estão conectados com um arquivo de projeto. Cor de fundo de mapa de tela ou destino no sistema de coordenadas de referência (CRS) são exemplos — fundo branco e WGS84 pode ser adequado para um projeto, enquanto fundo amarelo e projeção UTM são melhores para outra. Um exemplo de uso segue

  ```
  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]
  ```

  Como pode ver, o método `writeEntry()` é usado para todos os tipos de dados, mas em diversos métodos

existe para leitura um valor de configuração de retorno, e o correspondente tem de ser selecionado para cada tipo de dado.

- **map layer settings** — essas configurações estão relacionadas a uma instância específica de uma camada de mapa com um projeto. Eles *não* estão conectados com a fonte de uma camada de dados subjacente, por isso, se você criar duas instâncias da camada de mapa de um shapefile, eles não vão compartilhar as configurações. As configurações são armazenadas no arquivo de projeto, por isso, se o usuário abre o projeto novamente, as definições relacionadas com a camada vai estar lá novamente. Esta funcionalidade foi adicionada no QGIS v1.4. A API é semelhante à QSettings — leva e retorna instâncias QVariant

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

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

CHAPTER 12

Comunicação com o usuário

Esta seção mostra alguns métodos e elementos que devem ser usados para se comunicar com o usuário, a fim de manter a consistência na interface do usuário.

12.1 Mostrando mensagens. Classe `QgsMessageBar`

Usando caixas de mensagem pode ser uma má idéia, do ponto de vista da experiência do usuário. Para mostrar uma pequena linha de informação ou uma mensagem de aviso/erro, a barra de mensagens QGIS é geralmente uma opção melhor.

Usando a referência ao objeto de interface QGIS, você pode mostrar uma mensagem na barra de mensagem com o seguinte código

```
from qgis.gui import QgsMessageBar
iface.messageBar().pushMessage("Error", "I'm sorry Dave, I'm afraid I can't do that", level=QgsMes
```

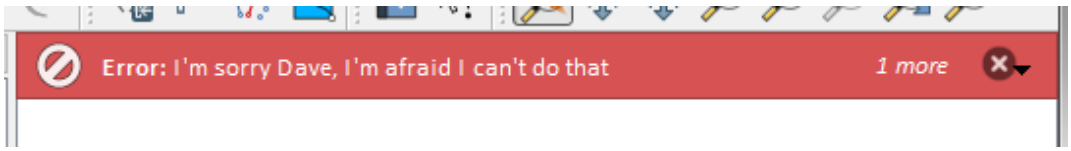


Figure 12.1: Barra de mensagem do QGIS

Você pode definir uma duração de mostrá-lo por um tempo limitado

```
iface.messageBar().pushMessage("Error", "Ooops, the plugin is not working as it should", level=Qgs
```

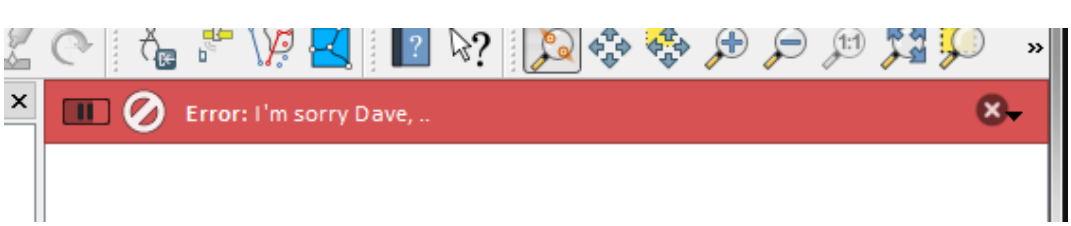


Figure 12.2: Barra de Mensagem QGIS como temporizar

Os exemplos acima mostram uma barra de erro, mas o parâmetro ‘‘ level‘‘ pode ser usado para criar mensagens de aviso ou mensagens de informação, usando as constantes `QgsMessageBar.WARNING` e

`QgsMessageBar.INFO` respectivamente.

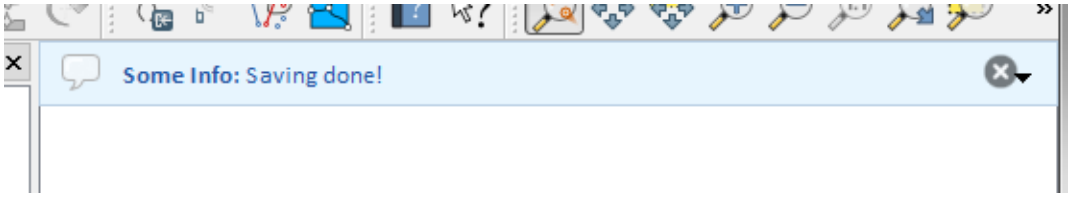


Figure 12.3: Barra de mensagens QGIS (informação)

Widgets podem ser adicionados à barra de mensagens, como por exemplo, um botão para mostrar mais informações

```
def showError():
    pass

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)
```

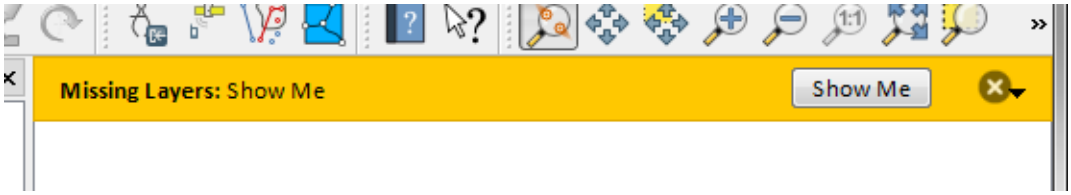


Figure 12.4: Barra de mensagens QGIS com um botão

Você ainda pode usar uma barra de mensagens em sua própria caixa de diálogo para que você não tenha de mostrar uma caixa de mensagem, ou se ela não faz sentido para mostrá-la na janela principal QGIS

```
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)
```

12.2 Mostrando progresso

As barras de progresso também pode ser colocado na barra de mensagem QGIS, uma vez que, como vimos, ele aceita widgets. Aqui está um exemplo que você pode tentar no console.

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

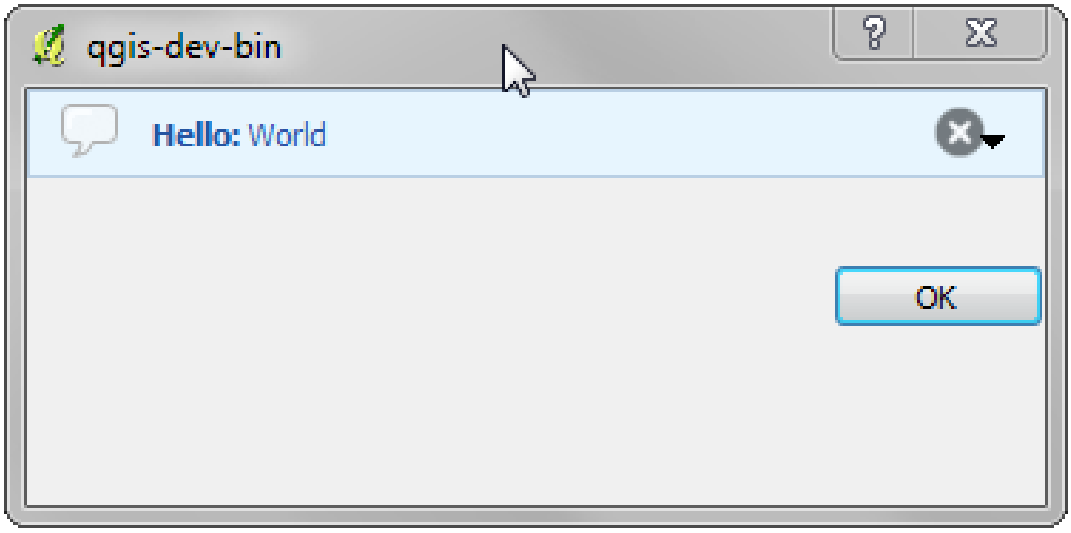


Figure 12.5: Barra de mensagens QGIS em diálogo personalizado

```
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()
```

Além disso, você pode usar a barra de status incorporada para relatar o progresso, como no exemplo a seguir

```
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()
```

12.3 Carregando

Você pode usar o sistema de registro de QGIS para registrar todas as informações que você deseja salvar sobre a execução de seu código.

```
# You can optionally pass a 'tag' and a 'level' parameters
QgsMessageLog.logMessage("Your plugin code has been executed correctly", 'MyPlugin', QgsMessageLo
QgsMessageLog.logMessage("Your plugin code might have some problems", level=QgsMessageLog.WARNING
QgsMessageLog.logMessage("Your plugin code has crashed!", level=QgsMessageLog.CRITICAL)
```

CHAPTER 13

Desenvolvimento de Complementos Python

É possível criar plugins usando a linguagem de programação Python. Em comparação com plugins clássicos criados em C++, esses devem ser mais fáceis de escrever, entender, manter e distribuir devido à natureza dinâmica da linguagem Python.

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

- UNIX/Mac: `~/.qgis2/python/plugins` and `(qgis_prefix)/share/qgis/python/plugins`
- Windows: `~/.qgis2/python/plugins` and `(qgis_prefix)/python/plugins`

Home directory (denoted by above `~`) on Windows is usually something like `C:\Documents and Settings\(user)` (on Windows XP or earlier) or `C:\Users\(user)`. Since QGIS is using Python 2.7, subdirectories of these paths have to contain an __init__.py file to be considered Python packages that can be imported as plugins.

Nota: By setting *QGIS_PLUGINPATH* to an existing directory path, you can add this path to the list of paths that are searched for plugins.

Passos:

1. *Ideia*: Tenha uma ideia sobre o que você quer fazer com seu novo plugin QGIS. Por que você quer fazê-lo? Qual problema você gostaria de resolver? Já existe outro plugin para este tipo de problema?
2. *Create files*: Create the files described next. A starting point (`__init__.py`). Fill in the *Plugin metadados* (`metadata.txt`) A main python plugin body (`mainplugin.py`). A form in QT-Designer (`form.ui`), with its `resources.qrc`.
3. *Write code*: Write the code inside the `mainplugin.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".

13.1 Escrevendo um complemento

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. The QGIS team also maintains an *Repositório oficial de complementos python*. Ready to create a plugin but no idea what to do? Python Plugin Ideas wiki page lists wishes from the community!

