
PyQGIS developer cookbook

Publicación 2.18

QGIS Project

08 de April de 2019

1	Introducción	1
1.1	Run Python code when QGIS starts	1
1.2	Python Console	2
1.3	Plugins Python	3
1.4	Aplicaciones Python	3
2	Cargar proyectos	7
3	Cargar capas	9
3.1	Capas Vectoriales	9
3.2	Capas ráster	11
3.3	Map Layer Registry	11
4	Usar las capas ráster	13
4.1	Detalles de la capa	13
4.2	Renderizador	13
4.3	Refreshing Layers	15
4.4	Valores de consulta	15
5	Usar capas vectoriales	17
5.1	Recuperando información sobre atributos	17
5.2	Selecting features	18
5.3	Iterando sobre la capa vectorial	18
5.4	Modifying Vector Layers	20
5.5	Modifying Vector Layers with an Editing Buffer	22
5.6	Usar índice espacial	23
5.7	Writing Vector Layers	23
5.8	Memory Provider	24
5.9	Appearance (Symbology) of Vector Layers	26
5.10	Más Temas	33
6	Manejo de Geometría	35
6.1	Construcción de Geometría	35
6.2	Acceso a Geometría	36
6.3	Geometría predicados y Operaciones	36
7	Soporte de Proyecciones	39
7.1	Sistemas de coordenadas de referencia	39
7.2	Projections	40
8	Using Map Canvas	41
8.1	Lienzo de mapa insertado	41
8.2	Utilizar las herramientas del mapa con el lienzo	42

8.3	Bandas elásticas y marcadores de vértices	43
8.4	Escribir herramientas de mapa personalizados	44
8.5	Escribir elementos de lienzo de mapa personalizado	45
9	Representación del Mapa e Impresión	47
9.1	Representación Simple	47
9.2	Representando capas con diferente SRC	48
9.3	Output using Map Composer	48
10	Expresiones, Filtros y Calculando Valores	51
10.1	Análisis de expresiones	52
10.2	Evaluar expresiones	52
10.3	Ejemplos	53
11	Configuración de lectura y almacenamiento	55
12	Comunicarse con el usuario	57
12.1	Showing messages. The QgsMessageBar class	57
12.2	Mostrando el progreso	58
12.3	Registro	59
13	Desarrollo de Plugins Python	61
13.1	Escribir un complemento	62
13.2	Contenido del complemento	63
13.3	Documentación	67
13.4	Traducción	67
14	Authentication infrastructure	71
14.1	Introducción	71
14.2	Glosario	71
14.3	QgsAuthManager the entry point	72
14.4	Adapt plugins to use Authentication infrastructure	75
14.5	Authentication GUIs	75
15	Configuración IDE para escribir y depurar complementos	79
15.1	Una nota sobre la configuración su IDE sobre Windows	79
15.2	Depure utilizando eclipse y PyDev	80
15.3	Depure utilizando PDB	84
16	Utilizar complemento Capas	85
16.1	Subclassing QgsPluginLayer	85
17	Compatibilidad con versiones antiguas de QGIS	87
17.1	Menu de plugins	87
18	Compartiendo sus plugins	89
18.1	Metadatos y nombres	89
18.2	Code and help	89
18.3	Repositorio oficial de complemento	90
19	Fragmentos de código	93
19.1	Cómo llamar a un método por un atajo de teclado	93
19.2	Como alternar capas	93
19.3	Cómo acceder a la tabla de atributos de los objetos espaciales seleccionados	94
20	Escribir nuevos complementos de procesamiento	95
20.1	Creating a plugin that adds an algorithm provider	95
20.2	Creating a plugin that contains a set of processing scripts	95

21 Biblioteca de análisis de redes	97
21.1 Información general	97
21.2 Contruir un gráfico	97
21.3 Análisis gráfico	99
22 Complementos de Python de QGIS Server	105
22.1 Server Filter Plugins architecture	105
22.2 Raising exception from a plugin	107
22.3 Writing a server plugin	107
22.4 Complemento control de acceso	110
Índice	113

Introducción

- Run Python code when QGIS starts
 - variable de entorno PYQGIS_STARTUP
 - El fichero `startup.py`
- Python Console
- Plugins Python
- Aplicaciones Python
 - Usando PyQGIS en scripts individuales
 - Usando PyQGIS en aplicaciones personalizadas
 - Running Custom Applications

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

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

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

- automatically run Python code when QGIS starts
- issue commands in Python console within QGIS
- create and use plugins in Python
- create custom applications based on QGIS API

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.

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

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 Run Python code when QGIS starts

Existen dos métodos distintos para ejecutar código Python cada vez que QGIS inicia.

1.1.1 variable de entorno PYQGIS_STARTUP

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.

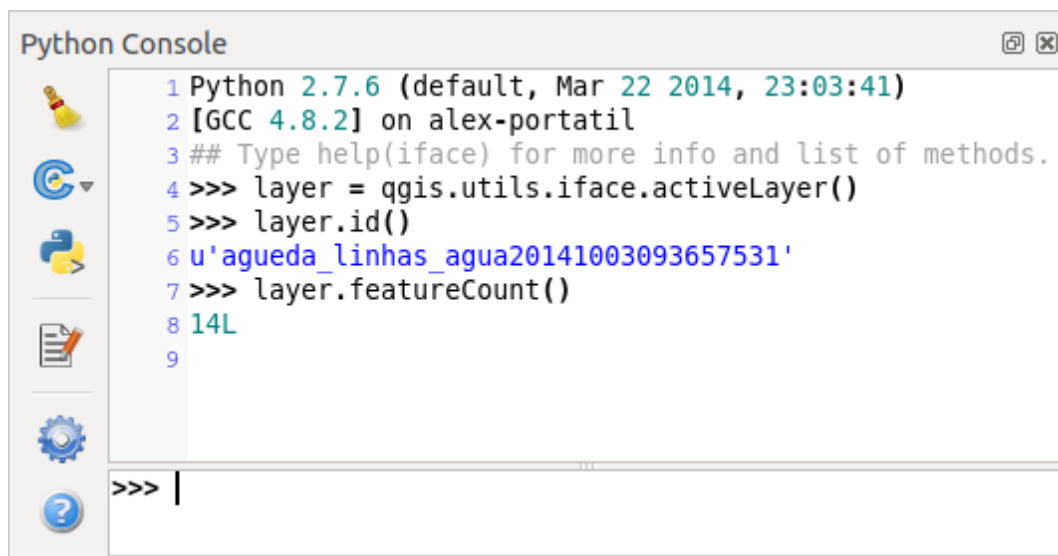
This method is something you will probably rarely need, but worth mentioning here because it is one of the several ways to run Python code within QGIS and because this code will run before QGIS initialization is complete. This method is very useful for cleaning `sys.path`, which may have undesirable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

1.1.2 El fichero `startup.py`

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 Python Console

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



```

Python Console
1 Python 2.7.6 (default, Mar 22 2014, 23:03:41)
2 [GCC 4.8.2] on alex-portatil
3 ## Type help(iface) for more info and list of methods.
4 >>> layer = qgis.utils.iface.activeLayer()
5 >>> layer.id()
6 u'aguada_linhas_agua20141003093657531'
7 >>> layer.featureCount()
8 14L
9
>>> |

```

Figure 1.1: Consola Python de 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 Plugins 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 *Desarrollo de Plugins Python* for detailed instructions.

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

1.4 Aplicaciones 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 Usando PyQGIS en scripts individuales

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 *Python Console* 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 Usando PyQGIS en aplicaciones personalizadas

The only difference between *Usando PyQGIS en scripts individuales* 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 Running Custom Applications

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

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

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

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

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

```
>>> import qgis.core
```

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

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

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

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

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

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

Cargar proyectos

Sometimes you need to load an existing project from a plugin or (more often) when developing a stand-alone QGIS Python application (see: *Aplicaciones 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()` functions 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 instantiate 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'))
```

Cargar capas

- Capas Vectoriales
- Capas ráster
- Map Layer Registry

Vamos a abrir algunas capas con datos. QGIS reconoce capas vectoriales y ráster. Además, están disponibles tipos de capas personalizadas, pero no se va a discutir de ellas aquí.

3.1 Capas Vectoriales

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!"
```

El identificador de la fuente de datos es una cadena y se especifica a cada proveedor de datos vectoriales. El nombre de la capa se utiliza en el widget de la lista de capa. Es importante validar si la capa se ha cargado satisfactoriamente. Si no fue así, se devuelve una instancia de capa no válida.

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.

La siguiente lista muestra cómo acceder a varias fuentes de datos utilizando los proveedores de datos vectoriales:

- 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")

```

- Los archivos GPX — el proveedor de datos “gpx” lee los caminos, rutas y puntos de interés desde archivos GPX. Para abrir un archivo, el tipo (caminos/ruta/punto de interés) se debe especificar como parte de la 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')

```

- Las geometrias basadas en WKB de MySQL, a través de OGR — la fuente de datos es la cadena de conexión a la tabla:

```

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

```

- Conexión WFS: se define con un URI y utiliza el proveedor WFS:

```

uri = "http://localhost:8080/geoserver/wfs?srsname=EPSG:23030&typename=union&version=1.0.0&re"
vlayer = QgsVectorLayer(uri, "my wfs layer", "WFS")

```

La uri se puede crear utilizando la librería estándar `urllib`:

```

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 Capas ráster

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")

```

This creates a new layer and adds it to the map layer registry (making it appear in the layer list) in one step.

Las capas ráster también se pueden crear desde el servicio WCS:

```

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](#)

Como alternativa se puede cargar una capa ráster desde un servidor WMS. Sin embargo actualmente no es posible acceder a las respuestas de `GetCapabilities` desde el API — se debe saber que capas desea:

```

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 Map Layer Registry

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

Adding a layer to the registry:

```

QgsMapLayerRegistry.instance().addMapLayer(layer)

```

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

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

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

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

Usar las capas ráster

- Detalles de la capa
- Renderizador
 - Rásters de una sola banda
 - Rásters multibanda
- Refreshing Layers
- Valores de consulta

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

4.1 Detalles de la capa

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 0x00000000F8A2048>
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 Renderizador

Cuando se carga una capa ráster, se asigna un estilo de renderizado basado en la escritura. Se puede cambiar en las propiedades de la capa ráster o de manera programática.

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`

Las capas ráster de una sola banda se puede dibujar ya sea en colores grises (valores bajos = negro, valores altos = blanco) o con un algoritmo de pseudocolor que asigna colores para los valores de una sola banda. Los ráster de una sola banda con una paleta además se pueden dibujar al utilizar su paleta. Capas multibanda suelen dibujarse mediante la asignación de las bandas de colores RGB. Otra posibilidad es utilizar una sola banda para el dibujo gris o pseudocolor.

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

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

4.2.1 Rásters de una sola 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)
```

Los colores del mapa de sombreado como se especifican por su mapa de color. El mapa de color se proporciona como una lista de elementos con valores de píxel y su color asociado. Hay tres modos de interpolación de valores:

- `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 Rásters multibanda

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 Refreshing Layers

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

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

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

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 Valores de consulta

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}
```

Usar capas vectoriales

- Recuperando información sobre atributos
- Selecting features
- Iterando sobre la capa vectorial
 - Accediendo atributos
 - Iterando sobre rasgos seleccionados
 - Iterando sobre un subconjunto de rasgos
- Modifying Vector Layers
 - Añadir objetos espaciales
 - Borrar objetos espaciales
 - Modify Features
 - Adding and Removing Fields
- Modifying Vector Layers with an Editing Buffer
- Usar índice espacial
- Writing Vector Layers
- Memory Provider
- Appearance (Symbology) of Vector Layers
 - Representador de Símbolo Único
 - Representador de símbolo categorizado
 - Graduated Symbol Renderer
 - Trabajo con Símbolos
 - * Working with Symbol Layers
 - * Creating Custom Symbol Layer Types
 - Creating Custom Renderers
- Más Temas

Esta sección resume varias acciones que pueden ser realizadas con las capas vectoriales

5.1 Recuperando información sobre atributos

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 Selecting features

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 Iterando sobre la capa vectorial

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() == Qgs.Point:
        x = geom.asPoint()
        print "Point: " + str(x)
    elif geom.type() == Qgs.Line:
        x = geom.asPolyline()
        print "Line: %d points" % len(x)
    elif geom.type() == Qgs.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 Accediendo atributos

Attributes can be referred to by their name.

