

---

# **PyQGIS developer cookbook**

*Version 2.18*

**QGIS Project**

16 February 2019



<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Exécuter un programme Python au démarrage de QGIS . . . . .	2
1.2	La Console Python . . . . .	2
1.3	Extensions Python . . . . .	3
1.4	Applications Python . . . . .	3
<b>2</b>	<b>Chargement de projets</b>	<b>7</b>
<b>3</b>	<b>Chargement de couches</b>	<b>9</b>
3.1	Couches vectorielles . . . . .	9
3.2	Couches raster . . . . .	11
3.3	Registre de couches cartographiques . . . . .	12
<b>4</b>	<b>Utiliser des couches raster</b>	<b>13</b>
4.1	Détails d'une couche . . . . .	13
4.2	Moteur de rendu . . . . .	13
4.3	Rafraîchir les couches . . . . .	15
4.4	Interrogation des données . . . . .	15
<b>5</b>	<b>Utilisation de couches vectorielles</b>	<b>17</b>
5.1	Récupérer les informations relatives aux attributs . . . . .	17
5.2	Sélection des entités . . . . .	18
5.3	Itérer sur une couche vecteur . . . . .	18
5.4	Modifier des couches vecteur . . . . .	20
5.5	Modifier des couches vecteur à l'aide d'un tampon d'édition . . . . .	22
5.6	Utilisation des index spatiaux . . . . .	23
5.7	Ecrire dans des couches vecteur . . . . .	24
5.8	Fournisseur de données en mémoire . . . . .	25
5.9	Apparence (Symbologie) des couches vecteur . . . . .	26
5.10	Sujets complémentaires . . . . .	33
<b>6</b>	<b>Manipulation de la géométrie</b>	<b>35</b>
6.1	Construction de géométrie . . . . .	35
6.2	Accéder à la Géométrie . . . . .	36
6.3	Prédicats et opérations géométriques . . . . .	36
<b>7</b>	<b>Support de projections</b>	<b>39</b>
7.1	Système de coordonnées de référence . . . . .	39
7.2	Projections . . . . .	40
<b>8</b>	<b>Utiliser le Canevas de carte</b>	<b>41</b>
8.1	Intégrer un canevas de carte . . . . .	41
8.2	Utiliser les outils cartographiques avec le canevas . . . . .	42

8.3	Contour d'édition et symboles de sommets . . . . .	43
8.4	Ecrire des outils cartographiques personnalisés . . . . .	44
8.5	Ecrire des éléments de canevas de carte personnalisés . . . . .	45
<b>9</b>	<b>Rendu cartographique et Impression</b>	<b>47</b>
9.1	Rendu simple . . . . .	47
9.2	Rendu des couches ayant différents SCR . . . . .	48
9.3	Sortie utilisant un composeur de carte . . . . .	48
<b>10</b>	<b>Expressions, Filtrage et Calcul de valeurs</b>	<b>51</b>
10.1	Analyse syntaxique d'expressions . . . . .	52
10.2	Évaluation des expressions . . . . .	52
10.3	Exemples . . . . .	53
<b>11</b>	<b>Lecture et sauvegarde de configurations</b>	<b>55</b>
<b>12</b>	<b>Communiquer avec l'utilisateur</b>	<b>57</b>
12.1	Afficher des messages: La classe <code>QgsMessageBar</code> . . . . .	57
12.2	Afficher la progression . . . . .	58
12.3	Journal . . . . .	59
<b>13</b>	<b>Développer des extensions Python</b>	<b>61</b>
13.1	Écriture d'une extension . . . . .	62
13.2	Contenu de l'extension . . . . .	63
13.3	Documentation . . . . .	67
13.4	Traduction . . . . .	67
<b>14</b>	<b>Infrastructure d'authentification</b>	<b>71</b>
14.1	Introduction . . . . .	71
14.2	Glossaire . . . . .	71
14.3	<code>QgsAuthManager</code> the entry point . . . . .	72
14.4	Adapt plugins to use Authentication infrastructure . . . . .	75
14.5	Authentication GUIs . . . . .	75
<b>15</b>	<b>Paramétrage de l'EDI pour la création et le débogage d'extensions</b>	<b>79</b>
15.1	Note sur la configuration de l'EDI sous Windows . . . . .	79
15.2	Débogage à l'aide d'Eclipse et PyDev . . . . .	80
15.3	Débogage à l'aide de PDB . . . . .	84
<b>16</b>	<b>Utiliser une extension de couches</b>	<b>85</b>
16.1	Héritage de <code>QgsPluginLayer</code> . . . . .	85
<b>17</b>	<b>Compatibilité avec les versions précédentes de QGIS</b>	<b>87</b>
17.1	Menu Extension . . . . .	87
<b>18</b>	<b>Publier votre extension</b>	<b>89</b>
18.1	Métadonnées et noms . . . . .	89
18.2	Code et aide . . . . .	90
18.3	Dépôt officiel des extensions Python . . . . .	90
<b>19</b>	<b>Extraits de code</b>	<b>93</b>
19.1	Comment appeler une méthode à l'aide d'un raccourci clavier . . . . .	93
19.2	Comment activer des couches: . . . . .	93
19.3	Comment accéder à la table attributaire des entités sélectionnées . . . . .	94
<b>20</b>	<b>Créer une extensions Processing</b>	<b>95</b>
20.1	Créer une extension qui ajoute un fournisseur d'algorithmes . . . . .	95
20.2	Créer une extension qui contient un jeu de scripts de traitements . . . . .	95