13.1.1 arquivos de complementos

Está é a estrutura de diretórios do nosso exemplo de complemento

```
PYTHON_PLUGINS_PATH/
  MyPlugin/
    __init__.py     --> *required*
    mainPlugin.py   --> *required*
    metadata.txt    --> *required*
    resources.qrc   --> *likely useful*
    resources.py    --> *compiled version, likely useful*
    form.ui         --> *likely useful*
    form.py         --> *compiled version, likely useful*
```

Qual o significado dos arquivos:

- `__init__.py` = The starting point of the plugin. It has to have the `classFactory()` method and may have any other initialisation code.
- `mainPlugin.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` = A tradução do arquivo .qrc descrito acima para Python.
- `form.ui` = A GUI creiada pelo Qt Designer.
- `form.py` = A tradução do formulário .ui descrito acima para Python.
- `metadata.txt` = Required for QGIS >= 1.8.0. Contains general info, version, name and some other metadata used by plugins website and plugin infrastructure. Since QGIS 2.0 the metadata from `__init__.py` are not accepted anymore and the `metadata.txt` is required.

Here is an online automated way 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.

Aviso: If you plan to upload the plugin to the *Repositório oficial de complementos python* you must check that your plugin follows some additional rules, required for plugin *Validação*

13.2 Conteúdo do complemento

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

13.2.1 Plugin metadados

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 to put this information.

Importante: Todos os metadados devem estar codificados em UTF-8.

Nome metadados	requerida	Notas
Nome	Verdadeiro	a short string containing the name of the plugin
qgisMinimumVersion	Verdadeiro	dotted notation of minimum QGIS version
qgisMaximumVersion	Falso	versão QGIS máxima com notação pontilhada
Descrição	Verdadeiro	texto pequeno que descreve o complemento, HTML não é permitido
sobre	Verdadeiro	texto longo que descreve o complemento em detalhes, HTML não é permitido
Versão	Verdadeiro	caracteres curtos com a versão com notação pontilhada
Autor	Verdadeiro	nome do autor
email	Verdadeiro	email of the author, not shown in the QGIS plugin manager or in the website unless by a registered logged in user, so only visible to other plugin authors and plugin website administrators
changelog	Falso	string, can be multiline, no HTML allowed
experimental	Falso	valor booleano, *True* ou *False*
desaprovado	Falso	valor booleano,'True' ou *False*', aplica-se a todo o plugin e não apenas para a versão carregada
etiquetas	Falso	lista separada por vírgulas, espaços são permitidos dentro das etiquetas individuais
página inicial	Falso	uma URL válida apontando para a página inicial do seu plugin
Repositório	Verdadeiro	uma URL válida para repositório do código fonte
rastreador	Falso	uma URL válida para o reporte de chamadas e erros
Ícone	Falso	a file name or a relative path (relative to the base folder of the plugin's compressed package) of a web friendly image (PNG, JPEG)
Categoria	Falso	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 or Web. For example, if your plugin will be available from *Raster* menu, add this to `metadata.txt`

```
category=Raster
```

Nota: If *qgisMaximumVersion* is empty, it will be automatically set to the major version plus *.99* when uploaded to the *Repositório oficial de complementos python*.

Um exemplo para este metadata.txt

```
; the next section is mandatory

[general]
name=HelloWorld
email=me@example.com
author=Just Me
qgisMinimumVersion=2.0
description=This is an example plugin for greeting the world.
    Multiline is allowed:
    lines starting with spaces belong to the same
    field, in this case to the "description" field.
    HTML formatting is not allowed.
about=This paragraph can contain a detailed description
    of the plugin. Multiline is allowed, HTML is not.
version=version 1.2
tracker=http://bugs.itopen.it
repository=http://www.itopen.it/repo
; end of mandatory metadata

; start of optional metadata
category=Raster
changelog=The changelog lists the plugin versions
    and their changes as in the example below:
    1.0 - First stable release
    0.9 - All features implemented
    0.8 - First testing release

; Tags are in comma separated value format, spaces are allowed within the
; tag name.
; Tags should be in English language. Please also check for existing tags and
; synonyms before creating a new one.
tags=wkt,raster,hello world

; these metadata can be empty, they will eventually become mandatory.
homepage=http://www.itopen.it
icon=icon.png

; experimental flag (applies to the single version)
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
```

13.2.2 __init__.py

This file is required by Python's import system. Also, QGIS requires that this file contains a `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's class from the `mainplugin.py` — in our case it's called `TestPlugin` (see below). This is how `__init__.py` should look like

```
def classFactory(iface):
  from mainPlugin import TestPlugin
  return TestPlugin(iface)

## any other initialisation needed
```

13.2.3 mainPlugin.py

This is where the magic happens and this is how magic looks like: (e.g. `mainPlugin.py`)

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

# initialize Qt resources from file resources.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.mainWin
    self.action.setObjectName("testAction")
    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.renderTe

  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 in the main plugin source file (e.g. `mainPlugin.py`) are:

- `__init__` –> Dá acesso à interface QGIS
- `initGui()` –> chamada quando o complemento é carregado
- *unload()*‘ –> chamada quando o complemento é descarregado

You can see that in the above example, the `addPluginToMenu()` 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.setObjectName("testMenu")
    self.menu.setTitle("MyMenu")

    self.action = QAction(QIcon(":/plugins/testplug/icon.png"), "Test plugin", self.iface.mainWin
    self.action.setObjectName("testAction")
    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()
```

Don't forget to set `QAction` and `QMenu` `objectName` to a name specific to your plugin so that it can be customized.

13.2.4 Arquivo de Recursos

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
```

Nota: In Windows environments, attempting to run the **pyrcc4** from Command Prompt or Powershell will probably result in the error "Windows cannot access the specified device, path, or file [...]". The easiest solution is probably to use the OSGeo4W Shell but if you are comfortable modifying the PATH environment variable or specifiying the path to the executable explicitly you should be able to find it at `<Your QGIS Install Directory>\bin\pyrcc4.exe`.

E isso é tudo... nada complicado :)

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.

13.3 Documentação

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.

13.4 Tradução

With a few steps you can set up the environment for the plugin localization so that depending on the locale settings of your computer the plugin will be loaded in different languages.

13.4.1 Software requirements

The easiest way to create and manage all the translation files is to install Qt Linguist. In a Debian-based GNU/Linux environment you can install it typing:

```
sudo apt-get install qt4-dev-tools
```

13.4.2 Files and directory

When you create the plugin you will find the `i18n` folder within the main plugin directory.

All the translation files have to be within this directory.

.pro file

First you should create a `.pro` file, that is a *project* file that can be managed by **Qt Linguist**.

In this `.pro` file you have to specify all the files and forms you want to translate. This file is used to set up the localization files and variables. A possible project file, matching the structure of our *example plugin*:

```
FORMS = ../form.ui
SOURCES = ../your_plugin.py
TRANSLATIONS = your_plugin_it.ts
```

Your plugin might follow a more complex structure, and it might be distributed across several files. If this is the case, keep in mind that `pylupdate4`, the program we use to read the `.pro` file and update the translatable string, does not expand wild card characters, so you need to place every file explicitly in the `.pro` file. Your project file might then look like something like this:

```
FORMS = ../ui/about.ui ../ui/feedback.ui \
        ../ui/main_dialog.ui
SOURCES = ../your_plugin.py ../computation.py \
          ../utils.py
```

Furthermore, the `your_plugin.py` file is the file that *calls* all the menu and sub-menus of your plugin in the QGIS toolbar and you want to translate them all.

Finally with the *TRANSLATIONS* variable you can specify the translation languages you want.

Aviso: Be sure to name the `ts` file like `your_plugin_ + language + .ts` otherwise the language loading will fail! Use 2 letters shortcut for the language (**it** for Italian, **de** for German, etc...)

.ts file

Once you have created the `.pro` you are ready to generate the `.ts` file(s) of the language(s) of your plugin.

Open a terminal, go to `your_plugin/i18n` directory and type:

```
pylupdate4 your_plugin.pro
```

you should see the `your_plugin_language.ts` file(s).

Open the `.ts` file with **Qt Linguist** and start to translate.

.qm file

When you finish to translate your plugin (if some strings are not completed the source language for those strings will be used) you have to create the `.qm` file (the compiled `.ts` file that will be used by QGIS).

Just open a terminal cd in `your_plugin/i18n` directory and type:

```
lrelease your_plugin.ts
```

now, in the `i18n` directory you will see the `your_plugin.qm` file(s).

13.4.3 Translate using Makefile

Alternatively you can use the makefile to extract messages from python code and Qt dialogs, if you created your plugin with Plugin Builder. At the beginning of the Makefile there is a LOCALES variable:

```
LOCALES = en
```

Add the abbreviation of the language to this variable, for example for Hungarian language:

```
LOCALES = en hu
```

Now you can generate or update the `hu.ts` file (and the `en.ts` too) from the sources by:

```
make transup
```

After this, you have updated `.ts` file for all languages set in the LOCALES variable. Use **Qt4 Linguist** to translate the program messages. Finishing the translation the `.qm` files can be created by the transcompile:

```
make transcompile
```

You have to distribute `.ts` files with your plugin.

13.4.4 Carregar o complemento

In order to see the translation of your plugin just open QGIS, change the language (*Settings* → *Options* → *Language*) and restart QGIS.

You should see your plugin in the correct language.

Aviso: If you change something in your plugin (new UIs, new menu, etc..) you have to **generate again** the update version of both `.ts` and `.qm` file, so run again the command of above.

CHAPTER 14

Authentication infrastructure

14.1 Introduction

User reference of the Authentication infrastructure can be read in the User Manual in the *authentication_overview* paragraph.

This chapter describes the best practices to use the Authentication system from a developer perspective.

Aviso: Authentication system API is more than the classes and methods exposed here, but it's strongly suggested to use the ones described here and exposed in the following snippets for two main reasons

1. Authentication API will change during the move to QGIS3
2. Python bindings will be restricted to the `QgsAuthManager` class use.

Most of the following snippets are derived from the code of Geoserver Explorer plugin and its tests. This is the first plugin that used Authentication infrastructure. The plugin code and its tests can be found at this link. Other good code reference can be read from the authentication infrastructure tests code

14.2 Glossary

Here are some definition of the most common objects treated in this chapter.

Master Password Password to allow access and decrypt credential stored in the QGIS Authentication DB

Authentication Database A *Master Password* crypted sqlite db `<user home>/.qgis2/qgis-auth.db` where *Authentication Configuration* are stored. e.g user/password, personal certificates and keys, Certificate Authorities

Authentication DB *Authentication Database*

Authentication Configuration A set of authentication data depending on *Authentication Method*. e.g Basic authentication method stores the couple of user/password.

Authentication config *Authentication Configuration*

Authentication Method A specific method used to get authenticated. Each method has its own protocol used to gain the authenticated level. Each method is implemented as shared library loaded dynamically during QGIS authentication infrastructure init.

14.3 QgsAuthManager the entry point

The QgsAuthManager singleton is the entry point to use the credentials stored in the QGIS encrypted *Authentication DB*:

`<user home>/.qgis2/qgis-auth.db`

This class takes care of the user interaction: by asking to set master password or by transparently using it to access crypted stored info.