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

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

```
print feature[0]
```

5.3.2 Iterando sobre rasgos seleccionados

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
```

Another option is the `Processing features()` method:

```
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 un subconjunto de rasgos

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

```
request = QgsFeatureRequest()
request.setFilterRect(areaOfInterest)
for 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 *Expresiones, Filtros y Calculando 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)
```

Truco: 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 Modifying Vector Layers

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

```
caps = layer.dataProvider().capabilities()
# 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 Añadir objetos espaciales

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 Borrar objetos espaciales

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

```

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

```

5.4.3 Modify Features

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

```

fid = 100 # ID of the feature we will modify

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

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

```

Truco: 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.).

Truco: Directly save changes using `with` based command

Using `with edit(layer)`: the changes will be committed automatically calling `commitChanges()` at the end. If any exception occurs, it will `rollback()` all the changes. See *Modifying Vector Layers with an Editing Buffer*.

5.4.4 Adding and Removing Fields

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 Modifying Vector Layers with an Editing Buffer

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

To find out whether a layer is in editing mode, use `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 Usar índice espacial

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:

- create spatial index — the following code creates an empty index

```
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())
```

- once spatial index is filled with some values, you can do some queries

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

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

5.7 Writing Vector Layers

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

There are two possibilities how to export a vector layer:

- from an instance of `QgsVectorLayer`

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

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

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

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

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

- directly from features

```
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 WKBTYP enum
5. layer's spatial reference (instance of
   QgsCoordinateReferenceSystem) - optional
6. driver name for the output file """
writer = QgsVectorFileWriter("my_shapes.shp", "CP1250", fields, Qgs.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 Memory Provider

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

The provider supports string, int and double fields.

The memory provider also supports spatial indexing, which is enabled by calling the provider's `createSpatialIndex()` function. Once the spatial index is created you will be able to iterate over fea-

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

index=yes Especifica que el proveedor utilizará un índice espacial

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

The following example of a URI incorporates all these options

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

The following example code illustrates creating and populating a memory provider

```
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.xMinimum(), 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 Appearance (Symbology) of Vector Layers

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

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

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

And with that reference, let us explore it a bit

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

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

Tipo	Clase	Descripción
singleSymbol	QgsSingleSymbolRendererV2	Renders all features with the same symbol
categorizedSymbol	QgsCategorizedSymbolRendererV2	Renders features using a different symbol for each category
graduatedSymbol	QgsGraduatedSymbolRendererV2	Renders features using a different symbol for each range of values

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

```
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 Representador de Símbolo Único

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:

- circle
- cuadrado

- cross
- rectangle
- Diamante
- pentagon
- triángulo
- equilateral_triangle
- star
- regular_star
- arrow
- filled_arrowhead
- x

To get the full list of properties for the first symbol layer of a symbol 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 Representador de símbolo categorizado

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 Trabajo con Símbolos

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

- `QgsMarkerSymbolV2` — for point features
- `QgsLineSymbolV2` — for line features
- `QgsFillSymbolV2` — for polygon features

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

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

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

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

De forma predeterminada el tamaño y ancho están en milímetros, los ángulos en grados.

Working with Symbol Layers

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

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

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

Output

```
EllipseMarker
FontMarker
SimpleMarker
SvgMarker
VectorField
```

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

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

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.

We will have to create metadata for the symbol layer

```

class FooSymbolLayerMetadata(QgsSymbolLayerV2AbstractMetadata):

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

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

    def createSymbolLayerWidget(self):
        return FooSymbolLayerWidget()

```

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

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

El último pase es adicionar esta capa símbolo al registro — y estamos listos.

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.defaultSymbol]

    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 RandomRenderrer(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 `QgsRenderrerV2Widget`. The following sample code creates a button that allows user to set symbol of the first symbol

```

class RandomRenderrerWidget(QgsRenderrerV2Widget):
    def __init__(self, layer, style, renderer):
        QgsRenderrerV2Widget.__init__(self, layer, style)
        if renderer is None or renderer.type() != "RandomRenderrer":
            self.r = RandomRenderrer()
        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 `RandomRenderrer` example

```

class RandomRenderrerMetadata(QgsRenderrerV2AbstractMetadata):
    def __init__(self):
        QgsRenderrerV2AbstractMetadata.__init__(self, "RandomRenderrer", "Random renderrer")

```

```
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 Más Temas

PENDIENTE:

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

Manejo de Geometría

- Construcción de Geometría
- Acceso a Geometría
- Geometría predicados y Operaciones

Points, linestrings and polygons that represent a spatial feature are commonly referred to as geometries. In QGIS they are represented with the `QgsGeometry` class.

A veces una geometría es realmente una colección simple (partes simples) geométricas. Tal geometría se llama geometría de múltiples partes. Si contiene un tipo de geometría simple, lo llamamos un punto múltiple, líneas múltiples o polígonos múltiples. Por ejemplo, un país consiste en múltiples islas que se pueden representar como un polígono múltiple.

Las coordenadas de las geometrías pueden estar en cualquier sistema de referencia de coordenadas (SRC). Cuando extrae características de una capa, las geometrías asociadas tendrán sus coordenadas en el SRC de la capa.

Description and specifications of all possible geometries construction and relationships are available in the [OGC Simple Feature Access Standards](#) for advanced details.

6.1 Construcción de Geometría

There are several options for creating a geometry:

- desde 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)])])
```

Coordinates are given using `QgsPoint` class.

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

Las geometrías multi-parte van un nivel más allá: multi-punto es una lista de puntos, multi-línea es una lista de polilíneas y multi-polígono es una lista de polígonos.

- desde well-known text (WKT)

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

- desde well-known binary (WKB)

```
>>> g = QgsGeometry()
>>> wkb = '0101000000000000000000004540000000000001440'.decode('hex')
>>> g.fromWkb(wkb)
```

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

6.2 Acceso a Geometría

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

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

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

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

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

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

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

6.3 Geometría predicados y Operaciones

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

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

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

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

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

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

Puede encontrar muchos ejemplos de algoritmos que se incluyen en QGIS y utilizan estos métodos para analizar y transformar los datos vectoriales. Aquí hay algunos enlaces al código de algunos de ellos.

Additional information can be found in following sources:

- Geometry transformation: [Reproject algorithm](#)
- Distance and area using the `QgsDistanceArea` class: [Distance matrix algorithm](#)
- [Multi-part to single-part algorithm](#)

Soporte de Proyecciones

- Sistemas de coordenadas de referencia
- Projections

7.1 Sistemas de coordenadas de referencia

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

- especificar SRC por su ID

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

QGIS utiliza tres tipos diferentes de ID para cada sistema de referencia:

- `PostgisCrsId` — los IDs utilizados dentro de la base de datos PostGIS
- `InternalCrsId` — IDs internamente utilizados en la base de datos de QGIS.
- `EpsgCrsId` — IDs asignados por la organización EPSG

Si no se especifica lo contrario en el segundo parámetro, PostGIS SRID se utiliza por defecto.

- especificar SRC por su well-known text (WKT)

```
wkt = 'GEOGCS["WGS84", DATUM["WGS84", SPHEROID["WGS84", 6378137.0, 298.257223563]],'
      PRIMEM["Greenwich", 0.0], UNIT["degree",0.017453292519943295],
      AXIS["Longitude",EAST], AXIS["Latitude",NORTH]]'
crs = QgsCoordinateReferenceSystem(wkt)
```

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

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

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

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

Accessing spatial reference system information

```
print "QGIS CRS ID:", crs.srsid()
print "PostGIS SRID:", crs.srid()
print "EPSG ID:", crs.epsg()
print "Description:", crs.description()
print "Projection Acronym:", crs.projectionAcronym()
print "Ellipsoid Acronym:", crs.ellipsoidAcronym()
print "Proj4 String:", crs.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 Projections

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

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

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

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

Using Map Canvas

- Lienzo de mapa insertado
- Utilizar las herramientas del mapa con el lienzo
- Bandas elásticas y marcadores de vértices
- Escribir herramientas de mapa personalizados
- Escribir elementos de lienzo de mapa personalizado

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

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

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

Para resumir, la arquitectura del lienzo de mapa consiste en tres conceptos:

- lienzo de mapa — para la visualización del mapa
- map canvas items — additional items that can be displayed in map canvas
- map tools — for interaction with map canvas

8.1 Lienzo de mapa insertado

El lienzo de mapa es un widget como cualquier otro widget Qt, por lo que utilizarlo es tan sencillo como crearlo y mostrarlo

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


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

Por defecto, el lienzo de mapa tiene un fondo negro y no utiliza anti-aliasing. Para establecer el fondo blanco y habilitar el anti-aliasing para suavisar la presentación

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

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

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

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

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

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

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

Después de ejecutar estos comandos, el lienzo debe mostrar la capa que se ha cargado.

8.2 Utilizar las herramientas del mapa con el lienzo

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

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

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

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

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

        self.setCentralWidget(self.canvas)

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

```

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

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

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

# create the map tools
self.toolPan = QgsMapToolPan(self.canvas)
self.toolPan.setAction(actionPan)
self.toolZoomIn = QgsMapToolZoom(self.canvas, False) # false = in
self.toolZoomIn.setAction(actionZoomIn)
self.toolZoomOut = QgsMapToolZoom(self.canvas, True) # true = out
self.toolZoomOut.setAction(actionZoomOut)

self.pan()

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

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

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

```

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

```

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

```

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

8.3 Bandas elásticas y marcadores de vértices

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

Para mostrar una polilínea

```

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 un polígono

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

Tenga en cuenta que los puntos de polígonos no es una lista simple: de hecho, es una lista de anillos que contienen lista de anillos del polígono: el primer anillo es el borde exterior, anillos adicionales (opcional) corresponden a los agujeros en el polígono.

Las bandas elásticas permiten algún tipo de personalización, es decir, para cambiar su color o ancho de línea

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

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

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

(en C++ es posible simplemente eliminar el elemento, sin embargo en Python `del r` sería simplemente suprimir la referencia y el objeto aún existirá ya que es propiedad del lienzo)

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

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

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

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

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

8.4 Escribir herramientas de mapa personalizados

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

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

Aquí esta un ejemplo de una herramienta de mapa para definir una extensión rectangular haciendo clic y arrastrando en el lienzo. Cuando se define el rectángulo, imprime su límite de coordenadas en la consola. Utiliza los elementos de la banda elástica descrita antes para mostrar el rectángulo seleccionado ya que se está definiendo.

```
class RectangleMapTool(QgsMapToolEmitPoint):
    def __init__(self, canvas):
        self.canvas = canvas
        QgsMapToolEmitPoint.__init__(self, self.canvas)
        self.rubberBand = QgsRubberBand(self.canvas, Qgs.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(Qgs.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 Escribir elementos de lienzo de mapa personalizado

PENDIENTE: cómo crear un elemento de lienzo de mapa

```

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)
```

Representación del Mapa e Impresión

- Representación Simple
- Representando capas con diferente SRC
- Output using Map Composer
 - Output to a raster image
 - Output to PDF

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

9.1 Representación Simple

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

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

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

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

render = QgsMapRenderer()

# set layer set
lst = [layer.id()] # 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 Representando capas con diferente SRC

If you have more than one layer and they have a different CRS, the simple example above will probably not work: to get the right values from the extent calculations you have to explicitly set the destination CRS and enable OTF reprojection as in the example below (only the renderer configuration part is reported)

```
...  
# set layer set  
layers = QgsMapLayerRegistry.instance().mapLayers()  
lst = layers.keys()  
renderer.setLayerSet(lst)  
  
# Set destination CRS to match the CRS of the first layer  
renderer.setDestinationCrs(layers.values()[0].crs())  
# Enable OTF reprojection  
renderer.setProjectionsEnabled(True)  
...
```

9.3 Output using Map Composer

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

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

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

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

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

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

- `mapa` — este elemento dice a las bibliotecas dónde ponen el propio mapa. Aquí creamos un mapa y estimamos sobre el tamaño de papel

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

- **etiqueta** — permite mostrar etiquetas. Es posible modificar su letra, color, alineación y margen.

