"

PyQGIS developer cookbook

Release 2.14

QGIS Project

08 August 2017

Contents

CHAPTER 1

Introduzione

Questo documento è da intendersi sia come un tutorial che come una guida di riferimento. Sebbene esso non contempli tutti i possibili casi d'uso, dovrebbe comunque fornire una buona panoramica delle funzionalità principali.

A partire dalla versione 0.9, QGIS ha il supporto per lo scripting opzionale utilizzando il linguaggio Python. Abbiamo optato per il Python perché è uno dei linguaggi maggiormente preferiti per lo scripting. Le dipendenze PyQGIS fanno riferimento a SIP e PyQt4. Il motivo per cui si utilizza SIP invece del più diffuso SWIG è che l'intero codice di QGIS dipende dalle librerie Qt. Le dipendenze Python per Qt (PyQt) sono eseguite anche utilizzando SIP e ciò consente un'integrazione senza interruzioni tra PyQGIS e PyQt.

Ci sono diversi modi per utilizzare Python nel desktop QGIS che sono trattati in dettaglio nelle seguenti sezioni:

- avviare automaticamente codice Python all'avvio di QGIS
- esegui comandi nella console Python all'interno di QGIS
- creare ed usare plugin in Python
- Creare un'applicazione personalizzata basata sulle API di QGIS

Python è utilizzabile anche in QGIS Server:

- dallla versione 2.8, i plugin Python sono utilizzabili anche su QGIS Server (see: Server Python Plugins)
- dalla versione 2.11 la libreria QGIS Server può essere usata per collegare QGIS Server con un applicazione Python.

Esiste un riferimento di *API di QGIS completa <http://qgis.org/api/>* _ ref che documenta le classi provenienti dalle librerie di QGIS. Le API di QGIS per Python sono quasi identiche a quelle in C++.

Una buona risorsa quando si tratta di plugin è scaricare alcuni plugin dal *plugin repository <http://plugins.qgis.org/>* e esaminarne il codice. Inoltre, la cartella `python/plugins/` nell'installazione QGIS contiene un plugin che puoi utilizzare per imparare a sviluppare plugin e come eseguire alcune delle attività più comuni.

1.1 avviare automaticamente codice Python all'avvio di QGIS

Esistono du metodi distinti per avviare codice Python all'avvio di QGIS.

1.1.1 Variabile di ambiente PYQGIS_STARTUP

Si può avviare codice Python subito prima dell'inizializzazione impostando la variabile d'ambiente `PYQGIS_STARTUP` al percorso di un file Python esistente.

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

1.1.2 The `startup.py` file

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

1.2 Console Python

Per chi utilizza gli script, è possibile sfruttare la console Python integrata, accessibile dal menu *Plugins* → *Python Console*. La console si apre come una finestra di dialogo non modale.

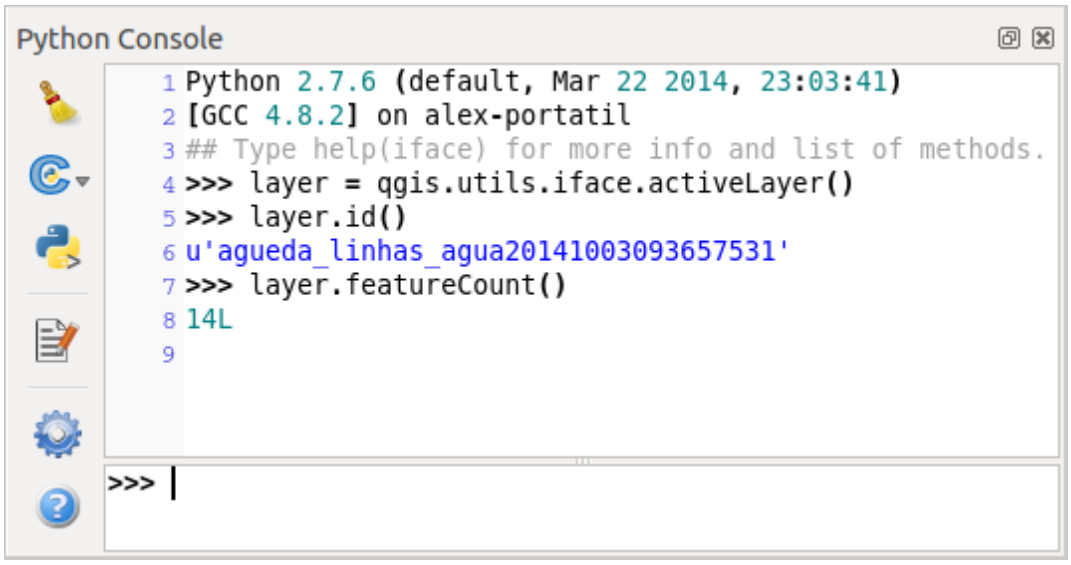

Figure 1.1: Console python di GIS

La schermata soprastante illustra come ottenere il layer attualmente selezionato nella lista dei layer, mostrare il suo ID e, se si tratta di un vettore, mostrare il conteggio delle geometrie. Per l'interazione con l'ambiente di QGIS, esiste una variabile `iface`, che è un'istanza della classe `QgsInterface`. Tale interfaccia consente di accedere alla mappa, ai menu, ai pannelli e alle parti di QGIS.

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
```

Per chi utilizza spesso la console, potrebbe essere utile impostare una scorciatoia per attivarla (all'interno del menu *Impostazioni* → *Configura scorciatoie...*)

1.3 Plugin Python

QGIS consente un miglioramento delle sue funzionalità utilizzando i plugin. Ciò era originariamente possibile solo con il linguaggio C++. Con l'aggiunta del supporto Python per QGIS, è altresì possibile utilizzare plugin scritti in Python. Il vantaggio principale rispetto ai plugin in C++ sono la sua semplicità di distribuzione (non è necessario la compilazione per ogni piattaforma) e un più facile sviluppo.

A partire dall'introduzione del supporto Python sono stati scritti molti plugin che includono diverse funzionalità. L'installatore di plugin consente agli utenti di scaricare facilmente, aggiornare e rimuovere i plugin Python. Vedi la pagina Repository dei Plugin Python per vari esempi di plugin.

Creare un plugin in python è semplice, vedi *Sviluppare Plugin Python* per avere istruzioni dettagliate.

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

1.4 Applicazioni Python

Spesso, quando si processano alcuni dati GIS, è comodo creare degli script per automatizzare il processo invece di ripetere la stessa procedura diverse volte. Con PyQGIS, ciò è perfettamente possibile — importa il modulo `qgis.core`, inizializzalo e sei pronto per il processamento.

Oppure potresti voler creare un'applicazione interattiva che usa alcune funzionalità di QGIS — misurare alcune dati, esportare una mappa in PDF o una qualunque altra operazione. Il modulo `qgis.gui` porta con sé alcune componenti della GUI, in particolare il widget di mappa che può essere incorporato molto facilmente all'interno dell'applicazione con il supporto per zoomare, spostare e/o qualsiasi altro strumento di mappa.

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.

Nota: *non* usare `qgis.py` come nome per il tuo script di prova — Python non sarà in grado di importare le dipendenze poiché il nome dello script le metterà in ombra.

1.4.1 Using PyQGIS in standalone scripts

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

```
from qgis.core import *

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

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

# load providers
qgs.initQgis()

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

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

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

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

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

1.4.2 Using PyQGIS in custom applications

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

```
from qgis.core import *

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

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

# load providers
qgs.initQgis()

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

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

Adesso puoi lavorare con l'API di QGIS — carica i layer ed esegui un po' di processamento oppure avvia la GUI con una mappa. Le possibilità sono infinite :-)

1.4.3 Avviare applicazioni personalizzate

Avrai bisogno di dire al tuo sistema dove cercare per le librerie di QGIS e i moduli Python appropriati nel caso in cui non si trovino in una posizione conosciuta — altrimenti Python lo chiederà:

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

Ciò può essere fissato impostando la variabile di ambiente `PYTHONPATH`. Nei comandi seguenti, `qgispath` dovrebbe essere sostituito con il tuo attuale percorso di installazione di QGIS.

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

Il percorso per i moduli PyQGIS è adesso noto, comunque essi dipendono dalle librerie `qgis_core` e `qgis_gui` libraries (i moduli Python servono solo come contenitori). Il percorso per tali librerie è tipicamente sconosciuto al sistema operativo, così ottieni di nuovo un errore di importazione (il messaggio potrebbe variare in funzione del sistema):

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

Correggi ciò aggiungendo le cartelle in cui si trovano le librerie di QGIS per cercare il percorso del collegamento dinamico:

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

Questi comandi possono essere inseriti in uno script che se ne occuperà all'avvio del programma. Quando si fa uso di applicazioni personalizzate utilizzando PyQGIS, esistono di solito due possibilità:

- richiedi all'utente di installare QGIS sulla sua piattaforma prima di installare la tua applicazione. L'installatore dell'applicazione dovrebbe guardare le posizioni predefinite delle librerie di QGIS e consentire all'utente di impostarne il percorso se non trovato. Questo approccio ha il vantaggio di essere più semplice, ma richiede all'utente di eseguier più passaggi.
- impacchetta QGIS insieme alla tua applicazione. Rilasciare l'applicazione può essere più impegnativo e il pacchetto sarà più grande, ma l'utente sarà risparmiato dall'onere di scaricare e installare parti addizionali del programma.

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

CHAPTER 2

Caricamento di progetti

A volte potreste avere bisogno di caricare un progetto esistente da un plugin opuure (piú frequentemente) quando si sviluppa un'applicazione QGIS Python stand-alone (riferimento: *Applicazioni Python*).

Per caricare un progetto all'interno dell'applicazione QGIS corrente serve un oggetto `QgsProject instance()` e si deve invocare il suo metodo `read()` passandogli l'oggetto `QFileInfo` che contiene il percorso da cui il progetto verrà caricato:

```
# 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'
```

Nel caso in cui si abbia bisogno di fare delle modifiche al progetto(ad esempio aggiungere o rimuovere alcuni layer) e salvare le modifiche, sará possibile chiamare il metodo `write()` dell'istanza del vostro progetto. Il metodo `write()` inoltre accetta opzionalmente `QFileInfo` che consente di specificare il percorso dove il progetto verrá salvato:

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

Sia `read()` che `write()` restituiscono un valore booleano che puó essere utilizzato per controllare che l'operazione si sia conclusa con successo.

Nota: Se stai scrivendo un'applicazione standalone, per sincronizzare il progetto caricato con la mappa è necessario inizializzare una classe `QgsLayerTreeMapCanvasBridge` come nell'esempio sotto:

```
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

Caricamento del vettore

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

3.1 Vector Layers

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
uri = "path/to/gpx/file.gpx?type=track"
vlayer = QgsVectorLayer(uri, "layer name you like", "gpx")
```

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

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

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

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

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

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

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

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

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

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

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

3.2 Raster Layers

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

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

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

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

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

Raster layers can also be created from a WCS service:

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

detailed URI settings can be found in provider documentation

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

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

3.3 Map Layer Registry

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

Adding a layer to the registry:

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

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

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

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

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

CHAPTER 4

Usare i raster

Questa sezione elenca le varie operazioni che si possono eseguire sui raster.

4.1 Dettagli del raster

Un raster consiste di una o piú bande — puó essere sia a banda singola che multi banda. Ogni banda rappresenta una matrice di valori. Una normale immagine a colori (e.g. una foto aerea) é un raster composto dalle bande ross, blu e verde. I raster a singola banda solitamente rappresentano o variabili continue (e.g. altitudine) oppure variabili discrete (e.g. uso della terra). In alcuni casi, un raster é associato ad una tavolozza ed i valori del raster si riferiscono ai colori memorizzati nella tavolozza:

```
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 Visualizzatore

Quando un raster viene caricato, esso ottiene un visualizzatore predefinito basato sul suo tipo. Può essere modificato sia attraverso le proprietà del raster che programmaticamente.

Per interrogare il visualizzatore corrente:

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

Per impostare un visualizzatore usa il metodo `setRenderer()` della classe `QgsRasterLayer`. Esistono diversi classi di visualizzatori disponibili (derivanti da `QgsRasterRenderer`):

- QgsMultiBandColorRenderer
- QgsPalettedRasterRenderer
- QgsSingleBandColorDataRenderer
- QgsSingleBandGrayRenderer
- QgsSingleBandPseudoColorRenderer

I raster a banda singola possono essere mostrati sia tramite scala di grigi (valori bassi = nero, valori alti = bianco) o con un algoritmo per pseudocolori che assegna i colori per i valori della singola banda. I raster a banda singola con una tavolozza possono inoltre essere mostrati usando la loro tavolozza. I raster multibanda sono solitamente mostrati mappando le bande con i colori RGB. Un'altra possibilitá é quella di utilizzare una singola banda in scala di grigio o con pseudocolori.

Le prossime sezioni spiegano come interrogare e modificare lo stile del raster. Dopo aver effettuato i cambiamenti, potrebbe essere necessario forzare l'aggiornamento della mappa, vedi *Aggiornare i Raster*.

TODO: miglioramenti sul contrasto, trasparenza (no data), min/max definiti dall'utente, statistiche sulla banda

4.2.1 Raster a Banda Singola

Ipotizziamo di voler visualizzare il nostro raster (assumendo che abbia una sola banda) con dei colori che vadano dal verde al giallo (per valori del pixel da 0 a 255), Nella prima fase prepareremo l'oggetto ‘‘QgsRasterShader‘‘ e configureremo la sua funzione shader:

```
>>> 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)
```

Lo shader mappa i colori così come specificato dalla sua mappa colore. La mappa colore è fornita come una lista di elementi avente il valore del pixel e il suo colore associato. Esistono tre modalità di interpolazione dei valori:

- linear (`INTERPOLATED`): il colore risultante é linearmente interpolato a partire dai valori dei colori della mappa al di sopra ed al di sotto del valore corrente
- discrete (`DISCRETE`): il colore é preso dai colori della mappa aventi valore maggiore od uguale
- exact (`EXACT`): il colore non é interpolato, vengono mostrati solo i pixel aventi valore uguale alla mappa dei colori

Nella seconda fase assoceremo questo shader al raster:

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

Il numero 1 nel codice qui sopra è il numero della banda (le bande raster sono indicizzate a partire da uno).

4.2.2 Raster Multi Banda

Come impostazione predefinita, QGIS mappa le prime tre bande con i valori rosso, verde e blu per creare un'immagine a colori (questo é lo stile `MultiBandColor`. In alcuni casi potrebbe essere utile modificare queste impostazioni. Il seguente codice scambia la banda rossa (1) con quella verde (2):

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

Nel caso in cui sia necessaria solo una banda per la visualizzazione del raster, si può scegliere la rappresentazione a banda singola — o a scala di grigi o falso colore.

4.3 Aggiornare i Raster

Quando si cambia la simbologia di un raster ed essere sicuri che i cambiamenti siano immediatamente visibili agli utenti, si possono invocare i seguenti metodi

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

La prima chiamata garantisce che l'immagine in cache del layer mostrato sia cancellata nel caso in cui la cache della visualizzazione sia attivata. Questa funzionalitá é disponibile a partire da QGIS 1.4, tale funzione non esiste nelle versioni precedenti — per essere sicuri che il codice funzioni con tutte le versioni di QGIS, controlleremo prima che il metodo esista.

La seconda chiamata emette un segnale che forza la mappa contenente il layer ad aggiornarsi.

Questi comandi non funzionano con raster WMS. In questo caso va fatto in maniera esplicita

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

Nel caso sia stata cambiata la simbologia del raster (far riferimento alle sezioni riguardanti raster e vettori a tal proposito), si potrebbe voler forzare QGIS ad aggiornare la simbologia raster nella widget della lista dei raster (legend). Ció puó essere fatto come segue (`iface` é un'istanza di `QgisInterface`)

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

4.4 Valori dell'interrogazione

Per eseguire un'interrogazione sui valori delle bande di un raster in un punto specifico

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

Il metodo ‘‘results‘‘in questo caso restituisce un dizionario, usando gli indici delle bande come chiavi, ed i valori delle bande come valori.

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

CHAPTER 5

Usare i Vettori

Questa sezione riassume le varie azioni che si possono eseguire con i vettori.

5.1 Retrieving information about attributes

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

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

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

5.2 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 Iterare un Vettore.

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 Accessing attributes

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 Iterare le caratteristiche selezionate

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 Iterare un sottoinsieme di caratteristiche

Nel caso si voglia iterare su un sottoinsieme di geometrie in un vettore, ad esempio quelle di un'area specifica, si deve aggiungere l'oggetto `QgsFeatureRequest` alla chiamata `getFeatures()`. Di seguito un esempio

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

If you need an attribute-based filter instead (or in addition) of a spatial one like shown in the example 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 *Espressioni, Filtraggio e Calcolo di Valori* 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)
```

Suggerimento: 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 Modificare i Vettori

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 Aggiungi Geometrie

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 Elimina Geometrie

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

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

5.4.3 Modifica Geometrie

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 })
```

Suggerimento: 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.)

5.4.4 Aggiungere e Rimuovere Campi

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

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

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 Modificare i Vettori con un Buffer di Modifica

Quando modifichi i vettori contenti in QGIS, devi prima attivare la modalità di modifica per un particolare layer, successivamente eseguire alcune modifiche e alla fine applicare (o annullare) i cambiamenti. Tutte le modifiche che fai non vengono scritte fino a quando non le applichi — si trovano in un buffer di modifica in memoria del vettore. È possibile usare questa funzionalità anche programmaticamente — è solo un altro metodo per modificare il vettore che è complementare all'uso diretto delle sorgenti dati. Usa questa opzione quando si forniscono alcuni strumenti dell'interfaccia grafica per la modifica del vettore, siccome questo permetterà all'utente di applicare/annullare e permetter l'uso di annulla/ripristina. Quando si applicano le modifiche, tutte le modifiche dal buffer di modifica sono salvate nella sorgente dati.

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

```
# 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):
  f = layer.getFeatures().next()
  f[0] = 5
  layer.updateFeature(f)
```

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 Usare l'Indice Spaziale

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)
```

- 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 Scrivere Vettori

Puoi scrivere file vettoriali usando la classe `QgsVectorFileWriter`. Essa supporta qualsiasi altro tipo di file vettoriale che supporta OGR (shapefile, GeoJSON, KML e altri).

Ci sono due possibilità per esportare un vettore:

- from an instance of `QgsVectorLayer`

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

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

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

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

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

- directly from features

```
# define fields for feature attributes. A 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 Shapef

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 Provider in Memoria

Il provider in memoria è previsto per essere usato principalmente da sviluppatori di plugin o applicazioni di terze parti. Esso non memorizza dati sul disco, permettendo agli sviluppatori di usarlo come backend veloce per alcuni layer temporanei.

Il provider supporta campi di tipo string, int e double.

Il provider in memoria supporta anche l'indicizzazione spaziale, che si abilita richiamando la funzione `createSpatialIndex()` del provider. Una volta creato l'indice spaziale sarai in grado di iterare più velocemente sugli elementi contenuti in regioni più piccole (poiché non è necessario scorrere tutti gli elementi, ma solo quelli nel rettangolo specificato).

Un provider in memoria è creato passando `"memory"` come stringa sorgente al costruttore `QgsVectorLayer`.

Il costruttore inoltre accetta un URI che definisce il tipo di geometria del vettore, una tra: `"Point"`, `"LineString"`, `"Polygon"`, `"MultiPoint"`, `"MultiLineString"`, o `"MultiPolygon"`.