<b>21 Bibliothèque d'analyse de réseau</b>	<b>97</b>
21.1 Information générale . . . . .	97
21.2 Construire un graphe . . . . .	97
21.3 Analyse de graphe . . . . .	99
<b>22 Extensions Python pour QGIS Server</b>	<b>105</b>
22.1 Architecture des extensions de filtre serveur . . . . .	105
22.2 Déclencher une exception depuis une extension . . . . .	107
22.3 Écriture d'une extension serveur . . . . .	107
22.4 Extension de contrôle d'accès . . . . .	110
<b>Index</b>	<b>115</b>



---

## Introduction

---

- Exécuter un programme Python au démarrage de QGIS
  - Variables d’environnement PYQGIS\_STARTUP
  - Le fichier : `startup.py`
- La Console Python
- Extensions Python
- Applications Python
  - Utiliser PyQGIS dans des scripts indépendants
  - Utiliser PyQGIS dans une application personnalisée
  - Exécuter des applications personnalisées

Ce document est à la fois un tutoriel et un guide de référence. Il ne liste pas tous les cas d’utilisation possibles, mais donne une bonne idée générale des principales fonctionnalités.

Dès la version 0.9, QGIS intégrait un support optionnel pour le langage Python. Nous avons choisi Python car c’est un des langages les plus adaptés pour la création de scripts. Les dépendances PyQGIS proviennent de SIP et PyQt4. Le choix de l’utilisation de SIP plutôt que de SWIG plus généralement répandu est dû au fait que le noyau de QGIS dépend des bibliothèques Qt. Les dépendances Python pour Qt (PyQt) sont opérées via SIP, ce qui permet une intégration parfaite de PyQGIS avec PyQt.

Il y a de nombreuses façons d’utiliser les dépendances Python de QGIS; elles sont mentionnées en détail dans les sections suivantes :

- exécuter automatiquement un programme Python quand QGIS démarre
- lancer des commandes dans la console Python de QGIS
- créer et utiliser des plugins en Python
- créer des applications personnalisées basées sur l’API QGIS

Les dépendances Python sont aussi disponibles pour QGIS Server:

- à partir de la version 2.8, les extensions Python sont également disponibles pour QGIS Server (see: [Server Python Plugins](#))
- depuis la version 2.11 (Master du 11-08-2015), la bibliothèque QGIS Server possède des dépendances Python qui peuvent être utilisées pour intégrer QGIS Server dans une application Python

Il y a une documentation [complète de l’API QGIS](#) qui détaille les classes des bibliothèques QGIS. L’API Python de QGIS est presque identique à l’API en C++.

Pour vous familiariser avec les extensions, vous pouvez en télécharger depuis le [Dépôt des Extensions Python](#) et examiner leur code. Aussi, le répertoire `python/plugins/` dans votre installation de QGIS contient des extensions que vous pouvez utiliser pour apprendre à développer les vôtres et à effectuer quelques-unes des tâches les plus courantes.

## 1.1 Exécuter un programme Python au démarrage de QGIS

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

### 1.1.1 Variables d'environnement PYQGIS\_STARTUP

Vous pouvez exécuter un programme Python juste avant la fin de l'initialisation de QGIS en affectant le chemin d'accès à un fichier Python existant à la variable d'environnement `PYQGIS_STARTUP`.

