"

PyQGIS developer cookbook

Publicación 2.18

QGIS Project

21 de February de 2019

Contents

CHAPTER 1

Introducción

Este documento pretende funcionar como un tutorial y como una guía referencia. Aunque no muestra todos los posibles casos de uso, debería dar una buena perspectiva de la funcionalidad principal.

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
- Comandos de objeto en la consola Python con QGIS
- Crear y usar extensiones en Python
- Crear aplicaciones personalizadas basadas en la API de QGIS

Python bindings are also available for QGIS Server:

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

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 Ejecutar código Python cuando QGIS inicie

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 undesireable paths, or for isolating/loading the initial environ without requiring a virt env, e.g. homebrew or MacPorts installs on Mac.

1.1.2 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 Consola Python

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:

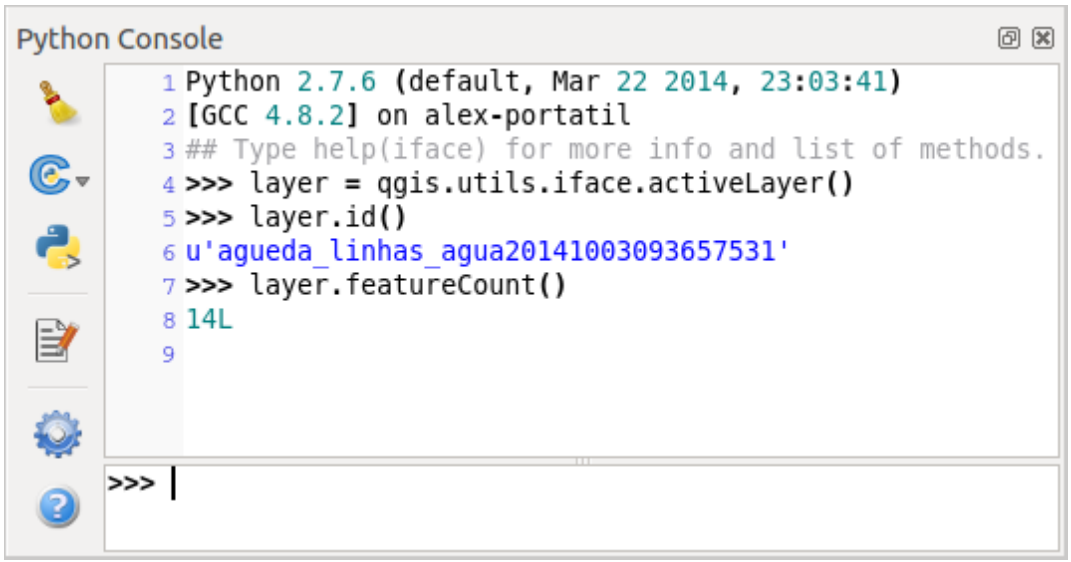


Figure 1.1: Consola Python de QGIS

La captura de pantalla superior muestra cómo añadir la capa seleccionada

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

A menudo, al procesar datos SIG, es práctico crear scripts para automatizar un proceso en lugar de realizar la misma tarea repetidas veces. Esto es perfectamente posible con PyGIS — importe el módulo `qgis.core`, inicialícelo y estará listo para el procesamiento.

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 *Consola Python* from within QGIS and look at the output from running `QgsApplication.prefixPath()`.

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

1.4.2 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.

CHAPTER 2

Cargar proyectos

Algunas veces se necesita cargar un proyecto existente desde un complemento o (más a menudo) al desarrollar una aplicación autónoma QGIS Python (vea : *Aplicaciones Python*).

Para cargar un proyecto en la aplicación QGIS actual necesita un objeto `QgsProject instance()` y llamar a su método `read()` pasando a un objeto `QFileInfo` que contenga la ruta desde donde el proyecto será cargado:

```
# 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'
```

En caso de que necesite hacer algunas modificaciones al proyecto (por ejemplo añadir o eliminar algunas capas) y guardad los cambios, se puede llamar al método `write()` de la instancia del proyecto. El método `write()` también acepta una opcional `QFileInfo` que le permite especificar una ruta donde el proyecto será almacenado:

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

Ambas funciones `read()` y `write()` regresan un valor booleano que se puede utilizar para validar si la operación fue un éxito.

Nota: Si esta escribiendo una aplicación autónoma de QGIS, con el fin de sincronizar el proyecto cargado con el lienzo que necesita para crear una instancia de una `QgsLayerTreeMapCanvasBridge` como el ejemplo siguiente:

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

CHAPTER 3

Cargar capas

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

Para cargar una capa vectorial, especificar el identificador de la fuente de datos de la capa, nombre y nombre del proveedor:

```
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.

La manera más rápida para abrir y desplegar una capa vectorial en QGIS es la función addVectorLayer del `QgisInterface`:

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

Esto crea una nueva capa y lo añade al registro de capa de mapa (haciendolo aparecer en la lista de capas) en un paso. La función regresa la instancia de la capa o *Nada* si la capa no puede cargarse.

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

- La librería OGR (archivos shape y muchos otros formatos de archivo) — la fuente de datos es la ruta del archivo.
  - para archivos shape:

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

  - para dxf (Observe las opciones internas en la fuente de datos uri):

    ```
    uri = "/path/to/dxffile/file.dxf|layername=entities|geometrytype=Point"
    vlayer = QgsVectorLayer(uri, "layer_name_you_like", "ogr")
    ```

- La base de datos PostGIS — la fuente de datos es una cadena con toda la información necesaria para crear una conexión a la base de datos PostgreSQL. La clase `QgsDataSourceURI` puede generar esta cadena para usted. Tenga en cuenta que QGIS tiene que ser compilado con la ayuda de Postgres, de lo contrario este proveedor no esta disponible:

```
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

Para acceder a los archivos ráster, se utiliza la librería GDAL. Es compatible con una amplia gama de formatos de archivo. En caso que tenga problemas al abrir algunos archivos, compruebe si su GDAL tiene implementado un formato en particular (no todos los formatos están disponibles por defecto). Para cargar un ráster desde un archivo, especificar su nombre de archivo y nombre base:

```
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!"
```

De manera similar que las capas vectoriales, ráster se pueden cargar utilizando la función addRasterLayer de la `QgisInterface`:

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

Esto crea una capa y lo agrega al registro de capa de mapa (haciendolo aparecer en la lista de capas) en un paso.

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')
```

Ajustes del URI detallado se pueden encontrar en documentación de proveedor

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 Registro de capa de mapa

Si desea utilizar las capas abiertas para la representación, no olvide de añadirlos al registro de capa de mapa. El registro capa de mapa asume la propiedad de las capas y se puede acceder más tarde desde cualquier parte de la aplicación por su ID único. Cuando se retira la capa de registro capa de mapa, que se elimina, también.

Añadir una capa al registro:

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

Las capas se destruyen de forma automática en la salida, por eso si se desea eliminar la capa de forma explícita, utilice:

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

Para una lista de capas cargadas e identificadores de capas, utilice:

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

CHAPTER 4

Usar las capas ráster

Esta sección enumera varias operaciones que se pueden hacer con capas ráster.

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 0x000000000F8A2048>
rlayer.extent().toString()
u'12.095833,48.552777 : 18.863888,51.056944'
rlayer.rasterType()
2  # 0 = GrayOrUndefined (single band), 1 = Palette (single band), 2 = Multiband
rlayer.bandCount()
3
rlayer.metadata()
u'<p class="glossy">Driver:</p>...'
rlayer.hasPyramids()
False
```

4.2 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.