L'URI può anche specificare il sistema di riferimento delle coordinate, i campi e l'indicizzazione del provider in memoria nell'URI. La sintassi è:

crs=definizione Specifica il sistema di riferimento delle coordinate dove la definizione può essere una qualsiasi delle forme accettate da `QgsCoordinateReferenceSystem.createFromString()`

index=yes Specifica che il provider userà un indice spaziale

field=nome:tipo(lunghezza,precisione) Specifica un attributo del vettore. L'attributo ha un nome, e facoltativamente un tipo (intero, double, o string), lunghezza, e precisione. Ci sono possono essere definizioni di campo multiple

The following example of a URI incorporates all these options

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

The following example code illustrates creating and populating a memory provider

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

```
pr = vl.dataProvider()

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

Quando un vettore deve essere visualizzato, l'aspetto dei dati è dato dal **visualizzatore** e dai **simboli** associati al vettore. I simboli sono classi che si occupano del disegno di rappresentazione visiva delle geometrie, mentre i visualizzatori determinano quale simbolo sarà usato per una particolare geometria.

Il visualizzatore per un dato vettore può essere ottenuto come mostrato sotto:

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

And with that reference, let us explore it a bit

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

Ci sono vari tipi di visualizzatori noti disponibili nella libreria di base di QGIS:

Tipo	Classe	Descrizione
singleSymbol	`QgsSingleSymbolRendererV2`	Visualizza tutte le geometria con lo stesso simbolo
categorizedSymbol	`QgsCategorizedSymbolRendererV2`	Visualizza le geometria usando un simbolo diverso per ogni categoria
graduatedSymbol	`QgsGraduatedSymbolRendererV2`	Visualizza le geometrie usando un simbolo diverso per ogni intervallo di valori

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

```
print QgsRendererV2Registry.instance().renderersList()
# Print:
[u'singleSymbol',
u'categorizedSymbol',
```

```
u'graduatedSymbol',
u'RuleRenderer',
u'pointDisplacement',
u'invertedPolygonRenderer',
u'heatmapRenderer']
```

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

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

5.9.1 Single Symbol Renderer

Puoi ottenere il simbolo usato per la visualizzazione chiamando il metodo `symbol()` e cambiarlo con il metodo `setSymbol()` (nota per gli sviluppatori C++: il visualizzatore assume la proprietà del simbolo.)

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`
- `square`
- `cross`
- `rectangle`
- `diamond`
- `pentagon`
- `triangle`
- `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 propery...
layer.rendererV2().symbol().symbolLayer(0).setName('square')
# ... but not all properties are accessible from methods,
# you can also replace the symbol completely:
props = layer.rendererV2().symbol().symbolLayer(0).properties()
props['color'] = 'yellow'
props['name'] = 'square'
layer.rendererV2().setSymbol(QgsMarkerSymbolV2.createSimple(props))
```

5.9.2 Categorized Symbol Renderer

Puoi interrogare e impostare il nome dell'attributo che è usato per la classificazione: usa i metodi `classAttribute()` e `setClassAttribute()`.

To get a list of categories

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

Dove `value()` è il valore usato per la distinzione tra le categorie, `label()` è un testo usato per la descrizione della categoria e il metodo `symbol()` restituisce il simbolo assegnato.

Il visualizzatore di solito memorizza anche il simbolo di origine e la scala di colori utilizzati per la classificazione: metodi `sourceColorRamp()` e `sourceSymbol()`.

5.9.3 Graduated Symbol Renderer

Questo visualizzatore è molto simile al visualizzatore simbolo categorizzato descritto sopra, ma invece di un valore di attributo per classe esso lavora con intervalli di valori e quindi può essere usato solo con attributi di tipo numerico.

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())
      )
```

puoi usare di nuovo `classAttribute()` per trovare il nome attributo di classificazione, i metodi `sourceSymbol()` e `sourceColorRamp()`. Inoltre, è presente il metodo `mode()` che determina come gli intervalli siano creati: usando intervalli uguali, quantili o qualche altro metodo.

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 Working with Symbols

Per la rappresentazione di simboli, è presente la classe di base `QgsSymbolV2` con tre classi derivate:

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

Ogni simbolo consiste in uno o più vettori simbolo (classi derivate da `QgsSymbolLayerV2`). I vettori simbolo creano la visualizzazione effettiva, la classe simbolo stessa serve solo come contenitore per i vettori simbolo.

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())
```

Per trovare il colore del simbolo usa il metodo `color()` e `setColor()` per cambiarlo. Con i simboli singoli inoltre è possibile interrogare per la dimensione del simbolo e la rotazione con i metodi `size()` e `angle()`, per i simboli linea è presente il metodo `width()` che restituisce la larghezza della linea.

La dimensione e la larghezza sono in millimetri per impostazione predefinita, gli angoli sono in gradi.

Working with Symbol Layers

Come detto prima, i vettori simbolo (sottoclassi di `QgsSymbolLayerV2`) determinano l'aspetto delle geometrie. Ci sono sono diverse classi di vettori simbolo di base per un uso generale. È possibile implementare nuovi tipi di vettori simbolo e quindi personalizzare arbitrariamente il modo con cui le geometria saranno visualizzate. Il metodo `layerType()` identifica unicamente la classe di vettore simbolo — quelli di base e di default sono i tipi di vettori simbolo SimpleMarker, SimpleLine e SimpleFill.

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

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

Output

```
EllipseMarker
FontMarker
SimpleMarker
SvgMarker
VectorField
```

La classe `QgsSymbolLayerV2Registry` gestisce un database di tutti i tipi di vettore simbolo disponibili.

Per accedere ai dati di vettore simbolo, usa il suo metodo `properties()` che restituisce un dizionario chiave-valore di proprietà che determinano l'aspetto. Ogni tipo di vettore simbolo ha un set specifico di proprietà che usa. Inoltre, ci sono i metodi generali `color()`, `size()`, `angle()`, `width()` con i loro set corrispondenti. Naturalmente la dimensione e l'angolo sono disponibili solo per vettori simbolo singolo e lo spessore per vettori simbolo linea.

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)
```

Il metodo `layerType()` determina il nome del vettore simbolo e deve essere unico tra tutti i vettori simbolo. Le proprietà sono usate per la persistenza degli attributi. Il metodo `clone()` deve restituire una copia del vettore simbolo con tutti gli stessi esatti attributi. Infine ci sono i metodi di visualizzazione: `startRender è chiamato prima della visualizzazione della prima geoemtria, :func:'stopRender()` quando la visualizzazione è stata fatta. E il metodo `renderPoint()` che fa la visualizzazione. Le coordinate del punto(i) sono già trasformate nelle coordinate in uscita.

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

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

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

        self.layer = None

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

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

    def symbolLayer(self):
        return self.layer

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

This widget can be embedded into the symbol properties dialog. When the symbol layer type is selected in symbol properties dialog, it creates an instance of the symbol layer and an instance of the symbol layer widget. Then it calls `setSymbolLayer()` method to assign the symbol layer to the widget. In that method the widget should update the UI to reflect the attributes of the symbol layer. `symbolLayer()` function is used to retrieve the symbol layer again by the properties dialog to use it for the symbol.

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

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

We will have to create metadata for the symbol layer

```
class FooSymbolLayerMetadata(QgsSymbolLayerV2AbstractMetadata):

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

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

  def createSymbolLayerWidget(self):
    return FooSymbolLayerWidget()

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

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

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

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 Further Topics

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

CHAPTER 6

Gestione della Geometria

Ci si riferisce comunemente a punti, linee e poligoni che rappresentano una caratteristica spaziale come geometrie. In QGIS sono rappresentate tramite la classe `QgsGeometry`. Tutti i possibili tipi di geometria sono mostrati nella pagina di discussione JST.

Alcune volte una geometria é effettivamente una collezione di geometrie (parti singole) piú semplici. Se contiene un tipo di geometria semplice, la chiameremo punti multipli, string multi linea o poligoni multipli. Ad esempio, un Paese formato da piú isole puó essere rappresentato come un poligono multiplo.

Le coordinate delle geometrie possono essere in qualsiasi sistema di riferimento delle coordinate (CRS). Quando si estraggono delle caratteristiche da un vettore, le geometrie associate avranno le coordinate nel CRS del vettore.

6.1 Costruzione della Geometria

Esistono diverse opzioni per creare una geometria:

- dalle coordinate

```
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)]])
```

  Le coordinate vengono fornite utilizzando la classe `QgsPoint`.

  Una polilinea (linestring) é rappresentata da una lista di punti. Un poligono é rappresentato da una lista di anelli lineari (i.e. linee chiuse). Il primo anello é l'anello esterno (confine), gli altri anelli opzionali sono buchi nel poligono.

  Le geometrie a parti multiple vanno ad un livello successivo: punti multipli é una lista di punti, una stringa multi linea é una linea di linee ed un poligono multiplo é una lista di poligoni.

- da well-known text (WKT)

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

- da well-known binary (WKB)

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

6.2 Accedere alla Geometria

Come prima cosa si deve individuare il tipo di geometria, utilizzando il metodo `wkbType()` — che restituisce un valore dell'enumerazione `QGis.WkbType`

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

Come alternativa, é possibile utilizzare il metodo `type()` che restituisce uno dei valori dell'enumerazione `QGis.GeometryType`. Esiste inoltre la funzione di aiuto `isMultipart()` per capire se la geometria é multiparte o meno.

Per estrarre informazioni dalla geometria esistono delle funzioni di accesso per ogni tipo di vettore. Come usare le funzioni di accesso

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

Nota: le tuple (x, y) non sono vere tuple, ma sono oggetti `QgsPoint`, i valori sono accessibili tramite i metodi `x()` e `y()`.

Per le geometrie multiparte esistono funzioni di accesso simili: `asMultiPoint()`, `asMultiPolyline()`, `asMultiPolygon()`.