14.3.1 Init the manager and set the master password

The following snippet gives an example to set master password to open the access to the authentication settings. Code comments are important to understand the snippet.

```
authMgr = QgsAuthManager.instance()
# check if QgsAuthManager has been already initialized... a side effect
# of the QgsAuthManager.init() is that AuthDbPath is set.
# QgsAuthManager.init() is executed during QGis application init and hence
# you do not normally need to call it directly.
if authMgr.authenticationDbPath():
    # already initilised => we are inside a QGIS app.
    if authMgr.masterPasswordIsSet():
        msg = 'Authentication master password not recognized'
        assert authMgr.masterPasswordSame( "your master password" ), msg
    else:
        msg = 'Master password could not be set'
        # The verify parameter check if the hash of the password was
        # already saved in the authentication db
        assert authMgr.setMasterPassword( "your master password",
                                          verify=True), msg
else:
    # outside qgis, e.g. in a testing environment => setup env var before
    # db init
    os.environ['QGIS_AUTH_DB_DIR_PATH'] = "/path/where/located/qgis-auth.db"
    msg = 'Master password could not be set'
    assert authMgr.setMasterPassword("your master password", True), msg
    authMgr.init( "/path/where/located/qgis-auth.db" )
```

14.3.2 Populate authdb with a new Authentication Configuration entry

Any stored credential is a *Authentication Configuration* instance of the QgsAuthMethodConfig class accessed using a unique string like the following one:

```
authcfg = 'fm1s770'
```

that string is generated automatically when creating an entry using QGIS API or GUI.

QgsAuthMethodConfig is the base class for any *Authentication Method*. Any Authentication Method sets a configuration hash map where authentication informations will be stored. Hereafter an useful snippet to store PKI-path credentials for an hypothetic alice user:

```
authMgr = QgsAuthManager.instance()
# set alice PKI data
p_config = QgsAuthMethodConfig()
p_config.setName("alice")
p_config.setMethod("PKI-Paths")
p_config.setUri("http://example.com")
p_config.setConfig("certpath", "path/to/alice-cert.pem" ))
p_config.setConfig("keypath", "path/to/alice-key.pem" ))
# check if method parameters are correctly set
assert p_config.isValid()

# register alice data in authdb returning the ''authcfg'' of the stored
# configuration
authMgr.storeAuthenticationConfig(p_config)
newAuthCfgId = p_config.id()
assert (newAuthCfgId)
```

Available Authentication methods

*Authentication Method*s are loaded dynamically during authentication manager init. The list of Authentication method can vary with QGIS evolution, but the original list of available methods is:

1. `Basic` User and password authentication
2. `Identity-Cert` Identity certificate authentication
3. `PKI-Paths` PKI paths authentication
4. `PKI-PKCS#12` PKI PKCS#12 authentication

The above strings are that identify authentication methods in the QGIS authentication system. In Development section is described how to create a new c++ *Authentication Method*.

Populate Authorities

```
authMgr = QgsAuthManager.instance()
# add authorities
cacerts = QSslCertificate.fromPath( "/path/to/ca_chains.pem" )
assert cacerts is not None
# store CA
authMgr.storeCertAuthorities(cacerts)
# and rebuild CA caches
authMgr.rebuildCaCertsCache()
authMgr.rebuildTrustedCaCertsCache()
```

Aviso: Due to QT4/OpenSSL interface limitation, updated cached CA are exposed to OpenSsl only almost a minute later. Hope this will be solved in QT5 authentication infrastructure.

Manage PKI bundles with QgsPkiBundle

A convenience class to pack PKI bundles composed on SslCert, SslKey and CA chain is the QgsPkiBundle class. Hereafter a snippet to get password protected:

```
# add alice cert in case of key with pwd
boundle = QgsPkiBundle.fromPemPaths( "/path/to/alice-cert.pem",
                                     "/path/to/alice-key_w-pass.pem",
                                     "unlock_pwd",
                                     "list_of_CAs_to_bundle" )
assert boundle is not None
assert boundle.isValid()
```

Refer to QgsPkiBundle class documentation to extract cert/key/CAs from the bundle.

14.3.3 Remove entry from authdb

We can remove an entry from *Authentication Database* using it's `authcfg` identifier with the following snippet:

```
authMgr = QgsAuthManager.instance()
authMgr.removeAuthenticationConfig( "authCfg_Id_to_remove" )
```

14.3.4 Leave authcfg expansion to QgsAuthManager

The best way to use an *Authentication Config* stored in the *Authentication DB* is referring it with the unique identifier `authcfg`. Expanding, means convert it from an identifier to a complete set of credentials. The best practice to use stored *Authentication Config*s, is to leave it managed automatically by the Authentication manager. The common use of a stored configuration is to connect to an authentication enabled service like a WMS or WFS or to a DB connection.

Nota: Take into account that not all QGIS data providers are integrated with the Authentication infrastructure. Each authentication method, derived from the base class QgsAuthMethod and support a different set of Providers. For example `Identity-Cert` method supports the following list of providers:

```
In [19]: authM = QgsAuthManager.instance()
In [20]: authM.authMethod("Identity-Cert").supportedDataProviders()
Out[20]: [u'ows', u'wfs', u'wcs', u'wms', u'postgres']
```

For example, to access a WMS service using stored credentials identified with `authcfg = 'fm1s770'`, we just have to use the `authcfg` in the data source URL like in the following snippet:

```
authCfg = 'fm1s770'
quri = QgsDataSourceURI()
quri.setParam("layers", 'usa:states')
quri.setParam("styles", '')
quri.setParam("format", 'image/png')
quri.setParam("crs", 'EPSG:4326')
quri.setParam("dpiMode", '7')
quri.setParam("featureCount", '10')
quri.setParam("authcfg", authCfg)    # <---- here my authCfg url parameter
quri.setParam("contextualWMSLegend", '0')
quri.setParam("url", 'https://my_auth_enabled_server_ip/wms')
rlayer = QgsRasterLayer(quri.encodedUri(), 'states', 'wms')
```

In the upper case, the `wms` provider will take care to expand `authcfg` URI parameter with credential just before setting the HTTP connection.

Aviso: Developer would have to leave `authcfg` expansion to the QgsAuthManager, in this way he will be sure that expansion is not done too early.

Usually an URI string, build using `QgsDataSourceURI` class, is used to set QGIS data source in the following way:

```
rlayer = QgsRasterLayer( quri.uri(False), 'states', 'wms')
```

Nota: The `False` parameter is important to avoid URI complete expansion of the `authcfg` id present in the URI.

PKI examples with other data providers

Other example can be read directly in the QGIS tests upstream as in test_authmanager_pki_ows or test_authmanager_pki_postgres.

14.4 Adapt plugins to use Authentication infrastructure

Many third party plugins are using httplib2 to create HTTP connections instead of integrating with `QgsNetworkAccessManager` and its related Authentication Infrastructure integration. To facilitate this integration an helper python function has been created called `NetworkAccessManager`. Its code can be found here.

This helper class can be used as in the following snippet:

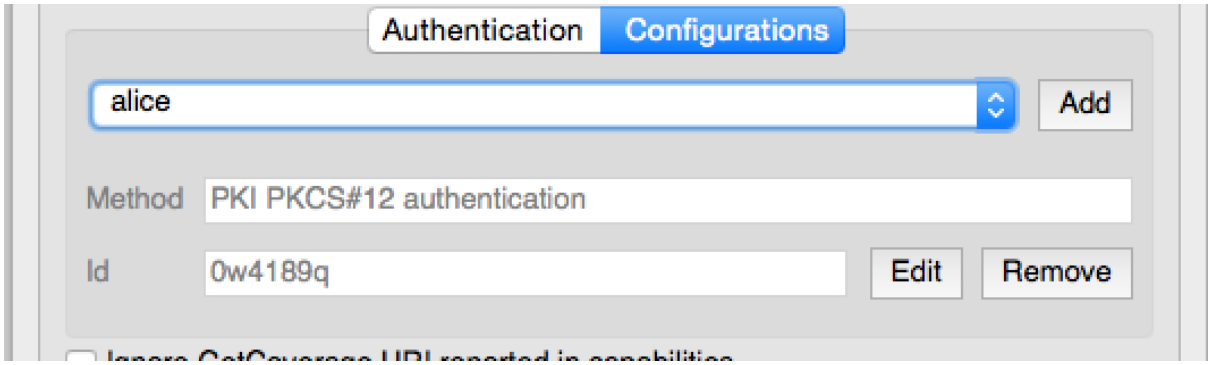
```
http = NetworkAccessManager(authid="my_authCfg", exception_class=My_FailedRequestError)
try:
  response, content = http.request( "my_rest_url" )
except My_FailedRequestError, e:
  # Handle exception
  pass
```

14.5 Authentication GUIs

In this paragraph are listed the available GUIs useful to integrate authentication infrastructure in custom interfaces.

14.5.1 GUI to select credentials

If it's necessary to select a *Authentication Configuration* from the set stored in the *Authentication DB* it is available in the GUI class QgsAuthConfigSelect



and can be used as in the following snippet:

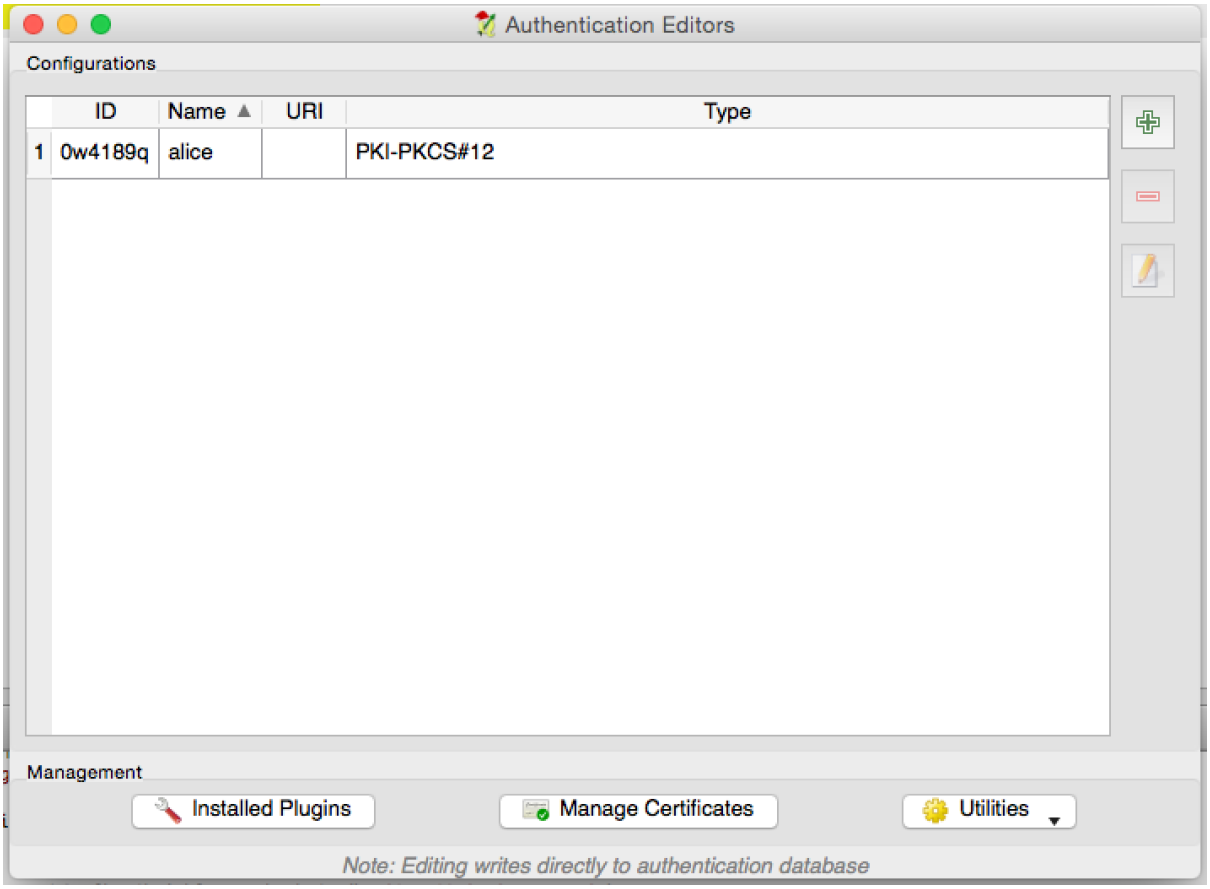
```
# create the instance of the QgsAuthConfigSelect GUI hierarchically linked to
# the widget referred with 'parent'
gui = QgsAuthConfigSelect( parent, "postgres" )
# add the above created gui in a new tab of the interface where the
```

```
# GUI has to be integrated
tabGui.insertTab( 1, gui, "Configurations" )
```

The above example is get from the QGIS source code The second parameter of the GUI constructor refers to data provider type. The parameter is used to restrict the compatible *Authentication Method*s with the specified provider.

14.5.2 Authentication Editor GUI

The complete GUI used to manage credentials, authorities and to access to Authentication utilities is managed by the class QgsAuthEditorWidgets



and can be used as in the following snippet:

```
# create the instance of the QgsAuthEditorWidgets GUI hierarchically linked to
# the widget referred with 'parent'
gui = QgsAuthConfigSelect( parent )
gui.show()
```

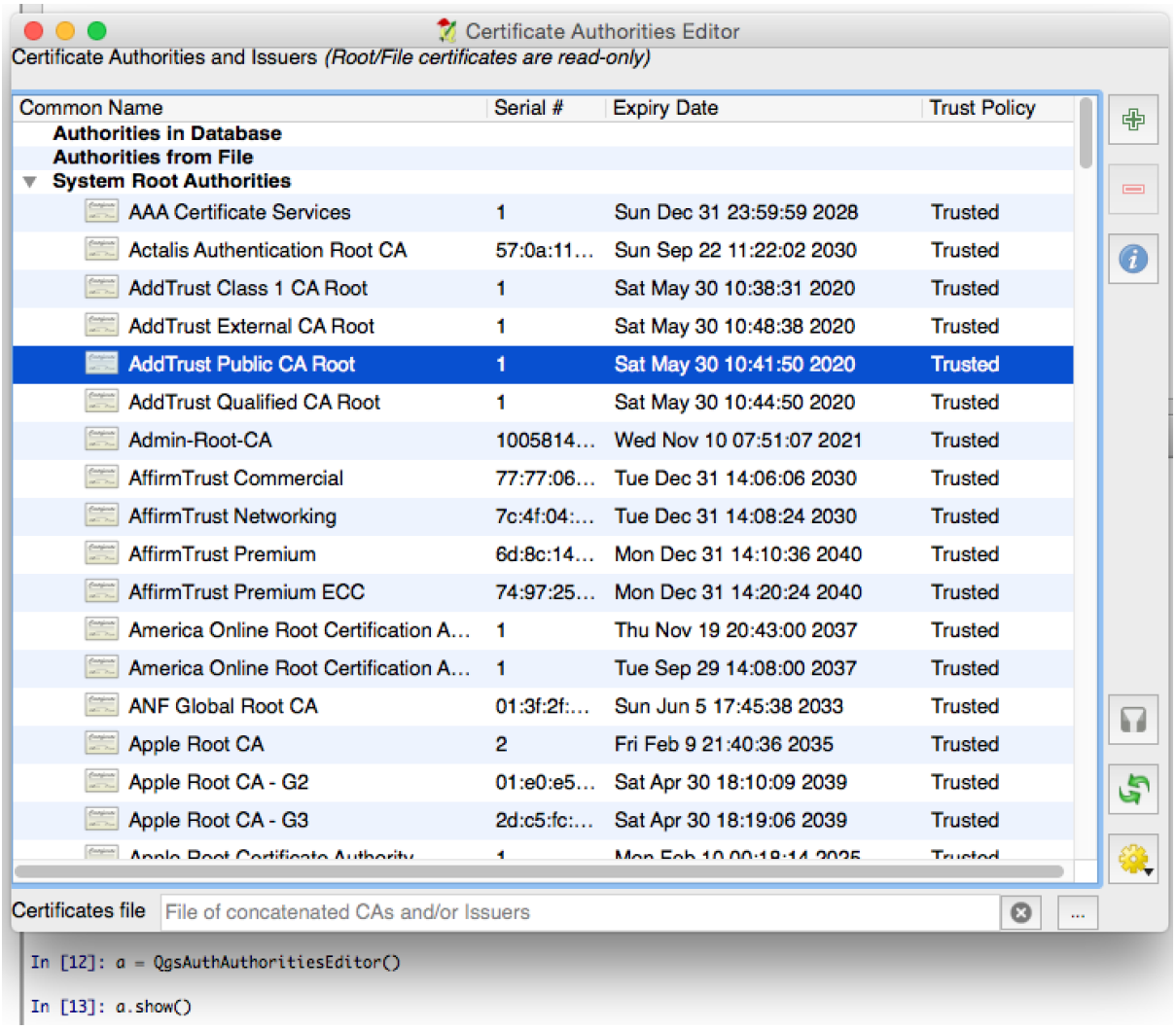
an integrated example can be found in the related test

14.5.3 Authorities Editor GUI

A GUI used to manage only authorities is managed by the class QgsAuthAuthoritiesEditor

and can be used as in the following snippet:

```
# create the instance of the QgsAuthAuthoritiesEditor GUI hierarchically
#  linked to the widget referred with 'parent'
```


Certificate Authorities Editor
Certificate Authorities and Issuers (Root/File certificates are read-only)
Common Name
Serial #
Expiry Date
Trust Policy
Authorities in Database
Authorities from File
System Root Authorities
AAA Certificate Services 1 Sun Dec 31 23:59:59 2028 Trusted
Actalis Authentication Root CA 57:0a:11… Sun Sep 22 11:22:02 2030 Trusted
AddTrust Class 1 CA Root 1 Sat May 30 10:38:31 2020 Trusted
AddTrust External CA Root 1 Sat May 30 10:48:38 2020 Trusted
AddTrust Public CA Root 1 Sat May 30 10:41:50 2020 Trusted
AddTrust Qualified CA Root 1 Sat May 30 10:44:50 2020 Trusted
Admin-Root-CA 1005814… Wed Nov 10 07:51:07 2021 Trusted
AffirmTrust Commercial 77:77:06… Tue Dec 31 14:06:06 2030 Trusted
AffirmTrust Networking 7c:4f:04:… Tue Dec 31 14:08:24 2030 Trusted
AffirmTrust Premium 6d:8c:14… Mon Dec 31 14:10:36 2040 Trusted
AffirmTrust Premium ECC 74:97:25… Mon Dec 31 14:20:24 2040 Trusted
America Online Root Certification A… 1 Thu Nov 19 20:43:00 2037 Trusted
America Online Root Certification A… 1 Tue Sep 29 14:08:00 2037 Trusted
ANF Global Root CA 01:3f:2f:… Sun Jun 5 17:45:38 2033 Trusted
Apple Root CA 2 Fri Feb 9 21:40:36 2035 Trusted
Apple Root CA - G2 01:e0:e5… Sat Apr 30 18:10:09 2039 Trusted
Apple Root CA - G3 2d:c5:fc:… Sat Apr 30 18:19:06 2039 Trusted
Certificates file
File of concatenated CAs and/or Issuers
...
In [12]: a = QgsAuthAuthoritiesEditor()
In [13]: a.show()

```
gui = QgsAuthAuthoritiesEditor( parent )
gui.show()
```

CHAPTER 15

Configurações de IDE para escrita e depuração de plugins

Embora cada programador tem seu editor de Texto/IDE preferido, aqui estão algumas recomendações para configurar o IDE populares para escrever e depurar plugins QGIS Python.

15.1 Uma nota na configuração IDE para Windows

No Linux não é necessária configuração adicional para desenvolver complementos. Se você estiver utilizando Windows, certifique-se de possuir as mesmas configurações de ambiente e utilizar as mesmas bibliotecas e interpretador utilizados pelo QGIS. A forma mais rápida de fazer isso, é modificar o arquivo em lote de inicialização do QGIS.

Se você utilizou o instalador OSGeo4W, o arquivo de inicialização pode ser encontrado dentro da pasta `bin` do diretório de instalação do OSGeo4W. Procure por algo como `C:\OSGeo4W\bin\qgis-unstable.bat`.

Para usar Pyscripter IDE, eis o que tem de fazer:

- Faça uma cópia do arquivo `qgis-unstable.bat` e renomeie para `pyscripter.bat`.
- Abra-o em um editor. E remova a última linha, a que inicia o QGIS.
- Adicione uma linha que aponte para o seu executável Pyscripter e adicione o argumento de linha de comando que define a versão do Phyton a ser utilizada (2.7 no caso do QGIS >= 2.0).
- Além disso, adicione o argumento de que aponta para a pasta onde PyScripter encontrará a dll Python usado pelo QGIS, você pode encontrá-la na pasta de arquivos binários da instalação OSGeoW

```
@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
```

Agora, quando você clicar duas vezes neste arquivo de lote, vai iniciar PyScripter, com o caminho correto.

Mais popular que o Pyscripter, o Eclipse é uma escolha comum entre os desenvolvedores. Nas seções seguintes, nós iremos explicar como configurá-lo para desenvolver e testar complementos. No Windows, para preparar o ambiente para a utilização do Eclipse, você também precisará criar um arquivo em lote para iniciá-lo.