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

- **leyenda**

```
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)
```

- **flecha**

- **imagen**

- **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)
```

- **tabla**

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

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

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

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

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

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

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

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

9.3.1 Output to a raster image

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

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

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

# render the composition
imagePainter = QPainter(image)
c.renderPage(imagePainter, 0)
imagePainter.end()

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

9.3.2 Output to PDF

The following code fragment renders a composition to a PDF file

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

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

Expresiones, Filtros y Calculando Valores

- Análisis de expresiones
- Evaluar expresiones
 - Expresiones Basicas
 - Expresiones con características
 - Manejar errores
- Ejemplos

QGIS tiene apoyo para el análisis de expresiones parecidas al SQL. Solo se reconoce un pequeño subconjunto de sintaxis SQL. Las expresiones pueden ser evaluados ya sea como predicados booleanos (regresando Verdadero o Falso) o como funciones (regresando un valor escalar). Vea *vector_expressions* en el Manual del usuario para obtener una lista completa de las funciones disponibles.

Se le da apoyo a tres tipos:

- numero - números enteros y números con decimales, e.j. 123, 3.14
- cadena - se tiene que encerrar en comas individuales: 'hola mundo'
- columna de referencia - cuando se evalúa, la referencia se substituye con el valor actual del campo. Los nombres no se escapan.

Los siguientes operadores están disponibles:

- operadores aritméticos: "+", "-", "/", ^
- paréntesis: para hacer cumplir la precedencia del operador: (1 + 1) * 3
- unario mas y menos: -12, +5
- funciones matemáticas: sqrt, sin, cos, tan, asin, acos, atan
- funciones de conversión: to_int, to_real, to_string, to_date
- funciones geométricas: \$area, \$length
- funciones de manejo de geometría: \$x, \$y, \$geometry, num_geometries, centroid

Se apoya las siguientes predicaciones:

- comparación: =, !=, >, >=, <, <=
- patrones iguales: LIKE (using % and _), ~ (expresión regular)
- lógica predicado: AND, OR, NOT
- revisión de valores NULO: IS NULL, IS NOT NULL

Ejemplos de predicado:

- 1 + 2 = 3
- sin(angulo) > 0

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

Ejemplo de escala de expresiones:

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

10.1 Análisis de expresiones

```
>>> 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 Evaluar expresiones

10.2.1 Expresiones Basicas

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

10.2.2 Expresiones con características

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

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

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

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

10.2.3 Manejar errores

```
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 Ejemplos

El siguiente ejemplo se puede utilizar para filtra capas y regresar cualquier característica que empata con el 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"
```

Configuración de lectura y almacenamiento

Muchas veces es útil para un complemento guardar algunas variables para que el usuario no tenga que introducir o seleccionar de nuevo la próxima vez que el complemento se ejecute.

Estas variables se pueden guardar y recuperar con ayuda de Qt y QGIS API. Para cada variable, se debe escoger una clave que será utilizada para acceder a la variable — para el color favorito del usuario podría utilizarse la clave “favourite_color” o cualquier otra cadena que tenga sentido. Es recomendable dar un poco de estructura al nombrar las claves.

We can make difference between several types of settings:

- **global settings** — they are bound to the user at particular machine. QGIS itself stores a lot of global settings, for example, main window size or default snapping tolerance. This functionality is provided directly by Qt framework by the means of `QSettings` class. By default, this class stores settings in system’s “native” way of storing settings, that is — registry (on Windows), .plist file (on 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)
```

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

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

```
proj = QgsProject.instance()

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

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

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

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

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

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

Comunicarse con el usuario

- Showing messages. The `QgsMessageBar` class
- Mostrando el progreso
- Registro

Esta sección muestra algunos métodos y elementos que deberían utilizarse para comunicarse con el usuario, con el objetivo de mantener la consistencia en la Interfaz del Usuario.

12.1 Showing messages. The `QgsMessageBar` class

Utilizar las bandejas de mensajes puede ser una mala idea desde el punto de vista de la experiencia de un usuario. Para mostrar una pequeña línea de información o mensajes de advertencia/error, la barra de mensajes de QGIS suele ser una mejor opción.

Utilizar la referencia a la interfaz objeto de QGIS, puede mostrar un mensaje en la barra de mensajes con el siguiente código

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

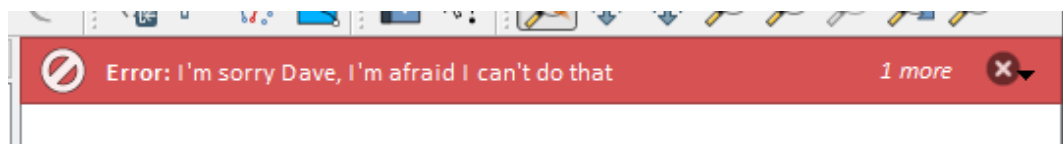


Figure 12.1: Barra de Mensajes de QGIS

Se puede establecer una duración para mostrarlo en un tiempo limitado

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

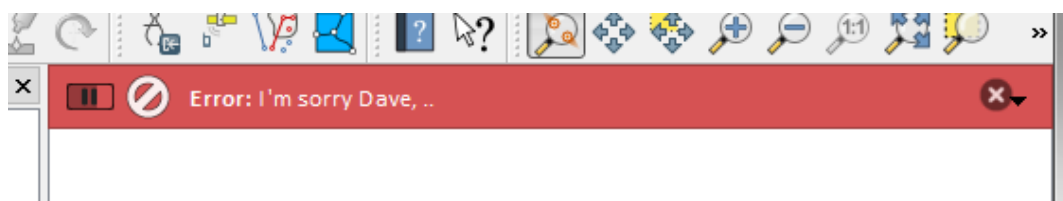


Figure 12.2: Barra de Mensajes de QGIS con temporizador

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

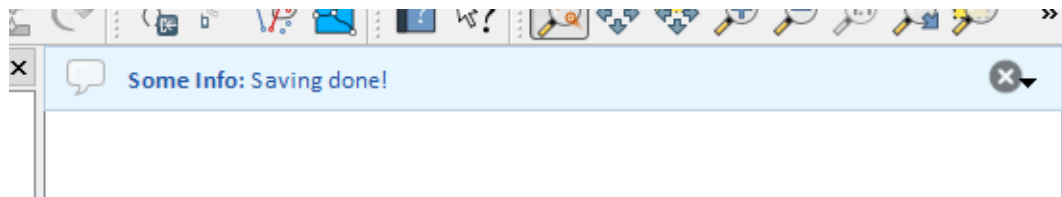


Figure 12.3: Barra de Mensajes de QGIS (información)

Se puede añadir complementos a la barra de mensajes, como por ejemplo un botón para mostrar más información

```
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)
```

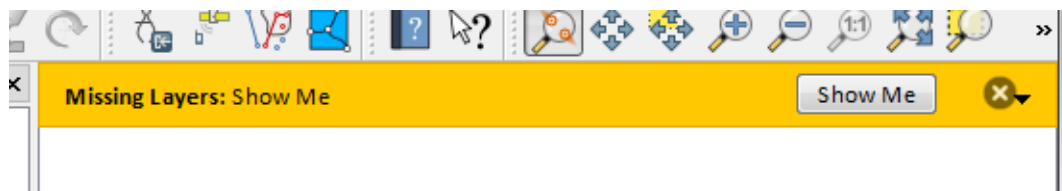


Figure 12.4: Barra de Mensajes de QGIS con un botón

Incluso puedes utilizar una barra de mensajes en tu propio cuadro de diálogo para no tener que mostrar la bandeja de mensajes o si no tiene sentido mostrarla en la pantalla principal de 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 el progreso

Las barras de progreso también pueden ponerse en la barra de Mensajes de QGIS, ya que, como hemos visto, admite complementos. Este es un ejemplo que puedes probar en la consola.

```
import time
from PyQt4.QtGui import QProgressBar
from PyQt4.QtCore import *
progressMessageBar = iface.messageBar().createMessage("Doing something boring...")
progress = QProgressBar()
progress.setMaximum(10)
```

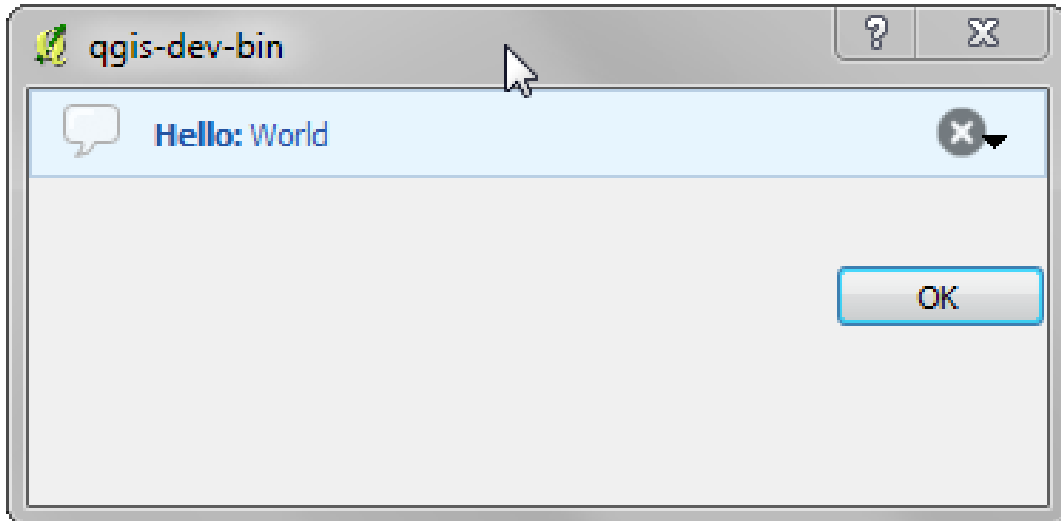


Figure 12.5: Barra de Mensajes de QGIS con un cuadro de diálogo personalizado

```

progress.setAlignment(Qt.AlignLeft|Qt.AlignVCenter)
progressMessageBar.layout().addWidget(progress)
iface.messageBar().pushWidget(progressMessageBar, iface.messageBar().INFO)
for i in range(10):
    time.sleep(1)
    progress.setValue(i + 1)
iface.messageBar().clearWidgets()

```

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

```

count = layers.featureCount()
for i, feature in enumerate(features):
    #do something time-consuming here
    ...
    percent = i / float(count) * 100
    iface.mainWindow().statusBar().showMessage("Processed {} %".format(int(percent)))
iface.mainWindow().statusBar().clearMessage()

```

12.3 Registro

Se puede utilizar el sistema de registro de QGIS para registrar toda la información de la ejecución de su código que se quiera guardar.

```

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

```

Desarrollo de Plugins Python

- Escribir un complemento
 - Archivos de complementos
- Contenido del complemento
 - Metadato del complemento
 - `__init__.py`
 - `mainPlugin.py`
 - Archivo de recurso
- Documentación
- Traducción
 - Requerimientos de software
 - Archivos y directorio
 - * archivo `.pro`
 - * archivo `.ts`
 - * archivo `.qm`
 - Translate using Makefile
 - Cargar el complemento

Es posible crear complementos en lenguaje de programación Python. En comparación con complementos clásicos escritos en C++, éstas deberían ser más fáciles de escribir, comprender, mantener y distribuir debido a la naturaleza dinámica del lenguaje Python.

Los complementos de Python están listados con complementos C++ en el administrador de complementos de QGIS. Se buscaron en estas rutas:

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

El directorio principal (denotado por el anterior `~`) en Windows es generalmente algo como `C:\Documents and Settings\user`. Desde que QGIS está utilizando Python 2.7, los subdirectorios de esta ruta deben contener un archivo `__init__.py` para ser considerados paquetes de Python que pueden ser importados como complementos.

Nota: Al establecer `QGIS_PLUGINPATH` a una ruta de directorio existente, se puede añadir la ruta a la lista de rutas donde se han buscado complementos.

