PyQGIS developer cookbook

Version 2.18

QGIS Project

08 April 2019

Contents

CHAPTER 1

Introduction

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

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

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

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

Python bindings are also available for QGIS Server:

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

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

A good resource when dealing with plugins is to download some plugins from plugin repository and examine their code. Also, the `python/plugins/` folder in your QGIS installation contains some plugin that you can use to learn how to develop such plugin and how to perform some of the most common tasks.

1.1 Run Python code when QGIS starts

Il y a deux façons distinctes d'exécuter un programme Python chaque fois que QGIS démarre.

1.1.1 Variables d'environnement PYQGIS_STARTUP

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

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

1.1.2 Le fichier : `startup.py`

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

1.2 Python Console

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

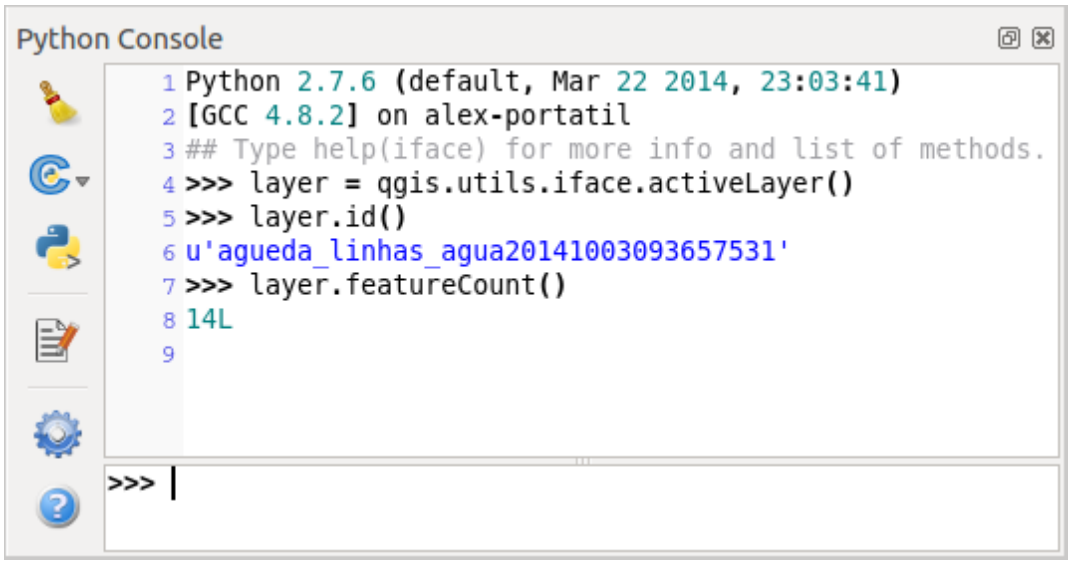


Figure 1.1: La Console Python de QGIS

The screenshot above illustrates how to get the layer currently selected in the layer list, show its ID and optionally, if it is a vector layer, show the feature count. For interaction with QGIS environment, there is a `iface` variable, which is an instance of `QgsInterface`. This interface allows access to the map canvas, menus, toolbars and other parts of the QGIS application.

For convenience of the user, the following statements are executed when the console is started (in future it will be possible to set further initial commands)

```
from qgis.core import *
import qgis.utils
```

For those which use the console often, it may be useful to set a shortcut for triggering the console (within menu *Settings* → *Configure shortcuts...*)

1.3 Extensions Python

QGIS allows enhancement of its functionality using plugins. This was originally possible only with C++ language. With the addition of Python support to QGIS, it is also possible to use plugins written in Python. The main advantage over C++ plugins is its simplicity of distribution (no compiling for each platform needed) and easier development.

Many plugins covering various functionality have been written since the introduction of Python support. The plugin installer allows users to easily fetch, upgrade and remove Python plugins. See the Python Plugin Repositories page for various sources of plugins.

Créer des extensions Python est simple. Voir *Développer des extensions Python* pour des instructions détaillées.

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

1.4 Applications Python

Often when processing some GIS data, it is handy to create some scripts for automating the process instead of doing the same task again and again. With PyQGIS, this is perfectly possible — import the `qgis.core` module, initialize it and you are ready for the processing.

Or you may want to create an interactive application that uses some GIS functionality — measure some data, export a map in PDF or any other functionality. The `qgis.gui` module additionally brings various GUI components, most notably the map canvas widget that can be very easily incorporated into the application with support for zooming, panning and/or any further custom map tools.

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

Note: do *not* use `qgis.py` as a name for your test script — Python will not be able to import the bindings as the script's name will shadow them.

1.4.1 Utiliser PyQGIS dans des scripts indépendants

Pour commencer un script indépendant, initialisez les ressources QGIS au début du script tel que dans le code suivant:

```
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()
```

Nous commençons par importer le module `qgis.core` et ensuite configurons le chemin du préfixe. Le chemin du préfixe est l'endroit où QGIS est installé sur votre système. Il est configuré dans le script en faisant appel à la méthode `setPrefixPath`. Le second argument de la méthode `setPrefixPath` est mis à `True`, ce qui contrôle si les chemins par défaut sont utilisés.

Le chemin d'installation de QGIS varie suivant XXXXX ; le moyen le plus simple pour trouver celle qui correspond à votre système est d'utiliser la *Python Console*

Une fois la configuration du chemin faite, nous sauvegardons une références à `QgsApplication` dans la variable `qgs`. Le second argument est défini à `False`, indiquant que nous n'envisageons pas d'utiliser une interface graphique étant donné que nous écrivons un script indépendant. `QgsApplication` étant configuré, nous chargeons les fournisseurs de données de QGIS et le registre de couches via la méthode `qgs.initQgis()`. Avec l'initialisation de QGIS, nous sommes désormais prêts à écrire le reste de notre script. A la fin, nous utilisons `qgs.exitQgis()` pour nous assurer de supprimer de la mémoire les fournisseurs de données et le registre de couches.

1.4.2 Utiliser PyQGIS dans une application personnalisée

La seule différence entre *Utiliser PyQGIS dans des scripts indépendants* et une application PyQGIS personnalisée réside dans le second argument lors de l'initialisation de `QgsApplication`. Passer `True` au lieu de `False` pour indiquer que nous allons utiliser une interface graphique.

```
from qgis.core import *

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

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

# load providers
qgs.initQgis()

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

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

Now you can work with QGIS API — load layers and do some processing or fire up a GUI with a map canvas. The possibilities are endless :-)

1.4.3 Exécuter des applications personnalisées

Vous devrez indiquer au système où trouver les librairies de QGIS et les modules Python appropriés s'ils ne sont pas à un emplacement connu — autrement, Python se plaindra:

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

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

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

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

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

Corrigez ce problème en ajoutant les répertoires d'emplacement des bibliothèques QGIS au chemin de recherche de l'éditeur dynamique de liens:

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

Ces commandes peuvent être écrites dans un script de lancement qui gérera le démarrage. Lorsque vous déployez des applications personnalisées qui utilisent PyQGIS, il existe généralement deux possibilités:

- require user to install QGIS on his platform prior to installing your application. The application installer should look for default locations of QGIS libraries and allow user to set the path if not found. This approach has the advantage of being simpler, however it requires user to do more steps.
- Créer un paquet QGIS qui contiendra votre application. Publier l'application sera plus complexe et le paquet d'installation sera plus volumineux mais l'utilisateur n'aura pas à télécharger et à installer d'autres logiciels.

Les deux modèles de déploiement peuvent être mélangés: déployer une application autonome sous Windows et Mac OS et laisser l'installation de QGIS par l'utilisateur (via son gestionnaire de paquets) pour Linux.

CHAPTER 2

Chargement de projets

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

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

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

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

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

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

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

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

CHAPTER 3

Chargement de couches

Ouvrons donc quelques couches de données. QGIS reconnaît les couches vectorielles et raster. En plus, des types de couches personnalisés sont disponibles mais nous ne les aborderons pas ici.

3.1 Couches vectorielles

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!"
```

L'identifiant de source de données est une chaîne de texte, spécifique à chaque type de fournisseur de données vectorielles. Le nom de la couche est utilisée dans le widget liste de couches. Il est important de vérifier si la couche a été chargée ou pas. Si ce n'était pas le cas, une instance de couche non valide est retournée.

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

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

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

La liste suivante montre comment accéder à différentes sources de données provenant de différents fournisseurs de données vectorielles:

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

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

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

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

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

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

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

Note: L'argument `False` passé à `uri.uri(False)` empêche l'expansion des paramètres du système d'authentification. si vous n'avez pas configuré d'authentification, cet argument n'a aucun effet.

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

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

Note: Le fournisseur de chaîne est structuré comme une URL, donc le chemin doit être préfixé avec `file://`. Il permet aussi d'utiliser les géométries formatées en WKT (well-known text) à la place des champs `x` et `y`, et permet de spécifier le système de référence géographique. Par exemple :

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

- Fichiers GPS — le fournisseur de données "gpx" lit les trajets, routes et points de passage d'un fichier gpx. Pour ouvrir un fichier, le type (trajet/route/point) doit être fourni dans l'url :

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

- SpatiaLite database — Similarly to PostGIS databases, `QgsDataSourceURI` can be used for generation of data source identifier:

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

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

- Géométries MySQL basées sur WKB, avec OGR — la source des données est la chaîne de connexion à la table :

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

- Connexion WFS : la connexion est définie par une URL et utilise le fournisseur de données `WFS`

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

L'url peut être crée en utilisant la bibliothèque standard : "urllib"

```
params = {
    'service': 'WFS',
    'version': '1.0.0',
```

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

Note: 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 Couches raster

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.

Les couches raster peuvent également être créées à partir d'un service WCS.

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

detailed URI settings can be found in provider documentation

Vous pouvez aussi charger une couche raster à partir d'un serveur WMS. Il n'est cependant pas encore possible d'avoir accès à la réponse de GetCapabilities à partir de l'API — vous devez connaître les couches que vous voulez :

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

3.3 Map Layer Registry

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

Adding a layer to the registry:

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

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

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

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

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

CHAPTER 4

Utiliser des couches raster

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

4.1 Détails d'une couche

A raster layer consists of one or more raster bands — it is referred to as either single band or multi band raster. One band represents a matrix of values. Usual color image (e.g. aerial photo) is a raster consisting of red, blue and green band. Single band layers typically represent either continuous variables (e.g. elevation) or discrete variables (e.g. land use). In some cases, a raster layer comes with a palette and raster values refer to colors stored in the palette:

```
rlayer.width(), rlayer.height()
(812, 301)
rlayer.extent()
<qgis._core.QgsRectangle object at 0x000000000F8A2048>
rlayer.extent().toString()
u'12.095833,48.552777 : 18.863888,51.056944'
rlayer.rasterType()
2  # 0 = GrayOrUndefined (single band), 1 = Palette (single band), 2 = Multiband
rlayer.bandCount()
3
rlayer.metadata()
u'<p class="glossy">Driver:</p>...'
rlayer.hasPyramids()
False
```

4.2 Moteur de rendu

Lorsqu'un raster est chargé, il récupère un moteur de rendu par défaut basé sur son type. Ce moteur peut être modifié dans les propriétés de la couche ou par programmation.

To query the current renderer:

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

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

- QgsMultiBandColorRenderer
- QgsPalettedRasterRenderer
- QgsSingleBandColorDataRenderer
- QgsSingleBandGrayRenderer
- QgsSingleBandPseudoColorRenderer

Les couches rasters mono-bande peuvent être affichées soit en niveaux de gris (faibles valeurs: noir, valeurs hautes = blanc) ou avec un algorithme de pseudo-couleurs qui affecte des couleurs aux valeurs de la bande unique. Les rasters mono-bande avec une palette peut être affichés en utilisant leur palette. Les couches multi-bandes sont affichées en calquant les bandes sur les couleurs RGB. L'autre possibilité est d'utiliser juste une bande pour le niveau de gris ou la pseudo-coleur.

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

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

4.2.1 Rasters mono-bande

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

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

Le shader affecte les couleurs comme indiqué par sa rampe de couleur. La rampe de couleur est fournie sous forme d'une liste contenant la valeur de pixel avec sa couleur associée. Il existe trois modes d'interpolation des valeurs:

- linear (`INTERPOLATED`): resulting color is linearly interpolated from the color map entries above and below the actual pixel value
- discrete (`DISCRETE`): color is used from the color map entry with equal or higher value
- exact (`EXACT`): color is not interpolated, only the pixels with value equal to color map entries are drawn

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

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

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

4.2.2 Rasters multi-bandes

By default, QGIS maps the first three bands to red, green and blue values to create a color image (this is the `MultiBandColor` drawing style. In some cases you might want to override these setting. The following code

interchanges red band (1) and green band (2):

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

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

4.3 Refreshing Layers

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

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

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

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

The second call emits signal that will force any map canvas containing the layer to issue a refresh.

With WMS raster layers, these commands do not work. In this case, you have to do it explicitly

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

In case you have changed layer symbology (see sections about raster and vector layers on how to do that), you might want to force QGIS to update the layer symbology in the layer list (legend) widget. This can be done as follows (`iface` is an instance of `QgisInterface`)

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

4.4 Interrogation des données

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

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

The `results` method in this case returns a dictionary, with band indices as keys, and band values as values.

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

CHAPTER 5

Utilisation de couches vectorielles

Cette section résume les diverses actions possibles sur les couches vectorielles.

5.1 Récupérer les informations relatives aux attributs

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()
```

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

5.2 Sélection des entités

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 Itérer sur une couche vecteur

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 Accès aux attributs

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 Itérer sur une sélection d'entités

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 Itérer sur un sous-ensemble d'entités

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

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

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

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

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

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

See *Expressions, Filtrage et Calcul de valeurs* for the details about the syntax supported by `QgsExpression`.

La requête peut être utilisée pour définir les données à récupérer de chaque entité, de manière à ce que l'itérateur ne retourne que des données partielles pour toutes les entités.

```
# 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)
```

Astuce: Speed features request

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

5.4 Modifier des couches vecteur

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'
```

En utilisant l'une des méthodes qui suivent pour l'édition de couches vectorielles, les changements sont directement validés dans le dispositif de stockage d'informations sous-jacent (base de données, fichier, etc.). Si vous désirez uniquement faire des changements temporaires, passez à la section suivante qui explique comment réaliser des *modifications à l'aide d'un tampon d'édition*.

Note: Si vous travaillez dans QGIS (soit à partir de la console, soit à partir d'une extension), il peut être nécessaire de forcer la mise à jour du canevas de cartes pour pouvoir voir les changements que vous avez effectués aux géométries, au style ou aux attributs

```
# 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 Ajout d'Entités

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 Suppression d'Entités

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 Modifier des Entités

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 })
```

Astuce: Favor QgsVectorLayerEditUtils class for geometry-only edits

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

Astuce: Directly save changes using `with` based command

Using `with edit(layer):` the changes will be commited automatically calling `commitChanges()` at the end. If any exception occurs, it will `rollBack()` all the changes. See *Modifier des couches vecteur à l'aide d'un tampon d'édition*.