Para criar o arquivo em lote, siga os passos a seguir:

- Localize a pasta onde está o arquivo `qgis_core.dll`. Normalmente ele pode ser encontrado em `C:\OSGeo4W\apps\qgis\bin`, mas se você compilou seu próprio QGIS, o arquivo estará na sua pasta de compilação em `output/bin/RelWithDebInfo`.
- Localize seu executável `eclipse.exe`.
- Crie o seguinte script e use para iniciar eclipse quando estiver desenvolvendo plugins QGIS.

```
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
```

15.2 Depuração usando Eclipse e PyDev

15.2.1 Instalação

Para usar Eclipse, certifique-se que tem instalado o seguinte:

- Eclipse
- Aptana Eclipse Plugin ou PyDev
- QGIS 2.x

15.2.2 Preparando QGIS

É preciso fazer uma preparação no próprio QGIS. Existem dois complementos necessários: **Remote Debug** e **Plugin reloader**.

- Vá para *Complementos* → *Gerenciar e Instalar Complementos...*
- Procure por *Remote Debug* (no memento este complemento ainda é experimental, portanto você precisa habilitar a exibição de complementos experimentais em *Opções* no menu à esquerda caso ele não apareça) . Instale ele.
- Procurar por *Plugin reloader* e instalá-lo. Isso permitirá que você recarregue um plugin em vez de ter que fechar e reiniciar QGIS para ter o plugin recarregado.

15.2.3 Configurando Eclipse

Em Eclipse, crie um novo projeto. Você pode selecionar*General Project* e vincular suas fontes reais mais tarde, por isso realmente não importa onde você coloca este projeto.

Agora clique com o botão direito no seu novo projeto e selecione *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.

Agora, na visão *Project Explorer*, sua árvore de origem aparece e você pode começar a trabalhar com o código. Você já tem destaque de sintaxe e todas as outras ferramentas IDE disponíveis.

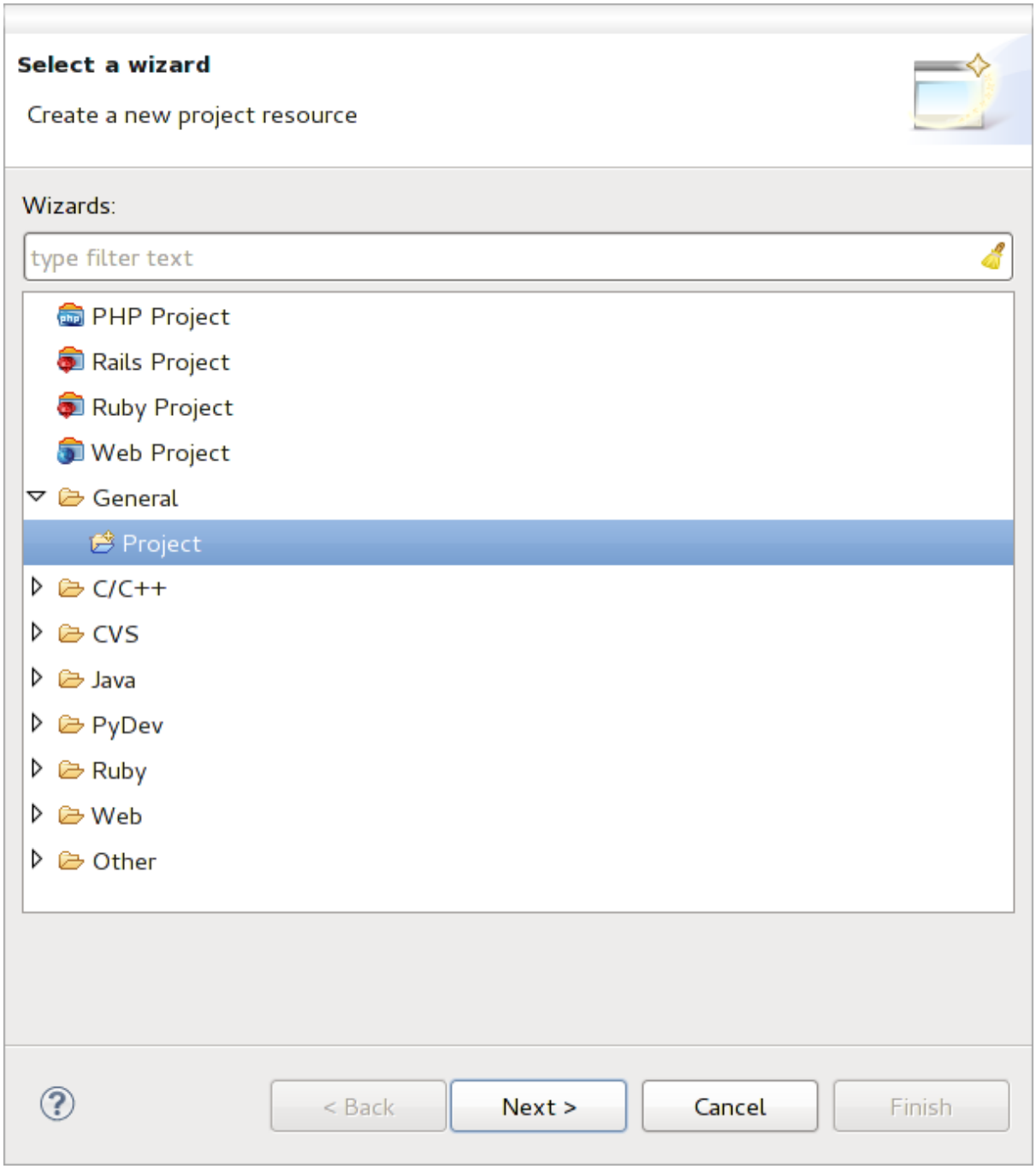


Figure 15.1: Projeto Eclipse

15.2.4 Configurando o depurador

Para obter o depurador trabalhando, alterne para a perspectiva de depuração no Eclipse (*Window → Open Perspective → Other → Debug*).

Agora inicie o servidor de depuração PyDev selecionando *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).

```
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 15.2: Ponto de interrupção

Uma coisa muito interessante que você pode fazer uso de agora é o console de depuração. Certifique-se de que a execução está parado em um ponto de interrupção, antes de prosseguir.

Abra a visualização da Console (*Window → Show view*). Ela vai mostrar console *Debug Server* que não é muito interessante. Mas há um botão **[Open Console]** que lhe permite mudar para uma console PyDev Debug mais interessante. Clique na seta ao lado do botão **[Open Console]** e escolha *PyDev Debug Console*. Uma janela se abre para perguntar-lhe qual console você quer começar. Escolha *PyDev Debug Console*. No caso de estar acinzentado e dizer-lhe para iniciar o depurador e selecionar o quadro válido, certifique-se de que você tem o depurador remoto ligado e estão atualmente em um ponto de interrupção.

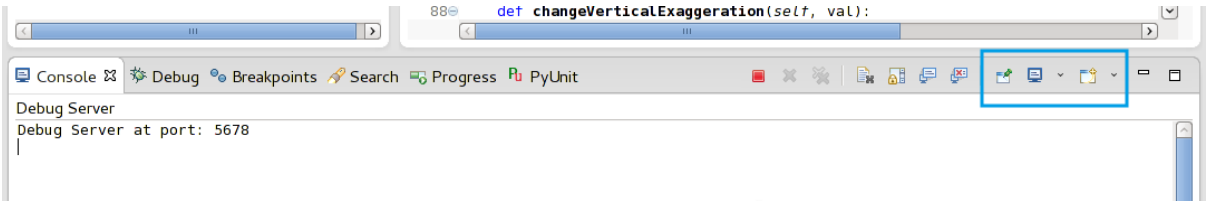


Figure 15.3: Consolde de depuração PyDev

Agora você tem um console interativo que lhe permite testar os comandos de dentro do contexto atual. Você pode manipular as variáveis ou fazer chamadas de API ou o que quer que você queira.

Um pouco chato é que, cada vez que você digitar um comando, o console volta para o Debug Server. Para parar esse comportamento, você pode clicar no botão *Pin Console* quando estiver na página Debug Server e ele deve se lembrar esta decisão, pelo menos, para a sessão de depuração atual.

15.2.5 Fazendo eclipse entender a API

Uma característica muito útil é ter Eclipse realmente informado sobre a QGIS API. Isto permite-lhe verificar o seu código de erros de digitação. Mas não é só isso, ele também permite que o Eclipse ajudá-lo com o completamento automático das importações de chamadas da API.

Para fazer isso, Eclipse analisa os arquivos da biblioteca do QGIS e recebe toda a informação lá. A única coisa que você tem a fazer é dizer ao Eclipse onde encontrar as bibliotecas.

Click *Window* → *Preferences* → *PyDev* → *Interpreter* → *Python*.

Você vai ver o seu interpretador Python configurado na parte superior da janela (no momento python2.7 para QGIS) e algumas guias na parte inferior. As guias interessantes para nós são *Libraries* e *Forced Builtins*

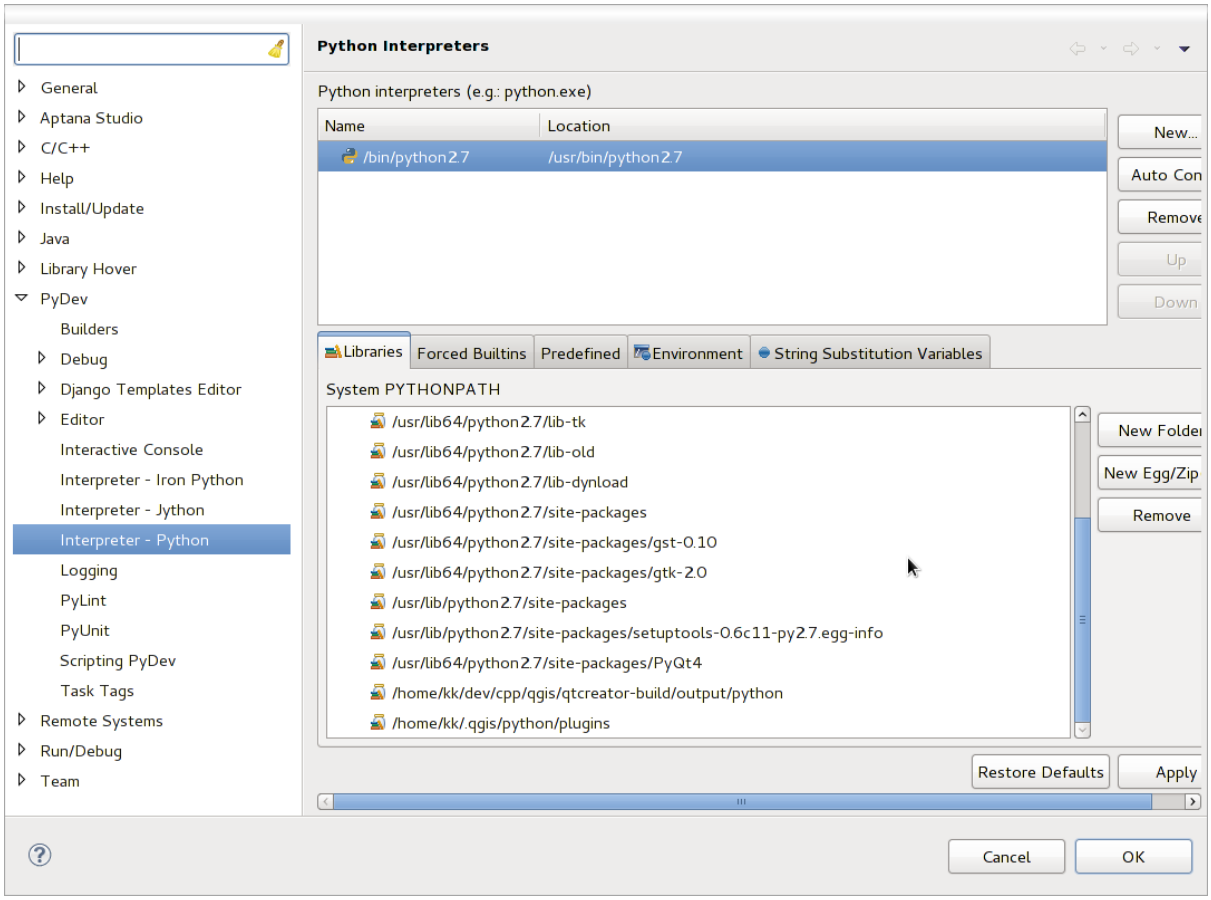


Figure 15.4: Consolde de depuração PyDev

Primeiro, abra a guia Bibliotecas. Adicione uma nova pasta e escolha a pasta python de sua instalação QGIS. Se você não sabe onde está pasta (não é a pasta plugins) abra QGIS, inicie um console python e basta digitar ‘‘ qgis‘‘ e pressione Enter. Ela vai mostrar que módulo qgis usa e seu caminho. Tira o final `/qgis/__init__.pyc` deste caminho e você tem o caminho que está procurando.

You should also add your plugins folder here (on Linux it is `~/.qgis2/python/plugins`).

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.

Clique *OK* e está feito.

Nota: Every time 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.

15.3 Depuração usando PDB

Se você não usar um IDE como o Eclipse, você pode depurar usando PDB, seguindo estes passos.

Primeiro adicione este código no local onde você gostaria de depurar

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

Em seguida, execute QGIS a partir da linha de comando.

On Linux do:

```
$ ./Qgis
```

On macOS do:

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

E quando o aplicativo atingir o seu ponto de interrupção você pode digitar no console!

TODO: Adicione informação de testes

CHAPTER 16

Usando Camadas de Plugins

Se o seu plugin usa seus próprios métodos para tornar uma camada de mapa, escrever o seu próprio tipo de camada com base em QgsPluginLayer pode ser a melhor maneira de implementar isso.

A FAZER: Verifique a exatidão e elabore bons casos de uso para QgsPluginLayer, ...

16.1 Subclasses QgsPluginLayer

Abaixo está um exemplo de uma implementação mínima QgsPluginLayer. É um trecho de exemplo do plugin Watermark

```
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
```

Os métodos para ler e escrever informações específicas para o arquivo de projeto também podem ser adicionados

```
def readXml(self, node):
  pass

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

Ao carregar um projeto que contém uma tal camada , uma classe de fábrica é necessária

```
class WatermarkPluginLayerType(QgsPluginLayerType):

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

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

Você também pode adicionar código para exibir informações personalizadas nas propriedades da camada

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

CHAPTER 17

Compatibilidade com versões anteriores do QGIS

17.1 menu Complementos

Se você colocar o plugin de itens do menu em um dos novos menus (*Raster*, *Vector*, *Database* or *Web'*), você deve modificar o código das funções : func: *initGUI ()* e:`unload()`. Uma vez que estes novos menus estão disponíveis somente no QGIS 2,0 e maior, o primeiro passo é verificar se a versão do QGIS em execução tem todas as funções necessárias. Se os novos menus estão disponíveis, vamos colocar nosso plugin neste menu, caso contrário, vamos usar o velho: *Plugins*. Aqui está um exemplo para menu *Raster*

```
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 18

Liberando seu complemento

Uma vez que o seu complemento estiver pronto e você achar que o complemento pode ser útil para algumas pessoas, não hesite em enviá-lo para *Repositório oficial de complementos python*. Nessa página você pode encontrar também pacotes de orientações sobre como preparar o complemento para funcionar bem com o instalador de complementos. Ou no caso de você gostaria de criar o seu próprio repositório de complementos, criando um arquivo XML simples que irá listar os complementos e seus metadados, para exemplos, ver outros repositórios de complementos.

Por favor tenha um cuidado especial com estas sugestões

18.1 Metadados e nome

- evitar o uso de um nome muito semelhante ao de complementos existentes
- Se seu complemento tem a funcionalda semelhante a de um complemento já existente, por favor explique as diferenças n campo Sobre, então o usuário saberá qual usar sem a necessidade de intalá-lo e testar.
- evitar repetir "plugin" no nome do complemento
- use o campo de descriçao em metadados para a descrição de 1 linha, e no campo Sobre para instruções mais detalhadas
- Adicione um repositório de código, um rastreador de erros, e uma página inicial; isso vai melhorar muito a possibilidade de colaboração, e pode ser feito muito facilmente com uma das infra-estruturas disponíveis na web (GitHub, GitLab, Bitbucket, etc.)
- Escolha etiquetas com cuidado, evite as não informativas (por exemplo vetor) e prefira as usualmente usada pelos outros (veja o website de complementos)
- Adicione um ícone próprio, não deixe o padrão; veja a interface QGIS por sugestão de estilos para usar

18.2 Código e ajuda

- não adicione arquivos gerados (ui_*.py, resources_rc.py, generated help files. . .) coisas inúteis (por exemplo. .gitignore) no repositório

- Adicione o complemento no menu apropriado (Vector, Raster, Web, Database)
- quando apropriado (complements realizando análises), considere acrescentar o complemento como uma sub=complemento do quadro de processamento: isso vai permitir que os usuários executem em lote, para integrá-lo em fluxos de trabalho mais complexos, e vai livrá-lo do fardo de projetar uma interface
- Adicione no mínimo uma pequena documentação, se for útil para teste e entendimento, dados de exemplo

18.3 Repositório oficial de complementos python

Você pode encontrar o repositório *oficial* de complementos python emt http://plugins.qgis.org/.

Para usar o repositório oficial, deve obter uma ID OSGEO do portal web OSGEO.

Depois de carregado o seu complemento será aprovado por um membro da equipe e você será notificado.

TODO: Insira um link para o documento de governança.

18.3.1 Permissões

Estas regras foram implementadas no repositório oficial de complementos:

- todos os usuários registrados podem adicionar um novo plugin
- A *equipe* de usuários podem aprovar ou reprovar todas as versões do plugin
- Os usuários que têm permissão especial *plugins.can_approve* automaticamente têm suas versões de extensão aprovado
- usuários que têm a permissão especial *plugins.can_approve* pode aprovar versões carregadas por outros, enquanto eles estão na lista do *proprietários* de complementos
- um complemento específico pode ser apagado e editado apenas pela *equipe* de usuários e pelos *proprietários* do complemento
- se um usuário sem permissão *plugins.can_approve* carrega uma nova versão, a versão do plugin é automaticamente não aprovado.

18.3.2 Gerenciamento de confiança

Os membros da equipe podem conceder *permissão* para criadores de complementos selecionado definindo permissão *plugins.can_approve* através da aplicação front-end.

A visão de detalhes do complemento oferece links diretos para conceder permissão para o criador do complemento ou o *proprietários* do complemento.

18.3.3 Validação

Metadados do complemento são automaticamente importados e validados a partir do pacote compactado quando o complemento é carregado.

Aqui estão algumas regras de validação que você deve estar ciente quando desejar fazer o upload de um complemento no repositório oficial:

1. o nome de sua pasta principal contendo seu complementeo deve conter apenas caracteres ASCII (A-Z e a-z), números e os caracteres underscore (_) e menos (-), e também não pode iniciar com um número.
2. `metadata.txt` é necessário
3. todos metadados requeridos e que devem estar presentes estão listados em *metadata table*
4. o campo *versão* metadados deve ser exclusivo

18.3.4 Estrutura dos complementos

Seguindo as regras de validação, o pacote compactado (.zip) de seu plugin deve ter uma estrutura específica para ser validado como um plugin funcional. Como o plugin será descompactado dentro da pasta de plugins de usuários ele deve ter seu próprio diretório dentro do arquivo .zip para não interfirir em outros plugins. Arquivos obrigatórios: `metadata.txt` e `__init__.py`. Seria melhor se tivesse um `README` e também um ícone representando o plugin (`resources.qrc`). A seguir, um exemplo de como um plugin.zip deve se parecer.

```
plugin.zip
  pluginfolder/
  |-- i18n
  |   |-- translation_file_de.ts
  |-- img
  |   |-- icon.png
  |   `-- iconsource.svg
  |-- __init__.py
  |-- Makefile
  |-- metadata.txt
  |-- more_code.py
  |-- main_code.py
  |-- README
  |-- resources.qrc
  |-- resources_rc.py
  `-- ui_Qt_user_interface_file.ui
```

CHAPTER 19

Fragmentos de código

Esta sección cuenta con fragmentos de código para facilitar el desarrollo de complementos.