Para consultar el renderizador actual:

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

Para establecer un renderizado utilice el método `setRenderer()` de `QgsRasterLayer`. Hay varios clases de renderizadores disponibles (derivado de `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.

Las siguientes secciones explican cómo consultar y modificar el estilo de dibujo de la capa. Después de hacer los cambios, es posible que desee forzar la actualización del lienzo del mapa, ver ref:*refresh-layer*.

TODO: mejoras de contraste, la transparencia (sin datos), máximos /mínimos definidos por el usuario, estadísticas de la banda

4.2.1 Rásters de una sola banda

Vamos a decir que queremos representar nuestra capa ráster (asuma sólo una banda) con rango de colores de verde a amarillo (para valores de píxel de 0 a 255). En la primer escena prepararemos el objeto `QgsRasterShader` y configuraremos su función de sombreado:

```
>>> 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`): el color resultante se interpola linealmente desde las entradas del mapa de color por encima y por debajo del valor real del píxel
- discrete (`DISCRETE`): el color se utiliza desde la entrada de mapa de color con valor igual o superior
- exact (`EXACT`): el color no es interpolado, solamente los píxeles con valor igual al color del mapa de entrada es dibujado

En el segundo paso asociaremos este sombreado con la capa ráster:

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

El número 1 en el código anterios es el número de la banda (bandas ráster estan indexadas de uno).

4.2.2 Rásters multibanda

Por defecto, en los mapas de QGIS las primeras tres bandas a valores de rojo, verde y azul para crear una imagen en color (este es el estilo de dibujo `MultiBandColor`. En algunos casos es posible que desee anular estos ajustes. El siguiente código intercambia la banda roja (1) y la banda verde (2):

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

En caso de que sólo una banda sea necesaria para visualizar del ráster, sólo una banda dibujada puede elegir — cualquiera de los niveles gris o pseudocolor.

4.3 Actualizar capas

Si se hace el cambio de la simbología de capa y le gustaría asegurarse de que los cambios son inmediatamente visibles para el usuario, llame a estos métodos

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

La primera llamada se asegurará de que la imagen en caché de la capa presentada se borra en caso de que el almacenamiento en caché este activado. Esta funcionalidad está disponible desde QGIS 1.4, en versiones anteriores no existe esta función — para asegurarse de que el código funciona con en todas las versiones de QGIS, primero comprobamos si existe el método.

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

La segunda llamada emite señal de que obligará a cualquier lienzo de mapa que contenga la capa de emitir una actualización.

Con capas ráster WMS, estos comandos no funcionan. En este caso, hay que hacerlo de forma explícita

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

En caso de que haya cambiado la simbología de capa (ver secciones acerca de capas ráster y vectoriales sobre cómo hacerlo), es posible que desee forzar QGIS para actualizar la simbología de capa en la lista de capas (leyenda) de widgets. Esto se puede hacer de la siguiente manera (`iface` es una instancia de `QgisInterface`)

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

4.4 Valores de consulta

Para hacer una consulta sobre el valor de las bandas de capa ráster en algún momento determinado

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

El método `results` en este caso regresa un diccionario, con índices de bandas como llaves, y los valores de la banda como valores.

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

CHAPTER 5

Usar capas vectoriales

Esta sección sumariza 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() == QGis.Point:
        x = geom.asPoint()
        print "Point: " + str(x)
    elif geom.type() == QGis.Line:
        x = geom.asPolyline()
        print "Line: %d points" % len(x)
    elif geom.type() == QGis.Polygon:
        x = geom.asPolygon()
        numPts = 0
        for ring in x:
            numPts += len(ring)
        print "Polygon: %d rings with %d points" % (len(x), numPts)
    else:
        print "Unknown"

    # fetch attributes
    attrs = feature.attributes()

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

5.3.1 Accediendo atributos

Attributes can be referred to by their name.

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

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

```
print feature[0]
```

5.3.2 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

Para borrar algunos elementos, solo provea una lista de los IDs de los elementos

```
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 commited 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

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

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

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

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

- 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 WKBTYPE enum
5. layer's spatial reference (instance of
   QgsCoordinateReferenceSystem) - optional
6. driver name for the output file """
writer = QgsVectorFileWriter("my_shapes.shp", "CP1250", fields, QGis.WKBPoint, None, "ESRI Sh

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

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

# delete the writer to flush features to disk
del writer
```

5.8 proveedor de memoria

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

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

5.9 Appearance (Symbology) of Vector Layers

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

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

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

And with that reference, let us explore it a bit

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

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

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 simbol instance you can follow the example code:

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

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

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

5.9.2 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` — para elementos puntuales
- `QgsLineSymbolV2` — para elementos lineales
- `QgsFillSymbolV2` — para elementos poligonales

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
```

Salida

```
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.defaultSymbo

  def symbolForFeature(self, feature):
```

```
    return random.choice(self.syms)

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

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

  def usedAttributes(self):
    return []

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

5.10 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

CHAPTER 6

Manejo de Geometría

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, lineas 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

Existen varias opciones para crear una geometría:

- 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)]])
```

  Las coordenadas son dadas usando la clase: *QgsPoint*.

  La Polilínea (Linestring) se representa mediante una lista de puntos. Un Polígono se representa por una lista de anillos lineales (Ej. polilíneas cerradas). El primer anillo es el anillo externo (límite), los subsecuentes anillos opcionales son huecos en el polígono.

  Las geometrías multi-parte van un nivel más allá: multi-punto es una lista de puntos, multi-linea 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 = '01010000000000000000004540000000000000001440'.decode('hex')
>>> g.fromWkb(wkb)
>>> g.exportToWkt()
'Point (42 5)'
```

6.2 Acceso a Geometría

En primer lugar, debe saber el tipo de geometría, el método `wkbType()` es una para usar — devuelve un valor de enumeración `QGis.WkbType`

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

Como alternativa, se puede utilizar el método `type()` que devuelve un valor de enumeración `QGis.GeometryType`. También hay una función auxiliar `isMultipart()` para averiguar si una geometría es multiparte o no.

Para extraer información de una geometría existen funciones de acceso para cada tipo vectorial. Cómo usar accesos

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

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

Para geometrias multiples existen funciones similares para accesar : `asMultiPoint()`, `asMultiPolyline()`, `asMultiPolygon()`.

6.3 Geometría predicados y Operaciones

QGIS utiliza la biblioteca GEOS para operaciones avanzadas de geometría como predicados de geometría (`contains()`, `intersects()`, ...) y operaciones de conjuntos (`union()`, `difference()`,. ..). También se puede calcular propiedades geométricas de geometrías, tales como el área (en el caso de polígonos) o longitudes (por polígonos y líneas)

Aquí tienes un pequeño ejemplo que combina iterar sobre las características de una determinada capa y realizar algunos cálculos geométricos en función de sus geometrías.

```
# 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.