5.4.4 Ajout et Suppression de Champs

Pour ajouter des champs (attributs) vous devez indiquer une liste de définitions de champs. Pour la suppression de champs, fournissez juste une liste des index des champs.

```
from PyQt4.QtCore import QVariant

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

if caps & QgsVectorDataProvider.DeleteAttributes:
    res = layer.dataProvider().deleteAttributes([0])
```

Après l'ajout ou la suppression de champs dans le pilote de données, les champs de la couche doivent être rafraîchis car les changements ne sont pas automatiquement propagés.

```
layer.updateFields()
```

5.5 Modifier des couches vecteur à l'aide d'un tampon d'édition

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

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

```
from PyQt4.QtCore import QVariant

# add two features (QgsFeature instances)
layer.addFeatures([feat1,feat2])
# delete a feature with specified ID
layer.deleteFeature(fid)

# set new geometry (QgsGeometry instance) for a feature
layer.changeGeometry(fid, geometry)
# update an attribute with given field index (int) to given value (QVariant)
layer.changeAttributeValue(fid, fieldIndex, value)

# add new field
layer.addAttribute(QgsField("mytext", QVariant.String))
# remove a field
layer.deleteAttribute(fieldIndex)
```

In order to make undo/redo work properly, the above mentioned calls have to be wrapped into undo commands. (If you do not care about undo/redo and want to have the changes stored immediately, then you will have easier work by *editing with data provider*.) How to use the undo functionality

```
layer.beginEditCommand("Feature triangulation")

# ... call layer's editing methods ...

if problem_occurred:
  layer.destroyEditCommand()
  return

# ... more editing ...

layer.endEditCommand()
```

The `beginEditCommand()` will create an internal "active" command and will record subsequent changes in vector layer. With the call to `endEditCommand()` the command is pushed onto the undo stack and the user will be able to undo/redo it from GUI. In case something went wrong while doing the changes, the `destroyEditCommand()` method will remove the command and rollback all changes done while this command was active.

To start editing mode, there is `startEditing()` method, to stop editing there are `commitChanges()` and `rollBack()` — however normally you should not need these methods and leave this functionality to be triggered by the user.

Vous pouvez également utiliser le: code: *with edit (layer)* -déclaration pour envelopper l'acceptation et l'annulation dans un bloc de code plus sémantique comme illustré dans l'exemple ci-dessous:

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

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

5.6 Utilisation des index spatiaux

Les index spatiaux peuvent améliorer fortement les performances de votre code si vous réalisez de fréquentes requêtes sur une couche vecteur. Imaginez par exemple que vous écrivez un algorithme d'interpolation et que pour une position donnée, vous devez déterminer les 10 points les plus proches dans une couche de points, dans l'objectif d'utiliser ces points pour calculer une valeur interpolée. Sans index spatial, la seule méthode pour QGIS de trouver ces 10 points est de calculer la distance entre tous les points de la couche et l'endroit indiqué et de comparer ces distances entre-elles. Cela peut prendre beaucoup de temps spécialement si vous devez répeter l'opération sur plusieurs emplacements. Si index spatial existe pour la couche, l'opération est bien plus efficace.

Vous pouvez vous représenter une couche sans index spatial comme un annuaire dans lequel les numéros de téléphone ne sont pas ordonnés ou indexés. Le seul moyen de trouver le numéro de téléphone d'une personne est de lire l'annuaire en commençant du début jusqu'à ce que vous le trouviez.

Les index spatiaux ne sont pas créés par défaut pour une couche vectorielle QGIS, mais vous pouvez les créer facilement. C'est ce que vous devez faire:

- 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)
  ```

- Alternativement, vous pouvez charger toutes les entités de la couche en une fois en utilisant un chargement en volume.

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

- Une fois que l'index est rempli avec des valeurs, vous pouvez lancer vos requêtes:

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

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

5.7 Writing Vector Layers

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

There are two possibilities how to export a vector layer:

- from an instance of `QgsVectorLayer`

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

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

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

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

Consultez les formats gérés par OGR pour trouver les noms de pilote valides. Vous devez indiquer la valeur dans la colonne" Code "comme nom du pilote. En option, vous pouvez définir si vous souhaitez exporter uniquement les fonctions sélectionnées, transmettre d'autres options spécifiques au pilote pour la création ou indiquer à l'auteur de ne pas créer d'attributs. Consultez la documentation pour connaître la syntaxe complète.

- directly from features

```
from PyQt4.QtCore import QVariant

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

""" create an instance of vector file writer, which will create the vector file.
Arguments:
1. path to new file (will fail if exists already)
2. encoding of the attributes
3. field map
4. geometry type - from WKBTYPE enum
5. layer's spatial reference (instance of
   QgsCoordinateReferenceSystem) - optional
6. driver name for the output file """
writer = QgsVectorFileWriter("my_shapes.shp", "CP1250", fields, QGis.WKBPoint, None, "ESRI Sh

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

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

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

5.8 Memory Provider

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

Le fournisseur gère les champs en chaînes de caractères, en entiers et en réels.

The memory provider also supports spatial indexing, which is enabled by calling the provider's

`createSpatialIndex()` function. Once the spatial index is created you will be able to iterate over features within smaller regions faster (since it's not necessary to traverse all the features, only those in specified rectangle).

A memory provider is created by passing `"memory"` as the provider string to the `QgsVectorLayer` constructor.

Le constructeur utilise également une URI qui définit le type de géométrie de la couche parmi: `"Point"`, `"LineString"`, `"Polygon"`, `"MultiPoint"`, `"MultiLineString"`, ou `"MultiPolygon"`.

L'URI peut également indiquer un système de coordonnée de référence, des champs et l'indexation. La syntaxe est la suivante:

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

index=yes Spécifie que le fournisseur utilisera un index spatial

field=nom:type(longueur,précision) Spécifie un attribut de la couche. L'attribut dispose d'un nom et optionnellement d'un type (integer, double ou string), d'une longueur et d'une précision. Il peut y avoir plusieurs définitions de champs.

L'exemple suivant montre une URI intégrant toutes ces options

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

L'exemple suivant illustre la création et le remplissage d'un fournisseur de données en mémoire

```
from PyQt4.QtCore import QVariant

# create layer
vl = QgsVectorLayer("Point", "temporary_points", "memory")
pr = vl.dataProvider()

# add fields
pr.addAttributes([QgsField("name", QVariant.String),
                  QgsField("age",  QVariant.Int),
                  QgsField("size", QVariant.Double)])
vl.updateFields() # tell the vector layer to fetch changes from the provider

# add a feature
fet = QgsFeature()
fet.setGeometry(QgsGeometry.fromPoint(QgsPoint(10,10)))
fet.setAttributes(["Johny", 2, 0.3])
pr.addFeatures([fet])

# update layer's extent when new features have been added
# because change of extent in provider is not propagated to the layer
vl.updateExtents()
```

Finalement, vérifions que tout s'est bien déroulé

```
# 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 Apparence (Symbologie) des couches vecteur

Lorsqu'une couche vecteur est en cours de rendu, l'apparence des données est assurée par un **moteur de rendu** et des **symboles** associés à la couche. Les symboles sont des classes qui gèrent le dessin de la représentation visuelle des entités alors que les moteurs de rendu déterminent quel symbole doit être utilisé pour une entité particulière.

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

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

Munis de cette référence, faisons un peu d'exploration:

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

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

Type	Classe	Description
singleSymbol	`QgsSingleSymbolRendererV2`	Affiche toutes les entités avec le même symbole.
categorizedSymbol	`QgsCategorizedSymbolRendererV2`	Affiche les entités en utilisant un symbole différent pour chaque catégorie.
graduatedSymbol	`QgsGraduatedSymbolRendererV2`	Affiche les entités en utilisant un symbole différent pour chaque plage de valeurs.

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']
```

Il est possible d'obtenir un extrait du contenu d'un moteur de rendu sous forme de texte, ce qui peut être utile lors du débogage:

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

5.9.1 Moteur de rendu à symbole unique

You can get the symbol used for rendering by calling `symbol()` method and change it with `setSymbol()` method (note for C++ devs: the renderer takes ownership of the symbol.)

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

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

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

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

`name` indique la forme du marqueur, et peut être l'une des valeurs suivantes :

- `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'}
```

Cela peut être utile si vous souhaitez modifier certaines propriétés:

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

5.9.2 Moteur de rendu à symboles catégorisés

You can query and set attribute name which is used for classification: use `classAttribute()` and `setClassAttribute()` methods.

Pour obtenir la liste des catégories

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

Where `value()` is the value used for discrimination between categories, `label()` is a text used for category description and `symbol()` method returns assigned symbol.

The renderer usually stores also original symbol and color ramp which were used for the classification: `sourceColorRamp()` and `sourceSymbol()` methods.

5.9.3 Moteur de rendu à symboles gradués

Ce moteur de rendu est très similaire au moteur de rendu par symbole catégorisé ci-dessus mais au lieu d'utiliser une seule valeur d'attribut par classe, il utilise une classification par plages de valeurs et peut donc être employé uniquement sur des attributs numériques.

Pour avoir plus d'informations sur les plages utilisées par le moteur de rendu:

```
for ran in rendererV2.ranges():
    print "%f - %f: %s %s" % (
        ran.lowerValue(),
        ran.upperValue(),
        ran.label(),
        str(ran.symbol())
      )
```

Vous pouvez à nouveau utiliser `classAttribute()` pour trouver le nom de l'attribut de classification ainsi que les méthodes `sourceSymbol()` et `sourceColorRamp()`. Il existe en plus une méthode `mode()` qui permet de déterminer comment les classes ont été créées: en utilisant des intervalles égaux, des quantiles ou tout autre méthode.

Si vous souhaitez créer votre propre moteur de rendu gradué, vous pouvez utiliser l'extrait de code qui est présenté dans l'exemple ci-dessous (qui créé simplement un arrangement en deux classes):

```
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 Travailler avec les symboles

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

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

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

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

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

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

La taille et la largeur sont exprimées en millimètres par défaut, les angles sont en degrés.

Travailler avec des couches de symboles

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

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

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

Output

```
EllipseMarker
FontMarker
SimpleMarker
SvgMarker
VectorField
```

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

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

Créer des types personnalisés de couches de symbole

Imaginons que vous souhaitez personnaliser la manière dont sont affichées les données. Vous pouvez créer votre propre classe de couche de symbole qui dessinera les entités de la manière voulue. Voici un exemple de marqueur qui dessine des cercles rouges avec un rayon spécifique.

```
class FooSymbolLayer(QgsMarkerSymbolLayerV2):

  def __init__(self, radius=4.0):
      QgsMarkerSymbolLayerV2.__init__(self)
      self.radius = radius
      self.color = QColor(255,0,0)

  def layerType(self):
     return "FooMarker"

  def properties(self):
      return { "radius" : str(self.radius) }

  def startRender(self, context):
    pass

  def stopRender(self, context):
      pass

  def renderPoint(self, point, context):
      # Rendering depends on whether the symbol is selected (QGIS >= 1.5)
      color = context.selectionColor() if context.selected() else self.color
      p = context.renderContext().painter()
      p.setPen(color)
      p.drawEllipse(point, self.radius, self.radius)

  def clone(self):
      return FooSymbolLayer(self.radius)
```

The `layerType()` method determines the name of the symbol layer, it has to be unique among all symbol layers. Properties are used for persistence of attributes. `clone()` method must return a copy of the symbol layer with all attributes being exactly the same. Finally there are rendering methods: `startRender()` is called before rendering first feature, `stopRender()` when rendering is done. And `renderPoint()` method which does the rendering. The coordinates of the point(s) are already transformed to the output coordinates.

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

En général, il est pratique d'ajouter une interface graphique pour paramétrer les attributs des couches de symbole pour permettre aux utilisateurs de personnaliser l'apparence. Dans le cadre de notre exemple ci-dessus, nous laissons l'utilisateur paramétrer le rayon du cercle. Le code qui suit implémente une telle interface:

```
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()"))
```

Cette interface peut être incorporée dans la boîte de dialogue sur les propriétés de symbole. Lorsque le type couche de symbole est sélectionné dans la boîte de dialogue des propriétés de symbole, cela créé une instance de la couche de symbole et une instance de l'interface. Ensuite, la méthode `setSymbolLayer()` est appelée pour affecter la couche de symbole à l'interface. Dans cette méthode, l'interface doit rafraîchir l'environnement graphique pour afficher les attributs de la couche de symbole. La fonction `symbolLayer()` est utilisée pour retrouver la couche de symbole des propriétés de la boîte de dialogue afin de l'utiliser pour le symbole.

A chaque changement d'attributs, l'interface doit émettre le signal `changed()` pour laisser les propriétés de la boîte de dialogue mettre à jour l'aperçu de sumbole.

Maintenant, il nous manque un dernier détail: informer QGIS de ces nouvelles classes. On peut le faire en ajoutant la couche de symbole au registre. Il est possible d'utiliser la couche de symbole sans l'ajouter au registre mais certaines fonctionnalités ne fonctionneront pas comme le chargement de fichiers de projet avec une couche de symbole personnalisée ou l'impossibilité d'éditer les attributs de la couche dans l'interface graphique.

Nous devons ensuite créer les métadonnées de la couche de symbole.

```
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.

La dernière étape consiste à ajouter la couche de symbole au registre et c'est terminé !

5.9.5 Créer ses propres moteurs de rendu

Il est parfois intéressant de créer une nouvelle implémentation de moteur de rendu si vous désirez personnaliser les règles de sélection des symboles utilisés pour l'affichage des entités. Voici quelques exemples d'utilisation: le symbole est déterminé par une combinaison de champs, la taille des symboles change selon l'échelle courante, etc.

Le code qui suit montre un moteur de rendu personnalisé simple qui crée deux symboles de marqueur et choisit au hasard l'un d'entre eux pour chaque entité.

```
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.

Le dernier élément qui manque concerne les métadonnées du moteur ainsi que son enregistrement dans le registre. Sans ces éléments, le chargement de couches avec le moteur de rendu ne sera pas possible et l'utilisateur ne pourra pas le sélectionner dans la liste des moteurs de rendus. Finissons notre exemple sur RandomRenderer:

```
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())
```

De la même manière que pour les couches de symbole, le constructeur des métadonnées attend le nom du moteur de rendu, le nom visible pour les utilisateurs et optionnellement le nom des icônes du moteur de rendu. La méthode `createRenderer()` fait passer une instance de `QDomElement` qui peut être utilisée pour restaurer l'état du moteur de rendu en utilisant un arbre DOM. La méthode `createRendererWidget()` créé l'interface graphique de configuration. Elle n'est pas obligatoire et peut renvoyer *None* si le moteur de rendu n'a pas d'interface graphique.

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 Sujets complémentaires

A FAIRE :

- création/modification des symboles
- working with style (`QgsStyleV2`)
- working with color ramps (`QgsVectorColorRampV2`)
- rule-based renderer (see this blogpost)
- Explorer les couches de symboles et les registres de rendus

CHAPTER 6

Manipulation de la géométrie

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

Parfois, une entité correspond à une collection d'éléments géométriques simples (d'un seul tenant). Une telle géométrie est appelée multi-parties. Si elle ne contient qu'un seul type de géométrie, il s'agit de multi-points, de multi-lignes ou de multi-polygones. Par exemple, un pays constitué de plusieurs îles peut être représenté par un multi-polygone.