19.1 Cómo llamar a un método por un atajo de teclado

No complemento adicione `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)
```

Para `unload()` adicione

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

O metódo que ~e chamado quando F7 é pressionada

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

19.2 Como alternar camadas

Desde QGIS 2.4 há uma nova estrutura de camada de API que permite o acesso direto à estrutura de camada na legenda. Aqui está um exemplo de como alternar a visibilidade da camada ativa

```
root = QgsProject.instance().layerTreeRoot()
node = root.findLayer(iface.activeLayer().id())
new_state = Qt.Checked if node.isVisible() == Qt.Unchecked else Qt.Unchecked
node.setVisible(new_state)
```

19.3 Cómo acceder a la tabla de atributos de los objetos espaciales seleccionados

```
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")
```

Este metódo requer um parâmetro (o novo valor para o campo de atributo da característica(s) selecionada) e pode ser chamado por

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

CHAPTER 20

Escrevendo um complemento de processamento

Depending on the kind of plugin that you are going to develop, it might be better option to add its functionality as a Processing algorithm (or a set of them). That would provide a better integration within QGIS, additional functionality (since it can be run in the components of Processing, such as the modeler or the batch processing interface), and a quicker development time (since Processing will take of a large part of the work).

Este documento descreve como criar um novo complemento que adicionar sua funcionalidade como um algoritmo de processamento.

Há dois mecanismos principais para fazer isso:

- Criando um complemento que adiciona um provedor de algoritmo: Estas opções são mais complexas, mas provém maior flexibilidade
- Criando um plugin que contém um conjunto de scripts de processamento: na solução mais simples, você só precisa de um conjunto de arquivos de scripts de processamento.

20.1 Criando um complemente que adiciona um provedor de algoritmo

Para criar um provedor de algoritmo, siga estes passos:

- Instale o complemento Plugin Builder
- Crie um novo complemento usando o Plugin Builder. Quando o Plugin Builder perguntar qual modelo usar, selecione “Processing provider”.
- The created plugin contains a provider with a single algorithm. Both the provider file and the algorithm file are fully commented and contain information about how to modify the provider and add additional algorithms. Refer to them for more information.

20.2 Criando um plugin que contém um conjunto de scripts de processamento

Para criar um conjunto de scripts de processamento, siga estes passos:

- Crie seus script como descrito no manual PyGIS. Todos os scripts que você deseja adicionar devem ser disponíveis na caixa de Processamento.

- In the *Scripts/Tools* group in the Processing toolbox, double-click on the *Create script collection plugin* item. You will see a window where you should select the scripts to add to the plugin (from the set of available ones in the toolbox), and some additional information needed for the plugin metadata.
- Clique em OK e o complemento será criado.
- You can add additional scripts to the plugin by adding scripts python files to the *scripts* folder in the resulting plugin folder.

CHAPTER 21

Biblioteca de análise de rede

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

- criar gráfico matemático para dado geográfico (camadas vetor polígono)
- implements basic methods from graph theory (currently only Dijkstra's algorithm)

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

21.1 Informação Geral

Resumidamente, um caso típico pode ser descrito assim:

1. criar gráfico para geodata (camada vetorial polígono usual)
2. rodar análise gráfica
3. usar resultados das análises (por exemplo, visualizá-los)

21.2 Elaborando um gráfico

A primeira coisa que você deve fazer — é preparar os dados de entrada, que é converter uma camada vetor em um gráfico. Todas as próximas ações irão usar este gráfico e não a camada.

As a source we can use any polyline vector layer. Nodes of the polylines become graph vertexes, 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 — the closest graph vertex or closest graph edge. In the latter case the edge will be split and a new vertex added.

Vector layer attributes and length of an edge can be used as the properties of an edge.

Converting from a vector layer to the graph is done using the Builder programming pattern. A graph is constructed using a 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 the

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 the 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 a strategy that computes travel time using edge length and speed value from attributes.

It's time to dive into the process.

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

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

Then some examples for creating a director

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

# use field with index 5 as source of information about road direction.
# one-way roads with direct direction have attribute value "yes",
# one-way roads with reverse direction have the value "1", and accordingly
# bidirectional roads have "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 a vector layer, that will be used as the source for the graph structure and information about allowed movement on each road segment (one-way or bidirectional movement, direct or reverse direction). The call looks like this

```
director = QgsLineVectorLayerDirector(vl, directionFieldId,
                                      directDirectionValue,
                                      reverseDirectionValue,
                                      bothDirectionValue,
                                      defaultDirection)
```

Aqui segue uma lista de todos os significados dos parâmetros:

- `vl` — vector layer used to build the graph
- `directionFieldId` — index of the attribute table field, where information about roads direction is stored. If `-1`, then don't use this info at all. An integer.
- `directDirectionValue` — field value for roads with direct direction (moving from first line point to last one). A string.
- `reverseDirectionValue` — field value for roads with reverse direction (moving from last line point to first one). A string.
- `bothDirectionValue` — field value for bidirectional roads (for such roads we can move from first point to last and from last to first). A string.
- `defaultDirection` — default road direction. This value will be used for those roads where field `directionFieldId` is not set or has some value different from any of the three values specified above. An integer. `1` indicates direct direction, `2` indicates reverse direction, and `3` indicates both directions.

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

```
properter = QgsDistanceArcProperter()
```

And tell the director about this strategy

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

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

- src — sistema de referência de coordenadas para uso. Argumento prioritário.
- otfAtivado — uso ou não da reprojeção "voo livre". Por padrão consta: *Verdadeiro* (use OTF).
- tolerânciaTopológica — tolerância topológica. Valor padrão é 0.
- elipsóideID — elipsóide usado. Por padrão "WGS84".

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

Também podemos definir vários pontos, que serão usados na análise. Por exemplo

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

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

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

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

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

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

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

21.3 Análise de Gráficos

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

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

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

- only one vertex has no incoming edges — the root of the tree
- todos os outros vértices têm apenas uma borda chegando
- if vertex B is reachable from vertex A, then the path from A to B is the single available path and it is optimal (shortest) on this graph

To get the shortest path tree use the 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 a new graph object (QgsGraph) and accepts three variables:

- fonte — gráfico de entrada
- 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 some very simple code to display the shortest path tree using the graph created with the `shortestTree()` method (select linestring layer in TOC and replace coordinates with your own). **Warning**: use this code only as an example, it creates a lots of QgsRubberBand objects and may be slow on large data-sets.

```
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())
```

21.3.1 Encontrando os caminhos mais curtos

To find the optimal path between two points the following approach is used. Both points (start A and end B) are "tied" to the graph when it is built. Then using the methods `shortestTree()` or `dijkstra()` we build the shortest path tree with root in the start point A. In the same tree we also find the end point B and start to walk through the tree from point B to point A. The 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 the path, in the form of the inverted list of vertexes (vertexes are listed in reversed order from end point to start point) 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 `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(-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)
```

21.3.2 Areas of availability

The area of availability for vertex A is the subset of graph vertexes that are accessible from vertex A and the cost of the paths from A to these vertexes are not greater that some value.

More clearly this can be shown with the following example: "There is a fire station. Which parts of city can a fire truck reach in 5 minutes? 10 minutes? 15 minutes?". Answers to these questions are fire station's areas of availability.

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

A more difficult problem is to get the borders of the area of availability. The bottom border is the set of vertexes that are still accessible, and the top border is the set of vertexes that are not accessible. In fact this is simple: it is the availability border based on the edges of the shortest path tree for which the source vertex of the edge is accessible and the target vertex of the edge is not.

Aqui está um exemplo

```
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))
```

CHAPTER 22

Complementos de servidores Python do QGIS

Python plugins can also run on QGIS Server (see *label_qgisserver*): by using the *server interface* (`QgsServerInterface`) a Python plugin running on the server can alter the behavior of existing core services (**WMS**, **WFS** etc.).

With the *server filter interface* (`QgsServerFilter`) we can change the input parameters, change the generated output or even by providing new services.

With the *access control interface* (`QgsAccessControlFilter`) we can apply some access restriction per requests.

22.1 Server Filter Plugins architecture

Server python plugins are loaded once when the FCGI application starts. They register one or more `QgsServerFilter` (from this point, you might find useful a quick look to the server plugins API docs). Each filter should implement at least one of three callbacks:

- `requestReady()`
- `responseComplete()`
- `sendResponse()`

All filters have access to the request/response object (`QgsRequestHandler`) and can manipulate all its properties (input/output) and raise exceptions (while in a quite particular way as we'll see below).

Here is a pseudo code showing a typical server session and when the filter's callbacks are called:

- **Get the incoming request**
  - create GET/POST/SOAP request handler
  - pass request to an instance of `QgsServerInterface`
  - call plugins `requestReady()` filters
  - **if there is not a response**
    - **if SERVICE is WMS/WFS/WCS**
      - **criar servidor WMS/WFS/WCS**

        call server's `executeRequest()` and possibly call `sendResponse()` plugin filters when streaming output or store the byte stream output and content type in the request handler
    - call plugins `responseComplete()` filters
  - call plugins `sendResponse()` filters
  - request handler output the response

The following paragraphs describe the available callbacks in details.

22.1.1 requestReady

This is called when the request is ready: incoming URL and data have been parsed and before entering the core services (WMS, WFS etc.) switch, this is the point where you can manipulate the input and perform actions like:

- authentication/authorization
- redirects
- add/remove certain parameters (typenames for example)
- raise exceptions

You could even substitute a core service completely by changing **SERVICE** parameter and hence bypassing the core service completely (not that this make much sense though).

22.1.2 sendResponse

This is called whenever output is sent to **FCGI** `stdout` (and from there, to the client), this is normally done after core services have finished their process and after responseComplete hook was called, but in a few cases XML can become so huge that a streaming XML implementation was needed (WFS GetFeature is one of them), in this case, `sendResponse()` is called multiple times before the response is complete (and before `responseComplete()` is called). The obvious consequence is that `sendResponse()` is normally called once but might be exceptionally called multiple times and in that case (and only in that case) it is also called before `responseComplete()`.

`sendResponse()` is the best place for direct manipulation of core service's output and while `responseComplete()` is typically also an option, `sendResponse()` is the only viable option in case of streaming services.

22.1.3 responseComplete

This is called once when core services (if hit) finish their process and the request is ready to be sent to the client. As discussed above, this is normally called before `sendResponse()` except for streaming services (or other plugin filters) that might have called `sendResponse()` earlier.

`responseComplete()` is the ideal place to provide new services implementation (WPS or custom services) and to perform direct manipulation of the output coming from core services (for example to add a watermark upon a WMS image).

22.2 Raising exception from a plugin

Some work has still to be done on this topic: the current implementation can distinguish between handled and unhandled exceptions by setting a `QgsRequestHandler` property to an instance of `QgsMapServiceException`, this way the main C++ code can catch handled python exceptions and ignore unhandled exceptions (or better: log them).

This approach basically works but it is not very "pythonic": a better approach would be to raise exceptions from python code and see them bubbling up into C++ loop for being handled there.

22.3 Writing a server plugin

A server plugins is just a standard QGIS Python plugin as described in *Desenvolvimento de Complementos Python*, that just provides an additional (or alternative) interface: a typical QGIS desktop plugin has access to QGIS application through the `QgisInterface` instance, a server plugin has also access to a `QgsServerInterface`.

To tell QGIS Server that a plugin has a server interface, a special metadata entry is needed (in *metadata.txt*)

```
server=True
```

The example plugin discussed here (with many more example filters) is available on github: QGIS HelloServer Example Plugin

22.3.1 Plugin files

Here's the directory structure of our example server plugin

```
PYTHON_PLUGINS_PATH/
  HelloServer/
    __init__.py     --> *required*
    HelloServer.py  --> *required*
    metadata.txt    --> *required*
```

22.3.2 __init__.py

This file is required by Python's import system. Also, QGIS Server requires that this file contains a `serverClassFactory()` function, which is called when the plugin gets loaded into QGIS Server when the server starts. It receives reference to instance of `QgsServerInterface` and must return instance of your plugin's class. This is how the example plugin *__init__.py* looks like:

```
# -*- coding: utf-8 -*-

def serverClassFactory(serverIface):
    from HelloServer import HelloServerServer
    return HelloServerServer(serverIface)
```

22.3.3 HelloServer.py

This is where the magic happens and this is how magic looks like: (e.g. `HelloServer.py`)

A server plugin typically consists in one or more callbacks packed into objects called QgsServerFilter.