6.3 Predicati ed Operazioni delle Geometrie

QGIS usa la libreria GEOS per operazioni avanzate sulle geometrie come i predicati (`contains()`, `intersects()`, ...) e operazioni di set (`union()`, `difference()`, ...). Inoltre la libreria calcola le proprietá geometriche della geometria come l'area (nel caso di poligoni) o le lunghezze (per linee e poligoni)

Di seguito un piccolo esempio che combina l'iterazione sulle caratteristiche di un vettore e l'esecuzione di alcuni calcoli geometrici basati sulle loro geometrie.

```
# 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()
```

Aree e perimetri non considerano il CRS quando vengono calcolate utilizzando questi metodi della classe `QgsGeometry`. Per un calcolo piú potente di area e distanza si puó utilizzare la classe `QgsDistanceArea`. Se le proiezioni vengono spente, i calcoli saranno planari, altrimenti verranno eseguiti sull'ellissoide. Quando un ellissoide non viene specificato si utilizzano i parametri del WGS84 per i calcoli.

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

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

É possibile trovare molti esempi di algoritmi che sono inclusi in QGIS ed utilizzare questi metodi per analizzare e trasformare i dati vettoriali. Di seguito i link al codice di alcuni di questi.

É possibile trovare ulteriori informazioni alle seguenti fonti:

- Trasformazione di geometria: Algoritmo di riproiezione
- Distanza ed area utilizzando la classe `QgsDistanceArea`: Algoritmo matrice distanza
- Algoritmo da parti multiple a parte singola

CHAPTER 7

Supporto alle proiezioni

7.1 Coordinate reference systems

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

- specify CRS by its ID

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

  QGIS uses three different IDs for every reference system:

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

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

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

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

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

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

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

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

Accessing spatial reference system information

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

7.2 Projections

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

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

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

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

CHAPTER 8

Area di mappa

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

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

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

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

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

8.1 Embedding Map Canvas

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

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

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

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

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

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

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

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

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

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

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

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

8.2 Using Map Tools with Canvas

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

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

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

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

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

    self.setCentralWidget(self.canvas)

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

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

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

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

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

    self.pan()

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

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

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

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

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

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

8.3 Rubber Bands and Vertex Markers

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

To show a polyline

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

To show a polygon

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

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

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

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

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

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

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

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

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

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

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

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

8.4 Writing Custom Map Tools

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

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

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

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

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

  def canvasPressEvent(self, e):
      self.startPoint = self.toMapCoordinates(e.pos())
      self.endPoint = self.startPoint
```

```
        self.isEmittingPoint = True
        self.showRect(self.startPoint, self.endPoint)

    def canvasReleaseEvent(self, e):
        self.isEmittingPoint = False
        r = self.rectangle()
        if r is not None:
          print "Rectangle:", r.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 Writing Custom Map Canvas Items

TODO: how to create a map canvas item

```
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

Visualizzazione e Stampa di una Mappa

Esistono generalmente due approcci per visualizzare i dati come mappa: o in modo veloce utilizzando `QgsMapRenderer` oppure producendo un risultato piú raffinato componendo la mappa con la classe `QgsComposition`.

9.1 Visualizzazione Semplice

Visualizzare alcuni layer utilizzando `QgsMapRenderer` — create il dispositivo di destinazione (`QImage`, `QPainter` etc.), configurare l'insieme di layer, dimensione del risultato ed eseguire la visualizzazione

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

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

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

render = QgsMapRenderer()

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

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

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

# do the rendering
render.render(p)
```

```
p.end()

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

9.2 Visualizzare layer con diversi SR

Nel caso in cui si abbia piú di un layer con un diverso SR, il semplice esempio precedente probabilmente non funzionerá: per ottenere i valori corretti dai calcoli dell'estensione si dovrá impostare esplicitamente l'SR di destinazione ed abilitare la riproiezione OTF come nel prossimo esempio (dove viene riportata unicamente la parte di visualizzazione)

```
...
# 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 Risultato utilizzando il Compositore di Stampe

Il compositore di mappe é uno strumento molto utile nel caso in cui si voglia produrre un risultato piú sofisticato rispetto alla semplice visualizzazione mostrata sopra. Utilizzando il compositore é possibile creare composizioni di mappe complesse composte da viste, etichette, legenda, tabelle ed altri elementi che sono solitamente presenti sulle mappe cartacee. La composizione puó essere esportata in PDF, immagini raster o stampata direttamente tramite una stampante.

Il compositore è costituito da un insieme di classi. Esse appartengono tutte alla libreria relativa al nucleo. L'applicativo QGIS ha una comoda GUI per la disposizione degli elementi, anche se non é disponibile nella libreria GUI. Se avete dimestichezza con Qt Graphics View framework <http://doc.qt.nokia.com/stable/graphicsview.html>'_, vi invitiamo a controllare la documentazione, dato che il compositore é basata su di essa. Controllate anche la 'documentazione Python dell'implementazione di <http://pyqt.sourceforge.net/Docs/PyQt4/qgraphicsview.html>'_.

La classe principale del composer é `QgsComposition` che deriva da `QGraphicsScene`. Creiamone una

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

Si noti che la composizione prende un'istanza di `QgsMapRenderer`. Il codice viene eseguito all'interno dell'applicazione QGIS e quindi utilizza la visualizzazione della mappa. La composizione utilizza diversi parametri della visualizzazione della mappa, soprattutto l'insieme predefinito di mappe e l'estensione corrente. Quando si utilizza il compositore in un'applicazione standalone, é possibile creare la propria istanza di mappa nello stesso modo mostrato nella sezione di cui sopra e passarla alla composizione.

É possibile aggiungere vari elementi (mappa. etichetta, ...) alla composizione — questi elementi devono essere discendenti della classe `QgsComposerItem`. Gli elementi attualmente supportati sono:

- mappa — questo elemento dice alle librerie dove posizionare la mappa stessa. Qui creiamo una mappa e la stiriamo sull'intera pagina

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

- etichetta — permetta la visualizzazione di etichette. É possibile modificarne il carattere, colore, allineamento e margine

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

- legenda

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

- barra di scala

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

- freccia
- immagine
- forma
- tabella

Come parametro predefinito il compositore appena creato ha posizione zero (angolo in alto a sinistra della pagina) e dimensione zero. La posizione e la dimensione sono sempre misurate in millimetri

```
# 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)
```

Una cornice viene disegnata attorno ad ogni elemento da impostazione predefinita. Come rimuovere la cornice

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

Oltre a creare gli elementi del compositore manualmente, QGIS fornisce il supporto per i modelli del compositore, che sono essenzialmente delle composizioni aventi tutti gli elementi salvati in un file .qpt (con sintassi XML). Purtroppo questa funzionalitá non é ancora disponibile nelle API.

Una volta che la composizione é pronta (gli elementi del compositore sono stati creati ed aggiunti alla composizione), possiamo procedere alla creazione di un risultato raster e/o vettoriale.

Le impostazioni predefinite del risultato per la composizione sono il formato di pagina A4 e risoluzione 300 DPI. É possibile cambiarle se necessario. La dimensione della pagina é specificata in millimetri

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

9.3.1 Esportare come immagine raster

Il seguente frammento di codice mostra come visualizzare una composizione come immagine raster

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

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

# render the composition
imagePainter = QPainter(image)
sourceArea = QRectF(0, 0, c.paperWidth(), c.paperHeight())
targetArea = QRectF(0, 0, width, height)
c.render(imagePainter, targetArea, sourceArea)
imagePainter.end()

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

9.3.2 Esportare come PDF

Il seguente frammento di codice visualizza la composizione come file 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

Espressioni, Filtraggio e Calcolo di Valori

QGIS offre supporto per l'analisi di espressioni SQL. Solo un piccolo sottoinsieme della sintassi SQL é supportato. Le espressioni possono essere valutate sia come predicati booleani (che restituiscono Vero o Falso) o come funzioni (che restituiscono un valore scalare). Vedi *vector_expressions* nel Manuale dell'Utente per una lista completa delle funzioni presenti.

Sono supportati tre tipi base:

- numero – sia numeri interi che decimali, e.g. `123`, `3.14`
- stringa – devono essere racchiuse tra apici singoli: `'hello world'`
- riferimento a colonna – durante la valutazione, il riferimento é sostituito con il valore del campo. I nomi non sono racchiusi tra apici.

Sono disponibili le seguenti operazioni:

- operatori aritmetici: `+`, `-`, `*`, `/`, `^`
- parentesi: per forzare la precedenza tra operatori: `(1 + 1) * 3`
- somma e sottrazione unari: `-12`, `+5`
- funzioni matematiche: `sqrt`, `sin`, `cos`, `tan`, `asin`, `acos`, `atan`
- funzioni di conversione: `to_int`, `to_real`, `to_string`, `to_date`
- funzioni sulla geometria: `$area`, `$length`
- funzioni di manipolazione della geometria: `$x`, `$y`, `$geometry`, `num_geometries`, `centroid`

Sono supportati i seguenti predicati:

- comparazione: `=`, `!=`, `>`, `>=`, `<`, `<=`
- pattern matching: `LIKE` (usando `%` e `_`), `~` (espressioni regolari)
- predicati logici: `AND`, `OR`, `NOT`
- controllo di valori NULL: `IS NULL`, `IS NOT NULL`

Esempi di predicati:

- `1 + 2 = 3`
- `sin(angolo) > 0`

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

Esempi di espressioni scalari:

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

10.1 Analisi di Espressioni

```
>>> 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 Valutazione di Espressioni

10.2.1 Espressioni Base

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

10.2.2 Espressioni con geometrie

L'esempio seguente valuterá l'espressione data su una geometria. "Column" é il nome del campo del layer.

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

Si puó anche utilizzare `QgsExpression.prepare()` per controllare piú di una geometria. L'utilizzo di `QgsExpression.prepare()` aumenterá la velocitá della valutazione.

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

10.2.3 Gestione degli errori