Les coordonnées des géométries peuvent être dans n'importe quel système de coordonnées de référence (SCR). Lorsqu'on accède aux entités d'une couche, les géométries correspondantes auront leurs coordonnées dans le SCR de la couche.

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

6.1 Construction de géométrie

There are several options for creating a geometry:

- à partir des coordonnées

  ```
  gPnt = QgsGeometry.fromPoint(QgsPoint(1,1))
  gLine = QgsGeometry.fromPolyline([QgsPoint(1, 1), QgsPoint(2, 2)])
  gPolygon = QgsGeometry.fromPolygon([[QgsPoint(1, 1), QgsPoint(2, 2),
                                       QgsPoint(2, 1)]])
  ```

  Coordinates are given using `QgsPoint` class.

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

  Les géométries multi-parties sont d'un niveau plus complexe: les multipoints sont une succession de points, les multilignes une succession de lignes et les multipolygones une succession de polygones.

- depuis un Well-Known-Text (WKT)

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

- depuis un Well-Known-Binary (WKB)

```
>>> g = QgsGeometry()
>>> wkb = '01010000000000000000004540000000000000001440'.decode('hex')
>>> g.fromWkb(wkb)
>>> g.exportToWkt()
'Point (42 5)'
```

6.2 Accéder à la Géométrie

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

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

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

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

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

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

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

6.3 Prédicats et opérations géométriques

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

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

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

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

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

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

Vous trouverez de nombreux exemples d'algorithmes inclus dans QGIS et utiliser ces méthodes pour analyser et modifier les données vectorielles. Voici des liens vers le code de quelques-uns.

Additional information can be found in following sources:

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

CHAPTER 7

Support de projections

7.1 Système de coordonnées de référence

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

- spécifier le SCR par son ID

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

  QGIS utilise trois identifiants différents pour chaque système de référence:

  - `PostgisCrsId` — Identifiants utilisés dans les bases de données PostGIS.
  - `InternalCrsId` — Identifiants utilisés dans la base de données QGIS.
  - `EpsgCrsId` — Identifiants définis par l'organisation EPSG.

  Sauf indication contraire dans le deuxième paramètre, le SRID de PostGIS est utilisé par défaut.

- spécifier le SCR par son 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.toProj4()
# check whether it's geographic or projected coordinate system
print "Is geographic:", crs.geographicFlag()
# check type of map units in this CRS (values defined in QGis::units enum)
print "Map units:", crs.mapUnits()
```

7.2 Projections

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

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

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

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

CHAPTER 8

Using Map Canvas

The Map canvas widget is probably the most important widget within QGIS because it shows the map composed from overlaid map layers and allows interaction 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.

Pour résumer, l'architecture du canevas de carte repose sur trois concepts:

- le canevas de carte — pour visualiser la carte
- map canvas items — additional items that can be displayed in map canvas
- map tools — for interaction with map canvas

8.1 Intégrer un canevas de carte

Le canevas de carte est un objet comme tous les autres objets Qt, on peut donc l'utiliser simplement en le créant et en l'affichant:

```
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.

Par défaut, le canevas de carte a un arrière-plan noir et n'utilise pas l'anticrénelage. Pour afficher un arrière-plan blanc et activer l'anticrénelage pour un rendu plus lisse:

```
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)])
```

Après exécution de ces commandes, le canevas de carte devrait afficher la couche chargée.

8.2 Utiliser les outils cartographiques avec le canevas

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 Contour d'édition et symboles de sommets

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.

Pour afficher une polyligne:

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

Pour afficher un polygone:

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

Veuillez noter que les points d'un polygone ne sont pas stockés dans une liste. En fait, il s'agit d'une liste d'anneaux contenants les anneaux linéaires du polygones: le premier anneau est la limite extérieure, les autres (optionnels) anneaux correspondent aux trous dans le polygone.

Les contours d'édition peut être personnalisés pour changer leur couleur ou la taille de la ligne:

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

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

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

(en C++, il est possible de juste supprimer l'objet mais sous Python `del r` détruira juste la référence et l'objet existera toujours étant donné qu'il appartient au canevas).

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 Ecrire des outils cartographiques personnalisés

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.

Voici un exemple d'outil cartographique qui permet de définir une emprise rectangulaire en cliquant et en déplaçant la souris sur le canevas. Lorsque le rectangle est dessiné, il exporte les coordonnées de ses limites dans la console. On utilise des éléments de contour d'édition décrits auparavant pour afficher le rectangle sélectionné au fur et à mesure de son dessin.

```
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 Ecrire des éléments de canevas de carte personnalisés

A FAIRE : Comment créer un objet de canevas de carte ?

```
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

Rendu cartographique et Impression

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

9.1 Rendu simple

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

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

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

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

render = QgsMapRenderer()

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

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

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

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

```
p.end()

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

9.2 Rendu des couches ayant différents SCR

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

```
...
# set layer set
layers = QgsMapLayerRegistry.instance().mapLayers()
lst = layers.keys()
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 Output using Map Composer

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

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

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

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

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

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

- carte — cet élément indique aux bibliothèques l'emplacement de la carte. Nous créons ici une carte et l'étirons sur toute la taille de la page

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

- étiquette — permet d'afficher des étiquettes. Il est possible d'en modifier la police, la couleur, l'alignement et les marges:

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

- légende

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

- Échelle graphique

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

- flèche
- image
- Forme simple
- Forme basée sur les nœuds

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

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

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

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

- table

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

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

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

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

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

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

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

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

9.3.1 Output to a raster image

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

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

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

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

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

9.3.2 Output to PDF

The following code fragment renders a composition to a PDF file

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

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

CHAPTER 10

Expressions, Filtrage et Calcul de valeurs

QGIS propose quelques fonctionnalités pour faire de l'analyse syntaxique d'expressions semblable au SQL. Seulement un petit sous-ensemble des syntaxes SQL est géré. Les expressions peuvent être évaluées comme des prédicats booléens (retournant Vrai ou Faux) ou comme des fonctions (retournant une valeur scalaire). Voir *vector_expressions* dans le manuel Utilisateur pour une liste complète des fonctions disponibles.

Trois types basiques sont supportés :

- nombre — aussi bien les nombres entiers que décimaux, par exemple `123`, `3.14`
- texte — ils doivent être entre guillemets simples: `'hello world'`
- référence de colonne — lors de l'évaluation, la référence est remplacée par la valeur réelle du champ. Les noms ne sont pas échappés.

Les opérations suivantes sont disponibles:

- opérateurs arithmétiques: `+`, `-`, `*`, `/`, `^`
- parenthèses: pour faire respecter la précédence des opérateurs: `(1 + 1) * 3`
- les unaires plus et moins: `-12`, `+5`
- fonctions mathématiques: `sqrt`, `sin`, `cos`, `tan`, `asin`, `acos`, `atan`
- fonctions de conversion : `to_int`, `to_real`, `to_string`, `to_date`
- fonctions géométriques: `$area`, `$length`
- Fonctions de manipulation de géométries : `$x`, `$y`, `$geometry`, `num_geometries`, `centroid`

Et les prédicats suivants sont pris en charge:

- comparaison: `=`, `!=`, `>`, `>=`, `<`, `<=`
- comparaison partielle: `LIKE` (avec % ou _), ~ (expressions régulières)
- prédicats logiques: `AND`, `OR`, `NOT`
- Vérification de la valeur NULL: `IS NULL`, `IS NOT NULL`

Exemples de prédicats:

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

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

Exemples d'expressions scalaires:

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

10.1 Analyse syntaxique d'expressions

```
>>> 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 Évaluation des expressions

10.2.1 Expressions basiques

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

10.2.2 Expressions avec entités

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

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

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

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

10.2.3 Gestion des erreurs

```
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 Exemples

L'exemple suivant peut être utilisé pour filtrer une couche et ne renverra que les entités qui correspondent au prédicat.

```
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

Lecture et sauvegarde de configurations

Il est souvent utile pour une extension de sauvegarder des variables pour éviter à l'utilisateur de saisir à nouveau leur valeur ou de faire une nouvelle sélection à chaque lancement de l'extension.

Ces variables peuvent être sauvegardées et récupérées grâce à Qt et à l'API QGIS. Pour chaque variable, vous devez fournir une clé qui sera utilisée pour y accéder — pour la couleur préférée de l'utilisateur, vous pourriez utiliser la clé "couleur_favorite" ou toute autre chaîne de caractères explicite. Nous vous recommandons d'utiliser une convention pour nommer les clés.

We can make difference between several types of settings:

- **global settings** — they are bound to the user at particular machine. QGIS itself stores a lot of global settings, for example, main window size or default snapping tolerance. This functionality is provided directly by Qt framework by the means of `QSettings` class. By default, this class stores settings in system's "native" way of storing settings, that is — registry (on Windows), .plist file (on macOS) or .ini file (on Unix). The QSettings documentation is comprehensive, so we will provide just a simple example

  ```
  def store():
    s = QSettings()
    s.setValue("myplugin/mytext", "hello world")
    s.setValue("myplugin/myint",  10)
    s.setValue("myplugin/myreal", 3.14)

  def read():
    s = QSettings()
    mytext = s.value("myplugin/mytext", "default text")
    myint  = s.value("myplugin/myint", 123)
    myreal = s.value("myplugin/myreal", 2.71)
  ```

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

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

  ```
  proj = QgsProject.instance()

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

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

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

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

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

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

CHAPTER 12

Communiquer avec l'utilisateur

Cette section montre quelques méthodes et éléments qui devraient être employés pour communiquer avec l'utilisateur dans l'objectif de conserver une certaine constance dans l'interface utilisateur

12.1 Showing messages. The `QgsMessageBar` class

Utiliser des boîtes à message est généralement une mauvaise idée du point de vue de l'expérience utilisateur. Pour afficher une information simple sur une seule ligne ou des messages d'avertissement ou d'erreur, la barre de message QGIS est généralement une meilleure option.

En utilisant la référence vers l'objet d'interface QGis, vous pouvez afficher un message dans la barre de message à l'aide du code suivant

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

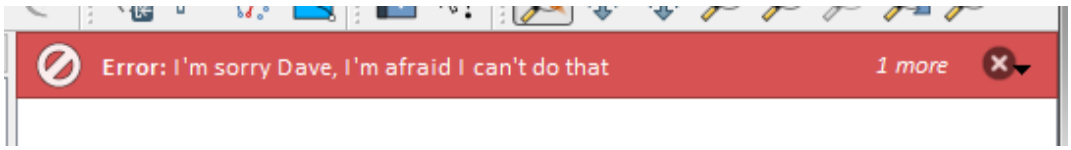


Figure 12.1: Barre de message de QGIS

Vous pouvez spécifier une durée pour que l'affichage soit limité dans le temps.

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

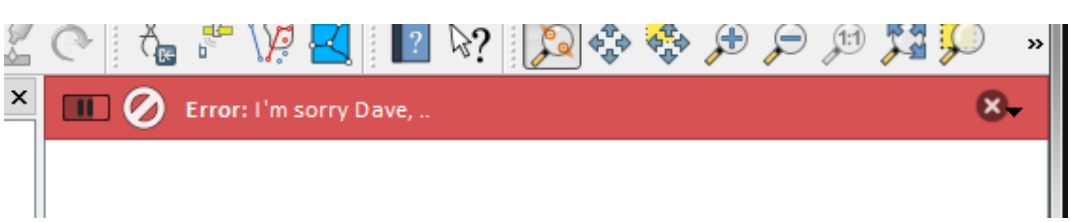


Figure 12.2: Barre de message de Qgis avec décompte

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

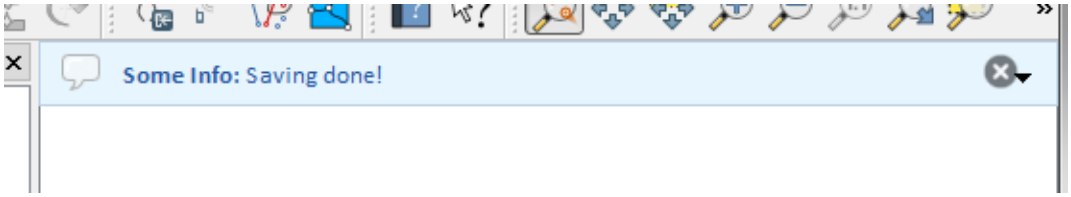


Figure 12.3: Barre de message QGis (info)

Des Widgets peuvent être ajoutés à la barre de message comme par exemple un bouton pour montrer davantage d'information

```
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)
```

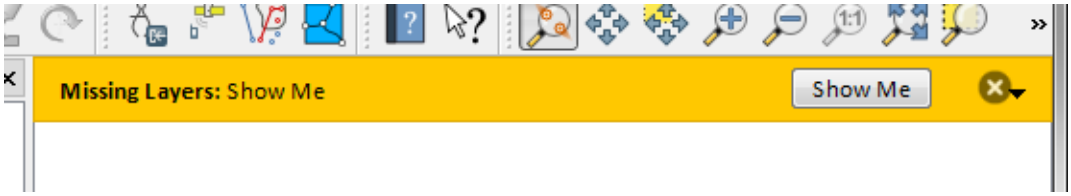


Figure 12.4: Barre de message QGis avec un bouton

Vous pouvez également utiliser une barre de message au sein de votre propre boîte de dialogue afin de ne pas afficher de boîte à message ou bien s'il n'y pas d'intérêt de l'afficher dans la fenêtre principale de QGis

```
class MyDialog(QDialog):
    def __init__(self):
        QDialog.__init__(self)
        self.bar = QgsMessageBar()
        self.bar.setSizePolicy( QSizePolicy.Minimum, QSizePolicy.Fixed )
        self.setLayout(QGridLayout())
        self.layout().setContentsMargins(0, 0, 0, 0)
        self.buttonbox = QDialogButtonBox(QDialogButtonBox.Ok)
        self.buttonbox.accepted.connect(self.run)
        self.layout().addWidget(self.buttonbox, 0, 0, 2, 1)
        self.layout().addWidget(self.bar, 0, 0, 1, 1)

    def run(self):
        self.bar.pushMessage("Hello", "World", level=QgsMessageBar.INFO)
```

12.2 Afficher la progression

Les barres de progression peuvent également être insérées dans la barre de message QGis car, comme nous l'avons déjà vu, cette dernière accepte les widgets. Voici un exemple que vous pouvez utilisez dans la console.