Each `QgsServerFilter` implements one or more of the following callbacks:

- `requestReady()`
- `responseComplete()`
- `sendResponse()`

The following example implements a minimal filter which prints *HelloServer!* in case the **SERVICE** parameter equals to "HELLO":

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

class HelloFilter(QgsServerFilter):

    def __init__(self, serverIface):
        super(HelloFilter, self).__init__(serverIface)

    def responseComplete(self):
        request = self.serverInterface().requestHandler()
        params = request.parameterMap()
        if params.get('SERVICE', '').upper() == 'HELLO':
            request.clearHeaders()
            request.setHeader('Content-type', 'text/plain')
            request.clearBody()
            request.appendBody('HelloServer!')
```

The filters must be registered into the **serverIface** as in the following example:

```
class HelloServerServer:
    def __init__(self, serverIface):
        # Save reference to the QGIS server interface
        self.serverIface = serverIface
        serverIface.registerFilter( HelloFilter, 100 )
```

The second parameter of `registerFilter()` allows to set a priority which defines the order for the callbacks with the same name (the lower priority is invoked first).

By using the three callbacks, plugins can manipulate the input and/or the output of the server in many different ways. In every moment, the plugin instance has access to the `QgsRequestHandler` through the `QgsServerInterface`, the `QgsRequestHandler` has plenty of methods that can be used to alter the input parameters before entering the core processing of the server (by using `requestReady()`) or after the request has been processed by the core services (by using `sendResponse()`).

The following examples cover some common use cases:

22.3.4 Modifying the input

The example plugin contains a test example that changes input parameters coming from the query string, in this example a new parameter is injected into the (already parsed) `parameterMap`, this parameter is then visible by core services (WMS etc.), at the end of core services processing we check that the parameter is still there:

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

class ParamsFilter(QgsServerFilter):
```

```
    def __init__(self, serverIface):
        super(ParamsFilter, self).__init__(serverIface)

    def requestReady(self):
        request = self.serverInterface().requestHandler()
        params = request.parameterMap( )
        request.setParameter('TEST_NEW_PARAM', 'ParamsFilter')

    def responseComplete(self):
        request = self.serverInterface().requestHandler()
        params = request.parameterMap( )
        if params.get('TEST_NEW_PARAM') == 'ParamsFilter':
            QgsMessageLog.logMessage("SUCCESS - ParamsFilter.responseComplete", 'plugin', QgsMessa
        else:
            QgsMessageLog.logMessage("FAIL    - ParamsFilter.responseComplete", 'plugin', QgsMessa
```

This is an extract of what you see in the log file:

```
src/core/qgsmessagelog.cpp: 45: (logMessage) [0ms] 2014-12-12T12:39:29 plugin[0] HelloServerServe
src/core/qgsmessagelog.cpp: 45: (logMessage) [1ms] 2014-12-12T12:39:29 Server[0] Server plugin He
src/core/qgsmessagelog.cpp: 45: (logMessage) [0ms] 2014-12-12T12:39:29 Server[0] Server python pl
src/mapserver/qgsgetrequesthandler.cpp: 35: (parseInput) [0ms] query string is: SERVICE=HELLO&req
src/mapserver/qgshttprequesthandler.cpp: 547: (requestStringToParameterMap) [1ms] inserting pair
src/mapserver/qgshttprequesthandler.cpp: 547: (requestStringToParameterMap) [0ms] inserting pair
src/mapserver/qgsserverfilter.cpp: 42: (requestReady) [0ms] QgsServerFilter plugin default reques
src/core/qgsmessagelog.cpp: 45: (logMessage) [0ms] 2014-12-12T12:39:29 plugin[0] HelloFilter.requ
src/mapserver/qgis_map_serv.cpp: 235: (configPath) [0ms] Using default configuration file path:
src/mapserver/qgshttprequesthandler.cpp: 49: (setHttpResponse) [0ms] Checking byte array is ok to
src/mapserver/qgshttprequesthandler.cpp: 59: (setHttpResponse) [0ms] Byte array looks good, setti
src/core/qgsmessagelog.cpp: 45: (logMessage) [0ms] 2014-12-12T12:39:29 plugin[0] HelloFilter.resp
src/core/qgsmessagelog.cpp: 45: (logMessage) [0ms] 2014-12-12T12:39:29 plugin[0] SUCCESS - Params
src/core/qgsmessagelog.cpp: 45: (logMessage) [0ms] 2014-12-12T12:39:29 plugin[0] RemoteConsoleFil
src/mapserver/qgshttprequesthandler.cpp: 158: (sendResponse) [0ms] Sending HTTP response
src/core/qgsmessagelog.cpp: 45: (logMessage) [0ms] 2014-12-12T12:39:29 plugin[0] HelloFilter.send
```

On the highlighted line the "SUCCESS" string indicates that the plugin passed the test.

The same technique can be exploited to use a custom service instead of a core one: you could for example skip a **WFS SERVICE** request or any other core request just by changing the **SERVICE** parameter to something different and the core service will be skipped, then you can inject your custom results into the output and send them to the client (this is explained here below).

22.3.5 Modifying or replacing the output

The watermark filter example shows how to replace the WMS output with a new image obtained by adding a watermark image on the top of the WMS image generated by the WMS core service:

```
import os

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


class WatermarkFilter(QgsServerFilter):

    def __init__(self, serverIface):
        super(WatermarkFilter, self).__init__(serverIface)
```

```
    def responseComplete(self):
        request = self.serverInterface().requestHandler()
        params = request.parameterMap( )
        # Do some checks
        if (request.parameter('SERVICE').upper() == 'WMS' \
                and request.parameter('REQUEST').upper() == 'GETMAP' \
                and not request.exceptionRaised() ):
            QgsMessageLog.logMessage("WatermarkFilter.responseComplete: image ready %s" % request
            # Get the image
            img = QImage()
            img.loadFromData(request.body())
            # Adds the watermark
            watermark = QImage(os.path.join(os.path.dirname(__file__), 'media/watermark.png'))
            p = QPainter(img)
            p.drawImage(QRect( 20, 20, 40, 40), watermark)
            p.end()
            ba = QByteArray()
            buffer = QBuffer(ba)
            buffer.open(QIODevice.WriteOnly)
            img.save(buffer, "PNG")
            # Set the body
            request.clearBody()
            request.appendBody(ba)
```

In this example the **SERVICE** parameter value is checked and if the incoming request is a **WMS GETMAP** and no exceptions have been set by a previously executed plugin or by the core service (WMS in this case), the WMS generated image is retrieved from the output buffer and the watermark image is added. The final step is to clear the output buffer and replace it with the newly generated image. Please note that in a real-world situation we should also check for the requested image type instead of returning PNG in any case.

22.4 Access control plugin

22.4.1 Plugin files

Here's the directory structure of our example server plugin:

```
PYTHON_PLUGINS_PATH/
  MyAccessControl/
    __init__.py    --> *required*
    AccessControl.py  --> *required*
    metadata.txt   --> *required*
```

22.4.2 __init__.py

This file is required by Python's import system. As for all QGIS server plugins, this file contains a `serverClassFactory()` function, which is called when the plugin gets loaded into QGIS Server when the server starts. It receives reference to instance of `QgsServerInterface` and must return instance of your plugin's class. This is how the example plugin *__init__.py* looks like:

```
# -*- coding: utf-8 -*-

def serverClassFactory(serverIface):
    from MyAccessControl.AccessControl import AccessControl
    return AccessControl(serverIface)
```

22.4.3 AccessControl.py

```
class AccessControl(QgsAccessControlFilter):

    def __init__(self, server_iface):
        super(QgsAccessControlFilter, self).__init__(server_iface)

    def layerFilterExpression(self, layer):
        """ Return an additional expression filter """
        return super(QgsAccessControlFilter, self).layerFilterExpression(layer)

    def layerFilterSubsetString(self, layer):
        """ Return an additional subset string (typically SQL) filter """
        return super(QgsAccessControlFilter, self).layerFilterSubsetString(layer)

    def layerPermissions(self, layer):
        """ Return the layer rights """
        return super(QgsAccessControlFilter, self).layerPermissions(layer)

    def authorizedLayerAttributes(self, layer, attributes):
        """ Return the authorised layer attributes """
        return super(QgsAccessControlFilter, self).authorizedLayerAttributes(layer, attributes)

    def allowToEdit(self, layer, feature):
        """ Are we authorise to modify the following geometry """
        return super(QgsAccessControlFilter, self).allowToEdit(layer, feature)

    def cacheKey(self):
        return super(QgsAccessControlFilter, self).cacheKey()
```

This example gives a full access for everybody.

It's the role of the plugin to know who is logged on.

On all those methods we have the layer on argument to be able to customise the restriction per layer.

22.4.4 layerFilterExpression

Used to add an Expression to limit the results, e.g.:

```
def layerFilterExpression(self, layer):
    return "$role = 'user'"
```

To limit on feature where the attribute role is equals to "user".

22.4.5 layerFilterSubsetString

Same than the previous but use the `SubsetString` (executed in the database)

```
def layerFilterSubsetString(self, layer):
    return "role = 'user'"
```

To limit on feature where the attribute role is equals to "user".

22.4.6 layerPermissions

Limit the access to the layer.

Return an object of type `QgsAccessControlFilter.LayerPermissions`, who has the properties:

- `canRead` to see him in the `GetCapabilities` and have read access.
- `canInsert` to be able to insert a new feature.
- `canUpdate` to be able to update a feature.
- `candelete` to be able to delete a feature.

Exemplo:

```
def layerPermissions(self, layer):
    rights = QgsAccessControlFilter.LayerPermissions()
    rights.canRead = True
    rights.canRead = rights.canInsert = rights.canUpdate = rights.canDelete = False
    return rights
```

To limit everything on read only access.

22.4.7 authorizedLayerAttributes

Used to limit the visibility of a specific subset of attribute.

The argument attribute return the current set of visible attributes.

Exemplo:

```
def authorizedLayerAttributes(self, layer, attributes):
    return [a for a in attributes if a != "role"]
```

To hide the 'role' attribute.

22.4.8 allowToEdit

This is used to limit the editing on a subset of features.

It is used in the `WFS-Transaction` protocol.

Exemplo:

```
def allowToEdit(self, layer, feature):
    return feature.attribute('role') == 'user'
```

To be able to edit only feature that has the attribute role with the value user.

22.4.9 cacheKey

QGIS server maintain a cache of the capabilities then to have a cache per role you can return the role in this method. Or return `None` to completely disable the cache.