```
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 Esempi

L'esempio seguente puó essere usato per filtrare un layer e restituire qualsiasi geometria che soddisfi il predicato.

```
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

Leggere E Memorizzare Impostazioni

Molte volte è utile per un plugin salvare alcune variabili in modo tale che l'utente non debba inserirle o selezionarle di nuovo la volta successiva che il plugin è eseguito.

Questa variabili possono essere salvate e recuperate con l'aiuto delle API di Qt e QGIS. Per ogni variabile, dovresti scegliere una chiave che sarà usata per accedere alla variabile — per il colore preferito dell'utente potresti usare la chiave "favourite_color" o una qualunque altra stringa significativa. È raccomandabile dare una qualche struttura alla denominazione delle chiavi.

Possiamo fare una differenza tra diversi tipi di impostazioni:

- **impostazioni globali** — sono legate all'utente per una specifica macchina. QGIS stesso memorizza un sacco di impostazioni globali, per esempio, la dimensione della finestra principale o la tolleranza predefinita di aggancio. Questa funzionalità è fornita direttamente dall'ambiente Qt per mezzo della classe `QSettings`. Per impostazione predefinita, questa classe memorizza le impostazioni nella maniera "nativa" del sistema di memorizzare le impostazioni, che è — il registro (su Windows), il file .plist (su Mac OS X) o il file .ini (su Unix). La documentazione di QSettings è completa, per cui forniremo solo un semplice esempio

  ```
  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)
  ```

  Il secondo parametro del metodo `value()` è opzionale e specifica il valore predefinito se non c'è nessun altro valore precedentemente impostato per il nome di impostazione passato.

- **impostazioni di progetto** — variano da progetto a progetto e inoltre sono collegate a un file di progetto. Il colore di sfondo della mappa o il sistema di riferimento delle coordinate (CRS) di destinazione ne sono un esempio — lo sfondo bianco e il WGS84 possono essere adatti per un progetto, mentre lo sfondo giallo e la proiezione UTM lo sono per un altro. Segue un esempio

  ```
  proj = QgsProject.instance()

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

  # read values
  ```

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

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

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

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

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

CHAPTER 12

Comunicare con l'utente

Questa sezione mostra alcuni metodi ed elementi che dovrebbero essere usati per comunicare con l'utente, in modo da mantenere la consistenza nell'interfaccia utente.

12.1 Mostrare i messaggi. La classe class:*QgsMessageBar*.

Utilizzare il box dei messaggi potrebbe essere una cattiva idea dal punto di vista dell'esperienza utente. Solitamente, per mostrare un messaggio di informazione o di errore/avvertimento, la barra dei messaggi di QGIS é l'opzione migliore.

Utilizzando il riferimento all'oggetto interfaccia di QGIS, é possibile mostrare un messaggio nell barra dei messaggi utilizzando il seguente codice

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

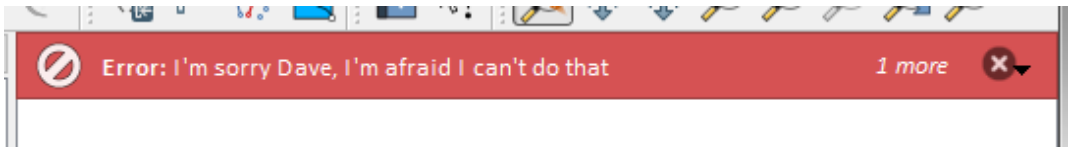

Figure 12.1: Barra dei messaggi di QGIS

É possibile impostare una durata per mostrarlo per un tempo limitato

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

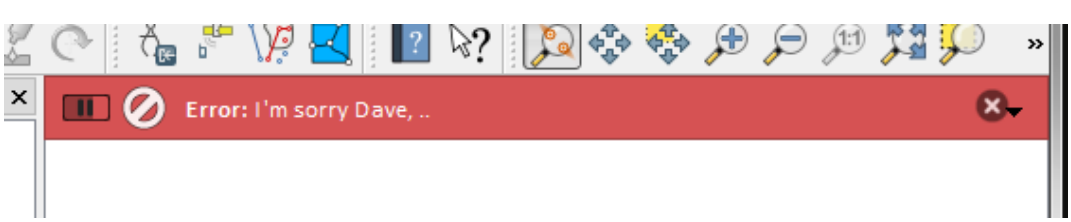

Figure 12.2: Barra dei messaggi di QGIS con timer

L'esempio precedente mostra una barra d'errore, ma il parametro `livello` puó essere usato per creare messaggi di avvertimento o di informazione, utilizzando rispettivamente le costanti `QgsMessageBar.WARNING` e

QgsMessageBar.INFO'.

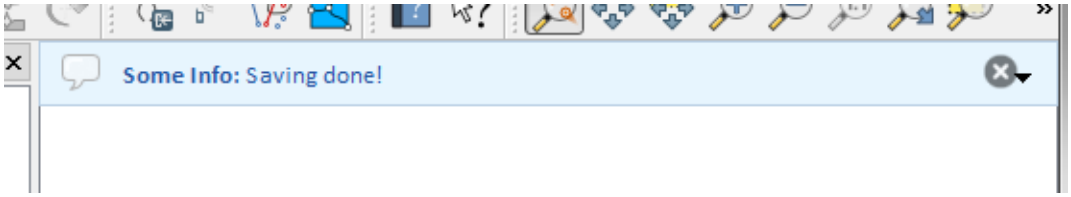

Figure 12.3: Barra dei messaggi di QGIS (informazioni)

I widget possono essere aggiunti alla barra dei messaggi, ad esempio il pulsante per mostrare piú informazioni

```
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)
```

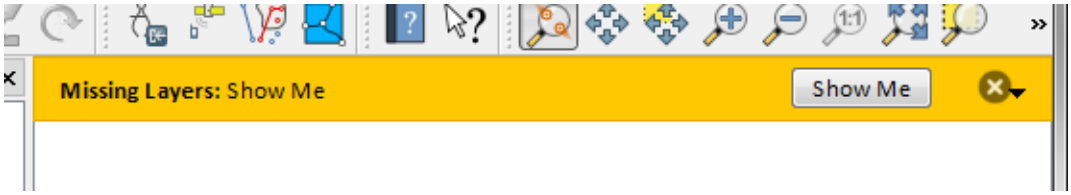

Figure 12.4: Barra dei messaggi di QGIS con un pulsante

É possibile usare una barra dei messaggi nella propria finestra di dialogo senza dover mostrare una finestra di messaggi, o nel caso in cui non abbia senso mostrarla nella finestra principale di 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 Mostrare l'avanzamento

Le barre di avanzamento si possono mettere anche nella barra dei messaggi di QGIS, dato che, come abbiamo visto, accetta i widget. Di seguito un esempio che potrete provare nella console.

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

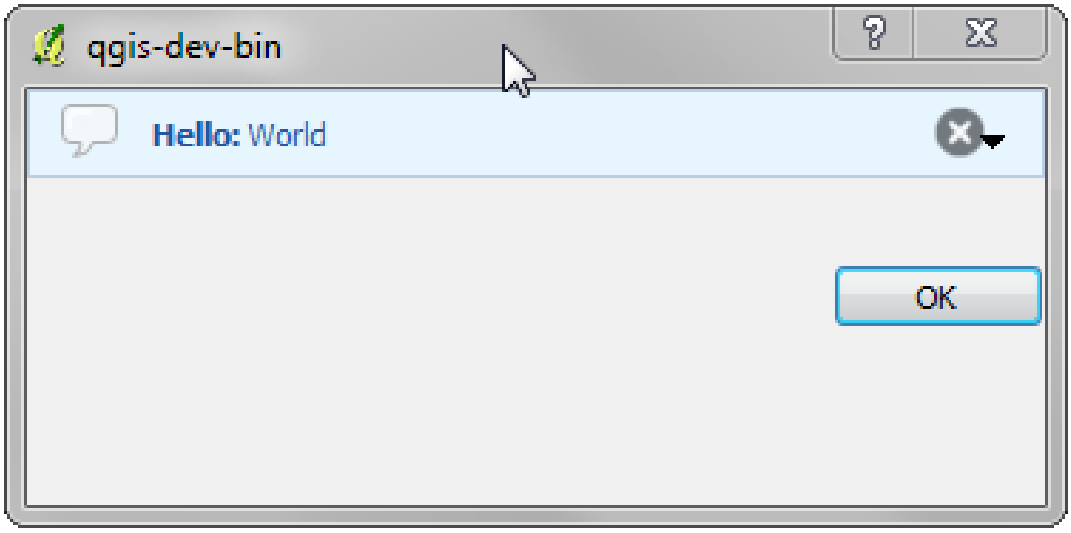

Figure 12.5: Barra dei messaggi di QGIS in una finestra di dialogo personalizzata

```
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()
```

Inoltre é possibile utilizzare la barra di stato integrata per mostrare un progresso, come nel prossimo esempio