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

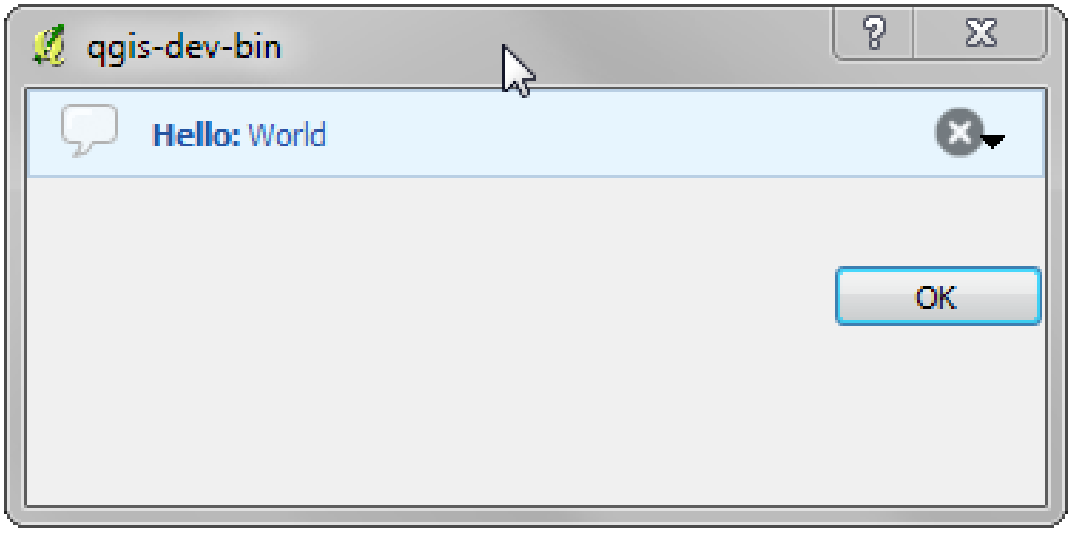


Figure 12.5: Barre de message QGis avec une boîte de dialogue personnalisée

```
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()
```

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

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

12.3 Journal

Vous pouvez utiliser le système de journal de QGis pour enregistrer toute information à conserver sur l'exécution de votre code.

```
# 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

Développer des extensions Python

Il est possible de créer des extensions dans le langage de programmation Python. Comparé aux extensions classiques développées en C++, celles-ci devraient être plus faciles à écrire, comprendre, maintenir et distribuer du fait du caractère dynamique du langage python.

Les extensions Python sont listées avec les extensions C++ dans le gestionnaire d'extension. Voici les chemins où elles peuvent être situées:

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

Sous Windows, le répertoire Maison (noté ci-dessus par `~`) est généralement situé dans un emplacement du type `C:\Documents and Settings\(utilisateur)` (sous Windows XP et inférieur) ou dans `C:\Utilisateurs\(utilisateur)`. Étant donné que QGIS utilise Python 2.7, les sous-répertoires de ces chemins doivent contenir un fichier __init__.py pour pouvoir les considérer comme des progiciels Python qui peuvent être importés en tant qu'extensions.

Note: En configurant *QGIS_PLUGINPATH* avec un chemin d'accès vers un répertoire existant, vous pouvez ajouter ce chemin d'accès à la liste des chemins d'accès qui est parcourue pour trouver les extensions.

Étapes :

1. *Idée*: Avoir une idée de ce que vous souhaitez faire avec votre nouvelle extension. Pourquoi le faites-vous? Quel problème souhaitez-vous résoudre? N'y a-t-il pas déjà une autre extension pour ce problème?
2. *Créer des fichiers*: Créer les fichiers décrits plus loin. Un point de départ (`__init.py__`). Remplissez les fichiers *Métadonnées de l'extension* (`metadata.txt`). Un corps principal de l'extension

(`mainplugin.py`). Un formulaire créé avec QT-Designer (`form.ui`), et son fichier de ressources `resources.qrc`.

3. *Écrire le code*: Écrire le code à l'intérieur du fichier `mainplugin.py`
4. *Test*: Fermez et ré-ouvrez QGIS et importez à nouveau votre extension. Vérifiez si tout est OK.
5. *Publier*: Publiez votre extension dans le dépôt QGIS ou créez votre propre dépôt tel un "arsenal" pour vos "armes SIG" personnelles.

13.1 Écriture d'une extension

Depuis l'introduction des extensions Python dans QGIS, un certain nombre d'extensions est apparu - sur le wiki du Dépôt des Extensions vous trouverez certaines d'entre elles et vous pourrez utiliser leur source pour en savoir plus sur la programmation avec PyQGIS ou pour savoir si vous ne dupliquez pas des efforts de développement. L'équipe QGIS maintient également un *Dépôt officiel des extensions Python*. Prêt à créer une extension, mais aucune idée de quoi faire ? Le wiki des Idées d'extensions Python liste les souhaits de la communauté !

13.1.1 Fichiers de l'extension

Vous pouvez voir ici la structure du répertoire de notre exemple d'extension

```
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*
```

A quoi correspondent ces fichiers?

- `__init__.py` = Le point d'entrée de l'extension. Il doit comporter une méthode `classFactory()` et peut disposer d'un autre code d'initialisation.
- `mainPlugin.py` = Le code principal de l'extension. Contient toutes les informations sur les actions de l'extension et le code principal.
- `resources.qrc` = Le document .xml créé par Qt Designer. Contient les chemins relatifs vers les ressources des formulaires.
- `resources.py` = La traduction en Python du fichier resources.qrc décrit ci-dessus.
- `form.ui` = L'interface graphique créée avec Qt Designer.
- `form.py` = La traduction en Python du fichier form.ui décrit ci-dessus.
- `metadata.txt` = Requis pour QGIS >= 1.8.0. Contient les informations générales, la version, le nom et d'autres métadonnées utilisées par le site des extensions et l'infrastructure de l'extension. A partir de QGIS 2.0, les métadonnées du fichier `__init__.py` ne seront plus acceptées et le fichier `metadata.txt` sera requis.

Vous trouverez ici une méthode automatisée en ligne pour créer les fichiers de base (le squelette) d'une classique extension Python sous QGIS.

Il existe également une extension QGIS nommée Plugin Builder qui crée un modèle d'extension depuis QGIS et ne nécessite pas de connexion Internet. C'est l'option recommandée car elle produit des sources compatibles avec la version 2.0.

Warning: Si vous projetez de déposer l'extension sur le *Dépôt officiel des extensions Python*, vous devez vérifier que votre extension respecte certaines règles supplémentaires, requises pour sa *Validation*.

13.2 Contenu de l'extension

Ici vous pouvez trouver des informations et des exemples sur ce qu'il faut ajouter dans chacun des fichiers de la structure de fichiers décrite ci-dessus.

13.2.1 Métadonnées de l'extension

Tout d'abord, le gestionnaire d'extensions a besoin de récupérer des informations de base sur l'extension par exemple son nom, sa description, etc. Le fichier `metadata.txt` est le bon endroit où mettre cette information.

Important: Toutes les métadonnées doivent être encodées en UTF-8.

Nom de la métadonnée	Requis	Notes
name	Vrai	texte court contenant le nom de l'extension
qgisMinimumVersion	Vrai	version minimum de QGIS en notation par points
qgisMaximumVersion	Faux	version maximum de QGIS en notation par points
description	Vrai	un texte court qui décrit l'extension. Le HTML n'est pas autorisé
about	Vrai	un texte long qui décrit l'extension en détail, pas de HTML autorisé
version	Vrai	texte court avec le numéro de version par points
author	Vrai	nom de l'auteur
email	Vrai	email de l'auteur, non affiché dans le gestionnaire de plugins QGIS ou dans le site Web, à moins d'être un utilisateur enregistré logué, donc seulement visible par les autres auteurs de plugin et par les administrateurs du site Web de plugin
changelog	Faux	texte, peut être multi-lignes, pas de HTML autorisé
experimental	Faux	indicateur booléen, *Vrai* ou *Faux*
deprecated	Faux	indicateur booléen, *Vrai* ou *Faux*, s'applique à l'extension entière et pas simplement à la version chargée
tags	Faux	liste séparée par une virgule, les espaces sont autorisés à l'intérieur des balises individuelles
homepage	Faux	une URL valide pointant vers la page d'accueil de l'extension
repository	Vrai	une URL valide pour le dépôt du code source
tracker	Faux	une URL valide pour les billets et rapports de bugs
icon	Faux	un nom de fichier ou un chemin relatif (relatif au dossier de base du package compressé du plugin) d'une image web sympa (PNG, JPEG)
category	Faux	soit *Raster*, *Vector*, *Database* ou *Web*

Par défaut, les extensions sont placées dans le menu *Extension* (nous verrons dans la section suivante comment ajouter une entrée de menu pour notre extension) mais elles peuvent également être placées dans les menus *Raster*, *Vecteur*, *Base de données* et *Internet*.

Une entrée "category" existe dans les métadonnées afin de spécifier cela, pour que l'extension soit classée en conséquence. Cette entrée de métadonnées est utilisée comme astuce pour les utilisateurs et leur dit où (dans quel menu) l'extension peut être trouvée. Les valeurs autorisées pour "category" sont : Vector, Raster, Database ou Web. Par exemple, si votre extension sera disponible dans le menu *Raster*, ajoutez ceci à `metadata.txt` :

```
category=Raster
```

Note: Si la variable *qgisMaximumVersion* est vide, elle sera automatiquement paramétrée à la version majeure plus *.99* lorsque l'extension sera chargée sur le *Dépôt officiel des extensions Python*.

Un exemple pour ce fichier 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

Ce fichier est requis par le système d'import de Python. QGIS impose aussi que ce fichier contienne une fonction `classFactory()` qui est appelée lorsque l'extension est chargée dans QGIS. Elle reçoit une référence vers une instance de la classe `QgisInterface` et doit renvoyer l'instance de la classe de l'extension située dans le fichier `mainplugin.py`. Dans notre cas, elle s'appelle `TestPlugin` (voir plus loin). Voici à quoi devrait ressembler le fichier `__init__.py` :

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

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

13.2.3 mainPlugin.py

C'est l'endroit où tout se passe et voici à quoi il devrait ressembler (ex: `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!"
```

Les seules fonctions de l'extension qui doivent exister dans le fichier source principal de l'extension (ex: `mainPlugin.py`) sont :

- `__init__` –> qui donne accès à l'interface de QGIS
- `initGui()` –> appelée lorsque l'extension est chargée.
- `unload()` –> chargée lorsque l'extension est déchargée.

Vous pouvez voir que dans l'exemple ci-dessus, la fonction `addPluginToMenu()` est utilisée. Elle ajoute l'entrée de menu correspondante au menu *Extension*. Il existe d'autres méthodes pour ajouter l'action dans un menu différent. Voici une liste de ces méthodes :

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

Toutes ont la même syntaxe que la méthode addPluginToMenu().

Ajouter votre extension dans un des menus prédéfinis est une méthode recommandée pour conserver la cohérence de l'organisation des entrées d'extensions. Toutefois, vous pouvez ajouter votre propre groupe de menus directement à la barre de menus, comme le montre l'exemple suivant :

```
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()
```

N'oubliez pas de paramétrer la propriété objectName de QAction et de QMenu avec un nom spécifique à votre extension pour qu'elle puisse être personnalisée.

13.2.4 Fichier de ressources

Vous pouvez voir que dans la fonction initGui(), nous avons utilisé une icône depuis le fichier ressource (appelé resources.qrc dans notre cas)

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

Il est bon d'utiliser un préfixe qui n'entrera pas en collision avec d'autres extensions ou toute autre partie de QGIS sinon vous risquez de récupérer des ressources que vous ne voulez pas. Vous devez juste générer un fichier python qui contiendra ces ressources. Cela peut être fait avec la commande **pyrcc4**.

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

Note: Dans les environnements Windows, tenter de lancer **pyrcc4** en mode commande ou depuis Powershell générera probablement une erreur du type "Windows ne peut pas accéder au périphérique, au répertoire, ou au fichier [...]". La solution la plus simple est certainement d'utiliser l'environnement OSGeo4W mais si vous vous sentez capable de modifier la variable d'environnement PATH ou de préciser de chemin vers l'exécutable de manière explicite vous devriez pouvoir le trouver dans <Votre répertoire d'installation de QGIS>\bin\pyrcc4.exe.

Et c'est tout ! Rien de bien compliqué :)

Si tout a été réalisé correctement, vous devriez pouvoir trouver et charger votre extension dans le gestionnaire d'extensions et voir un message dans la console lorsque l'icône de barre d'outils ou l'entrée de menu appropriée

est sélectionnée.

Lorsque vous travaillez sur une extension réelle, il est sage d'écrire l'extension dans un autre répertoire et de créer un fichier makefile qui générera l'interface graphique et les fichiers ressources en terminant par l'installation de l'extension dans l'installation QGIS.

13.3 Documentation

La documentation sur l'extension peut être écrite sous forme de fichiers d'aide HTML. Le module `qgis.utils` fournit une fonction, `showPluginHelp()`, qui ouvrira le fichier d'aide dans un navigateur, de la même manière que pour l'aide de QGIS.

La fonction `showPluginHelp()` recherche les fichiers d'aide dans le même dossier que le module d'appel. elle recherchera, dans l'ordre, `index-ll_cc.html`, `index-ll.html`, `index-en.html`, `index-en_us.html` et `index.html`, affichant celui qu'elle trouve en premier. Ici, `ll_cc` est pour la locale de QGIS. Ceci permet d'inclure des traductions multiples dans la documentation de l'extension.

La fonction `showPluginHelp()` prend également les paramètres packageName qui identifie une extension spécifique pour laquelle une aide sera affichée; filename qui peut remplacer "index" dans les noms de fichiers à rechercher; section qui est le nom d'une ancre HTML dans le document où le navigateur doit se positionner.

13.4 Traduction

En peu d'étapes, vous pouvez configurer un environnement pour la traduction de votre extension, de telle sorte que, selon les paramètres de langue de l'ordinateur, l'extension sera chargée dans différentes langues.

13.4.1 Prérequis logiciels

La façon la plus facile de créer et gérer les fichiers de traduction est d'installer Qt Linguist. Dans un environnement Linux, cela peut se faire en tapant:

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

13.4.2 Fichiers et répertoire

Une fois l'extension créée, vous verrez un dossier `i18n` à la racine du dossier de l'extension.

Tous les fichiers de traduction doivent être à l'intérieur de ce répertoire.

Fichier .pro

D'abord, vous devez créer un fichier `.pro`, qui est un fichier *projet* que **Qt Linguist** peut traiter.

Dans ce fichier `.pro` vous devez préciser tous les fichiers et tous les formulaires que vous voulez traduire. Ce fichier est utilisé pour paramétrer les fichiers et les variables de localisations. Un fichier projet possible, correspondant à la structure de notre :ref:'example plugin<plugin_files_architecture>':

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

Votre extension peut suivre une structure plus complexe, et elle peut être distribuée Si c'est le cas, gardez en tête que `pylupdate4`, le programme que nous avons utilisé pour lire le fichier `.pro` et mettre à jour la chaîne de caractères traduisible, ne développe pas les caractères génériques, vous devez donc placer tous les fichiers explicitement dans le fichier '' .pro''. Votre fichier de projet pourrait ressembler à ceci:

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

En plus, le fichier `your_plugin.py` est celui qui *appelle* tous les menus et sous-menus de votre plugin dans la barre d'outils de QGIS et vous voulez tous les traduire.

Enfin, à l'aide de la variable *TRANSLATIONS*, vous pouvez spécifier les langues que vous souhaitez en traduction.

Warning: Assurez-vous de nommer le fichier `.ts` comme combinaison de `votre_extension` + `langue` + `.ts`, sinon, le chargement de la langue échouera! Utilisez un code en 2 lettres pour votre langue (**it** pour l'Italien; **fr** pour le Français, etc...)

fichier .ts

Une fois le fichier `.pro` créé, vous êtes en capacité de générer les fichiers `.ts` pour les différentes langues de votre extension.