Pasos:

1. *Idea:* ¿Tiene una idea acerca de lo que quiere hacer con su nuevo complemento de QGIS. ¿Por qué lo hace? ¿Qué problema desea resolver? ¿Existe ya otro complemento para ese problema?
2. *Crear archivos:* Crear los archivos descritos a continuación. Un punto inicial (`__init__.py`). Relene el *Metadato del complemento* (`metadata.txt`) Un cuerpo de complemento python principal (`mainplugin.py`). Una forma en QT-Designer (`form.ui`), con su `resources.qrc`.

3. *Escribir código*: Escribir el código dentro del `mainplugin.py`
4. *Prueba*: Cerrar y abrir QGIS e importar el su complemento de nuevo. Comprobar si todo está bien.
5. *Publicar*: Publica su complemento en el repositorio de QGIS o hacer su propio repositorio como un “arsenal” de “armas SIG” personales.

13.1 Escribir un complemento

Desde la introducción de los complementos de Python en QGIS, una serie de complementos han aparecido - en [Plugin Repositories wiki page](#) se pueden encontrar algunos de ellos, puede utilizar su fuente para aprender más acerca de la programación con PyQGIS o averiguar si no está duplicando el esfuerzo de desarrollo. El equipo de QGIS también mantiene un *Repositorio oficial de complemento*. ¿Listo para crear un complemento pero ni idea de qué hacer? [Python Plugin Ideas wiki page](#) listas de deseos de la comunidad!

13.1.1 Archivos de complementos

Aquí está la estructura de directorios de nuestro ejemplo 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*
```

Cuál es el significado de los archivos:

- `__init__.py` = El punto de partida del complemento. Se tiene que tener el método `classFactory()` y puede tener cualquier otro código de inicialización.
- `mainPlugin.py` = El código principal de trabajo del complemento. Contiene toda la información acerca de las acciones del complemento y el código principal.
- `resources.qrc` = El documento .xml creado por Qt Designer. Contiene rutas relativas a los recursos de las formas.
- `resources.py` = La traducción del archivo .qrc descrito anteriormente para Python.
- `form.ui` = La GUI creada por Qt Designer.
- `form.py` = La traducción de la form.ui descrito anteriormente para Python.
- `metadata.txt` = Required for QGIS >= 1.8.0. contiene información general, versión, nombre y algunos otros metadatos utilizado por complementos del sitio web o el complemento de infraestructura. Dado que QGIS 2.0 los metadatos de `__init__.py` no son aceptados más y el `metadata.txt` es necesario.

[Aquí](#) es una manera automatizada en línea de crear los archivos básicos (esqueleto) de un complemento típico de QGIS Python.

También hay un complemento QGIS llamado [Plugin Builder](#) que crea la plantilla del complemento desde QGIS y no requiere conexión a Internet. Esta es la opción recomendada, ya que produce 2.0 fuentes compatibles.

Advertencia: Si su plan para actualizar el complemento del *Repositorio oficial de complemento* se debe validar que su complemento siga algunas reglas adicionales, necesarias para complementos *Validación*

13.2 Contenido del complemento

Aquí se puede encontrar información y ejemplos sobre lo que se debe añadir en cada uno de los archivos de la estructura de archivos descrito anteriormente.

13.2.1 Metadato del complemento

En primer lugar, Administrador de complementos necesita recuperar cierta información básica sobre el complemento tales como su nombre, descripción, etc. El archivo `metadata.txt` es el lugar adecuado para colocar esta información.

Importante: Todos los metadatos deben estar en codificación UTF-8.

Nombre del metadato	Necesario	Notas
nombre	Verdadero	una cadena corta contiene el nombre del complemento
qgisMinimumVersion	Verdadero	notación de la versión mínima de QGIS
qgisMaximumVersion	Falso	notación de la versión máxima de QGIS
descripción	Verdadero	texto corto que describe el complemento, no se permite HTML
acerca de	Verdadero	texto más largo que describe el complemento en detalles, no se permite HTML
versión	Verdadero	cadena corta con la notación versión punteado
autor	Verdadero	nombre del autor
correo electrónico	Verdadero	correo electrónico del autor, no se muestra en el administrador de complementos de QGIS o en el sitio web a menos registradas por un usuario conectado, por lo que sólo son visibles para otros autores de plugins y administradores de sitios web plug-in
registro de cambios	Falso	cadena, puede ser multilínea, no se permite HTML
experimental	Falso	bandera booleana, <i>True</i> o <i>False</i>
obsoleto	Falso	bandera booleana, <i>True</i> or <i>False</i> , se aplica a todo complemento y no solo a la versión actualizada
etiquetas	Falso	lista separada por comas, se permiten espacios dentro de las etiquetas individuales
página principal	Falso	una URL válida que apunte a la página principal de su complemento
repositorio	Verdadero	una URL válida para el repositorio del código fuente
rastreador	Falso	una URL válida para las entradas e informes de errores
icono	Falso	un nombre de archivo o una ruta relativa (relativa a la carpeta base del paquete de complemento comprimido) de una imagen de web amigable (PNG, JPEG)
categoría	Falso	uno de <i>Ráster</i> , <i>Vector</i> , <i>Base de Datos</i> y <i>Web</i>

Por defecto, los complementos se colocan en el menú *Complementos* (veremos en la siguiente sección sobre cómo añadir una entrada de menú para su complemento), pero también pueden ser colocados en los menús *Raster*, *Vector*, *Database* and *Web*.

Una entrada de metadato correspondiente “categoría” para especificar que existe, por lo que el complemento se puede clasificar en consecuencia. Esta entrada de metadatos se utiliza como consejo para los usuarios y les dice dónde (en qué menú) el complemento se puede encontrar. Los valores permitidos para “categoría” son: vector,

ráster, base de datos o web. Por ejemplo, si su complemento estará disponible desde el menú *Ráster*, añadir esto a `metadata.txt`

```
category=Raster
```

Nota: Si `qgisMaximumVersion` está vacía, se ajustará automáticamente a la versión principal `.99` cuando se actualizan a el *Repositorio oficial de complemento*.

Un ejemplo 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`

El archivo es necesario por el sistema de importación de Python. También, QGIS requiere que este archivo contenga una función `classFactory()`, que se llama cuando el complemento se carga a QGIS. Recibe referencia a instancia de `QgisInterface` y debe volver la instancia de la clase de su complemento desde el

mainplugin.py — en nuestro caso se llama TestPlugin (ver más abajo). Esta es como `__init__.py` debe ser.

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

```
## any other initialisation needed
```

13.2.3 mainPlugin.py

Aquí es donde sucede la magia y así es como la magia se ve: (por ejemplo 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.mainWindow)
        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.renderTest)
```

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

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

Las únicas funciones de complemento que deben existir en el archivo principal fuente (por ejemplo mainPlugin.py) son:

- `__init__` -> que da acceso a la interfaz de QGIS

- `initGui()` -> se llama cuando se carga el complemento
- `unload()` -> se llama cuando se descarga el complemento

Se puede ver que en el ejemplo anterior, el `addPluginToMenu()` se utiliza. Esto añadirá la acción del menú correspondiente al menú *Complementos*. Métodos alternativos existen para añadir la acción a diferentes menús. Aquí esta la lista de esos métodos:

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

Todos ellos tienen la misma sintaxis como el método `addPluginToMenu()`

Añadir el menú de su complemento a uno de aquellos métodos predefinidos se recomienda mantener la coherencia en la forma en que se organizan las entradas de complementos. Sin embargo, puede agregar a su grupo de menú personalizado directamente a la barra de menú, como el siguiente ejemplo demuestra:

```
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.mainWindow())
    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()
```

No olvide establecer `QAction` y `QMenu` `objectName` a un nombre específico a su complemento para que pueda ser personalizado.

13.2.4 Archivo de recurso

Se puede ver que en `initGui()` hemos utilizado un icono desde el archivo fuente (llamado `resources.qrc` en nuestro caso)

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

Es bueno utilizar un prefijo que no colisionará con otros o cualquier parte de QGIS, de lo contrario podría obtener los recursos que no quería. Ahora sólo tiene que generar un archivo de Python que contendrá los recursos. Está hecho con el comando **pyrcc4**

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

Nota: En entornos de Windows, intente ejecutar el **pyrcc4** desde símbolo de sistema o Powershell probablemente resultará en el error "Windows no puede tener acceso al dispositivo especificado, la ruta, o el archivo [...]". La solución más sencilla es probablemente usar el shell OSGeo4W pero si se siente cómodo modificar la variable de entorno `PATH` o especificando la ruta del ejecutable de forma explícita debería ser capaz de encontrarlo en <Su directorio de instalación QGIS>\bin\pyrcc4.exe.

y eso es todo... nada complicado :)

Si ha hecho todo correctamente debe ser capaz de encontrar y cargar sus complementos en el administrador y ver un mensaje en consola cuando el icono en la barra de herramientas o el elemento del menú apropiado es seleccionado.

Cuando se trabaja en un complemento real es aconsejable escribirlo en otro directorio (de trabajo) y crear un makefile que generará los archivos de interfaz de usuario + recursos e instalar el complemento a la instalación de QGIS.

13.3 Documentación

La documentación para el complemento puede estar escrita como archivos de ayuda HTML. El módulo `qgis.utils` proporciona una función, `showPluginHelp()` que abrirá el explorador de archivos de ayuda, de la misma manera como otra ayuda QGIS.

La función `showPluginHelp()` busca los archivos de ayuda en el mismo directorio que el módulo de llamadas. Se buscará, a la vez, `index-ll_cc.html`, `index-ll.html`, `index-en.html`, `index-en_us.html` y `index.html`, mostrando lo que se encuentra en primer lugar. Aquí `ll_cc` esta la configuración regional de QGIS. Esto permite múltiples traducciones de la documentación que se incluyen con el complemento.

La función `showPluginHelp()` también puede tener parámetros de nombre de paquete, que identifica un complemento específico para el que se mostrará la ayuda, nombre de archivo, que puede sustituir “índice” en los nombres de los archivos que se buscan, y la sección, que es el nombre de una etiqueta de anclaje html en el documento sobre el que se colocará el navegador.

13.4 Traducción

Con unos pocos pasos se puede instalar el entorno para el complemento de localización que depende de la configuración regional de su computadora, el complemento se cargará en diferentes idiomas.

13.4.1 Requerimientos de software

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 Archivos y directorio

Cuando se crea el complemento encontrará la carpeta `i18n` dentro del directorio de complementos principal.