```
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 Logging

É possibile utilizzare il sistema di logging di QGIS per annotare tutte le informazioni che riguardano l'esecuzione del codice che si vogliono salvare.

```
# 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

Sviluppare Plugin Python

É possibile creare plugin nel linguaggio di programmazione Python. A differenza dei classici plugin scritti in C++ questi dovrebbero essere piú facili da scrivere, capire, mantenere e distribuire grazie alla natura dinamica del linguaggio Python.

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

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

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

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

Passi:

1. *Idea*: Avere un'idea su cosa si vuole fare con il nuovo plugin QGIS. Perché lo fai? Esiste giá un altro plugin per questo problema?
2. *Create files*: Create the files described next. A starting point (`__init__.py`). Fill in the *Metadati del plugin* (`metadata.txt`) A main python plugin body (`mainplugin.py`). A form in QT-Designer (`form.ui`), with its `resources.qrc`.
3. *Write code*: Write the code inside the `mainplugin.py`
4. *Test*: Chiudi e ri-apri QGIS e importa nuovamente il tuo plugin. Controlla se tutto va bene.

5. *Publish*: Publish your plugin in QGIS repository or make your own repository as an "arsenal" of personal "GIS weapons".

13.1 Scrivere un plugin

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

13.1.1 File del plugin

Here's the directory structure of our example plugin

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

Qual é il significato dei files:

- `__init__.py` = The starting point of the plugin. It has to have the `classFactory()` method and may have any other initialisation code.
- `mainPlugin.py` = The main working code of the plugin. Contains all the information about the actions of the plugin and the main code.
- `resources.qrc` = The .xml document created by Qt Designer. Contains relative paths to resources of the forms.
- `resources.py` = La traduzione in Python del file .qrc descritto sopra.
- `form.ui` = The GUI created by Qt Designer.
- `form.py` = La traduzione in Python del file form.ui descritto sopra.
- `metadata.txt` = Required for QGIS >= 1.8.0. Contains general info, version, name and some other metadata used by plugins website and plugin infrastructure. Since QGIS 2.0 the metadata from `__init__.py` are not accepted anymore and the `metadata.txt` is required.

Here is an online automated way of creating the basic files (skeleton) of a typical QGIS Python plugin.

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

Avvertimento: If you plan to upload the plugin to the *Repository di plugin python ufficiale* you must check that your plugin follows some additional rules, required for plugin *Validazione*

13.2 Contenuto del plugin

Qui puoi trovare informazioni ed esempi su cosa aggiungere in ognuno dei file nella struttura descritta sopra.

13.2.1 Metadati del plugin

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

Importante: Tutti i metadati devono codificati in UTF-8.

Nome del metadato	Obbligatorio	Note
nome	Vero	una string contenente il nome del plugin
qgisMinimumVersion	Vero	notazione puntata della versione minima di QGIS
qgisMaximumVersion	Falso	notazione puntata della massima versione di QGIS
descrizione	Vero	short text which describes the plugin, no HTML allowed
about	Vero	longer text which describes the plugin in details, no HTML allowed
versione	Vero	stringa con la notazione puntata della versione
autore	Vero	nome dell'autore
email	Vero	email of the author, not shown in the QGIS plugin manager or in the website unless by a registered logged in user, so only visible to other plugin authors and plugin website administrators
elenco cambiamenti	Falso	stringa, puó essere su piú righe, HTML non consentito
sperimentale	Falso	flag booleano, 'True ' o 'False '
dismesso	Falso	il flag booleano, *True* or *False*, si applica all'intero plugin e non solo alla versione caricata
etichette	Falso	comma separated list, spaces are allowed inside individual tags
homepage	Falso	una URL valida che punta alla homepage del plugin
repository	Vero	una URL valida per il repository del codice sorgente
tracker	Falso	una URL valida per i tickets ed i bug report
icona	Falso	a file name or a relative path (relative to the base folder of the plugin's compressed package) of a web friendly image (PNG, JPEG)
categoria	Falso	uno tra *Raster*, *Vector*, *Database* e *Web*

By default, plugins are placed in the *Plugins* menu (we will see in the next section how to add a menu entry for your plugin) but they can also be placed the into *Raster*, *Vector*, *Database* and *Web* menus.

A corresponding "category" metadata entry exists to specify that, so the plugin can be classified accordingly. This metadata entry is used as tip for users and tells them where (in which menu) the plugin can be found. Allowed values for "category" are: Vector, Raster, Database or Web. For example, if your plugin will be available from *Raster* menu, add this to `metadata.txt`

```
category=Raster
```

Nota: Se *qgisMaximumVersion* é vuoto, viene automaticamente impostato alla versione maggiore piú *.99* quando viene caricato in *Repository di plugin python ufficiale*.

An example for this metadata.txt

```
; the next section is mandatory

[general]
name=HelloWorld
email=me@example.com
author=Just Me
```

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

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

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

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

; experimental flag (applies to the single version)
experimental=True

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

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

13.2.2 __init__.py

This file is required by Python's import system. Also, QGIS requires that this file contains a `classFactory()` function, which is called when the plugin gets loaded to QGIS. It receives reference to instance of `QgisInterface` and must return instance of your plugin's class from the `mainplugin.py` — in our case it's called `TestPlugin` (see below). This is how `__init__.py` should look like

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

## any other initialisation needed
```

13.2.3 mainPlugin.py

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

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

```
# initialize Qt resources from file resources.py
import resources

class TestPlugin:

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

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

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

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

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

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

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

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

The only plugin functions that must exist in the main plugin source file (e.g. `mainPlugin.py`) are:

- `__init__` –> which gives access to QGIS interface
- `initGui()` –> called when the plugin is loaded
- `unload()` –> called when the plugin is unloaded

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

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

Hanno tutti la stessa sintassi del metodo `addPluginToMenu()`.

Aggiungere il menu del tuo plugin ad uno di questi metodi predefiniti é raccomandato per mantenere la consistenza riguardo all'organizzazione dei plugin. É comunque possibile aggiungere un gruppo personalizzato alla barra dei menu, come dimostrato nel prossimo esempio:

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

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

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

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

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

13.2.4 Resource File

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

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

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

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

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

And that's all... nothing complicated :)

If you've done everything correctly you should be able to find and load your plugin in the plugin manager and see a message in console when toolbar icon or appropriate menu item is selected.

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

13.3 Documentazione

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

The `showPluginHelp()` function looks for help files in the same directory as the calling module. It will look for, in turn, `index-ll_cc.html`, `index-ll.html`, `index-en.html`, `index-en_us.html` and

`index.html`, displaying whichever it finds first. Here `ll_cc` is the QGIS locale. This allows multiple translations of the documentation to be included with the plugin.

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

13.4 Translation

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

13.4.1 Software requirements

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

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

13.4.2 Files and directory

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

All the translation files have to be within this directory.

.pro file

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

In this `.pro` file you have to specify all the files and forms you want to translate. This file is used to set up the localization files and variables. An example of the pro file is:

```
FORMS = ../ui/*

SOURCES = ../your_plugin.py

TRANSLATIONS = your_plugin_it.ts
```

In this particular case all your UIs are placed in the `../ui` folder and you want to translate all of them.

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

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

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

.ts file

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

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

```
lupdate your_plugin.pro
```

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

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

.qm file

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

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

```
lrelease your_plugin.ts
```

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

13.4.3 Load the plugin

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

You should see your plugin in the correct language.

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

CHAPTER 14

IDE settings for writing and debugging plugins

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

14.1 A note on configuring your IDE on Windows

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

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

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

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

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

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

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

To create that batch file, follow these steps:

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

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

14.2 Debugging using Eclipse and PyDev

14.2.1 Installation

To use Eclipse, make sure you have installed the following

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

14.2.2 Preparing QGIS

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

- Go to *Plugins* → *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.
- Search for *Plugin reloader* and install it as well. This will let you reload a plugin instead of having to close and restart QGIS to have the plugin reloaded.

14.2.3 Setting up Eclipse

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

Now right-click your new project and choose *New* → *Folder*.

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

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

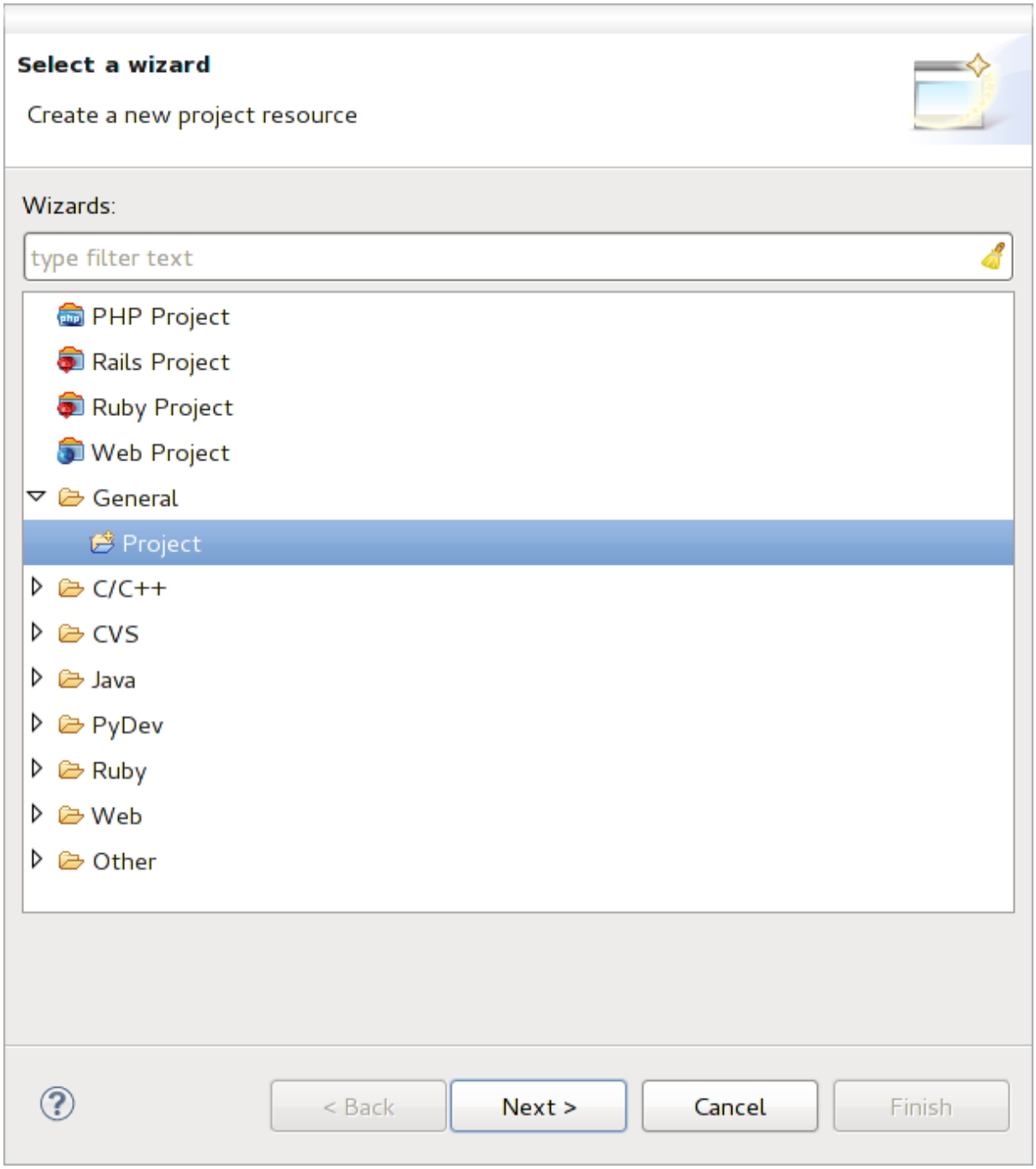

Figure 14.1: Eclipse project

14.2.4 Configuring the debugger

To get the debugger working, switch to the Debug perspective in Eclipse (*Window → Open Perspective → Other → Debug*).

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

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

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

```
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 14.2: Breakpoint

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

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.