Ouvrez un terminal, allez dans le dossier `votre_extension/i18n` et saisissez:

```
pylupdate4 your_plugin.pro
```

vous devriez voir le(s) fichier(s) `votre_extension_langue.ts`.

Ouvrir le fichier `.ts` avec **Qt Linguist** et commencer à traduire.

fichier .qm

Une fois la traduction de votre extension finie (si certains textes ne sont pas traduits, ils apparaîtront dans la langue originale), vous devez créer le fichier `.qm` (la version compilée du fichier `.ts` qui sera utilisée par QGIS).

Ouvrez un terminal, allez dans le dossier `votre_extension/i18n` et saisissez:

```
lrelease your_plugin.ts
```

maintenant, dans le répertoire `i18n` tu verras les fichiers `ton_plugin.qm`.

13.4.3 Translate using Makefile

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

```
LOCALES = en
```

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

```
LOCALES = en hu
```

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

```
make transup
```

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

```
make transcompile
```

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

13.4.4 Charger le plugin

Afin d'exécuter la version traduite de votre extension, ouvrez QGIS, modifiez la langue (*Préférences* → *Options* → *Langue*) et redémarrez QGIS.

Vous devriez voir votre extension dans la bonne langue.

Warning: Si vous effectuez une modification dans votre extension (nouvelle interface, nouveau menu, etc...) vous devrez à nouveau exécuter les commandes ci-dessus afin de **regénérer** des versions actualisées des fichiers `.ts` et `.qm`.

CHAPTER 14

Infrastructure d'authentification

14.1 Introduction

Les infrastructures d'authentification de la référence utilisatieur peuvent être lu dans le manuel d'utilisateur le paragraphe "*authentication_overview* .

Ce chapitre décrit les les bonnes pratiques de développement pour l'utilisation du système d'authentification.

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

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

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

14.2 Glossaire

Voici quelques définitions des principaux éléments étudiés dans ce chapitre.

Mot de passe principal Mot de passe permettant l'accès et le décryptage des informations stockées dans la base de données d'authentification de QGIS.

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

Base de données d'authentification *Base de données d'authentification*

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

Authentication config *Authentication Configuration*

Méthode d'authentification Une méthode pour s'authentifier. Chaque méthode a son propre protocole utilisé pour accorder le statut 'authentifié'. Chaque méthode est mise à disposition comme une librairie chargée dynamiquement pendant la phase d'initialisation de l'infrastructure d'authentification de QGIS.

14.3 QgsAuthManager the entry point

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

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

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

14.3.1 Init the manager and set the master password

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

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

14.3.2 Populate authdb with a new Authentication Configuration entry

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

```
authcfg = 'fm1s770'
```

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

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

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

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

Available Authentication methods

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

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

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

Populate Authorities

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

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

Manage PKI bundles with QgsPkiBundle

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

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

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

14.3.3 Remove entry from authdb

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

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

14.3.4 Leave authcfg expansion to QgsAuthManager

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

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

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

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

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

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

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

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

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

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

PKI examples with other data providers

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

14.4 Adapt plugins to use Authentication infrastructure

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

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

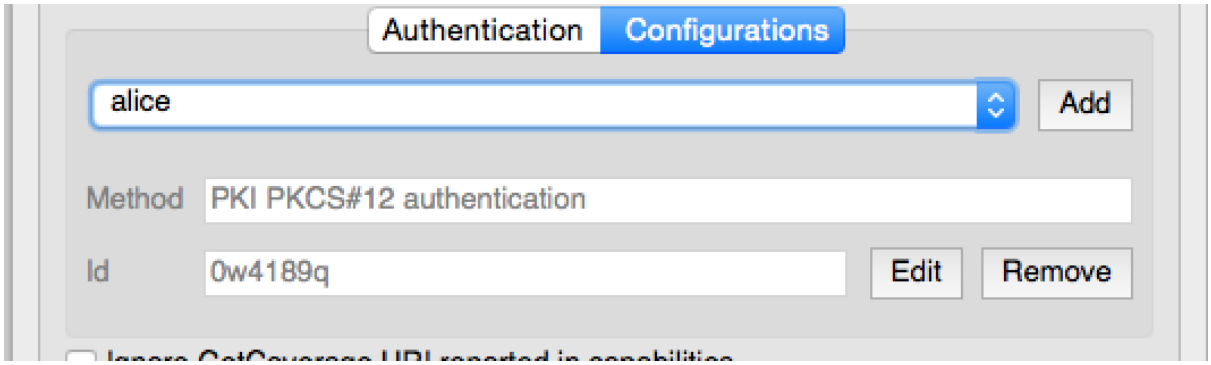
```
http = NetworkAccessManager(authid="my_authCfg", exception_class=My_FailedRequestError)
try:
  response, content = http.request( "my_rest_url" )
except My_FailedRequestError, e:
  # Handle exception
  pass
```

14.5 Authentication GUIs

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

14.5.1 GUI to select credentials

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



and can be used as in the following snippet:

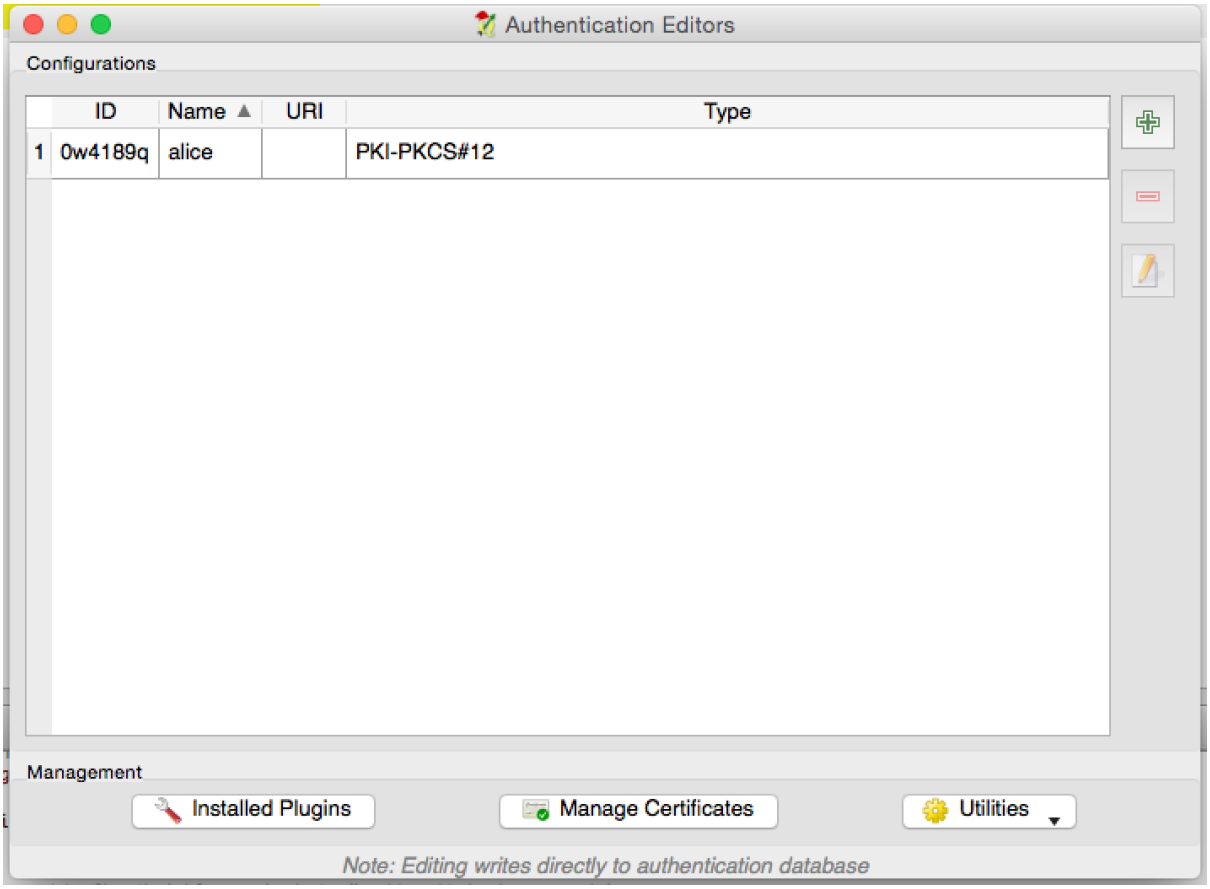
```
# create the instance of the QgsAuthConfigSelect GUI hierarchically linked to
# the widget referred with 'parent'
gui = QgsAuthConfigSelect( parent, "postgres" )
# add the above created gui in a new tab of the interface where the
```

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

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

14.5.2 Authentication Editor GUI

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



and can be used as in the following snippet:

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

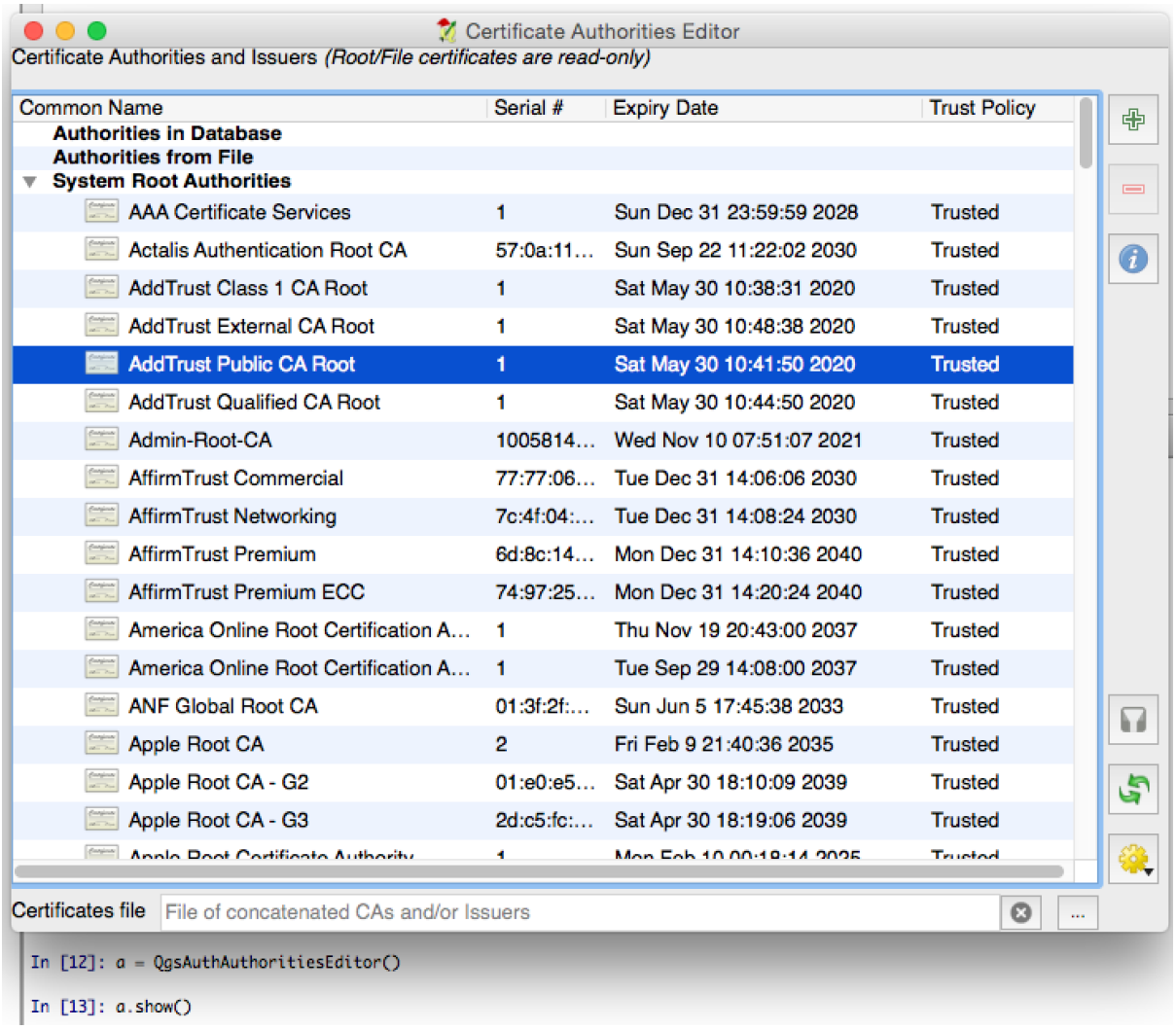
an integrated example can be found in the related test

14.5.3 Authorities Editor GUI

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

and can be used as in the following snippet:

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


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

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

CHAPTER 15

Paramétrage de l'EDI pour la création et le débogage d'extensions

Bien que chaque développeur dispose de son EDI/éditeur de texte préféré, voici quelques recommandations pour paramétrer les EDI populaires pour créer et déboguer des extensions QGIS en Python.

15.1 Note sur la configuration de l'EDI sous Windows

Sous GNU/Linux, il n'y a pas besoin de configuration supplémentaire pour développer des extensions. En revanche, sous Windows, vous devez vous assurer que vous disposez des mêmes variables d'environnement et que vous utilisez les mêmes bibliothèques et interpréteurs que QGIS. Le moyen le plus simple consiste à modifier le fichier batch de démarrage de QGIS.

Si vous avez utilisé l'installateur OSGeo4W, vous pouvez le trouver dans le dossier `bin` de votre installation OSGeo4W. Cherchez quelque chose du genre `C:\OSGeo4W\bin\qgis-unstable.bat`.

Voici ce que vous avez à faire pour utiliser l'IDE Pyscripter:

- Faites une copie du fichier `qgis-unstable.bat` et renommez-le en `pyscripter.bat`.
- Ouvrez-le dans un éditeur et supprimez la dernière ligne, celle qui lance QGIS.
- Ajoutez une ligne qui pointe vers l'exécutable de Pyscripter et ajoutez l'argument de ligne de commande qui paramètre la version de Python à employer (2.7 dans le cas de QGIS >= 2.0).
- Ajoutez également un argument qui pointe vers le répertoire où Pyscripter peut trouver les DLL Python utilisées par QGIS. Vous pouvez le trouver dans le répertoire bin de votre installation OSGeo4W

```
@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
```

Désormais, lorsque vous cliquerez sur ce fichier batch, il lancera Pyscripter avec le chemin correct.

Plus populaire que Pyscripter, Eclipse est un choix courant parmi les développeurs. Dans les sections qui suivent, nous allons expliquer comment le configurer pour le développement et les tests des extensions. Pour préparer votre environnement d'utilisation d'Eclipse sous Windows, vous devriez également créer un fichier batch et l'utiliser pour lancer Eclipse.

Pour créer ce fichier de commandes, suivez ces étapes:

- Trouvez le répertoire où est stocké le fichier `qgis_core.dll`. Normalement, il s'agit de `C:\OSGeo4W\apps\qgis\bin` mais si vous avez compilé votre propre application QGIS, il sera dans votre répertoire de compilation `output/bin/RelWithDebInfo`.
- Localisez votre exécutable `eclipse.exe`.
- Créez le script qui suit et utilisez-le pour démarrer Eclipse lorsque vous développez des extensions QGIS.