Todos los archivos de traducción tienen que estar dentro de este directorio.

archivo `.pro`

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
```

Por otra parte, el archivo `your_plugin.py` es el que *llama* a todos el menú, y sub-menús de su complemento en la barra de herramientas de QGIS y desea traducir a todos.

Finalmente con la variable `TRANSLATIONS` se puede especificar los idiomas de traducción que desee.

Advertencia: Asegúrese de nombrar el archivo `ts` como `your_plugin_ + language + .ts` de otra manera el idioma cargado fallará! Utilice 2 letras para el idioma (**it** para italiano, **de** para alemán, etc...)

archivo .ts

Una vez que ha creado el `.pro` ya tiene que generar los archivo(s) de los idioma(s) de su complemento.

Abra una terminal, vaya al directorio `your_plugin/i18n` y escriba:

```
pylupdate4 your_plugin.pro
```

Debería ver los archivo(s) `your_plugin_language.ts`

Abra el archivo `.ts` con **Qt Linguist** e inicie a traducir.

archivo .qm

Cuando termine de traducir su complemento (si algunas cadenas no son completadas se utilizarán las cadenas de estos en el idioma origen), tiene que crear el archivo `.qm` (El archivo compilado `.ts` será utilizado por QGIS).

Sólo abra una terminal `cd` en su carpeta `your_plugin/i18n` y escriba:

```
lrelease your_plugin.ts
```

ahora, en el directorio `i18n`, verá los archivo(s) `your_plugin.qm`.

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 Cargar el complemento

con el fin de ver la traducción de su complemento sólo abra QGIS, cambie el idioma (*Configuración* → *Opciones* → *Idioma*) y reinicie QGIS.

Debería ver su complemento con el idioma correcto.

<p>Advertencia: Si cambia algo en su complemento (nuevos UIs, nuevo menú, etc...) tiene que generar de nuevo la versión actualizada de ambos archivos <code>.ts</code> y <code>.qm</code>, así que ejecute de nuevo el comando de arriba.</p>

Authentication infrastructure

- Introducción
- Glosario
- QgsAuthManager the entry point
 - Init the manager and set the master password
 - Populate authdb with a new Authentication Configuration entry
 - * Available Authentication methods
 - * Populate Authorities
 - * Manage PKI bundles with QgsPkiBundle
 - Remove entry from authdb
 - Leave authcfg expansion to QgsAuthManager
 - * PKI examples with other data providers
- Adapt plugins to use Authentication infrastructure
- Authentication GUIs
 - GUI to select credentials
 - Authentication Editor GUI
 - Authorities Editor GUI

14.1 Introducción

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.

Advertencia: 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 Glosario

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

Contraseña maestra 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

Autenticación BD *Authentication Database*

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

Configuración de autenticación *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 = 'fmls770'
```

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 hypothetical 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 Methods 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 Autenticación

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()
```

Advertencia: 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
bundle = QgsPkiBundle.fromPemPaths( "/path/to/alice-cert.pem",
                                   "/path/to/alice-key_w-pass.pem",
                                   "unlock_pwd",
                                   "list_of_CAs_to_bundle" )

assert bundle is not None
assert bundle.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 its `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 Configs*, 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.

Advertencia: 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:

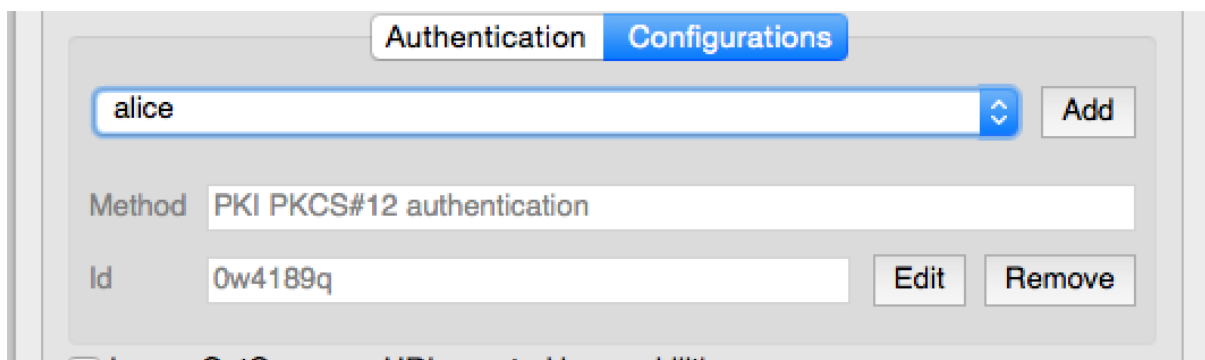
```
http = NetworkAccessManager(authid="my_authCfg", exception_class=My_FailedRequestError)
try:
    response, content = http.request( "my_rest_url" )
except My_FailedRequestError, e:
    # Handle exception
    pass
```

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:

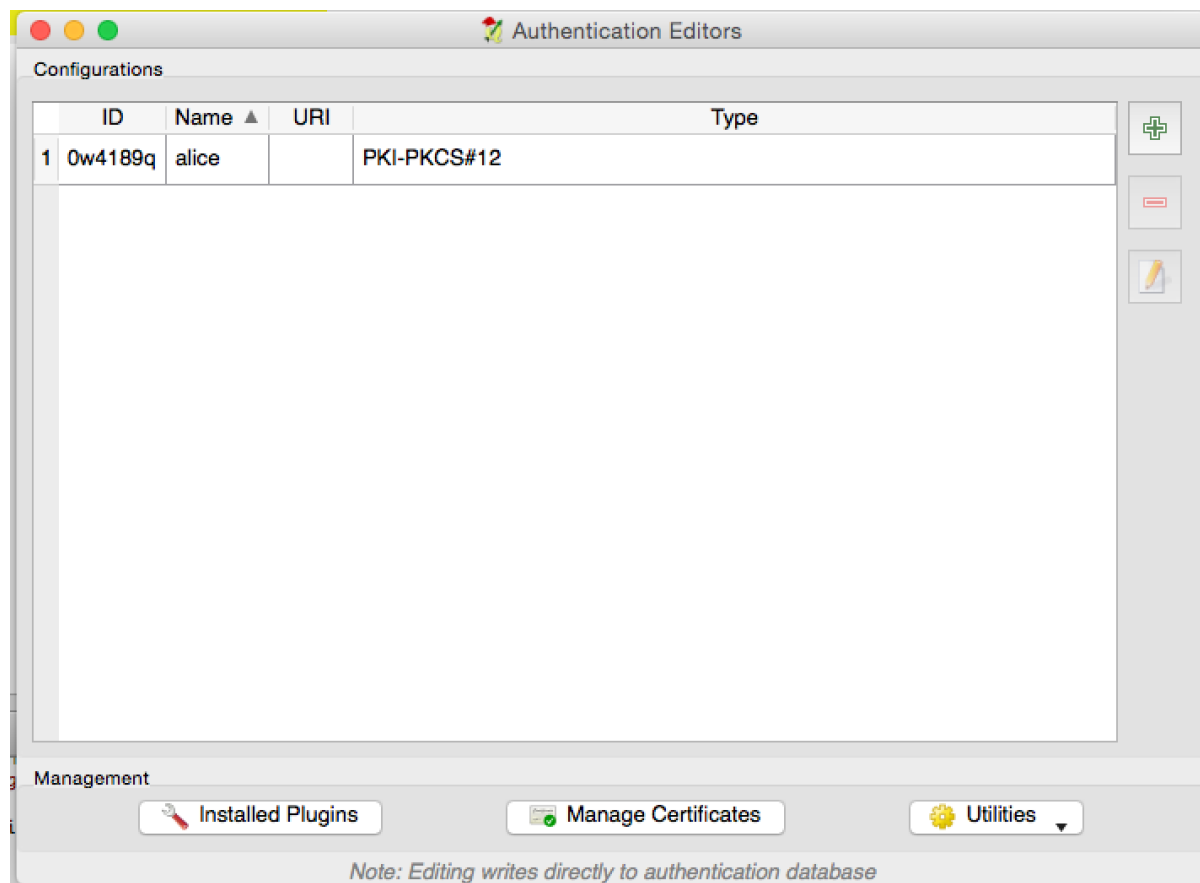
```
# create the instance of the QgsAuthConfigSelect GUI hierarchically linked to
# the widget referred with 'parent'
gui = QgsAuthConfigSelect( parent, "postgres" )
# add the above created gui in a new tab of the interface where the
```

```
# 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 Methods* 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()
```

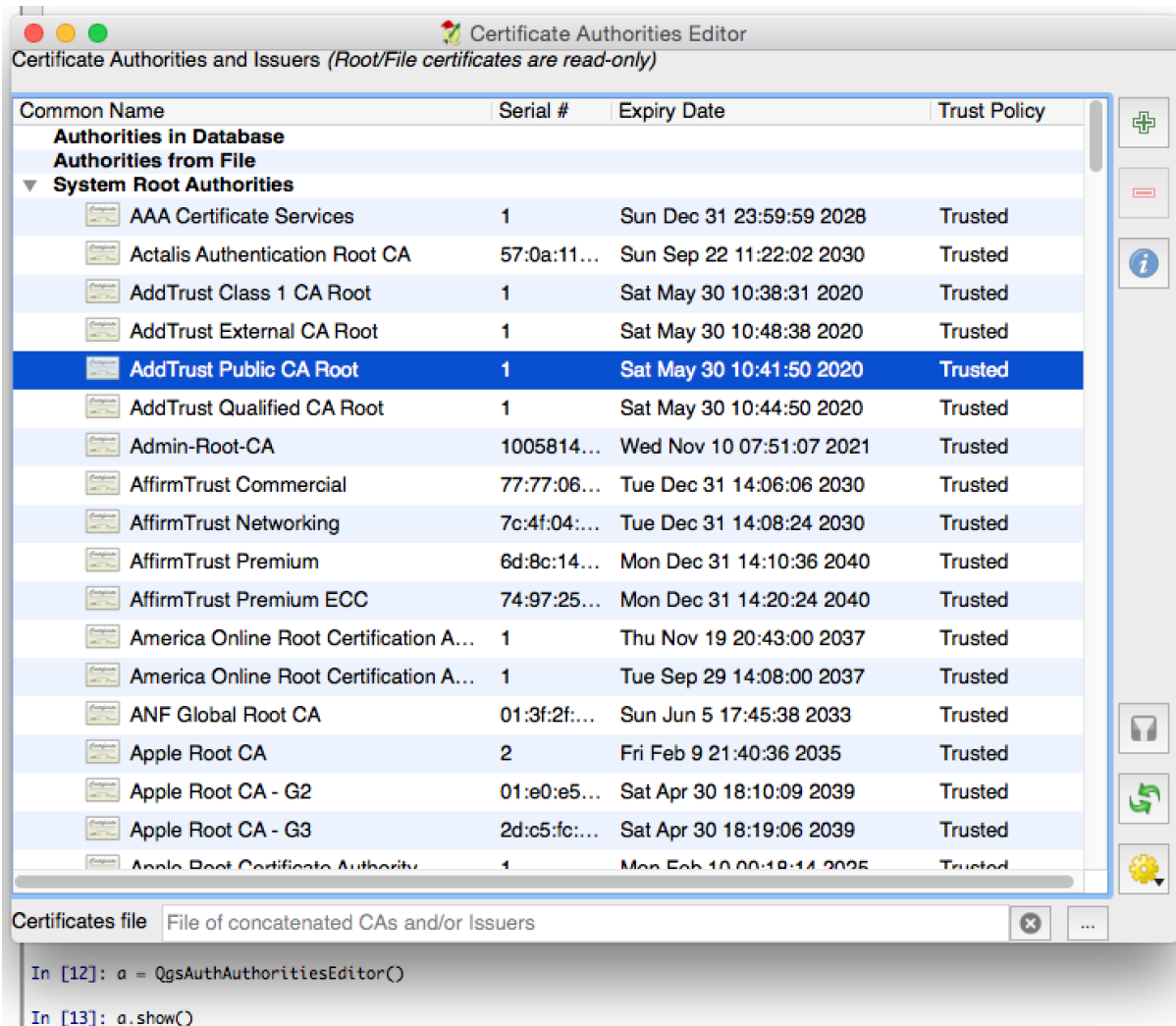
an integrated example can be found in the related test

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'
```



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

Configuración IDE para escribir y depurar complementos

- Una nota sobre la configuración su IDE sobre Windows
- Depure utilizando eclipse y PyDev
 - Instalación
 - Preparación QGIS
 - Configuración de eclipse
 - Configurando el depurador
 - Hacer que eclipse entienda el API
- Depure utilizando PDB

Aunque cada programador tiene si editos IDE/texto preferido, aquí están algunas recomendaciones para configurar los IDE's populares para escribir y depurar complementos Python QGIS.

15.1 Una nota sobre la configuración su IDE sobre Windows

Sobre Linux no hay configuración adicional necesaria para desarrollar complementos. Pero sobre Windows se necesita asegurar que tiene la misma configuración de entorno y utilizan la mismas librerías y como interprete QGIS. La manera más rápida de hacer esto, es modificar el archivo de inicio por lotes de QGIS.