Información adicional puede ser encontrada en las siguientes fuentes:

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

CHAPTER 7

Soporte de Proyecciones

7.1 Sistemas de coordenadas de referencia

Los sistemas de referencia de coordenadas (SRC) están encapsuladas por la clase `QgsCoordinateReferenceSystem`. Las instancias de esta clase pueden ser creados por varias formas diferentes:

- especificar SRC por su ID

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

  QGIS utliza 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",0.0174532925199433295],'
       AXIS["Longitude",EAST], AXIS["Latitude",NORTH]]'
crs = QgsCoordinateReferenceSystem(wkt)
```

- crear un SRC inválido y después utilizar una de las funciones `create*()` para inicializarlo. En el siguiente ejemplo usamos una cadena Proj4 para inicializar la proyección

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

Es aconsejable comprobar si la creación (es decir, operaciones de búsqueda en la base de datos) del SRC ha tenido éxito: `isValid()` debe regresar `True`.

Tenga en cuenta que para la inicialización de los sistemas de referencia QGIS tiene que buscar valores apropiados en su base de datos interna `srs.db`. Así, en caso de de crear una aplicación independiente, necesita definir las rutas correctamente con `QgsApplication.setPrefixPath()` de lo contrario, no podrá encontrar la base de datos. Si se están ejecutando los comandos de consola python QGIS o desarrolla un complemento que no le importe: todo está ya preparado para usted.

Accediendo a información del sistema de referencia espacial

```
print "QGIS CRS ID:", crs.srsid()
print "PostGIS SRID:", crs.srid()
print "EPSG ID:", crs.epsg()
print "Description:", crs.description()
print "Projection Acronym:", crs.projectionAcronym()
print "Ellipsoid Acronym:", crs.ellipsoidAcronym()
print "Proj4 String:", crs.toProj4()
# check whether it's geographic or projected coordinate system
print "Is geographic:", crs.geographicFlag()
# check type of map units in this CRS (values defined in QGis::units enum)
print "Map units:", crs.mapUnits()
```

7.2 Proyecciones

Se puede hacer la transformación entre diferentes sistemas de referencia espacial al utilizar la clase `QgsCoordinateTransform`. La forma más fácil de usar que es crear origen y el destino de SRC y construir la instancia :class: *QgsCoordinateTransform* con ellos. Luego llame simplemente repetidamente la función `transform()` para hacer la transformación. Por defecto se hace la transformación hacia adelante, pero es capaz de hacer también la transformación inversa

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

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

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

CHAPTER 8

Usando el Lienzo de Mapa

El widget del lienzo del mapa es probablemente el widget más importante dentro de QGIS porque muestra el mapa integrado de capas de mapas superpuestos y permite la interacción con el mapa y las capas. El lienzo muestra siempre una parte del mapa definido por el alcance del lienzo actual. La interacción se realiza mediante el uso de **herramientas de mapa**: hay herramientas para desplazamiento, zum, la identificación de las capas, de medida, para editar vectores y otros. Al igual que en otros programas de gráficos, siempre hay una herramienta activa y el usuario puede cambiar entre las herramientas disponibles.

El lienzo de mapa esta implementado como clase `QgsMapCanvas` en el modulo `qgis.gui`. La implementación se basa en el marco Qt Graphics View. Este marco generalmente proporciona una superficie y una vista donde elementos gráficos personalizados se colocan y el usuario puede interacturar con ellos. Vamos a suponer que está lo suficientemente familiarizado con Qt para comprender los conceptos de la escena gráfica, visión y objetos. Si no es así, asegúrese de leer la vista general del marco.

Cada vez que el mapa ha sido desplazado, zum para acercar /alejar (o algún otra acción que desencadene una actualización), el mapa se representa de nuevo dentro de la extensión actual. Las capas se presentan a una imagen (usando la clase `QgsMapRenderer`) y esa imagen después se muestra en el lienzo. El elemento de gráficos (en términos del marco de vista de gráficos Qt) responsable de mostrar el mapa es la clase `QgsMapCanvasMap`. Esta clase también controla la actualización del mapa. Además este elemento que actúa como un fondo, puede haber más **elementos de lienzo de mapas**. Los elementos de lienzo de mapas típico son bandas de goma (utilizadas para medir, edición vectorial, etc.) o marcadores de vértices. Los elementos de lienzo generalmente se utilizan para dar una retroalimentación visual para herramientas del mapa, por ejemplo, cuando se crea un nuevo polígono, la herramienta del mapa crea una banda borrador en el elemento de mapa que muestra la figura actual del polígono. Todos los elementos de lienzo de mapa están en subclases de `QgsMapCanvasItem` que añade más funciónalidad a los objetos básicos‘‘QGraphicsItem‘‘.

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

- lienzo de mapa — para la visualización del mapa
- Los elementos de lienzo de mapa — los elementos adicionales que se pueden desplegar en un lienzo de mapa
- herramientas de mapa — para interactuar con el lienzo del mapa

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()
```

Esto produce una ventana independiente con el lienzo de mapa. Puede también ser incrustado en un widget existente o ventana. Al utilizar archivo ui y Qt Designer, coloque un `QWidget` sobre el formulario y promuévalo a una nueva clase: establezca `QgsMapCanvas` como nombre de clase y `qgis.gui` como archivo de encabezado. La utilidad `pyuic4` se hará cargo de ella. Esta es una forma conveniente de incrustar el lienzo. La otra posibilidad es escribir manualmente el código para construir el lienzo del mapa y otros widgets (como hijos de una ventana principal o diálogo) y crea un diseño.

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)
```

(En caso de que se esté preguntando, `Qt` viene del modulo `PyQt4.QtCore` y `Qt.white` es uno de lo que predefine las instancias `QColor`.)

Ahora es tiempo de añadir algunas capas. Primero, abriremos una capa y lo añadiremos al registro capa de mapa. A continuación, vamos a establecer la extensión del lienzo y la lista de capas para el lienzo

```
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

El siguiente ejemplo construye una ventana que contiene un lienzo de mapa y herramientas de mapa básicos para desplazar y hacer zum. Las acciones se crean para la activación de cada herramienta: el desplazamiento se hace con `QgsMapToolPan`, el zum acercar/alejar con un par de instancias `QgsMapToolZoom`. Las acciones se establecen como comprobables y posteriormente asignados a las herramientas para permitir la manipulación automática de activar/desactivar el estado de las acciones – cuando una herramienta de mapa se activa, su acción es marcada como seleccionada y la acción de la herramienta anterior es deseleccionable. Las herramientas de mapa se activan utilizando el método `setMapTool()`.

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

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

    self.canvas = QgsMapCanvas()
```

```
    self.canvas.setCanvasColor(Qt.white)

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

    self.setCentralWidget(self.canvas)

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

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

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

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

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

    self.pan()

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

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

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

Se puede colocar el código anterior a un archivo, por ejemplo `mywnd.py` y probarlo en la consola de Python dentro de QGIS. Este código colocará la capa seleccionada actualmente dentro del lienzo recién creado.

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

Sólo asegúrese que el archivo `mywnd.py` se encuentra dentro de la ruta de búsquedas de Python (`sys.path`). Si no es así, simplemente puede añadirlo: `sys.path.insert(0, '/my/path')` — de lo contrario la declaración de la importación fallará, al no encontrar el modulo.

8.3 Bandas elásticas y marcadores de vértices

Para mostrar algunos datos adicionales en la parte superior del mapa en el lienzo, utilice los elementos del lienzo de mapa. Es posible crear clases de elementos del lienzo personalizada (cubiertas más abajo), sin embargo, hay dos clases de elementos de lienzo útiles para mayor comodidad `QgsRubberBand` para dibujar polilíneas o polígonos, y `QgsVertexMarker` para dibujar puntos. Ambos trabajan con coordenadas de mapa, por lo que la figura se mueve/ se escala de forma automática cuando el lienzo está siendo desplazado o haciendo zum.