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

15.2 Débogage à l'aide d'Eclipse et PyDev

15.2.1 Installation

Afin d'utiliser Eclipse, assurez-vous d'avoir installé

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

15.2.2 Préparation de QGIS

Il faut faire un peu de préparation dans QGIS lui-même. Deux extensions sont nécessaires : **Remote Debug** et **Plugin reloader**.

- Allez dans *Extension* → *Installer/Gérer les extensions*.
- Cherchez l'extension *Remote Debug* (pour l'instant, elle est en version expérimentale et vous devrez donc activer les extensions expérimentales dans l'onglet *Paramètres* pour pouvoir l'afficher). Installez-la.
- Cherchez l'extension *Plugin reloader* et installez-la de la même manière. Elle vous permettra de recharger une extension sans avoir à redémarrer QGIS.

15.2.3 Configuration d'Eclipse

Sous Eclipse, créez un nouveau projet. Vous pouvez choisir *Projet Général* et relier vos sources réels plus tard. L'endroit où vous placez le projet n'est donc pas vraiment important.

Maintenant faites un clic-droit sur votre nouveau projet et choisissez *Nouveau* → *Dossier*.

Cliquez sur **[Avancé]** et choisissez *Lier à un emplacement alternatif (répertoire lié)*. Dans le cas où vous avez déjà des fichiers sources que vous voulez déboguer, choisissez les. Si ce n'est pas le cas, créez un répertoire tel qu'expliqué auparavant.

Désormais, votre arbre de fichiers sources est présent dans la vue *Explorateur de Projet* et vous pouvez commencer à travailler avec le code. Vous pouvez profiter dès maintenant de la coloration syntaxique ainsi que des autres puissants outils de votre EDI.

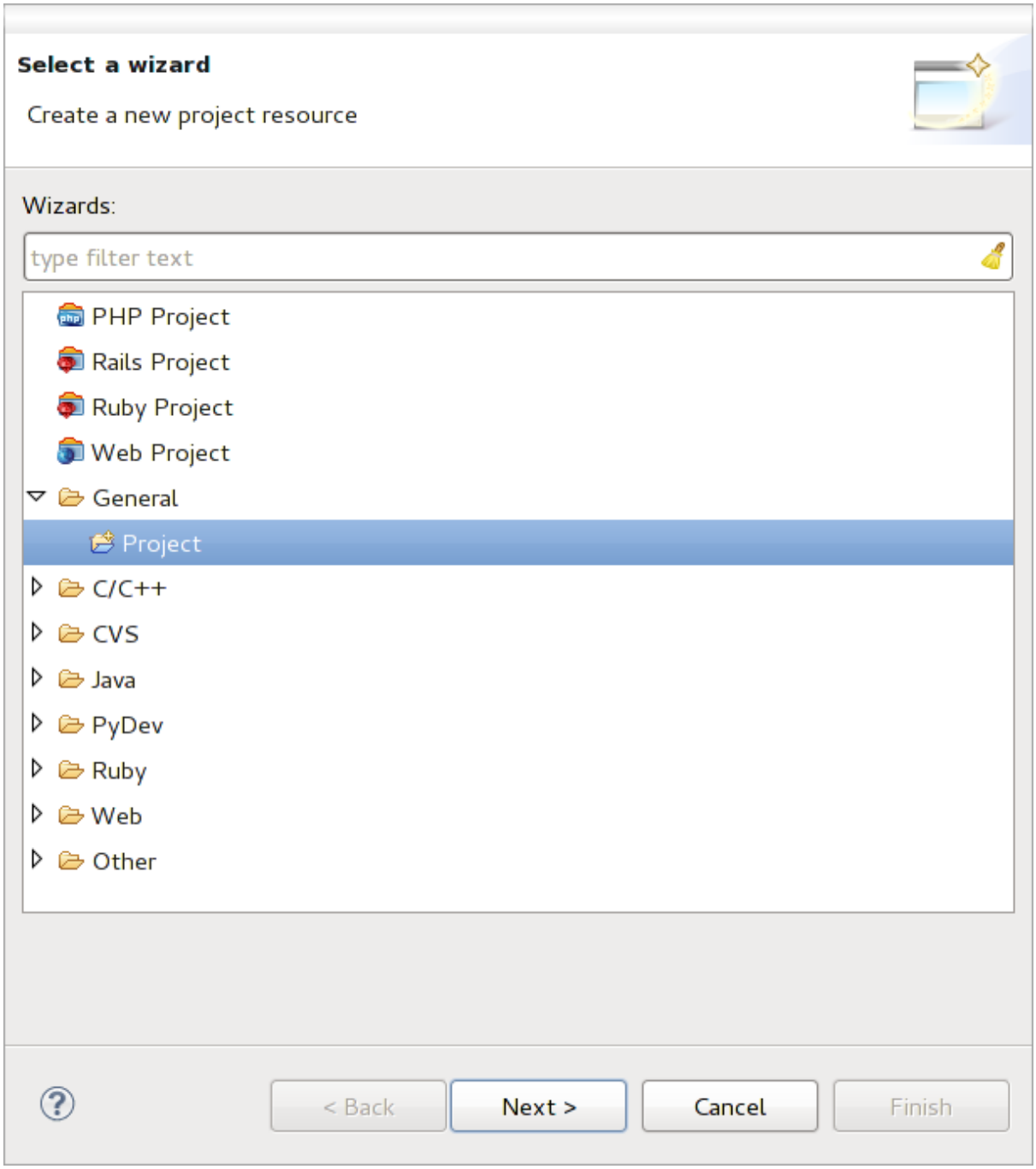


Figure 15.1: Projet Eclipse

15.2.4 Configurer le débogueur

Pour faire fonctionner le débogueur, basculez dans la perspective de Débogage d'Eclipse (*Fenêtre → Ouvrir une perspective → Autre → Debug*).

Maintenant, démarrez le serveur de débogage PyDev en choisissant *PyDev=>Démarrez Serveur de Débogage*.

Eclipse attend maintenant une connexion de QGIS au serveur de débogage. Lorsque QGIS se connectera au serveur de débogage, cela permettra à ce dernier de contrôler les scripts Python. C'est pour cela que nous avons installé l'extension *Remote Debug*. Démarrez QGIS au cas où ce n'est pas déjà fait et cliquez sur le symbole du bogue.

Maintenant, vous pouvez paramétrer un point d'arrêt et, dès que le code y parviendra, son exécution sera stoppée et vous pourrez inspecter l'état courant de votre extension. (Le point d'arrêt est le point vert dans l'image ci-dessous. On peut le marquer en double-cliquant sur l'espace en blanc à gauche de la ligne où vous voulez poser le point d'arrêt).

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

Figure 15.2: Point d'arrêt

Une chose très intéressante que vous pouvez désormais utiliser est la console de débogage. Assurez-vous que l'exécution est parvenue à un point d'arrêt avant de commencer.

Ouvrez la vue Console ((*Fenêtre → Montrer la vue*). Elle montrera la console *Serveur de débogage* ce qui n'est pas très intéressant. Mais il existe un bouton **[Ouvrir Console]** qui vous permet de basculer vers la console de débogage PyDev. Cliquez sur la flèche près de **[Ouvrir Console]** et choisissez *Console PyDev*. Une fenêtre apparaît, vous demandant quelle console vous souhaitez lancer. Choisissez *Console PyDev Debug*. Dans le cas où ce choix est grisé et qu'on vous demande de démarrer le débogueur et de sélectionner le cadre valide, assurez-vous que le débogueur à distance est bien connecté et que vous êtes sur un point d'arrêt.

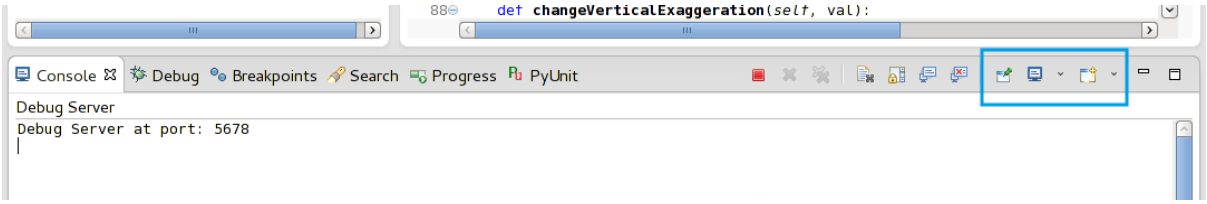


Figure 15.3: Console de débogage de PyDev

vous avez maintenant une console interactive qui vous permet de tester n'importe quelle commande du contexte courant. Vous pouvez manipuler les variables ou lancer des appels à l'API, ou ce que vous voulez.

Un point un peu ennuyeux: chaque fois que vous saisissez une commande, la console bascule vers le Serveur de Débogage. Pour stopper ce comportement, vous pouvez cliquer sur le bouton *Attacher la Console* lorsque vous êtes sur la page du serveur de débogage. Ce choix devrait perdurer tout le long de la session de débogage courante.

15.2.5 Permettre à Eclipse de comprendre l'API

Une fonctionnalité très pratique est de faire en sorte qu'Eclipse tienne compte de l'API de QGIS. Cela vous permet de vérifier les erreurs de syntaxe. Cela permet également à Eclipse de vous aider grâce au complément automatique du code en fonction des appels à l'API.

Pour faire tout cela, Eclipse analyse les fichiers de bibliothèque QGIS et en récupère toute l'information utile. La seule chose que vous avez à faire est de dire à Eclipse où trouver ces bibliothèques.

Cliquez sur *Fenêtre* → *Préférences* → *PyDev* → *Interpreteur* → *Python*.

Vous pourrez observer la configuration de l'interpréteur Python dans la partie supérieure de la fenêtre (pour le moment Python 2.7) ainsi que des onglets dans la partie inférieure. Les onglets qui vous intéressent sont nommés *Bibliothèques* et *Compilation forcée*.

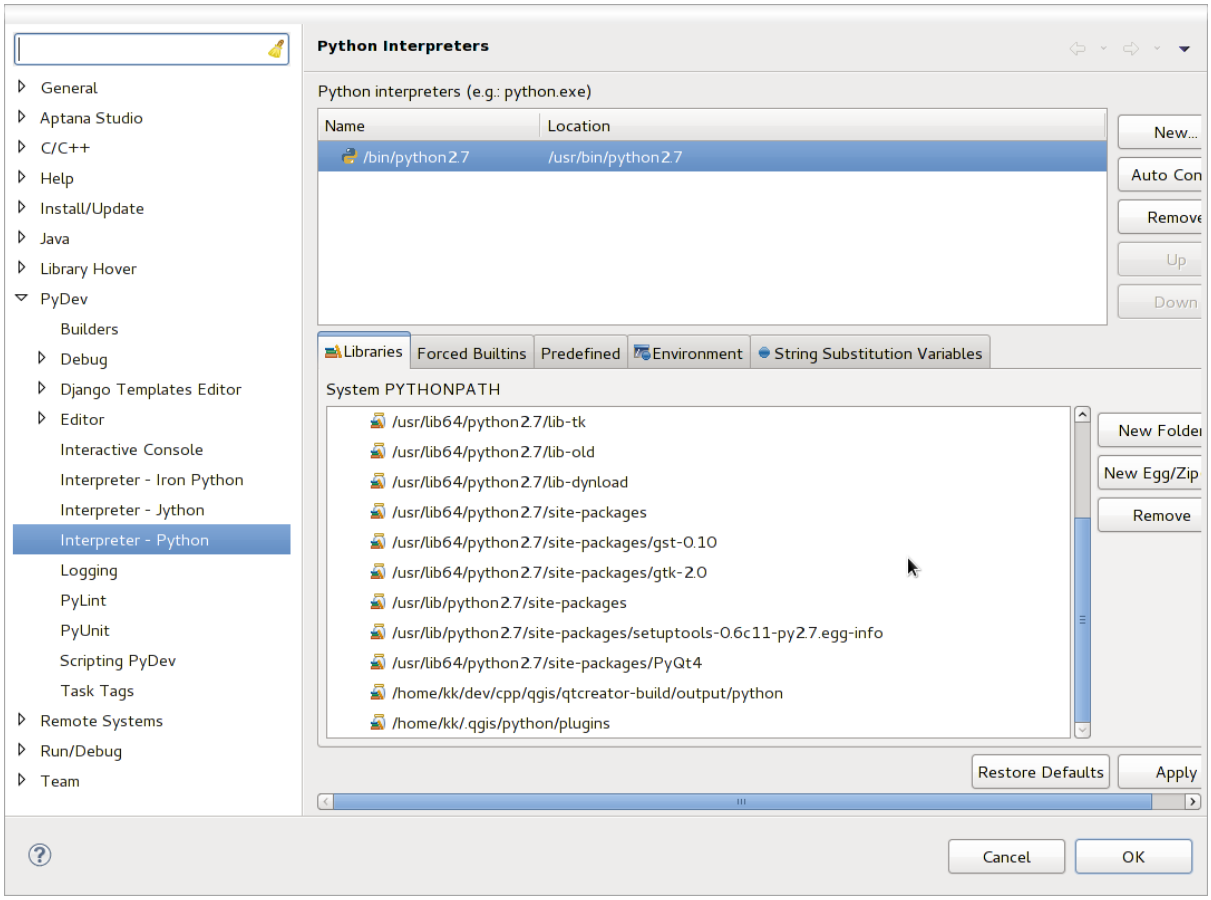


Figure 15.4: Console de débogage de PyDev

Ouvrez d'abord l'onglet Bibliothèques. Ajoutez un nouveau répertoire et choisissez le répertoire Python de votre installation QGIS. Si vous ne savez pas où est situé ce répertoire (il ne s'agit pas du répertoire des extensions), ouvrez QGIS et démarrez une console Python et entrez simplement `qgis` en pressant Entrée. Cela vous montrera quel module QGIS est utilisé ainsi que son chemin. Supprimer la fin du chemin qui contient `/qgis/__init__.pyc` et vous avez l'emplacement que vous cherchez.

Vous devriez également ajouter le répertoire de vos extensions (sous Linux, c'est `~/.qgis2/python/plugins`).

Ensuite, allez dans l'onglet *Compilation forcée*, cliquez sur *Nouveau...* et saisissez `qgis`. Cela permettra à Eclipse d'analyser l'API QGIS. Vous pouvez également ajouter l'API de PyQt4. Il doit sans doute être déjà présent dans votre onglet Bibliothèques.

Cliquer sur *OK* et c'est fini.

Note: Chaque fois que l'API de QGIS évolue (ex: si vous avez compilé la branche master de QGIS et que le fichier sip a changé), vous devriez retourner sur cette page et cliquer simplement sur *Appliquer*. Eclipse se chargera d'analyser toutes les bibliothèques.

15.3 Débogage à l'aide de PDB

Si vous n'utilisez pas d'EDI comme Eclipse, vous pouvez déboguer vos extensions en utilisant PDB et en suivant les étapes qui suivent.

D'abord, ajoutez ce code à l'endroit que vous souhaitez déboguer:

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

Ensuite exécutez QGIS depuis la ligne de commande.

Sur Linux, faites:

```
$ ./Qgis
```

Sur macOS, faites:

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

Lorsque votre application atteint le point d'arrêt, vous pouvez taper des commandes dans la console !