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

Para usar `IDE Pyscripter`, Aquí esta lo que tiene que hacer:

- Make a copy of `qgis-unstable.bat` and rename it `pyscripter.bat`.
- Abra esto en un editor. Y elimine la ultima linea, una que inicia con QGIS.
- Añadir una linea que apunte al ejecutable Pyscripter y añada el argumento en la linea de comando que establece la versión de Python a ser utilizado (2.7 en el caso de QGIS 2.0)
- También añadimos el argumento que apunta en la carpeta donde el Pyscripter puede encontrarse el dll de Python utilizado por QGIS, se puede encontrar esto bajo la carpeta bin de su instalación 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
```

Ahora cuando haga clic en el archivo por lotes, iniciará el Pyscripter, con la ruta correcta.

Más popular que Pyscripter, Eclipse es una opción común entre los desarrolladores. En las siguientes secciones, vamos a explicar cómo configurarlo para desarrollar y probar los complementos. Para preparar el entorno para el uso de Eclipse en Windows, también debe crear un archivo por lotes y utilizarlo para iniciar Eclipse.

To create that batch file, follow these steps:

- Busque la carpeta donde el archivo `:qgis_core.dll` reside. Normalmente esto es `C:\OSGeo4W\apps\qgis\bin`, pero cuando se ha compilado su aplicación QGIS esto es su carpeta de compilación en `output/bin/RelWithDebInfo`
- Busque su ejecutable file:`eclipse.exe`.
- Cree el siguiente script y utilice esto para iniciar eclipse cuando desarrolle complementos en 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 Depure utilizando eclipse y PyDev

15.2.1 Instalación

Para utilizar Eclipse, asegurese que ha instalado lo siguiente

- Eclipse
- Complemento Aptana Eclipse o PyDev
- QGIS 2.x

15.2.2 Preparación QGIS

Hay un poco de preparación que hacer sobre sí mismo en QGIS. Dos complementos son de interés: *Depuración remota* y *Recargador de complementos*.

- Go to *Plugins* → *Manage and Install plugins...*
- Search for *Remote Debug* (at the moment it's still experimental, so enable experimental plugins under the *Options* tab in case it does not show up). Install it.
- Busque *Recargador de complementos* e instálelo también. Esto dejara que recargue un complemento en lugar de tener que cerrar y reiniciar QGIS para tener recargado el complemento.

15.2.3 Configuración de eclipse

En Eclipse, crear un nuevo proyecto. Se puede seleccionar *Proyecto General* y enlazar su fuente real después por lo que realmente no importa donde se coloque el proyecto.

Ahora haga clic derecho sobre su nuevo proyecto y elija *Nuevo* → *Carpeta*.

Haga clic en [**Avanzado**] y elija *Enlace a la ubicación alterna (Carpeta enlazada)*. En caso que ya tenga los recursos que desea depurar, elija estos, en caso que no lo haga, cree una carpeta como ya se explicó.

Ahora en la vista *Explorador de proyecto*, su arbol de recursos aparece y puede empezar a trabajar con el código. Ya se tiene la sintaxis resaltada y todas las otras herramientas del IDE de gran alcance disponible.

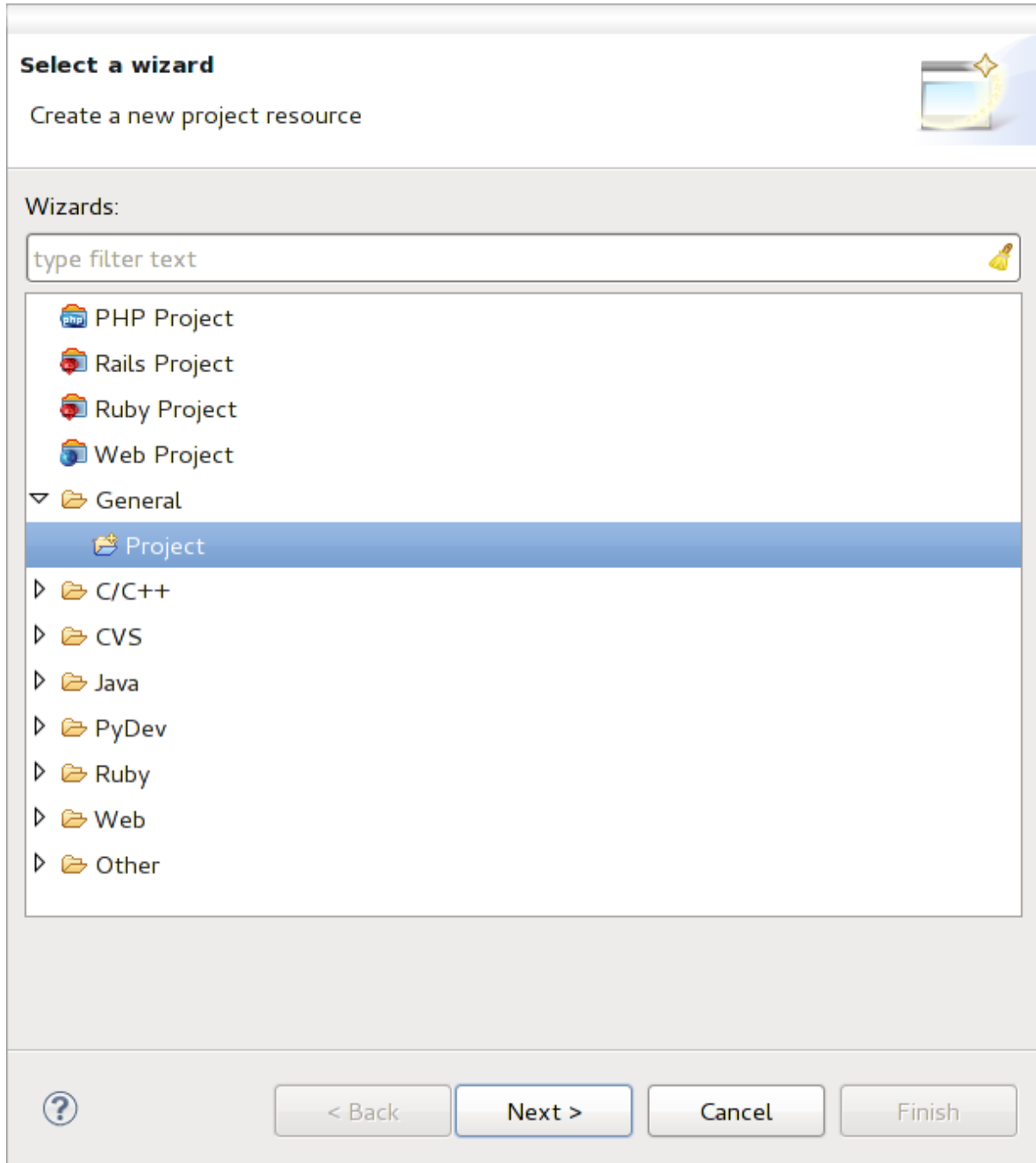


Figure 15.1: Proyecto de Eclipse

15.2.4 Configurando el depurador

Para conseguir el funcionamiento, cambie a la perspectiva de Depurar en Eclipse (*Window* → *Abrir perspectiva* → *Otro* → *Depurar*).

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

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

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

```

87         self.verticalExaggerationChanged.emit(val)
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 )
103

```

Figure 15.2: Punto de interrupción

Una cosa muy interesante que puede hacer uso de ahora es la consola de depuración. Asegurese que la ejecución esta detenida actualmente en un punto de interrupción, antes de continuar.

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

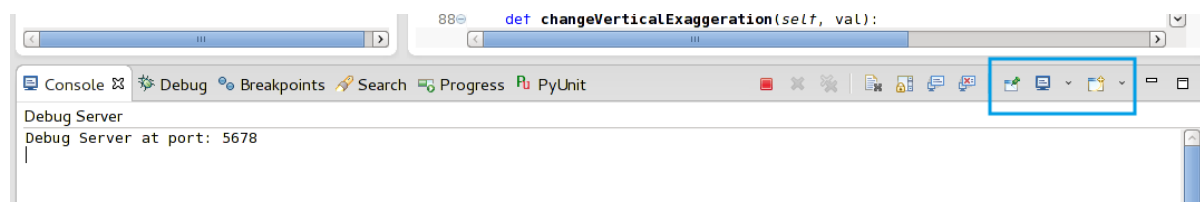


Figure 15.3: consola de depuración PyDev

Se tiene ahora una consola interactiva que le permite probar cualquier comando desde dentro del contexto actual. Se pueden manipular variables o hacer llamadas al API o lo que quiera.

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

15.2.5 Hacer que eclipse entienda el API

Una característica muy útil es tener Eclipse que realmente conozca acerca de la API de QGIS. Esto le permite comprobar el código de errores tipográficos. Pero no sólo esto, sino que también permite que Eclipse le ayude con la terminación automática de las importaciones a llamadas a la API.

Para ello, Eclipse analiza los archivos de la biblioteca de QGIS y recibe toda la información que hay. Lo único que tiene que hacer es decirle a Eclipse, donde se encuentran las librerías.

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

Verá su interpretador de python configurado en la parte superior de la ventana (en este momento python2.7 para QGIS) y algunas pestañas en la parte inferior. Las pestañas interesantes para nosotros son *Librerías* y *Elementos incluidos forzados*

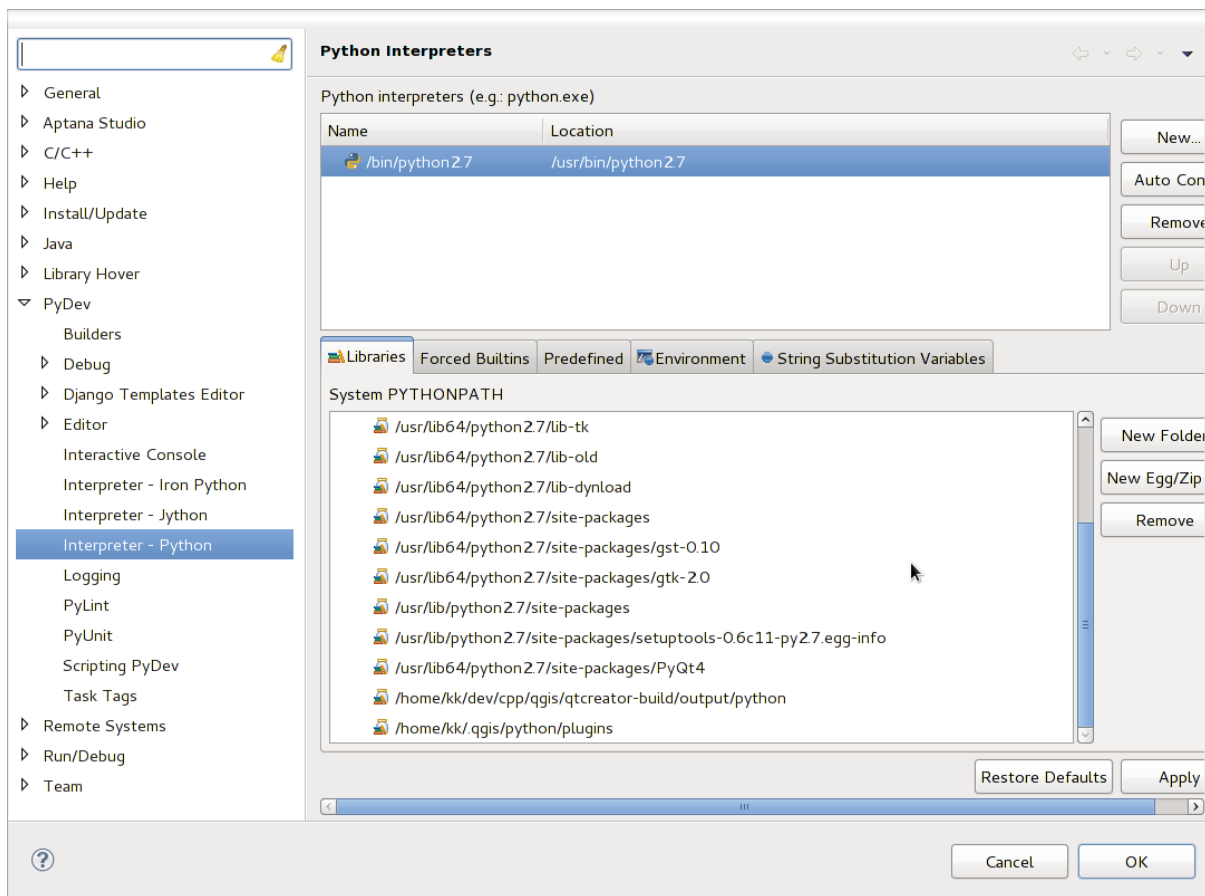


Figure 15.4: consola de depuración PyDev

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

You should also add your plugins folder here (on Linux 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.

Haga clic en *Aceptar* y ya está.

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 Depure utilizando PDB

Si no usa un IDE como Eclipse, puede depurar usando PDB, siguiendo los siguientes pasos.

Primero adicione este código en el lugar donde desea depurar

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

Entonces ejecute QGIS desde la línea de comando.

Sobre Linux haga:

```
$ ./Qgis
```

Sobre Mac OS haga:

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

Y cuando la aplicación pegue su punto de interrupción puede escribir en la consola!

PENDIENTE: Adicionar información de prueba

Utilizar complemento Capas

Si su complemento utiliza métodos propios para representar una capa de mapa, escribir su propio tipo de capa basado en `QgsPluginLayer` puede ser la mejor forma de implementarla.

TODO: Comprobar la corrección y elaborar buenos casos de uso de `QgsPluginLayer`

16.1 Subclassing `QgsPluginLayer`

A continuación es un ejemplo de una implementación mínima de `QgsPluginLayer`. Es un extracto del [Complemento de ejemplo de marca de agua](#)

```
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
```

Métodos de lectura y escritura de información específica para el archivo del proyecto también puede ser añadido

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

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

Al cargar un proyecto que contiene una capa de este tipo, se necesita una clase de fábrica

```
class WatermarkPluginLayerType(QgsPluginLayerType):
    def __init__(self):
        QgsPluginLayerType.__init__(self, WatermarkPluginLayer.LAYER_TYPE)

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

Se puede también añadir código para mostrar información personalizada en las propiedades de la capa

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

Compatibilidad con versiones antiguas de QGIS

17.1 Menu de plugins

Si se coloca las entradas del menú de complementos en uno de los nuevos menús (*Ráster, Vector, Base de Datos o Web*), se debe modificar el código de las funciones `initGui()` y `unload()`. Dado que estos nuevos menús sólo están disponibles en QGIS 2.0 y versiones superiores, el primer paso es comprobar que la versión de QGIS que se está ejecutando tenga todas las funciones. Si los nuevos menús están disponibles, colocaremos nuestros complementos bajo estos menús, de lo contrario utilizaremos el anterior menú *Complementos*. Aquí un ejemplo para el menú *Ráster*

```
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)
```

Compartiendo sus plugins

- Metadatos y nombres
- Code and help
- Repositorio oficial de complemento
 - Permisos
 - Trust management
 - Validación
 - Estructura de complemento

Una vez que su complemento esté listo y usted piense que puede ser útil para los demás, no dude en subirlo a *Repositorio oficial de complemento*. En esa página encontrará guías de empaquetado del complemento para que éste funcione correctamente con el instalador de complementos. O en el caso de que usted quiera configurar su propio repositorio de complementos, cree un archivo XML para listar los complementos y sus metadatos. Para ver ejemplos mire otros *repositorios de complementos*.

Please take special care to the following suggestions:

18.1 Metadatos y nombres

- avoid using a name too similar to existing plugins
- if your plugin has a similar functionality to an existing plugin, please explain the differences in the About field, so the user will know which one to use without the need to install and test it
- avoid repeating “plugin” in the name of the plugin itself
- use the description field in metadata for a 1 line description, the About field for more detailed instructions
- include a code repository, a bug tracker, and a home page; this will greatly enhance the possibility of collaboration, and can be done very easily with one of the available web infrastructures (GitHub, GitLab, Bitbucket, etc.)
- choose tags with care: avoid the uninformative ones (e.g. vector) and prefer the ones already used by others (see the plugin website)
- add a proper icon, do not leave the default one; see QGIS interface for a suggestion of the style to be used

18.2 Code and help

- do not include generated file (ui_*.py, resources_rc.py, generated help files...) and useless stuff (e.g. .gitignore) in repository
- add the plugin to the appropriate menu (Vector, Raster, Web, Database)

- when appropriate (plugins performing analyses), consider adding the plugin as a subplugin of Processing framework: this will allow users to run it in batch, to integrate it in more complex workflows, and will free you from the burden of designing an interface
- include at least minimal documentation and, if useful for testing and understanding, sample data.

18.3 Repositorio oficial de complemento

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

In order to use the official repository you must obtain an OSGEO ID from the [OSGEO web portal](#).

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

PENDIENTE: Insert a link to the governance document

18.3.1 Permisos

These rules have been implemented in the official plugin repository:

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

18.3.2 Trust management

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

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

18.3.3 Validación

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

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

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

18.3.4 Estructura de complemento

Following the validation rules the compressed (.zip) package of your plugin must have a specific structure to validate as a functional plugin. As the plugin will be unzipped inside the users plugins folder it must have it's own directory inside the .zip file to not interfere with other plugins. Mandatory files are: `metadata.txt` and `__init__.py`. But it would be nice to have a `README` and of course an icon to represent the plugin (`resources.qrc`). Following is an example of how a `plugin.zip` should look like.

```
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
```

Fragmentos de código

- Cómo llamar a un método por un atajo de teclado
- Como alternar capas
- Cómo acceder a la tabla de atributos de los objetos espaciales seleccionados

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

En el complemento añadir a la `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 añadir `unload()`

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

El método que se llama cuando se presiona F7

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

19.2 Como alternar capas

Desde QGIS 2.4 hay un nuevo API de árbol de capas que permite acceder directamente al árbol de capas en la leyenda. Aquí un ejemplo de cómo alternar la visibilidad de la capa activa.

```
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")
```

El método requiere un parámetro (el nuevo valor para el campo de atributo de los objeto(s) espaciales seleccionados) y puede ser llamado por

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

Escribir nuevos complementos de procesamiento

- Creating a plugin that adds an algorithm provider
- Creating a plugin that contains a set of processing scripts

En función del tipo de complemento que vayas a desarrollar, puede ser más recomendable añadir la funcionalidad en cuestión como un algoritmo de Procesamiento (o un conjunto de algoritmos). Esta opción permite una mejor integración en QGIS, añadiendo la funcionalidad (ya que así podrá ser ejecutada dentro de cualquiera los componentes de menú Procesos, como el modelador gráfico o la interfaz de procesos por lotes), y optimizando el tiempo de desarrollo (ya que el complemento de Procesos hará gran parte del trabajo).

This document describes how to create a new plugin that adds its functionality as Processing algorithms.

There are two main mechanisms for doing that:

- Creating a plugin that adds an algorithm provider: This options is more complex, but provides more flexibility
- Creating a plugin that contains a set of processing scripts: The simplest solution, you just need a set of Processing script files.

20.1 Creating a plugin that adds an algorithm provider

To create an algorithm provider, follow these steps:

- Instala el complemento Plugin Builder.
- Crea un nuevo complemento, usando el Plugin Builder. En el cuadro de diálogo del Plugin Builder, selecciona “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 Creating a plugin that contains a set of processing scripts

To create a set of processing scripts, follow these steps:

- Create your scripts as described in the PyQGIS cookbook. All the scripts that you want to add, you should have them available in the Processing toolbox.
- 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.
- Click on OK and the plugin will be created.

- You can add additional scripts to the plugin by adding scripts python files to the *scripts* folder in the resulting plugin folder.

Biblioteca de análisis de redes

- Información general
- Contruir un gráfico
- Análisis gráfico
 - Encontrar la ruta más corta
 - Áreas de disponibilidad

A partir de la versión [ee19294562](#) (QGIS \geq 1.8) la nueva librería de análisis de redes se agregó a la librería de análisis de núcleo de QGIS. La librería:

- Crear gráfico matemático de datos geográficos (capas vectoriales de polilínea)
- implementa métodos básicos de la teoría de grafos (actualmente sólo el algoritmo Dijkstra)

La librería de análisis de redes fue creada por funciones básicas de exportación del complemento núcleo Road-Graph y ahora se puede utilizar los metodos en complementos o directamente de la consola Python.

21.1 Información general

Brevemente, un caso de uso típico se puede describir como:

1. Crear gráfica de geodatos (normalmente de capa vectorial de polilíneas)
2. ejecutar análisis gráfico
3. utilizar resultados de análisis (por ejemplo, visualizarlos)

21.2 Contruir un gráfico

Lo primero que hay que hacer — es preparar la entrada de datos, que es convertir una capa vectorial en un gráfico. Todas las acciones adicionales utilizarán esta gráfica, no la capa.

Como fuente podemos utilizar una capa vectorial de polilínea. Los nodos de las polilíneas se convierten en vértices del gráfico, y los segmentos de la polilínea son bordes de gráfico. Si varios nodos tienen la misma coordenada entonces ellos tienen el mismo vértice gráfico. Por lo que dos líneas que tienen un nodo en común se conectarán entre sí.

Además durante la creación del gráfico se puede “arreglar” (“atar”) a la capa vectorial de entrada cualquier número de puntos adicionales. Para cada punto adicional se encontrará una coincidencia — el vértice gráfico más cercano o el borde gráfico más cercano. En el último caso el borde será dividido y un nuevo vértice se añadirá.

Los atributos de la capa vectorial y la longitud de un borde se puede utilizar como las propiedades de un borde.

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.

Es tiempo de sumergirse en el proceso.

Antes que nada, para utilizar esta librería debemos importar el modulo de análisis de redes

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

Después algunos ejemplos para crear un 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)
```

Para construir un director debemos pasar a una capa vectorial, que se utilizará como fuente para la estructura gráfica y la información sobre el movimiento permitido en cada segmento de carretera (movimiento unidireccional o bidireccional, dirección directa o inversa). La llamada se parece a esto

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

Y aquí esta la lista completa de lo que significan estos parámetros:

- `vl` — la capa vectorial utilizada para construir la gráfica
- `directionFieldId` — índice de la tabla de atributos de campo, donde se almacena información acerca de dirección de carreteras. Si `-1`, entonces no utilice esta información en absoluto. Un entero.
- `directDirectionValue` — el valor del campo de carreteras con dirección directa (mover desde la primer punto de línea a la última). Un texto.
- `reverseDirectionValue` — valor del campo de carreteras con dirección inversa (mover del último punto de línea al primero). Un texto.
- `bothDirectionValue` — valor de campo para carreteras bidireccionales (para cada carretera podemos mover del primer punto al último y del último al primero). Un texto.
- `defaultDirection` — dirección de carretera predeterminada. Este valor se utilizará para esos caminos donde el campo `directionFieldId` no esta establecido o tiene algun valore diferente de cualquiera de los tres valores especificados anteriormente. Un entero. `1` indica la dirección directa, `2` la dirección inversa, y `3` ambas direcciones.

Es necesario entonces crear una estrategia para calcular propiedades de borde

```
properter = QgsDistanceArcProperter()
```

Y decirle al director sobre esta estrategia

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

Ahora podemos utilizar el constructor, que creará el grafo. El constructor de la clase `QgsGraphBuilder` tomar varios argumentos:

- `src` — sistema de referencia de coordenadas a utilizar. Argumento obligatorio.
- `otfEnable` — utilizar la reproyección ‘al vuelo’ o no. Por defecto `const:True` (utilizar OTF).
- `topologyTolerance` — tolerancia topologica. Por defecto el valor es 0.
- `ellipsoidID` — ellipsoid a utilizar. Por defecto “WGS84”.

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

También podemos definir varios puntos, que se utilizarán en el análisis. Por ejemplo

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

Ahora todo está en su lugar para que podamos construir el gráfico y “atar” a estos puntos

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

Construir el grafo puede tomar tiempo (que depende del número de elementos y tamaño de una capa). `tiedPoints` es una lista con coordenadas de puntos “tied”. Cuando la operación de construcción se finalizo podemos obtener la gráfica y utilizarlo para el análisis

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

Con el siguiente código podemos obtener el índice del vértice de nuestros puntos

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

21.3 Análisis gráfico

El análisis de redes es utilizado para encontrar respuestas a dos preguntas: que vértices estan conectados y cómo encontrar la ruta más corta. Para resolver estos problemas la librería de análisis de redes proporciona el algoritmo Dijkstra.

El algoritmo Dijkstra encuentra la ruta más corta de uno de los vértices del grafo a todos los otros y los valores de los parámetros de optimización, El resultado puede ser representado como un árbol de la ruta más corta.

El árbol del camino más corto es un grafo ponderado dirigido (o más precisamente – árbol) con las siguientes propiedades:

- sólo un vértice no tiene bordes entrantes — la raíz del árbol
- todos los otros vértices sólo tienen un borde entrante
- Si el vértice B es accesible desde el vértice A, entonces el camino de A a B es la única ruta disponible y es optima (más corta) en este grafo

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.

El método `shortestTree()` es útil cuando se desea caminar al rededor del árbol del camino más corto. Siempre crea un nuevo objeto grafo (`QgsGraph`) y acepta tres variables:

- `fuelle` — gráfico de entrada
- `startVertexIdx` — índice del punto en el árbol (la raíz del árbol)
- `criterionNum` — número de propiedad de borde a utilizar (iniciar de 0).

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

El método `dijkstra()` tiene los mismos argumentos, pero regresa dos arrays. En el primer elemento del array `i` contiene el índice del borde entrante o `-1` si no hay bordes entrantes. En el segundo elemento del array `i` contiene la distancia de la raíz del árbol al vértice `i` o `DOUBLE_MAX` si el vértice `i` es inalcanzable de la raíz.