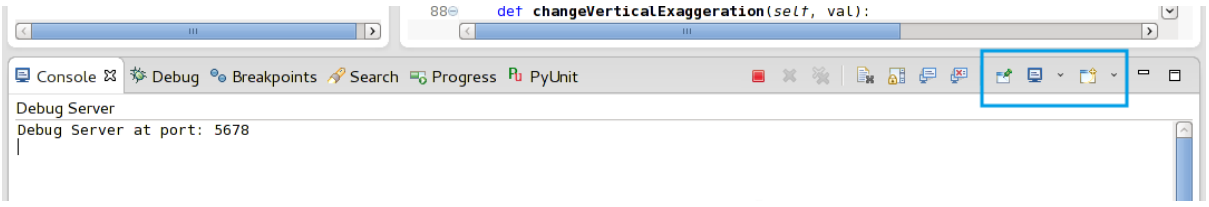

Figure 14.3: PyDev Debug Console

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

A little bit annoying is, that 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.

14.2.5 Making eclipse understand the API

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

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

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

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

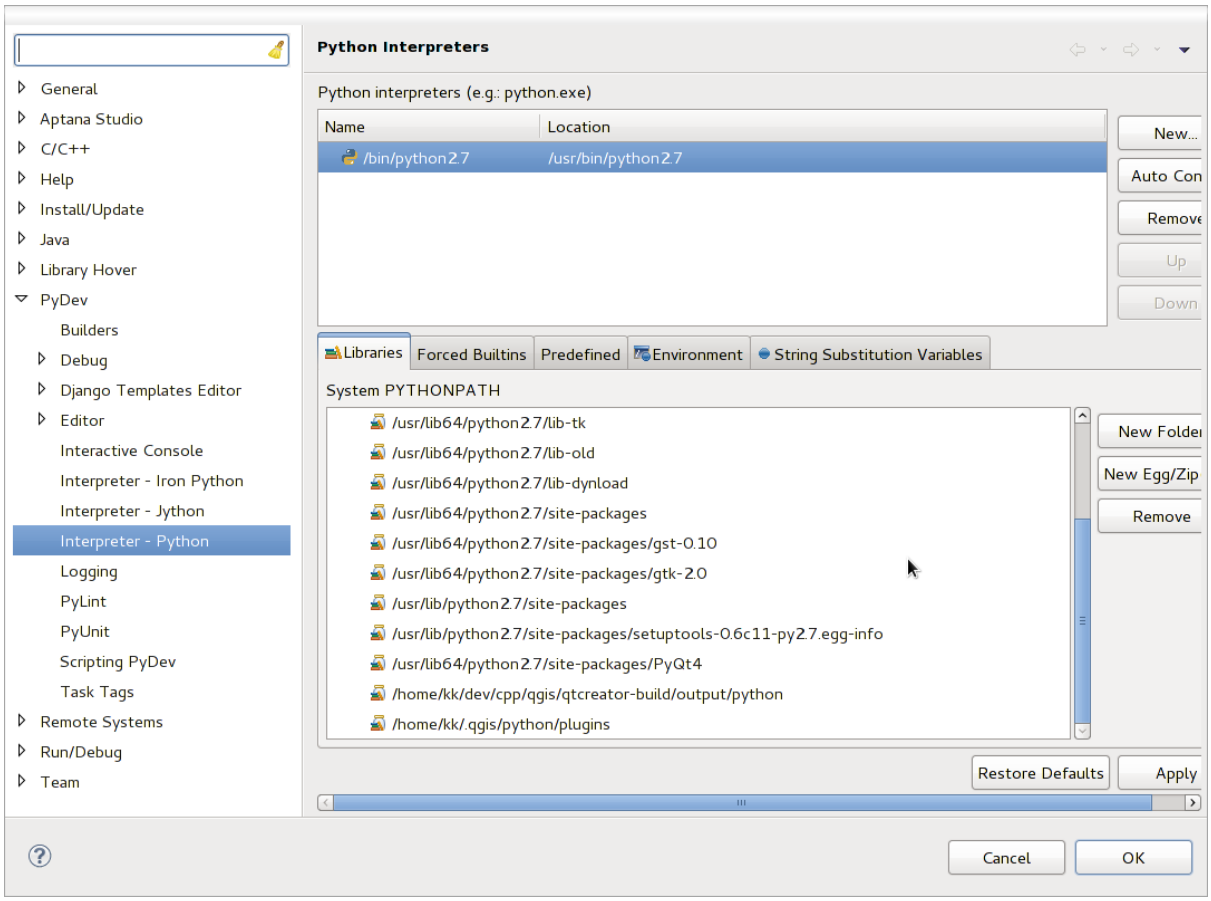

Figure 14.4: PyDev Debug Console

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.

Click *OK* and you're done.

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.

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

14.3 Debugging using PDB

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

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

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

Then run QGIS from the command line.

On Linux do:

```
$ ./Qgis
```

On Mac OS X do:

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

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

TODO: Add testing information

CHAPTER 15

Using Plugin Layers

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

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

15.1 Subclassing QgsPluginLayer

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

```
class WatermarkPluginLayer(QgsPluginLayer):

  LAYER_TYPE="watermark"

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

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

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

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

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

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

```
class WatermarkPluginLayerType(QgsPluginLayerType):

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

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

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

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

CHAPTER 16

Compatibilitá con versioni precedenti di QGIS

16.1 Menu dei plugin

Se posizionate le voci di menú del vostro plugin in uno dei nuovi menu (*Raster*, *Vector*, *Database* o *Web*), dovreste modificare il codice delle funzioni `initGui()` e `unload()`. Dato che questi menu sono disponibili solo in QGIS 2.0 o superiori, il primo passo é quello di controllare che la versione di QGIS in esecuzione abbia tutte le funzioni necessarie. Se i nuovi menu sono disponibili, inseriremo il nostro plugin in questo menu, altrimenti utilizzeremo il vecchio menu *Plugins*. Di seguito un esempio per il menu *Raster*

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

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

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

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

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

CHAPTER 17

Rilascio del tuo plugin

Una volta che il tuo plugin è pronto e ritieni che esso possa essere utile per altre persone, non esitare a caricarlo su *Repository di plugin python ufficiale*. In questa pagina puoi trovare anche linee guida su come preparare il plugin per farlo funzionare bene con l'installatore di plugin. Oppure, nel caso in cui tu preferisca impostare un tuo repository di plugin personale, crea un semplice file XML che elencherà i plugin e i loro metadati, per esempi vedi altri repository di plugin.

Fai particolare attenzione ai seguenti suggerimenti:

17.1 Metadati e nomi

- evita di usare un nome molto simile a quello di plugin esistenti
- se il tuo plugin offre funzionalità simili a quelle di un plugin esistente, evidenzia le differenze nel campo A proposito, così l'utente saprà quale dei due usare senza la necessità di installarlo e testarlo
- evita di ripetere "plugin" nel nome del plugin stesso
- usa il campo descrizione nei metadati per una descrizione di una riga, il campo A proposito per informazioni più dettagliate
- includi un repository del codice, un bug tracker, e una pagina iniziale; ciò migliorerà notevolmente la possibilità di collaborazione, e può essere fatto molto facilmente con una delle infrastrutture web disponibili (GitHub, GitLab, Bitbucket, etc.)
- scegli i tag con cura: evita quelli non esplicativi (ad es. vettore) e preferisci quelli già utilizzati da altri (vedi il sito web dei plugin)
- aggiungi un'icona appropriata, non lasciare quella predefinita; visita l'interfaccia di QGIS per farti un'idea sullo stile da usare

17.2 Codice e guida

- non includere i file generati (ui_*.py, resources_rc.py, file di guida generati...) e materiale inutile (ad es. gitignore) nel repository

- aggiungi il plugin al menu appropriato (Vettore, Raster, Web, Database)
- quando necessario (plugin che eseguono analisi). considera la possibilità di aggiungere il plugin come sottoplugin dell'ambiente di Processing: ciò consentirà agli utenti di eseguirlo in serie, di integrarlo in flussi di lavoro più complessi ed eviterà l'onere di progettare un'interfaccia
- includi almeno la documentazione minima e, se utili per il testing e la compresione, dati campione.

17.3 Repository di plugin python ufficiale

Puoi trovare il repositori di plugin python *ufficiale* su http://plugins.qgis.org/.

Per usare il repository ufficiale devi ottenere un ID OSGEO dal **'portale web OSGEO<http://www.osgeo.org/osgeo_userid/>'_**.

Una volta che avrai caricato il tuo plugin, esso sarà approvato da un membro del personale e ciò ti verrà notificato.