A FAIRE : Ajouter des informations sur les tests

CHAPTER 16

Utiliser une extension de couches

Si votre extension utilise ses propres méthodes pour faire le rendu de la couche cartographique, écrire votre propre type de couche basé sur QgsPluginLayer pourrait être la meilleure façon de l'implémenter.

À FAIRE : Vérifier que ce qui suit est correct et ajouter des détails sur de bons cas d'utilisation de QgsPluginLayer, ...

16.1 Héritage de QgsPluginLayer

Voici un exemple d'implémentation minimaliste d'un QgsPluginLayer. Il est issu d'un extrait de l'extension Watermark

```
class WatermarkPluginLayer(QgsPluginLayer):

  LAYER_TYPE="watermark"

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

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

Des méthodes pour lire et écrire les informations spécifiques du fichier de projet peuvent également être ajoutées :

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

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

Lors du chargement d'un projet contenant une telle couche, une classe "factory" est indispensable :

```
class WatermarkPluginLayerType(QgsPluginLayerType):

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

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

Vous pouvez également ajouter du code pour afficher une information personnalisée dans les propriétés de la couche :

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

CHAPTER 17

Compatibilité avec les versions précédentes de QGIS

17.1 Menu Extension

Si vous placez les entrées de menu de votre extension dans l'un des nouveaux menus (*Raster*, *Vecteur*, *Base de données* ou *Internet*), vous devriez modifier le code des fonctions `initGui()` et `unload()`. Etant donné que ces menus ne sont disponibles qu'à partir de QGIS 2.0, la première étape est de vérifier que la version utilisée de QGIS dispose des fonctions nécessaires. Si les nouveaux menus sont disponibles, votre extension sera placée dans ce menu sinon, le menu *Extension* sera utilisé à la place. Voici un exemple pour le menu *Raster*

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

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

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

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

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

CHAPTER 18

Publier votre extension

Une fois que l'extension est prête et que vous pensez qu'elle pourra être utile à d'autres, n'hésitez pas à la téléverser sur *Dépôt officiel des extensions Python*. Sur cette page, vous pouvez également trouver un guide d'empaquetage sur comment préparer l'extension pour qu'elle fonctionne correctement avec l'installateur d'extensions. Dans le cas où vous souhaitez mettre en place votre propre dépôt d'extensions, créez un unique fichier XML qui listera vos extensions ainsi que leur métadonnées Pour des exemples, consultez les autres dépôts d'extension.

Veuillez noter avec grands soins les suggestions suivantes :

18.1 Métadonnées et noms

- évitez d'utiliser un nom similaire à celui d'une extension existante
- si votre extension présente une fonctionnalité similaire à celle d'une extension existante, veuillez s'il vous plaît en expliquer les différences dans le champ À propos, de sorte que l'utilisateur sache laquelle utiliser sans avoir à l'installer et la tester
- éviter de répéter "extension" dans le nom de l'extension
- utilisez le champ description des métadonnées pour donner une description en 1 ligne et le champ À propos pour faire une description plus détaillée
- inclure un dépôt du code, un suiveur de bogues et une page d'accueil ; cela augmentera significativement les possibilités de collaboration et peut se faire très facilement avec l'une des infrastructures disponibles sur le Web (GitHub, GitLab, Bitbucket, etc.)
- choisissez les mots-clé avec soin : évitez ceux qui ne sont pas informatifs (par ex. vecteur) et préférez ceux qui sont déjà utilisés par d'autres (voir le site Web des extensions)
- ajoutez une icône évocatrice, n'utilisez pas celle par défaut ; voir l'interface QGIS pour des suggestions de styles à utiliser

18.2 Code et aide

- ne fournissez pas les fichiers générés (ui_*.py, resources_rc.py, fichiers d'aide générés...) et les fichiers inutiles (par ex. .gitignore) dans le dépôt
- ajoutez l'extension dans le menu approprié (Vecteur, Raster, Web, Base de données)
- Lorsque c'est possible (l'extension fait des analyses) considérez l'ajout de l'extension en tant qu'extension du module de traitement : cela permettra aux utilisateurs de l'utiliser dans des lots, de l'intégrer dans des flux de traitement complexes et vous évitera d'avoir à créer une interface
- ajoutez au moins une documentation minimale et, si cela est utile pour la tester et la comprendre, un échantillon de données.

18.3 Dépôt officiel des extensions Python

Vous pouvez trouver le dépôt *officiel* des extensions python à http://plugins.qgis.org/.

Afin d'utiliser le dépôt officiel, vous devez détenir un identifiant OSGEO, à partir du portail web OSGEO.

Une fois que vous avez téléversé votre extension, elle sera approuvée par un membre du staff et une notification vous sera adressée.

A FAIRE : Insérer un lien vers le document de gouvernance

18.3.1 Permissions

Ces règles ont été implémentées dans le dépôt officiel des extensions :

- tout utilisateur enregistré peut ajouter une nouvelle extension
- les utilisateurs membres du *staff* sont habilités à approuver ou non chacune des versions de toutes les extensions
- Les utilisateurs qui ont l'autorisation spéciale *plugins.can_approve* ont leurs versions d'extension automatiquement approuvées
- Les utilisateurs ayant l'autorisation spéciale *plugins.can_approve* peuvent approuver les versions téléversées par d'autres, dès lors qu'ils sont dans la liste des *propriétaires* de l'extension
- une extension particulière peut être effacée et éditer uniquement par les utilisateurs de *l'équipe* et par leurs *propriétaires*
- Si un utilisateur ne disposant pas de la permission *plugins.can_approve* téléverse une nouvelle version, cette version de l'extension est automatiquement signalée comme non approuvée.

18.3.2 Gestion de la confiance

Les membres de l'équipe peuvent ajouter un niveau de confiance à certains créateurs d'extension en paramétrant la permission dans la variable *plugins.can_approve* depuis l'application frontale.

La vue détaillée de l'extension montre les liens directs pour modifier le niveau de confiance du créateur d'extension ou des *propriétaires* de l'extension.

18.3.3 Validation

Les métadonnées de l'extension sont importées et validées automatiquement à partir du paquet compressé lorsque l'extension est envoyée.

Voici quelques règles de validation auxquelles vous devriez faire attention quand vous souhaitez charger votre extension sur le dépôt officiel:

1. le nom du répertoire principal de votre extension ne doit contenir que des caractères ASCII (A-Z et a-z), des chiffres et les caractères trait de soulignement (_) et signe moins (-), et il ne peut pas commencer avec un chiffre
2. `metadata.txt` est requis
3. Toutes les métadonnées requises listées dans *metadata table* doivent être présentes.
4. Le champ de métadonnée *version* doit être unique

18.3.4 Structure d'une extension

Le paquet compressé (.zip) de votre extension, suivant les règles de validation, doit avoir une structure spécifique pour être validé en tant qu'extension fonctionnelle. Étant donné que l'extension doit être décompressée à l'intérieur du répertoire des extensions de l'utilisateur, elle doit disposer de sont propre répertoire au sein de l'archive .zip pour ne pas interférer avec les autres extensions. Les fichiers obligatoires sont: `metadata.txt` et `__init__.py`. Il serait également appréciable de fournir un fichier `README` ainsi qu'une icône pour représenter l'extension (`resources.qrc`). Voici à quoi devrait ressembler le contenu d'une archive zip contenant une extension:

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

CHAPTER 19

Extraits de code

Cette section présente des extraits de code pour faciliter le développement d'extensions.

19.1 Comment appeler une méthode à l'aide d'un raccourci clavier

Ajoutez ce qui suit à la méthode `initGui()` de l'extension:

```
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)
```

Pour décharger l'extension, ajoutez ce qui suit à la méthode `unload()` de l'extension:

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

La méthode est appelée lors d'un appui sur F7:

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

19.2 Comment activer des couches:

Depuis QGIS 2.4, il existe une nouvelle API d'arbre de couches qui permet un accès direct à l'arbre des couches de la légende. Voici un exemple qui présente une méthode pour activer la visibilité d'une couche active:

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

19.3 Comment accéder à la table attributaire des entités sélectionnées

```
def changeValue(self, value):
  layer = self.iface.activeLayer()
  if(layer):
    nF = layer.selectedFeatureCount()
    if (nF > 0):
      layer.startEditing()
      ob = layer.selectedFeaturesIds()
      b = QVariant(value)
      if (nF > 1):
        for i in ob:
        layer.changeAttributeValue(int(i), 1, b) # 1 being the second column
      else:
        layer.changeAttributeValue(int(ob[0]), 1, b) # 1 being the second column
      layer.commitChanges()
    else:
      QMessageBox.critical(self.iface.mainWindow(), "Error",
        "Please select at least one feature from current layer")
  else:
    QMessageBox.critical(self.iface.mainWindow(), "Error", "Please select a layer")
```

La méthode utilise un paramètre (la nouvelle valeur du champ d'attribut de l'entité sélectionnée) et elle peut être appelée de la manière suivante:

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

CHAPTER 20

Créer une extensions Processing

Suivant le type d'extension que vous voulez développer, il sera parfois plus judicieux d'ajouter sa fonctionnalité sous forme d'un algorithme Processing (ou un ensemble d'algorithmes). Cela vous apportera une meilleure intégration au sein de QGIS, des fonctionnalités supplémentaires (puisqu'elle pourra être également lancée dans les composants de Processing comme le modeleur ou l'interface de traitements par lots), ainsi qu'un temps de développement plus court (puisque Processing va gérer une grande partie du travail).

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

There are two main mechanisms for doing that:

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

20.1 Creating a plugin that adds an algorithm provider

To create an algorithm provider, follow these steps:

- Installez l'extension Plugin Builder
- Créez une nouvelle extension à l'aide de Plugin Builder. Lorsque l'application vous demande le modèle à utiliser, sélectionnez "Processing Provider".
- L'extension créée contient un fournisseur disposant d'un seul algorithme. Les fichiers du fournisseur et de l'algorithme sont correctement commentés et contiennent de l'information sur comment modifier le fournisseur et comment ajouter de nouveaux algorithmes. S'y référerer pour plus d'informations.

20.2 Creating a plugin that contains a set of processing scripts

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

- Create your scripts as described in the PyQGIS cookbook. All the scripts that you want to add, you should have them available in the Processing toolbox.
- In the *Scripts/Tools* group in the Processing toolbox, double-click on the *Create script collection plugin* item. You will see a window where you should select the scripts to add to the plugin (from the set of available ones in the toolbox), and some additional information needed for the plugin metadata.
- Click on OK and the plugin will be created.

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

CHAPTER 21

Bibliothèque d'analyse de réseau

Depuis la révision ee19294562 (QGIS >= 1.8), la nouvelle bibliothèque d'analyse de réseau a été ajoutée à la bibliothèque principale d'analyse de QGIS. La bibliothèque :

- créé un graphe mathématique à partir de données géographiques (couches vecteurs de polylignes)
- implémente des méthodes simples de la théorie des graphes (pour l'instant, uniquement avec l'algorithme Dijkstra).

La bibliothèque d'analyse de réseau a été créée en exportant les fonctions de l'extension principale RoadGraph. Vous pouvez en utiliser les méthodes dans des extensions ou directement dans la console Python.

21.1 Information générale

Voici un résumé d'un cas d'utilisation typique:

1. créer un graphe depuis les données géographiques (en utilisant une couche vecteur de polylignes)
2. lancer une analyse de graphe
3. utiliser les résultats d'analyse (pour les visualiser par exemple)

21.2 Construire un graphe

La première chose à faire est de préparer les données d'entrée, c'est à dire de convertir une couche vecteur en graphe. Les actions suivantes utiliseront ce graphe et non la couche.

Comme source de données, on peut utiliser n'importe quelle couche vecteur de polylignes. Les nœuds des polylignes deviendront les sommets du graphe et les segments des polylignes seront les arcs du graphes. Si plusieurs nœuds ont les mêmes coordonnées alors ils composent le même sommet de graphe. Ainsi, deux lignes qui ont en commun un même nœud sont connectées ensemble.

Pendant la création d'un graphe, il est possible de "forcer" ("lier") l'ajout d'un ou de plusieurs points additionnels à la couche vecteur d'entrée. Pour chaque point additionnel, un lien sera créé: le sommet du graphe le plus proche ou l'arc de graphe le plus proche. Dans le cas final, l'arc sera séparé en deux et un nouveau sommet sera ajouté.

Les attributs de la couche vecteur et la longueur d'un segment peuvent être utilisés comme propriétés du segment.

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.

Il est temps de plonger dans le processus.

D'abord, nous devrions importer le module networkanalysis pour utiliser la bibliothèque

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

Ensuite, quelques exemples pour créer un directeur

```
# 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)
```

Pour construire un directeur, il faut lui fournir une couche vecteur qui sera utilisée comme source pour la structure du graphe ainsi que des informations sur les mouvements permis sur chaque segment de route (sens unique ou déplacement bidirectionnel, direct ou inversé). L'appel au directeur se fait de la manière suivante

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

Voici la liste complète de la signification de ces paramètres:

- `vl` — couche vecteur utilisée pour construire le graphe
- `directionFieldId` — index du champ de la table d'attribut où est stocké l'information sur la direction de la route. Si `-1` est utilisé, cette information n'est pas utilisée. Un entier.
- `directDirectionValue` — valeur du champ utilisé pour les routes avec une direction directe (déplacement du premier point de la ligne au dernier). Une chaîne de caractères.
- `reverseDirectionValue` — valeur du champ utilisé pour les routes avec une direction inverse (déplacement du dernier point de la ligne au premier). Une chaîne de caractères.
- `bothDirectionValue` — valeur du champ utilisé pour les routes bidirectionelles (pour ces routes, on peut se déplacer du premier point au dernier et du dernier au premier). Une chaîne de caractères.
- `defaultDirection` — direction par défaut de la route. Cette valeur sera utilisée pour les routes où le champ `directionFieldId` n'est pas paramétré ou qui a une valeur différente des trois valeurs précédentes. Un entier `1` indique une direction directe, `2` indique une direction inverse et `3` indique les deux directions.

Il est ensuite impératif de créer une stratégie de calcul des propriétés des arcs:

```
properter = QgsDistanceArcProperter()
```

Et d'informer le directeur à propos de cette stratégie:

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

Nous pouvons maintenant utiliser le constructeur qui créera le graphe. Le constructeur de la classe QgsGraphBuilder utilise plusieurs arguments:

- crs — système de coordonnées de référence à utiliser. Argument obligatoire.
- otfEnabled — utiliser ou non la projection "à la volée". La valeur par défaut est const:*True* (oui, utiliser OTF).
- topologyTolerance — la tolérance topologique. La valeur par défaut est 0.
- ellipsoidID — ellipsoïde à utiliser. Par défaut "WGS84".