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 personalizacion, es decir, para cambiar su color o ancho de línea

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

Los elementos del lienzo están ligados a la escena del lienzo. Para ocultarlos temporalmente (y para mostrarlos de nuevo, utiliza el combo `hide()` y `show()`. Para eliminar por completo el tema, hay que eliminarlo de la escena del lienzo

```
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)

La banda elástica puede también ser utilizado para dibujar puntos, sin embargo la clase `QgsVertexMarker` es más adecuado para esto (`QgsRubberBand` sólo dibuja un rectangulo alrededor del punto deseado). Cómo utilizar el marcador de vértices

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

Este dibujará una cruz roja en posición [0,0]. Es posible personalizar el tipo de icono, tamaño, color y ancho de pluma

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

Para ocultar temporalmente de los marcadores de vértices y borrarlos del lienzo, lo mismo aplica para las bandas elásticas.

8.4 Escribir herramientas de mapa personalizados

Puede escribir sus herramientas personalizadas, para implementar un comportamiento personalizado a las acciones realizadas por los usuarios en el lienzo.

Las herramientas de mapa deben heredar de la clase `QgsMapTool` o cualquier clase derivada, y seleccione como herramienta activa en el lienzo utilizando el método `setMapTool()` como ya hemos visto.

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 limite de coordenadas en la consola. Utiliza los elementos de la banda elástica descrita antes para mostrar el rectángulo seleccionado ya que se esta definiendo.

```
class RectangleMapTool(QgsMapToolEmitPoint):
  def __init__(self, canvas):
      self.canvas = canvas
```

```
        QgsMapToolEmitPoint.__init__(self, self.canvas)
        self.rubberBand = QgsRubberBand(self.canvas, QGis.Polygon)
        self.rubberBand.setColor(Qt.red)
        self.rubberBand.setWidth(1)
        self.reset()

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

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

    def canvasReleaseEvent(self, e):
        self.isEmittingPoint = False
        r = self.rectangle()
        if r is not None:
          print "Rectangle:", r.xMinimum(), r.yMinimum(), r.xMaximum(), r.yMaximum()

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

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

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

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

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

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

        return QgsRectangle(self.startPoint, self.endPoint)

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

8.5 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)
```

CHAPTER 9

Representación del Mapa e Impresión

Generalmente hay dos maneras de importar datos para renderizar en el mapa; hacerlo rápido usando `QgsMapRenderer` o producir una salida afinada componiendo el mapa con `QgsComposition` clase y amigos.

9.1 Representación Simple

Hacer algunas capas usando `QgsMapRenderer` — crea un destino en el dispositivo de pintura (`QImage`, `QPainter` etc.), establecer conjunto de capas, extensión, salida y renderizar.

```
# 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

Si tiene más de una capa y ellas tienen diferente SRC, el sencillo ejemplo de arriba probablemente no funcionará: para obtener los valores correctos en los cálculos de dimensiones, tiene que fijar explicitamente el SRC de destino y activar reproyección al vuelo como en el ejemplo de abajo (solo la parte de configuración de representación se indica)

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

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

9.3 Producción usando el Diseñador de impresión

El Diseñador de Mapas es una herramienta muy útil si desea hacer diseños más sofisticados que la representación simple mostrada arriba. Usando el diseñador es posible crear diseños complejos de mapa incluyendo vistas de mapas, etiquetas, leyendas, tablas y otros elementos que usalmente están presentes en mapas impresos. Los diseños pueden exportarse a PDF, imágenes ráster o directamente impresos en una impresora.

El diseñador consiste de un grupo de clases. Todos ellos pertenecen a la biblioteca central. La aplicación QGIS tiene un GUI conveniente para colocar elementos, a pesar de que no esta disponible en la biblioteca GUI. Si no esta familiarizado con Qt Graphics View framework, se le recomienda que revise la documentación ahora, porque el diseñador esta basada en ella. También revise Documentación de Python de la implementación de QGraphicView.

El compositior central de clase es: `QgsComposition` which is derived from `QGraphicsScene`. Vamos a crear uno

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

Anote que la composición toma una instancia de:class:*QgsMapRenderer*. El el código esperamos que se corra dentro de la aplicación de QGIS y así crear los mapas dentro del cambas. La composición utiliza varios parámetros dentro del creador de mapas, mas importante las capas de mapa y el tamaño se pone por defecto. Cuanto se utiliza el compositor como aplicación independiente se puede crear su propio renderizador como se demuestra en la selección arriba y pasar al compositor.

Es posible agregar carios elementos (mapa, etiquetas, ...) a la composicion – estos elemenos tienen que descender de `QgsComposerItem` class.. Actualmente el apoyo a estos itemes son:

- mapa — este elemento dice a las bibliotecas dónde ponen el propio mapa. Aquí creamos un mapa y estiramos 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

Por defecto el los ítemes del compositor nuevo no tienen posición (esquina superior izquierda de la pagina) ademas de un tamaño de cero. La posición y el tamaño siempre se miden en milímetros.

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

Un marco es dibujado alrededor de cada elemento por defecto. Cómo quitar el marco

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

Además de crear a mano los elementos del diseñador, QGIS soporta las plantillas de composición que escencialmente son diseños con todos sus elementos guardados en un archivo .qpt (con formato XML). Por desgracia esta funcionalidad no está disponible todavía en la API.

Una vez que la composición está lista (los elemntos del compositor han sido añadidos al diseño), podemos proceder a producir una salida ráster y/o vectorial.

La configuración por defecto para el dseño son: tamaño de página A4 con una resolución de 300 DPI. Se puede cambiar de ser necesario. El tamaño de papel está especificado en milímetros.

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

9.3.1 Exportar como imagen ráster

El siguiente fragmento de código muestra cómo exportar una composición a una imagen ráster

```
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 Exportar como PDF

El siguiente fragmento de código exporta una composición a un archivo PDF

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

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

CHAPTER 10

Expresiones, Filtros y Calculando Valores

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 predicciones:

- 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 caracteristicas

El siguiente ejemplo evalua las expresiones dadas con un elemento. "Columna" es el nombre del campo en una capa.

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

También puedes utilizar `QgsExpression.prepare()` Si necesitas mas de una característica, utilizando, Using `QgsExpression.prepare()` puede aumentar la velocidad de la cual se corre evaluación.

```
>>> 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"
```

CHAPTER 11

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.

Podemos hacer la diferencia entre varios tipos de ajustes:

- **configuración global** — están vinculados al usuario en una máquina particular. QGIS almacena una gran cantidad de ajustes globales, por ejemplo, el tamaño de la ventana principal o tolerancia de autoensamblado predeterminado. Esta funcionalidad es proporcionada directamente por el marco Qt por medio de la clase `QSettings`. Por defecto, esta clase almacena la configuración en el sistema "nativo" de la configuración de almacenamiento, que es — registro (en Windows), archivo .plist (en macOS) o archivo .ini (en Unix). La Documentación QSettings es amplia, por lo que vamos a proporcionar un solo ejemplo sencillo.

  ```
  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)
  ```

  El segundo parámetro del método `value()` es opcional y especifica el valor por defecto si no hay valor previo establecido para el nombre de la configuración aprobada.

- **Configuración del proyecto** — varia entre los diferentes proyectos y por lo tanto están conectados con un archivo de proyecto. El color fondo del lienzo de mapa o destino del sistema de referencia de coordenadas (SRC) son ejemplos —fondo blanco y WGS84 podría ser adecuado para un proyecto, mientras que el fondo amarillo y la proyección UTM son mejores para otro. Un ejemplo de uso es el siguiente

  ```
  proj = QgsProject.instance()

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

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

Como puede ver, el método `writeEntry()` es utilizado para todo tipo de dato, pero existen varios métodos para leer el valor de configuración de nuevo, y la correspondiente tiene que ser seleccionada para cada tipo de datos.