```
(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
```

La misma cosa pero utilizando el método `dijkstra()`

```
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 Encontrar la ruta más corta

Para encontrar la ruta óptima entre dos puntos, el siguiente enfoque se utiliza. Ambos puntos (inicio A y fin B) son “tied” al grafo cuando es construido. Entonces utiliza los métodos `shortestTree()` o `dijkstra()`, nosotros construiremos el árbol de la ruta más corta con raíz en el punto inicial A. En el mismo árbol también encontramos el punto final B y comenzamos a avanzar a través del árbol del punto B al punto A. Todo el algoritmo se puede escribir como

```

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

```

En este punto tenemos la ruta, en el formulario de la lista invertida de vértices (los vértices están listados en orden invertida del punto final al punto inicial) que serán visitados durante el viaje por este camino.

Aquí esta el código de ejemplo para la consola Python de QGIS (deberá seleccionar la capa linestring en TOC y en el código reemplazar las coordenadas por las suyas) que usa el método `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)
```

Y aquí esta el mismo ejemplo pero sin utilizar el método `dijkstra()`

```
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 Áreas de disponibilidad

El área de la disponibilidad para el vértice A es el subconjunto de vértices del grafo que son accesibles desde el vértice A y el costo de los caminos de la A a estos vértices son no es mayor que cierto valor.

Más claramente esto se puede demostrar con el siguiente ejemplo: “Hay una estación de bomberos ¿Qué partes de la ciudad puede un camión de bomberos alcanzar en 5 minutos? 10 minutos? 15 minutos?”. Las respuestas a estas preguntas son las zonas de la estación de bomberos de la disponibilidad.

Para encontrar las zonas de disponibilidad podemos utilizar el método `dijkstra()` de la clase `QgsGraphAnalyzer`. Es suficiente para comparar los elementos del array de costos con un valor predefinido. Si el costo [i] es menor que o igual a un valor predefinido, entonces el vértice i esta dentro de la zona de disponibilidad, de lo contrario esta fuera.

Un problema más difícil es conseguir los límites de la zona de disponibilidad. El borde inferior es el conjunto de vértices que son todavía accesibles, y el borde superior es el conjunto de vértices que no son accesibles. De hecho esto es simple: es la frontera disponibilidad basado en los bordes del árbol de ruta más corta para los que el vértice origen del contorno es más accesible y el vértice destino del borde no lo es.

Aquí tiene un ejemplo

```

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))
```

Complementos de Python de QGIS Server

- Server Filter Plugins architecture
 - requestReady
 - sendResponse
 - responseComplete
- Raising exception from a plugin
- Writing a server plugin
 - Archivos de complementos
 - `__init__.py`
 - `HelloServer.py`
 - Modificando la entrada
 - Modificar o reemplazar la salida
- Complemento control de acceso
 - Archivos de complementos
 - `__init__.py`
 - `AccessControl.py`
 - `layerFilterExpression`
 - `layerFilterSubsetString`
 - `layerPermissions`
 - `authorizedLayerAttributes`
 - `allowToEdit`
 - `cacheKey`

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**
 - * **Si SERVICE es WMS/WFS/WCS**
 - **create WMS/WFS/WCS server**
 - 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
- redirije
- 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 *Desarrollo de Plugins 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 Archivos de complementos

Aquí está la estructura de directorio de nuestro complemento servidor de ejemplo

```
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()`).

Los siguientes ejemplos cubren algunos casos comunes de uso:

22.3.4 Modificando la entrada

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', QgsMessageLog.INFO)
    else:
        QgsMessageLog.logMessage("FAIL - ParamsFilter.responseComplete", 'plugin', QgsMessageLog.INFO)

```

Esto es un extracto de lo que puede ver en el archivo de log:

```

src/core/qgsmessageolog.cpp: 45: (logMessage) [0ms] 2014-12-12T12:39:29 plugin[0] HelloServerService
src/core/qgsmessageolog.cpp: 45: (logMessage) [1ms] 2014-12-12T12:39:29 Server[0] Server plugin HelloServerService
src/core/qgsmessageolog.cpp: 45: (logMessage) [0ms] 2014-12-12T12:39:29 Server[0] Server python plugin HelloServerService
src/mapserver/qgsgetrequesthandler.cpp: 35: (parseInput) [0ms] query string is: SERVICE=HELLO&request=HELLO
src/mapserver/qgshttprequesthandler.cpp: 547: (requestStringToParameterMap) [1ms] inserting pair 'SERVICE'='HELLO'
src/mapserver/qgshttprequesthandler.cpp: 547: (requestStringToParameterMap) [0ms] inserting pair 'request'='HELLO'
src/mapserver/qgsserverfilter.cpp: 42: (requestReady) [0ms] QgsServerFilter plugin default requestReady
src/core/qgsmessageolog.cpp: 45: (logMessage) [0ms] 2014-12-12T12:39:29 plugin[0] HelloFilter.requestReady
src/mapserver/qgis_map_serv.cpp: 235: (configPath) [0ms] Using default configuration file path: /usr/share/qgis/
src/mapserver/qgshttprequesthandler.cpp: 49: (setHttpResponse) [0ms] Checking byte array is ok to use
src/mapserver/qgshttprequesthandler.cpp: 59: (setHttpResponse) [0ms] Byte array looks good, setting response
src/core/qgsmessageolog.cpp: 45: (logMessage) [0ms] 2014-12-12T12:39:29 plugin[0] HelloFilter.responseComplete
src/core/qgsmessageolog.cpp: 45: (logMessage) [0ms] 2014-12-12T12:39:29 plugin[0] SUCCESS - ParamsFilter.responseComplete
src/core/qgsmessageolog.cpp: 45: (logMessage) [0ms] 2014-12-12T12:39:29 plugin[0] RemoteConsoleFilter
src/mapserver/qgshttprequesthandler.cpp: 158: (sendResponse) [0ms] Sending HTTP response
src/core/qgsmessageolog.cpp: 45: (logMessage) [0ms] 2014-12-12T12:39:29 plugin[0] HelloFilter.sendResponse

```

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 Modificar o reemplazar la salida

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 Complemento control de acceso

22.4.1 Archivos de complementos

Aquí está la estructura de directorio de nuestro complemento servidor de ejemplo:

```

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()
```

Este ejemplo otorga acceso total para todos.

Es rol del complemento saber quién ha ingresado.

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

22.4.4 layerFilterExpression

Usado para agregar una Expresión para limitar los resultados, ej.:

```
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

Limitar el acceso a la capa.

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.

Ejemplo:

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

Para limitar todo en acceso de solo lectura.

22.4.7 `authorizedLayerAttributes`

Usado para limitar la visibilidad de un subconjunto específico de atributo.

El atributo del argumento devuelve el conjunto actual de atributos visibles.

Ejemplo:

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

Para ocultar el atributo 'rol'.

22.4.8 `allowToEdit`

Esto es usado para limitar la edición de un subconjunto de objetos espaciales.

It is used in the WFS-Transaction protocol.

Ejemplo:

```
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.

-
-
- API, 1
 - Autenticación BD, **72**
 - Authentication Configuration, **72**
 - Authentication Database, **72**
 - Authentication Method, **72**
 - Calculating values, 50
 - Categorized symbology renderer, 27
 - Configuración de autenticación, **72**
 - Console
 - Python, 2
 - Contraseña maestra, **71**
 - Custom
 - Renderer, 31
 - Custom applications
 - Python, 3
 - Running, 4
 - Delimited text files
 - Loading, 10
 - Environment
 - PYQGIS_STARTUP, 1
 - Expressions, 50
 - Evaluating, 52
 - Parsing, 52
 - Filtering, 50
 - Geometry
 - Access to, 36
 - Construction, 35
 - Handling, 33
 - Predicates and operations, 36
 - GPX files
 - Loading, 10
 - Graduated symbol renderer, 28
 - Iterating features, 18
 - Loading
 - Delimited text files, 10
 - GPX files, 10
 - MySQL geometries, 10
 - OGR layers, 9
 - PostGIS layers, 9
 - Projects, 7
 - Raster layers, 11
 - Spatialite layers, 10
 - Vector layers, 9
 - WFS vector, 10
 - WMS raster, 11
 - Map canvas, 40
 - Custom canvas items, 45
 - Custom map tools, 44
 - Embedding, 41
 - Map tools, 42
 - Rubber bands, 43
 - Vertex markers, 43
 - map canvas
 - architecture, 41
 - Map layer registry, 11
 - Adding a layer, 11
 - Map printing, 46
 - Map rendering, 46
 - Simple, 47
 - Memory layer, 24
 - Metadata, 107
 - metadata, 107
 - metadata.txt, 63, 107
 - MySQL geometries
 - Loading, 10
 - OGR layers
 - Loading, 9
 - Output
 - PDF, 50
 - Raster image, 50
 - Using Map Composer, 48
 - Plugin layers, 84
 - Subclassing QgsPluginLayer, 85
 - Plugins
 - Access attributes of selected features, 93
 - Adding shortcut, 93
 - Code snippets, 67
 - Debugging, 78
 - Developing, 59, 69
 - Documentación, 67
 - Implementing help, 67
-

- Initialisation, 64
- Metadata, 63
- metadata.txt, 107
- Processing algorithm, 94
- Releasing, 87
- Repositorio oficial de complemento, 90
- Resource file, 66
- Toggle layers, 93
- Traducción, 67
- User interaction, 56
- Writing, 62
- Writing code, 62
- plugins
 - testing, 84
- PostGIS layers
 - Loading, 9
- Projections, 40
- Projects
 - Loading, 7
- PyQGIS
 - Vector layers, 17
- PYQGIS_STARTUP
 - Environment, 1
- Python
 - Authentication infrastructure, 69
 - Console, 2
 - Custom applications, 3
 - Developing plugins, 59
 - Developing server plugins, 104
 - Plugins, 2
 - Standalone scripts, 3
 - startup, 1
 - startup.py, 2
- Querying
 - Raster layers, 15
- Raster
 - Raster layers, 12
- Raster layers
 - Details, 13
 - Loading, 11
 - Multi band, 14
 - Querying, 15
 - Raster, 12
 - Refreshing, 15
 - Renderizador, 13
 - Single band, 14
- Refreshing
 - Raster layers, 15
- Renderer
 - Custom, 31
- resources.qrc, 66
- Running
 - Custom applications, 4
- Selecting features, 17
- Server plugins
 - Developing, 104
 - server plugins
 - metadata.txt, 107
- Settings
 - Global, 55
 - Map layer, 56
 - Project, 55
 - Reading, 53
 - Storing, 53
- Single symbol renderer, 26
- Sistemas de coordenadas de referencia, 39
- Spatial index, 23
- SpatiaLite layers
 - Loading, 10
- Standalone scripts
 - Python, 3
- startup
 - Python, 1
- Symbol layers
 - Creating custom types, 29
 - Working with, 29
- Symbology
 - Categorized symbol renderer, 27
 - Graduated symbol renderer, 28
 - Single symbol renderer, 26
- Symbols
 - Working with, 28
- Vector layers
 - Creating, 23
 - Editing, 20
 - Loading, 9
 - Symbology, 25
- WFS vector
 - Loading, 10
- WMS raster
 - Loading, 11