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

Nous pouvons également définir plusieurs points qui seront utilisés dans l'analyse, par exemple:

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

Maintenant que tout est en place, nous pouvons construire le graphe et lier ces points dessus:

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

La construction du graphe peut prendre du temps (qui dépend du nombre d'entités dans la couche et de la taille de la couche). `tiedPoints` est une liste qui contient les coordonnées des points liés. Lorsque l'opération de construction est terminée, nous pouvons récupérer le graphe et l'utiliser pour l'analyse:

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

Avec le code qui suit, nous pouvons récupérer les index des arcs de nos points:

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

21.3 Analyse de graphe

L'analyse de graphe est utilisée pour trouver des réponses aux deux questions: quels arcs sont connectés et comment trouver le plus court chemin ? Pour résoudre ces problèmes la bibliothèque d'analyse de graphe fournit l'algorithme de Dijkstra.

L'algorithme de Dijkstra trouve le plus court chemin entre un des arcs du graphe par rapport à tous les autres en tenant compte des paramètres d'optimisation. Ces résultats peuvent être représentés comme un arbre du chemin le plus court.

L'arbre du plus court chemin est un graphe pondéré de direction (plus précisément un arbre) qui dispose des propriétés suivantes:

- Seul un arc n'a pas d'arcs entrants: la racine de l'arbre.
- Tous les autres arcs n'ont qu'un seul arc entrant.
- Si un arc B est atteignable depuis l'arc A alors le chemin de A vers B est le seul chemin disponible et il est le chemin optimal (le plus court) sur ce graphe.

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.

La méthode `shortestTree()` est utile lorsque vous voulez approcher l'arbre du chemin le plus court. Elle créé toujours un nouvel objet de graphe (QgsGraph) et elle accepte trois variables:

- source — graphe en entrée

- startVertexIdx — index du point sur l'arbre (la racine de l'arbre)
- criterionNum — nombre de propriétés d'arc à utiliser (en partant de 0).

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

La méthode `dijkstra()` dispose des mêmes arguments mais retourne deux tableaux. Dans le premier, l'élément i contient l'index de l'arc à suivre ou -1 s'il n'y a pas d'arc à suivre. Dans le second tableau, l'élément i contient la distance depuis la racine de l'arbre jusqu'au sommet i ou la valeur DOUBLE_MAX si le sommet est inaccessible depuis la racine.

```
(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
```

Même chose mais en utilisant la méthode `dijkstra()`.

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

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

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

```
builder = QgsGraphBuilder(crs)

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

graph = builder.graph()

idStart = graph.findVertex(pStart)

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

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

21.3.1 Trouver les chemins les plus courts

Pour trouver le chemin optimal entre deux points, on peut utiliser l'approche suivante. Les deux points (départ en A et arrivée en B) sont "liés" au graphe lors de sa construction. En utilisant les méthodes `shortestTree()` ou `dijkstra()`, nous construisons alors l'arbre du chemin le plus court avec une racine qui démarre par le point A. Dans le même arbre, nous trouvons notre point B et commençons à traverser l'arbre du point B vers le point A. L'algorithme complet peut être écrit de la façon suivante

```
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
```

A ce niveau, nous avons le chemin, sous la forme d'une liste inversée d'arcs (les arcs sont listés dans un ordre inversé, depuis le point de la fin vers le point de démarrage) qui seront traversés lors de l'évolution sur le chemin.

Voici le code d'exemple pour la console Python de QGIS qui utilise la méthode `shortestTree()` (vous devrez sélectionner la couche de polylignes dans la légende et remplacer les coordonnées dans le code par les vôtres):

```
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)
```

Et voici le même exemple mais avec la méthode `dijkstra()`:

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

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

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

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

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

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

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

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

if tree[idStop] == -1:
  print "Path not found"
else:
  p = []
  curPos = idStop
```

```
  while curPos != idStart:
    p.append(graph.vertex(graph.arc(tree[curPos]).inVertex()).point())
    curPos = graph.arc(tree[curPos]).outVertex();

  p.append(tStart)

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

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

21.3.2 Surfaces de disponibilité

La surface de disponibilité d'un arc A est le sous-ensemble des arcs du graphe qui sont accessibles à partir de l'arc A et où le coût des chemins à partir de A vers ces arcs ne dépasse pas une certaine valeur.

Plus clairement, cela peut être illustré par l'exemple suivant: "Il y a une caserne de pompiers. Quelles parties de la ville peuvent être atteintes par un camion de pompier en 5 minutes ? 10 minutes ? 15 minutes ?" La réponse à ces questions correspond aux surface de disponibilité de la caserne de pompiers.

Pour trouver les surfaces de disponibilité, nous pouvons utiliser la méthode `dijkstra()` de la classe `QgsGraphAnalyzer`. Elle suffit à comparer les éléments du tableau de coût avec une valeur prédéfinie. si le coût[i] est inférieur ou égal à la valeur prédéfinie, alors l'arc i est à l'intérieur de la surface de disponibilité, sinon il est situé en dehors.

Un problème plus difficile à régler est d'obtenir les frontières de la surface de disponibilité. La frontière inférieure est constituée par l'ensemble des arcs qui sont toujours accessibles et la frontière supérieure est composée des arcs qui ne sont pas accessibles. En fait, c'est très simple: c'est la limite de disponibilité des arcs de l'arbre du plus court chemin pour lesquels l'arc source de l'arc est accessible et l'arc cible ne l'est pas.

Voici un exemple:

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

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

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

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

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

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

idStart = graph.findVertex(tStart)
```

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

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

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

CHAPTER 22

Extensions Python pour QGIS Server

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

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

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

22.1 Architecture des extensions de filtre serveur

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()`

Tous les filtres ont accès à l'objet de requête/réponse (`QgsRequestHandler`) et peuvent manipuler toutes les propriétés (entrée/sortie) et déclencher des exceptions (à l'aide d'une méthode un peu particulière comme nous le verrons ci-dessous).

Voici un pseudo-code présentant une session serveur typique et quand les fonctions de retour des filtres sont appelées:

- **Récupérer la requête entrante**
  - Créer un gestionnaire de requête GET/POST/SOAP.
  - Passer la requête à une instance de la classe `QgsServerInterface`.
  - Appeler la fonction `requestReady()` des filtres d'extension.
  - **S'il n'y a pas de réponse**
    - **Si SERVICE vaut WMS/WFS/WCS.**
      - **Créer un serveur WMS/WFS/WCS.**

        Appeler la fonction du serveur `executeRequest()` et appeler la fonction des filtres d'extension `sendResponse()` lors de l'envoi du flux vers la sortie ou alors conserver le flux binaire et le type de contenu dans le gestionnaire de requête.
    - Appeler la fonction `responseComplete()` des filtres d'extension.
  - Appeler la fonction `sendResponse()` des filtres d'extension.
  - Demander au gestionnaire d'émettre la réponse.

Les paragraphes qui suivent décrivent les fonctions de rappel disponibles en détails.

22.1.1 requestReady

Cette fonction est appelée lorsque la requête est prêt: l'URL entrante et ses données ont été analysées et juste avant de passer la main aux services principaux (WMS, WFS, etc.), c'est le point où vous pouvez manipuler l'entrée et dérouler des actions telles que:

- l'authentification/l'autorisation
- les redirections
- l'ajout/suppression de certains paramètres (les noms de type par exemple)
- le déclenchement d'exceptions

Vous pouvez également substituer l'intégralité d'un service principal en modifiant le paramètre **SERVICE** et complètement outrepasser le service (ce qui n'a pas beaucoup d'intérêt).

22.1.2 sendResponse

Cette fonction est appelée lorsque la sortie est envoyée vers la sortie **FCGI** ‘‘stdout‘‘(et ainsi, vers le client); cette action est habituellement réalisée après la fin du traitement des services principaux et après que le signal responseComplete ait été appelé. Dans un certain nombre de rares cas, le contenu XML peut devenir si volumineux qu'il a fallu implémenter une gestion de flux XML (GetFeature pour WFS est l'une d'entre elles) et dans ce cas, `sendResponse()` est appelée plusieurs fois avant que la réponse soit complète (et avant que `responseComplete()` soit appelée). La conséquence naturelle est que `sendResponse()` est normalement appelée une fois mais peut, exceptionnellement, être appelée plusieurs fois et dans ce cas (et uniquement dans ce cas), elle est appelée avant `responseComplete()`.

`sendResponse()` est le meilleur moment pour la manipulation directe des sorties des services principaux et alors que la fonction `responseComplete()` est généralement une option, `sendResponse()` est la seule option viable pour le cas des services en flux.

22.1.3 responseComplete

Cette fonction est appelée lorsque les services principaux (si lancés) terminent leur processus et que la requête est prête à être envoyée au client. Comme indiqué ci-dessus, elle est normalement appelée avant `sendResponse()` sauf pour les services en flux (ou les filtres d'extension) qui peuvent avoir lancé `sendResponse()` plus tôt.

`responseComplete()` est le moment adéquat pour fournir de nouvelles implémentations de service (WPS ou services personnalisés) et pour effectuer des manipulations directes de la sortie des services principaux (comme par exemple pour ajouter un filigrane sur une image WMS).

22.2 Déclencher une exception depuis une extension

Il reste encore du travail sur ce sujet: l'implémentation actuelle gère les exceptions gérées ou non en paramétrant la propriété `QgsRequestHandler` avec une instance de `QgsMapServiceException`. De cette manière, le code C++ peut capturer les exceptions Python gérées et ignorer les exceptions non gérées (ou mieux, les journaliser).

Cette approche fonctionne globalement mais elle n'est pas très "pythonesque": une meilleure approche consisterait à déclencher des exceptions depuis le code Python et les faire remonter dans la boucle principale C++ pour y être traitées.

22.3 Écriture d'une extension serveur

A server plugins is just a standard QGIS Python plugin as described in *Développer des extensions 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`.

Pour indiquer à QGIS Server qu'une extension dispose d'une interface serveur, une métadonnée spécifique est requise (dans *metadata.txt*)

```
server=True
```

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

22.3.1 Fichiers de l'extension

Vous pouvez voir ici la structure du répertoire de notre exemple d'extension pour serveur

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

22.3.2 __init__.py

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

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

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

22.3.3 HelloServer.py

C'est l'endroit où tout se passe et voici à quoi il devrait ressembler : (ex. `HelloServer.py`)

Une extension côté serveur consiste typiquement en une ou plusieurs fonctions de rappel empaquetées sous forme d'objets appelés QgsServerFilter.

Chaque `QgsServerFilter` implémente une ou plusieurs des fonctions de rappel suivantes:

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

L'exemple qui suit implémente un filtre minimaliste qui affiche *HelloServer!* pour le cas où le paramètre **SERVICE** vaut "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!')
```

Les filtres doivent être référencés dans la variable **serverIface** comme indiqué dans l'exemple suivant:

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

Le second paramètre de `registerFilter()` permet de définir une priorité indiquant l'ordre des fonctions de rappel ayant le même nom (une priorité faible est invoquée en premier).

En utilisant les trois fonctions de rappel, les extensions peuvent manipuler l'entrée et/ou la sortie du serveur de plusieurs manières. A chaque instant, l'instance de l'extension a accès à la classe `QgsRequestHandler` au travers de la classe `QgsServerInterface`, `QgsRequestHandler` dispose de nombreuses méthodes qui peuvent être utilisée pour modifier les paramètres d'entrée avant qu'ils intègrent le processus principal du serveur (à l'aide de `requestReady()`) ou après que la requête ait été traitée par les services principaux (en utilisant `sendResponse()`).

Les exemples suivants montrent quelques cas d'utilisation courants :

22.3.4 Modifier la couche en entrée

L'extension d'exemple contient un test qui modifie les paramètres d'entrée provenant de la requête; dans cet exemple, un nouveau paramètre est injecté dans *parameterMap* (qui est déjà analysé), ce paramètre est alors visible dans les services principaux (WMS, etc.), à la fin du processus des services principaux, nous vérifions que le paramètre est toujours présent:

```
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
```

Ceci est un extrait de ce que vous pouvez voir dans le fichier log:

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

Sur la ligne en surbrillance, la chaîne "SUCCESS" indique que le plugin a réussi le test.

La même technique peut être employée pour utiliser un service personnalisé à la place d'un service principal: vous pouviez par exemple sauter une requête **WFS SERVICE** ou n'importe quelle requête principale en modifiant le paramètre **SERVICE** par quelque-chose de différent et le service principal ne serait alors pas lancé; vous pourriez ensuite injecter vos resultats personnalisés dans la sortie et les renvoyer au client (ceci est expliqué ci-dessous).

22.3.5 Modifier ou remplacer la couche en sortie

L'exemple du filtre de filigrane montre comment remplacer la sortie WMS avec une nouvelle image obtenue par l'ajout d'un filigrane plaqué sur l'image WMS générée par le service principal WMS:

```
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)
```

Dans cet exemple, la valeur du paramètre **SERVICE** est vérifiée. Si la requête entrante est de type **WMS GETMAP** et qu'aucune exception n'a été déclarée par une extension déjà déclarée ou par un service principal (WMS dans notre cas), l'image WMS générée est récupérée depuis le tampon de sortie et l'image du filigrane est ajoutée. L'étape finale consiste à vider le tampon de sortie et à le remplacer par l'image nouvellement générée. Merci de prendre note que dans une situation réelle, nous devrions également vérifier le type d'image requêtée au lieu de retourner du PNG quoiqu'il arrive.

22.4 Extension de contrôle d'accès

22.4.1 Fichiers de l'extension

Voici l'arborescence de notre exemple d'extension serveur:

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

22.4.2 __init__.py

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

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

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

22.4.3 AccessControl.py

```
class AccessControl(QgsAccessControlFilter):

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

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

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

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

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

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

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

Cet exemple donne un accès total à tout le monde.

C'est le rôle de l'extension de connaître qui est connecté dessus.

Pour toutes ces méthodes nous avons la couche passée en argument afin de personnaliser la restriction par couche.

22.4.4 layerFilterExpression

Utilisé pour ajouter une expression pour limiter les résultats, ex:

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

Pour limiter aux entités où l'attribut role vaut "user".

22.4.5 layerFilterSubsetString

Comme le point précédent mais utilise `SubsetString` (exécuté au niveau de la base de données).

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

Pour limiter aux entités où l'attribut role vaut "user".

22.4.6 layerPermissions

Limiter l'accès à la couche.

Renvoie un objet de type `QgsAccessControlFilter.LayerPermissions` qui dispose des propriétés suivantes:

- `canRead` pour autoriser l'affichage dans les requêtes `GetCapabilities` et pour permettre l'accès en lecture.
- `canInsert` pour autoriser l'insertion de nouvelles entités.
- `canUpdate` pour autoriser les mises à jour d'entités.
- `candelete` pour autoriser les suppressions d'entités.

Exemple :

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

Pour tout limiter à un accès en lecture seule.

22.4.7 authorizedLayerAttributes

Utilisé pour limiter la visibilité d'un sous-groupe d'attribut spécifique.

L'argument attributes renvoie la liste des attributs réellement visibles.

Exemple :

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

Cache l'attribut 'role'.

22.4.8 allowToEdit

Il permet de limiter l'édition à un sous-ensemble d'entités.

Il est utilisé dans le protocole `WFS-Transaction`.

Exemple :

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

Pour limiter l'édition aux entités dont l'attribut role contient la valeur user.

22.4.9 cacheKey

QGIS Server conserve un cache du capabilties donc pour avoir un cache par rôle vous pouvez retourner le rôle dans cette méthode. Ou retourner `None` pour complètement désactiver le cache.