- **Ajustes de la capa de mapa** — estos ajustes están relacionados a un caso en particular de una capa de mapa con un proyecto. Ellos *no* están conectados con la fuente de datos subyacentes de una capa, por lo que si crea dos instancias de capa de mapa de un archivo shape, no van a compartir los ajustes. La configuración se almacena en el archivo del proyecto, por lo que si el usuario abre el archivo de nuevo, los ajustes relacionados a la capa estarán allí de nuevo. Esta funcionalidad se ha añadido en QGIS v1.4. El API es similar a QSettings —que toma y regresa instancias QVariant

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

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

CHAPTER 12

Comunicarse con el usuario

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 Mostrar mensajes. La :class:'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 barrar 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
```

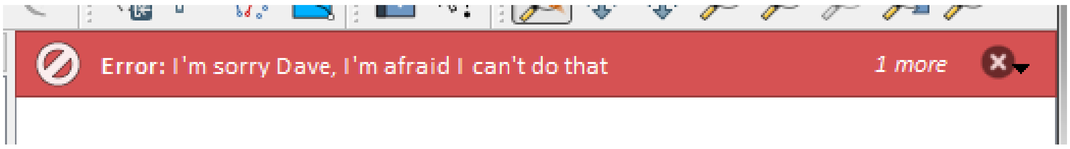


Figure 12.1: Barra de Mensajes de QGIS

Se puede establecer una duración para mostrarlo en un tiempo limitado

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

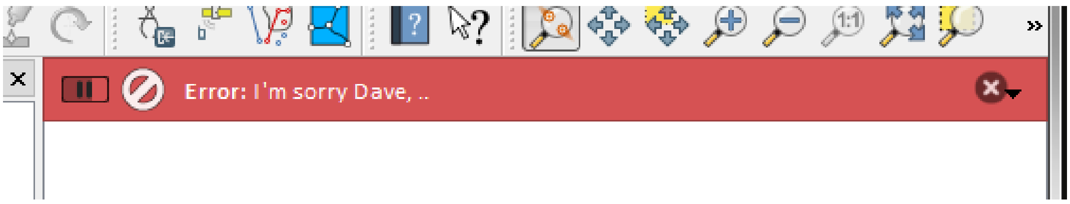


Figure 12.2: Barra de Mensajes de QGIS con temporizador

Los ejemplos anteriores muestran una barra de error, pero el parámetro `nivel` puede utilizarse para crear mensajes de alerta o de información, utilizando las constantes `QgsMessageBar.WARNING` y

`QgsMessageBar.INFO` respectivamente.

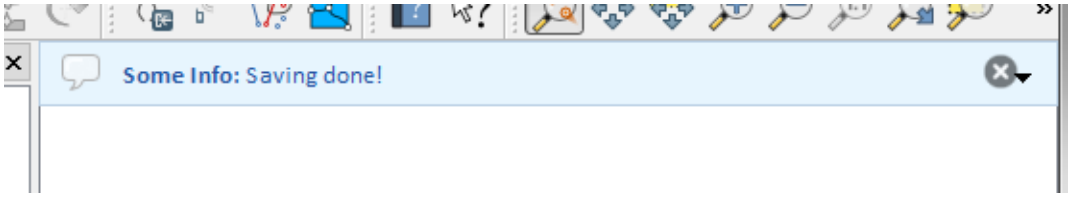


Figure 12.3: Barra de Mensajes de QGIS (información)

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)
```

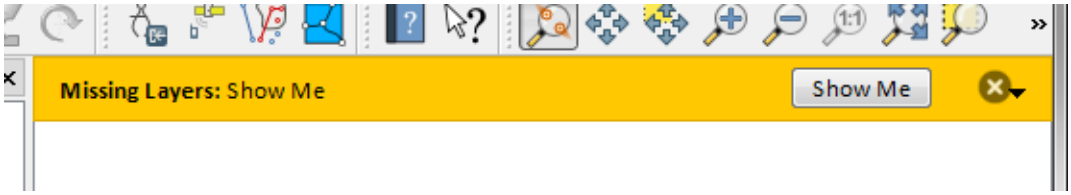


Figure 12.4: Barra de Mensajes de QGIS con un botón

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...")
```

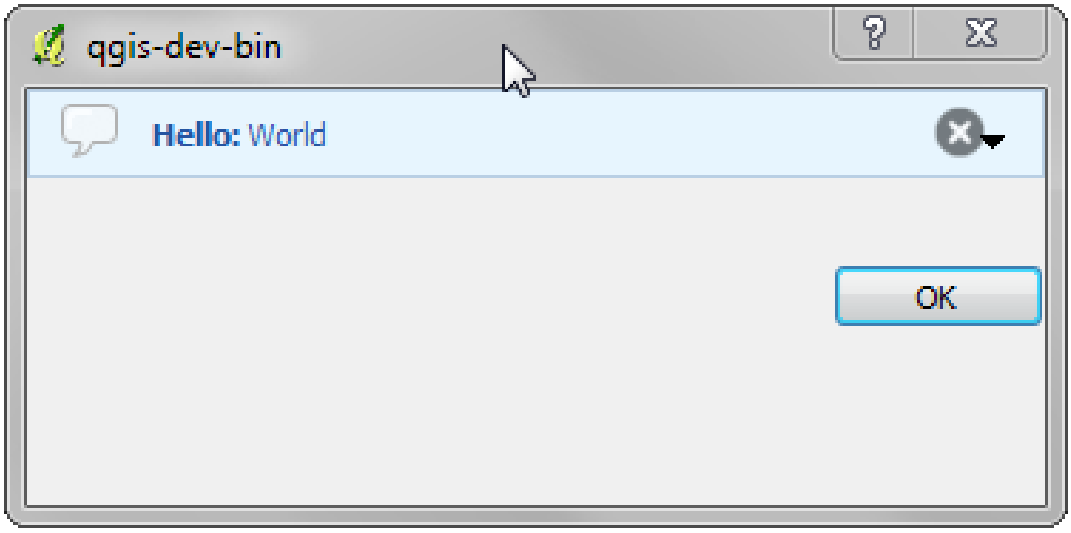


Figure 12.5: Barra de Mensajes de QGIS con un cuadro de diálogo personalizado

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

Además, puedes utilizar una barra de estatus incorporada para mostrar el progreso, como en el siguiente ejemplo

```
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', QgsMessageLo
QgsMessageLog.logMessage("Your plugin code might have some problems", level=QgsMessageLog.WARNING
QgsMessageLog.logMessage("Your plugin code has crashed!", level=QgsMessageLog.CRITICAL)
```

CHAPTER 13

Desarrollo de Plugins Python

Es posible crear complementos en lenguaje de programación Python. En comparación con complementos clásicos escritos en C++, éstas deberían ser más fácil 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 esta 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`). Rellene 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 eb 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, *True* o *False*
obsoleto	Falso	bandera booleana, *True* or *False*, 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 *Ráster*, *Vector*, *Base de Datos* y *Web*

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.mainWin
    self.action.setObjectName("testAction")
    self.action.setWhatsThis("Configuration for test plugin")
    self.action.setStatusTip("This is status tip")
    QObject.connect(self.action, SIGNAL("triggered()"), self.run)

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

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

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

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

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

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

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.mainWin
    self.action.setObjectName("testAction")
    self.action.setWhatsThis("Configuration for test plugin")
    self.action.setStatusTip("This is status tip")
    QObject.connect(self.action, SIGNAL("triggered()"), self.run)
    self.menu.addAction(self.action)

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

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

No olvide establecer `QAction` y `QMenu` `objectName` a un nombre especifico 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 comodo modificar la variable de entorno PATH o especificando la ruta del ejecutable de forma explicita 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 ves 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.

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

CHAPTER 14

Authentication infrastructure

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 = 'fm1s770'
```