TODO: Inserisci un collegamento al documento di amministrazione

17.3.1 Privilegi

Queste regole sono state implementate nel repository di plugin ufficiale:

- ogni utente registrato può aggiungere un nuovo plugin
- gli utenti facenti parte del *personale* possono approvare o rifiutare tutte le versioni del plugin
- gli utenti che hanno il privilegio speciale *plugins.can_approve* ricevono l'approvazione automatica delle versioni da loro caricate
- gli utenti che hanno il privilegio speciale *plugins.can_approve* possono approvare versioni caricate da altri finché si trovano nella lista dei *proprietari* del plugin
- uno specifico plugin può essere cancellato e modificato soltanto dai membri del *personale* e dai *proprietari* del plugin
- se un utente senza il privilegio *plugins.can_approve* carica una nuova versione, la versione del plugin è automaticamente rifiutata.

17.3.2 Gestione dell'attendibilità

I membri del personale possono accordare l'*affidabilità* ai creatori del plugin selezionato impostando il privilegio *plugins.can_approve* attraverso l'applicazione front-end.

La vista dei dettagli del plugin offre collegamenti diretti per accordare l'affidabilità al creatore del plugin o ai *proprietari* del plugin.

17.3.3 Validazione

I metadati del plugin sono importati automaticamente e validati dal pacchetto compresso quando il plugin è caricato.

Ecco alcune regole di validazione che dovresti conoscere quando desideri caricare un plugin nel repository ufficiale:

1. il nome della cartella principale contenente il tuo plugin deve contenere solo caratteri ASCII (A-Z e a-z), numeri e i caratteri trattino basso (_) e meno (-), inoltre non può iniziare con un numero
2. È richiesto il file `metadata.txt`
3. tutti i metadati richiesti elencati nella *tabella metadati* devono essere presenti

4. il campo metadati *version* deve essere unico

17.3.4 Struttura del plugin

Seguendo le regole di convalida, il pacchetto compresso (.zip) del tuo plugin deve avere una struttura specifica per validarsi come un plugin efficiente. Siccome il plugin sarà scompattato all'interno della cartella dei plugin dell'utente, esso deve avere la sua cartella propria dentro il file .zip per non interferire con gli altri plugin. I file obbligatori sono: `metadata.txt` e `__init__.py`. Ma sarebbe apprezzato avere un file `LEGGIMI` e certamente un'icona per rappresentare il plugin (`resources.qrc`). Di seguito un esempio di come un file plugin.zip dovrebbe apparire.

```
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 18

Frammenti di codice

Questa sezione contiene frammenti di codice per facilitare lo sviluppo dei plugin.

18.1 Come invocare un metodo tramite scorciatoia da tastiera

Nel plug-in aggiungere a `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)
```

Aggiungere a `unload()`

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

Il metodo che viene invocato quando si preme F7

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

18.2 Come impostare/rimuovere i layers

A partire da QGIS 2.4 é disponibile una nuova API dell'albero dei layer che consente un accesso diretto all'albero dei layer direttamente dalla legenda. Questo é un esempio di come attivare/rimuovere la visibilitá del layer attivo.

```
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)
```

18.3 Come accedere alla tabella degli attributi di una caratteristica selezionata

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

Il metodo richiede un parametro (il nuovo valore per il campo attributo delle caratteristiche selezionate()) e puó essere invocato da

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

CHAPTER 19

Scrivere un plugin di Processing

In funzione del tipo di plugin che andrai a sviluppare, la migliore opzione sarebbe quella di aggiungerlo come algoritmo di Processing (o un set di essi). Ciò consentirebbe una migliore integrazione all'interno di QGIS, funzionalità aggiuntive (poiché può essere eseguito nei moduli di Processing, come il modellatore o l'interfaccia di processing in serie), e tempistiche di sviluppo più rapide (siccome Processing farà gran parte del lavoro).

Questo documento descrive come creare un nuovo plugin che aggiunge le sue funzionalità come algoritmo di Processing.

Esistono due modalità principali per fare ciò:

- Creare un plugin che aggiunge una sorgente per l'algoritmo: Questa opzione è più complessa, ma fornisce maggiore flessibilità
- Creare un plugin che contiene un insieme di script di processing: La soluzione più semplice, hai solo bisogno di un insieme di file di script di Processing.

19.1 Creare un plugin che aggiunge una sorgente per l'algoritmo

Per creare una sorgente per l'algoritmo, segui questi passaggi:

- Installa il plugin Plugin Builder
- Crea un nuovo plugin usando il Plugin Builder. Quando il Plugin Builder ti chiederà il modello da usare, seleziona "Sorgente di Processing".
- Il plugin creato contiene una sorgente con un singolo algoritmo. Il file della sorgente e dell'algoritmo sono entrambi commentati e contengono informazioni su come modificare la sorgente e gli algoritmi aggiuntivi. Fai riferimento ad essi per maggiori informazioni.

19.2 Creare un plugin che contiene un insieme di script di processing

Per creare un insieme di script di processing, segui questi passaggi:

- Crea i tuoi script come descritto nel PyQGIS cookbook. Tutti gli script che vuoi aggiungere dovrebbero essere disponibili negli strumenti di Processing.
- Nel gruppo *Script/Strumenti* negli strumenti di Processing, fai doppio click sull'elemento *Crea plugin da collezione di script*. Vedrai una finestra in cui dovrebbe essere possibile selezionare gli script da aggiungere

al plugin (da un insieme di script disponibili negli strumenti), e alcune informazioni aggiuntive necessarie per i metadati del plugin.

- Clicca OK e il plugin verrà creato.
- Puoi aggiungere ulteriori script al plugin aggiungendo file di script python alla cartella *script* nella cartella del plugin risultante.

CHAPTER 20

Libreria per l'analisi di reti

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

- creates mathematical graph from geographical data (polyline vector layers)
- implements basic methods from graph theory (currently only Dijkstra's algorithm)

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

20.1 General information

Briefly, a typical use case can be described as:

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

20.2 Building a graph

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

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

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

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

Converting from a vector layer to the graph is done using the Builder programming pattern. A graph is constructed using a so-called Director. There is only one Director for now: QgsLineVectorLayerDirector. The director sets the basic settings that will be used to construct a graph from a line vector layer, used by the builder to create the

graph. Currently, as in the case with the director, only one builder exists: QgsGraphBuilder, that creates QgsGraph objects. You may want to implement your own builders that will build a graphs compatible with such libraries as BGL or NetworkX.

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

It's time to dive into the process.

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

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

Then some examples for creating a director

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

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

To construct a director we should pass a vector layer, that will be used as the source for the graph structure and information about allowed movement on each road segment (one-way or bidirectional movement, direct or reverse direction). The call looks like this

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

And here is full list of what these parameters mean:

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

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

```
properter = QgsDistanceArcProperter()
```

And tell the director about this strategy

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

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

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

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

Also we can define several points, which will be used in the analysis. For example

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

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

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

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

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

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

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

20.3 Graph analysis

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

graph = builder.graph()

idStart = graph.findVertex(pStart)

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

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

Same thing but using `dijkstra()` method

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

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

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

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

graph = builder.graph()
```

```
idStart = graph.findVertex(pStart)

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

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

20.3.1 Finding shortest paths

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

if tree[idStop] == -1:
  print "Path not found"
else:
  p = []
  curPos = idStop
  while curPos != idStart:
    p.append(graph.vertex(graph.arc(tree[curPos]).inVertex()).point())
    curPos = graph.arc(tree[curPos]).outVertex();

  p.append(tStart)

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

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

20.3.2 Areas of availability

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

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

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

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

Here is an example

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

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

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

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

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

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

idStart = graph.findVertex(tStart)

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

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

```
      upperBound.append(i)
  i = i + 1

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

CHAPTER 21

QGIS Server Python Plugins

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.

21.1 Server Filter Plugins architecture

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

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

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

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

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

21.1.1 requestReady

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

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

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

21.1.2 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.

21.1.3 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).

21.2 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.

21.3 Writing a server plugin

A server plugins is just a standard QGIS Python plugin as described in *Sviluppare Plugin 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

21.3.1 Plugin files

Here's the directory structure of our example server plugin

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

21.3.2 __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)
```

21.3.3 HelloServer.py

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

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

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

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

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

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

class HelloFilter(QgsServerFilter):

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

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

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

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

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

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

The following examples cover some common use cases:

21.3.4 Modifying the input

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

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

class ParamsFilter(QgsServerFilter):
```

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

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

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

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

```
src/core/qgsmessagelog.cpp: 45: (logMessage) [0ms] 2014-12-12T12:39:29 plugin[0] HelloServerServe
src/core/qgsmessagelog.cpp: 45: (logMessage) [1ms] 2014-12-12T12:39:29 Server[0] Server plugin Hel
src/core/qgsmessagelog.cpp: 45: (logMessage) [0ms] 2014-12-12T12:39:29 Server[0] Server python plu
src/mapserver/qgsgetrequesthandler.cpp: 35: (parseInput) [0ms] query string is: SERVICE=HELLO&requ
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 request
src/core/qgsmessagelog.cpp: 45: (logMessage) [0ms] 2014-12-12T12:39:29 plugin[0] HelloFilter.reque
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, settin
src/core/qgsmessagelog.cpp: 45: (logMessage) [0ms] 2014-12-12T12:39:29 plugin[0] HelloFilter.respo
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] RemoteConsoleFilt
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.sendR
```

On line 13 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).

21.3.5 Modifying or replacing the output

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

```
import os

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


class WatermarkFilter(QgsServerFilter):

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

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

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

21.4 Access control plugin

21.4.1 Plugin files

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

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

21.4.2 __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)
```

21.4.3 AccessControl.py

```
class AccessControl(QgsAccessControlFilter):

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

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

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

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

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

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

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

This example gives a full access for everybody.

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

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

21.4.4 layerFilterExpression

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

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

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

21.4.5 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".

21.4.6 layerPermissions

Limit the access to the layer.

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

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

Example:

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

To limit everything on read only access.

21.4.7 authorizedLayerAttributes

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

The argument attribute return the current set of visible attributes.

Example:

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

To hide the 'role' attribute.

21.4.8 allowToEdit

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

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

Example:

```
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.

21.4.9 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.