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

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

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

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

Available Authentication methods

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

1. `Basic` User and password authentication
2. `Identity-Cert` Identity certificate authentication
3. `PKI-Paths` PKI paths authentication
4. `PKI-PKCS#12` PKI PKCS#12 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
boundle = QgsPkiBundle.fromPemPaths( "/path/to/alice-cert.pem",
                                     "/path/to/alice-key_w-pass.pem",
                                     "unlock_pwd",
                                     "list_of_CAs_to_bundle" )
assert boundle is not None
assert boundle.isValid()
```

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

14.3.3 Remove entry from authdb

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

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

14.3.4 Leave authcfg expansion to QgsAuthManager

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

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

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

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

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

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

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:

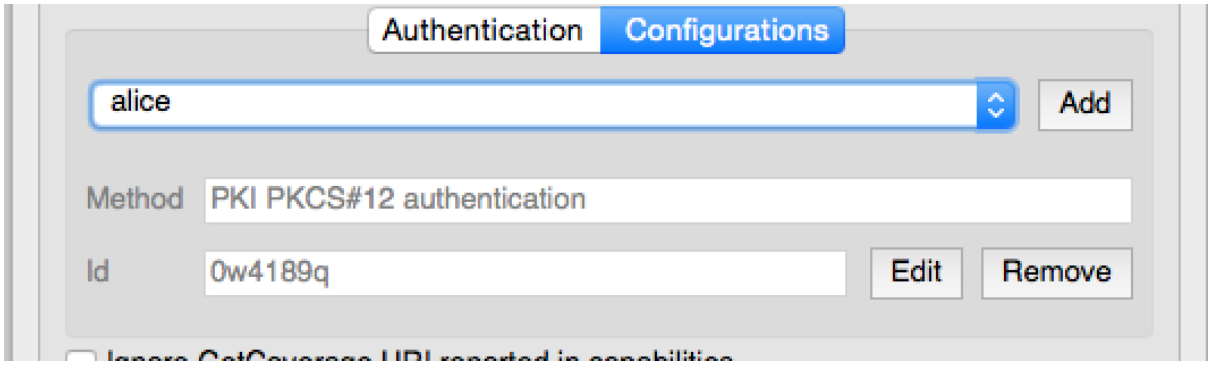
```
http = NetworkAccessManager(authid="my_authCfg", exception_class=My_FailedRequestError)
try:
  response, content = http.request( "my_rest_url" )
except My_FailedRequestError, e:
  # Handle exception
  pass
```

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:

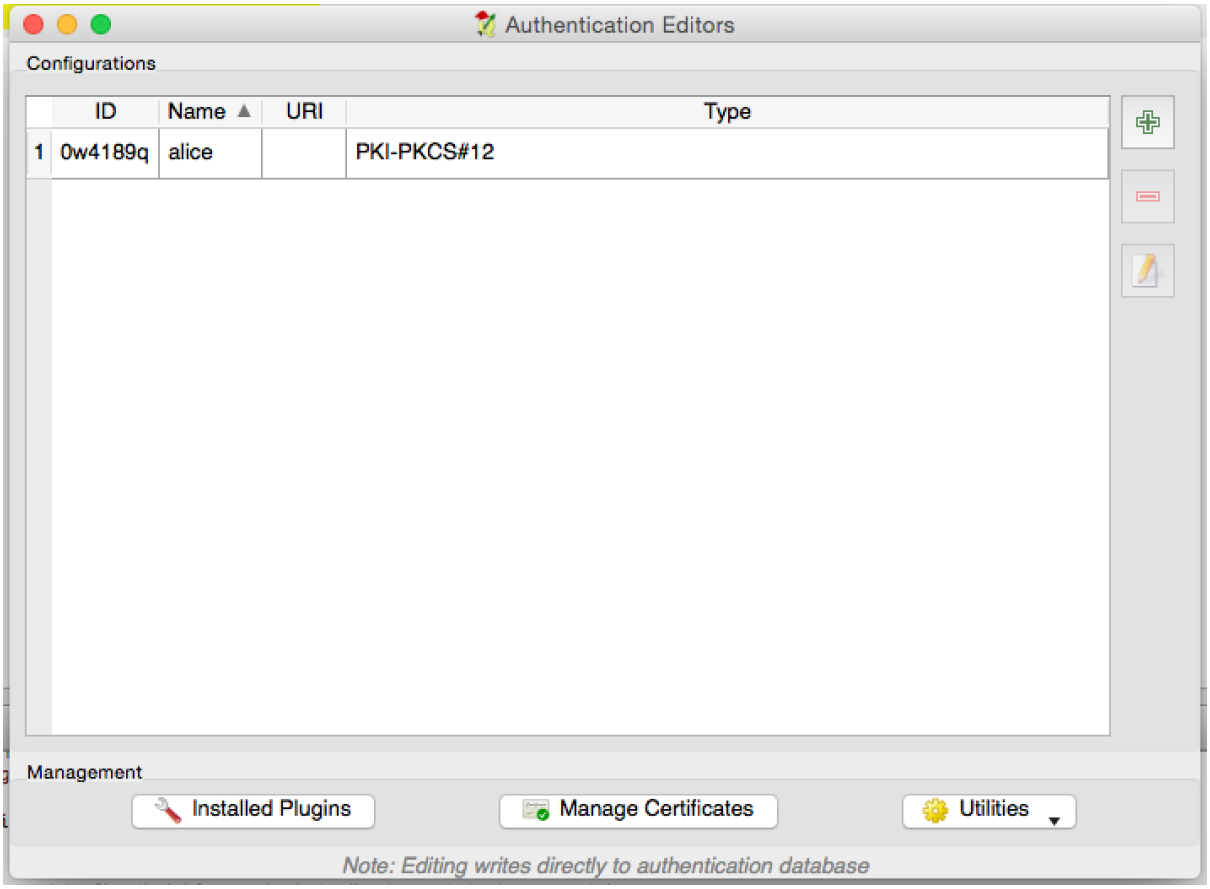
```
# create the instance of the QgsAuthConfigSelect GUI hierarchically linked to
# the widget referred with 'parent'
gui = QgsAuthConfigSelect( parent, "postgres" )
# add the above created gui in a new tab of the interface where the
```

```
# 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 Method*s with the specified provider.

14.5.2 Authentication Editor GUI

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



and can be used as in the following snippet:

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

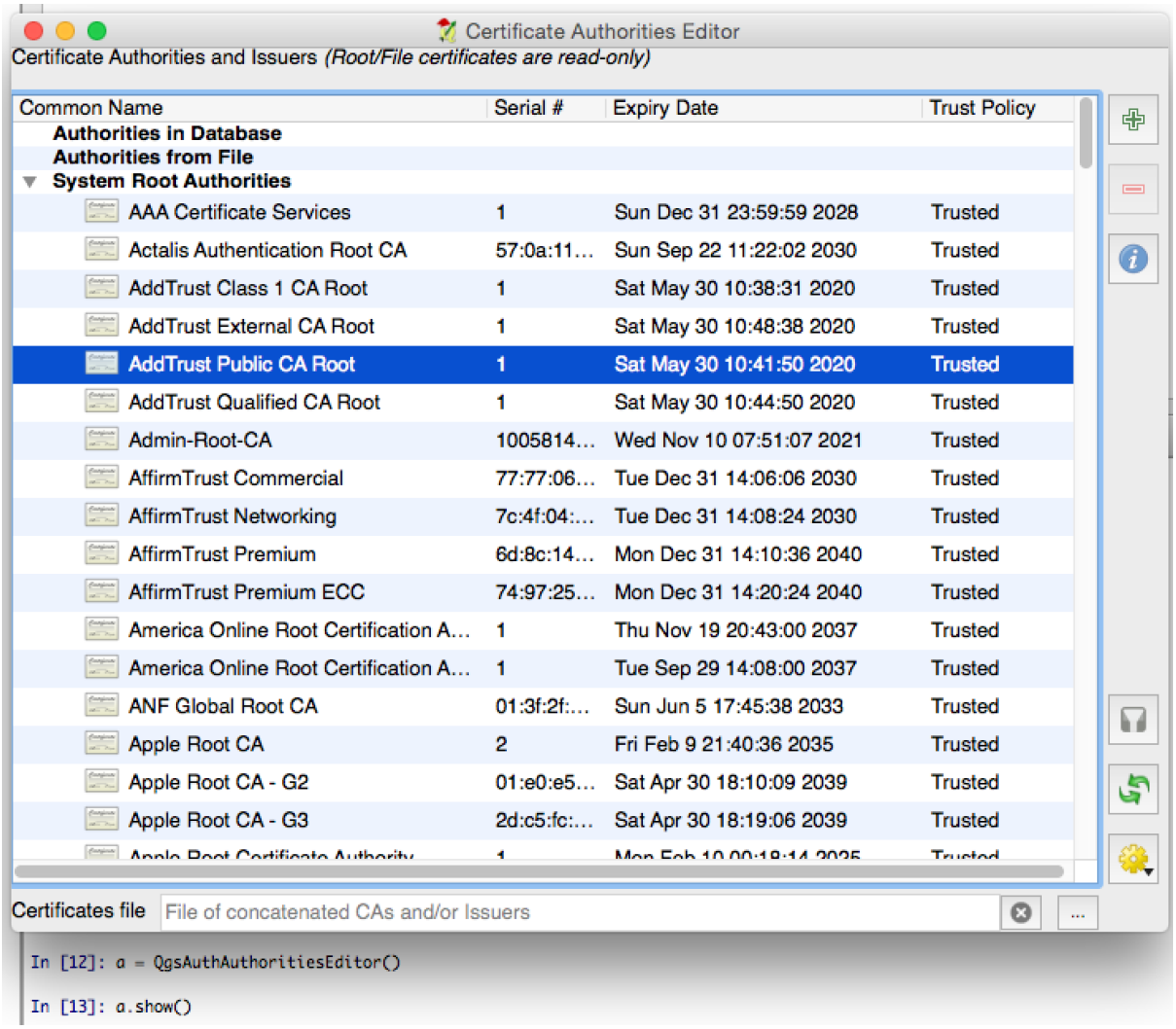
an integrated example can be found in the related test

14.5.3 Authorities Editor GUI

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

and can be used as in the following snippet:

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


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

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

CHAPTER 15

Configuración IDE para escribir y depurar complementos

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 clie 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:OSGeo4Wappsqgisbin`, pero cuando se ha compilado su aplicacion 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.

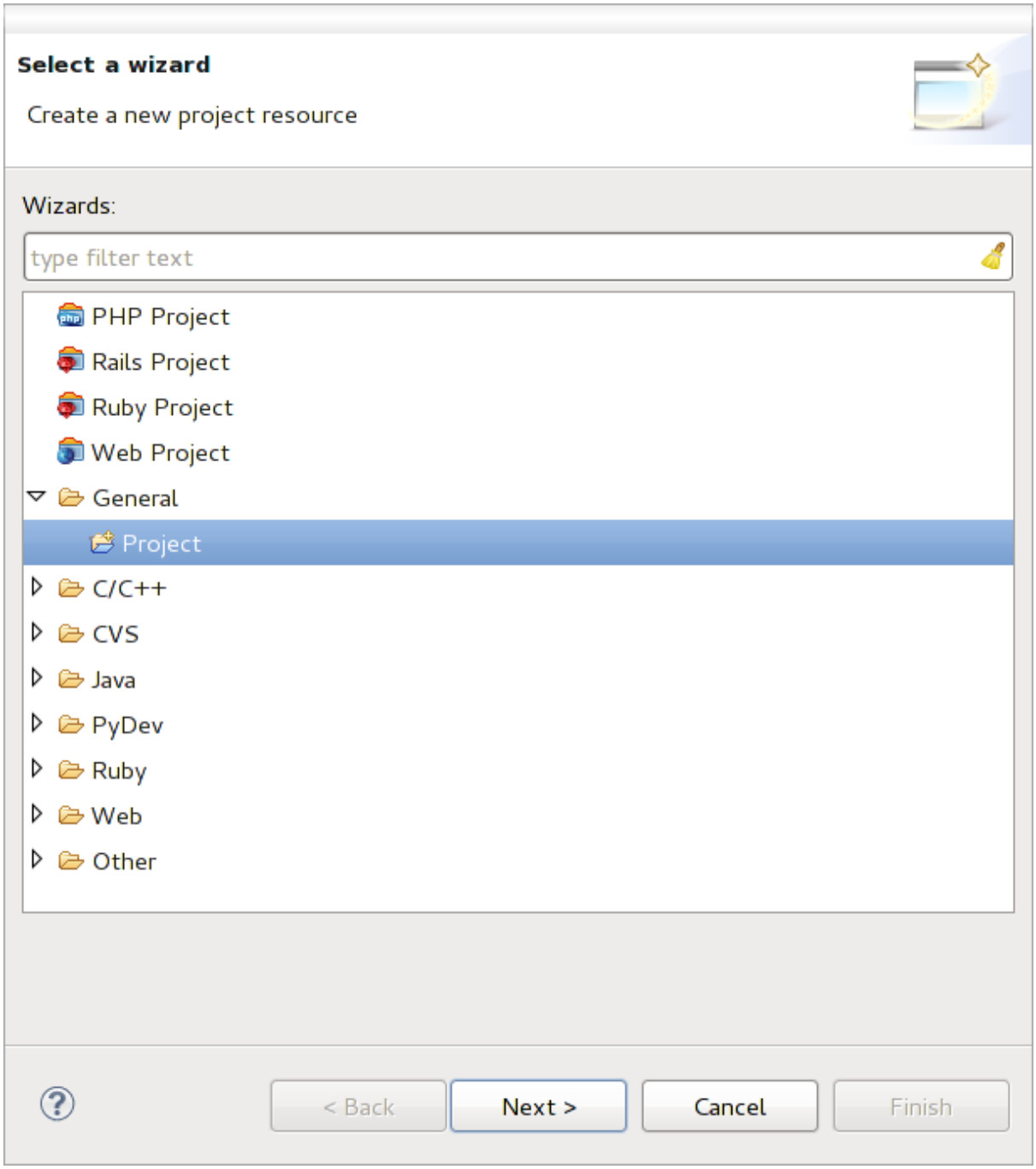


Figure 15.1: Proyecto de Eclipse

15.2.4 Configurando el depurador

Para consegir 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).

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

Figure 15.2: 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.

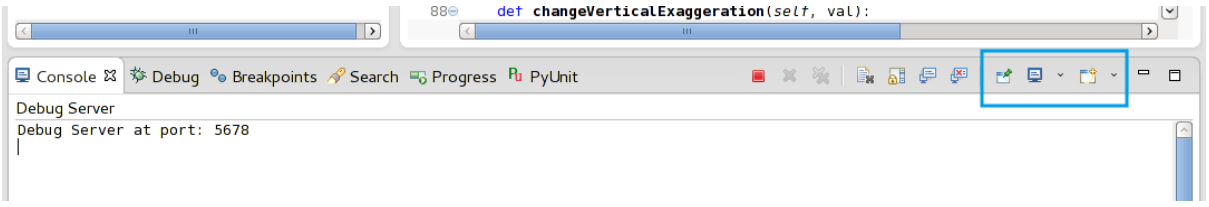


Figure 15.3: consola de depuración PyDev

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 incluídos forzados*

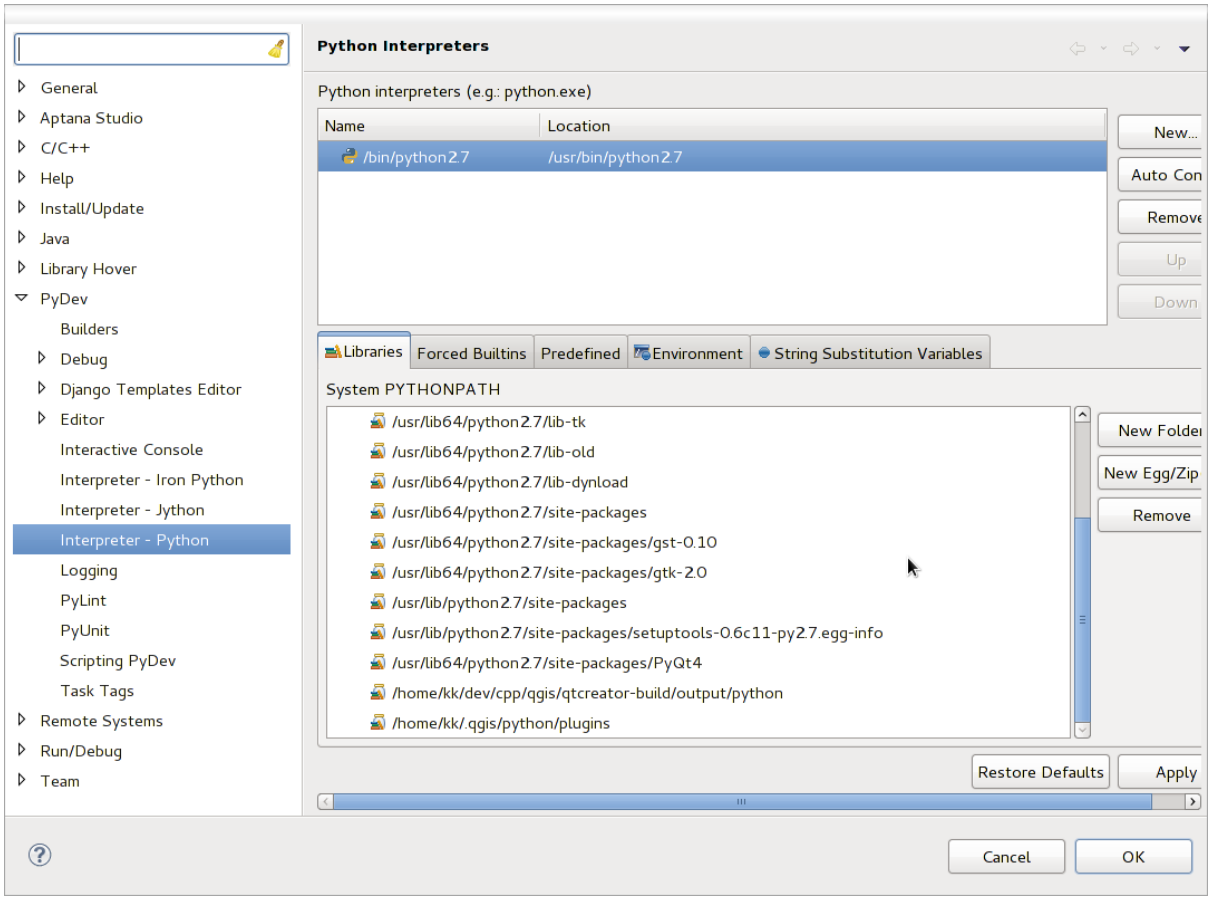


Figure 15.4: consola de depuración PyDev

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 esta.

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

CHAPTER 16

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 especifica 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
```

CHAPTER 17

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 esta 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.mainWindo
  self.action.setWhatsThis("Configuration for test plugin")
  self.action.setStatusTip("This is status tip")
  QObject.connect(self.action, SIGNAL("triggered()"), self.run)

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

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

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

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

CHAPTER 18

Compartiendo sus plugins

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 repositiorio 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
```

CHAPTER 19

Fragmentos de código

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

19.1 Cómo llamar a un método por un atajo de teclado

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)
```

CHAPTER 20

Escribir nuevos complementos de procesamiento

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).

Este documento describe como crear un nuevo complemento, para añadir funcionalidad a los algoritmos del menú Procesos.

Existen dos mecanismos para lograr esto:

- Crear un complemento que incluya un proveedor de algoritmos. Esta es la opción más compleja, pero nos da una mayor flexibilidad.
- Crear un complemento que contenga una serie de archivos con el código necesario para los procesos. Es la solución más simple, ya que solamente necesitaríamos los archivos con el código de los Procesos.

20.1 Crear un complemento que incluya un proveedor de algoritmos

Para crear un proveedor de algoritmos, hay que seguir los pasos que se indican a continuación:

- 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 Crear un complemento que contenga una serie de scripts de procesamiento

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.
- Haga click sobre Aceptar y el complemento se creará.
- You can add additional scripts to the plugin by adding scripts python files to the *scripts* folder in the resulting plugin folder.

CHAPTER 21

Biblioteca de análisis de redes

A partir de la versión ee19294562 (QGIS >= 1.8) la nueva librería de análisis de redes se agregó a la librería de análisis de nucleo 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 RoadGraph 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 mimso vértice gráfico. Por lo que dos líneas que tienen un nodo en común se conectaran entre si.

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.

Convertir de una capa vectorial a una gráfica se hace utilizando el **'Patrón de la programación del constructor<http://en.wikipedia.org/wiki/Builder_pattern>'_**. Una gráfica se construye utilizando un llamado director.

Hay solo un Director por ahora: QgsLineVectorLayerDirector. El director establece la configuración básica que se utilizará para construir una gráfica de una capa vectorial de línea, utilizado por el constructor para crear la gráfica. Actualmente, con en el caso con el director, solo un constructor existe: QgsGraphBuilder, que crea objetos QgsGraph. Se puede querer implementar su propio constructor que construya un grafo compatible con cada librería como BGL or NetworkX.

Para calcular las propiedades del borde el patrón de programación se utiliza strategy. Por ahora solo QgsDistanceArcProperter estrategicamente esta disponible, que toma en cuenta la longitud de la ruta. Se puede implemenetar su propia estrategia que utilizará todos los parametros necesarios. Por ejemplo, el complemento RoadGraph utiliza una estrategía que calcula el tiempo de viaje mediante la longitud del borde y el valor de la velocidad de los atributos.

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 cámino 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

Para obtener el árbol de la ruta más corta utilice los métodos `shortestTree()` y `dijkstra()` de la clase QgsGraphAnalyzer. Es recomendable utilizar el método `dijkstra()` porque funciona más rápido y utiliza memoria más efectivamente.

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:

- fuente — 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)
```

Aquí hay algunos sencillos ejemplos de código para mostrar el árbol de la ruta más corta usando el grafo creado con el método `shortestTree()` (seleccione la capa de linestring en TOC y reemplace las coordenadas con las propias). **Advertencia**: utilice este código sólo como ejemplo, cree muchos objetos QgsRubberBand y puede ser lento en grandes conjuntos de datos .

```
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 optima 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 raiz 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))
```

CHAPTER 22

Complementos de Python de QGIS Server

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', QgsMessa
        else:
            QgsMessageLog.logMessage("FAIL    - ParamsFilter.responseComplete", 'plugin', QgsMessa
```

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

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

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

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

22.3.5 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.