Vous aurez probablement rarement besoin de cette méthode, mais il est utile de la mentionner ici car c'est une des façons d'exécuter un programme Python dans QGIS et parce que ce programme s'exécutera avant que l'initialisation de QGIS ne soit complète. Cette méthode est très utile pour nettoyer `sys.path`, qui peut contenir des chemins d'accès indésirables, ou pour isoler/charger l'environnement initial sans avoir recours à un environnement virtuel, par ex. homebrew ou MacPorts sur Mac.

### 1.1.2 Le fichier : `startup.py`

Chaque fois que QGIS démarre, le fichier nommé `startup.py` est recherché dans le répertoire Python principal de l'utilisateur (généralement : `.qgis2/python`). Si ce fichier existe, il est exécuté par l'interpréteur Python intégré.

## 1.2 La Console Python

Il est possible de tirer partie de la console Python intégrée pour créer des scripts et les exécuter. La console peut être ouverte grâce au menu : *Extension* → *Console Python*. La console s'ouvre comme une fenêtre utilitaire non modale :



```
Python Console
1 Python 2.7.6 (default, Mar 22 2014, 23:03:41)
2 [GCC 4.8.2] on alex-portatil
3 ## Type help(iface) for more info and list of methods.
4 >>> layer = qgis.utils.iface.activeLayer()
5 >>> layer.id()
6 u'agueda_linhas_agua20141003093657531'
7 >>> layer.featureCount()
8 14L
9
>>> |
```

Figure 1.1: La Console Python de QGIS

La capture d'écran ci-dessus montre comment récupérer la couche sélectionnée dans la liste des couches, afficher son identifiant et éventuellement, si c'est une couche vecteur, afficher le nombre d'entités. Pour interagir avec l'environnement de QGIS, il y a une variable `iface`, instance de la classe `QgsInterface`. Cette interface permet d'accéder au canevas de carte, aux menus, barres d'outils et autres composantes de l'application QGIS.

A la convenance de l'utilisateur, les déclarations qui suivent sont exécutées lors du lancement de la console (à l'avenir, il sera possible de paramétrer d'autres commandes d'initialisation):



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

Pour ceux qui utilisent souvent la console, il peut être utile de définir un raccourci pour déclencher la console (dans le menu *Préférences* → *Configurer les raccourcis...*)

## 1.3 Extensions Python

QGIS permet d'enrichir ses fonctionnalités à l'aide d'extensions. Au départ, ce n'était possible qu'avec le langage C++. Avec l'ajout du support de Python dans QGIS, il est également possible d'utiliser les extensions écrites en Python. Le principal avantage sur des extensions C++ est leur simplicité de distribution (pas de compilation nécessaire pour chaque plate-forme) et la facilité du développement.

De nombreuses extensions couvrant diverses fonctionnalités ont été écrites depuis l'introduction du support de Python. L'installateur d'extensions permet aux utilisateurs de facilement chercher, mettre à niveau et supprimer les extensions Python. Voir la page du [Dépôt des Extensions Python](#) pour diverses sources d'extensions.

Créer des extensions Python est simple. Voir [Développer des extensions Python](#) pour des instructions détaillées.

---

**Note:** Les extensions Python sont aussi disponibles dans QGIS Server (*label\_qgisserver*), voyez [Extensions Python pour QGIS Server](#) pour plus d'information.

---

## 1.4 Applications Python

Souvent lors du traitement de données SIG, il est très pratique de créer des scripts pour automatiser le processus au lieu de faire la même tâche encore et encore. Avec PyQGIS, cela est parfaitement possible — importez le module `qgis.core`, initialisez-le et vous êtes prêt pour le traitement.

Vous pouvez aussi souhaiter créer une application interactive utilisant certaines fonctionnalités SIG — mesurer des données, exporter une carte en PDF ou toute autre fonction. Le module `qgis.gui` vous apporte différentes composantes de l'interface, le plus notable étant le canevas de carte qui peut être facilement intégré dans l'application, avec le support du zoom, du déplacement ou de tout autre outil personnalisé de cartographie.

Les applications personnalisées de PyQGIS ou les scripts

Note: *ne pas* utiliser `qgis.py` comme nom de script test — Python ne sera pas en mesure d'importer les dépendances étant donné qu'elles sont occultées par le nom du script.

### 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 *La Console Python*

Une fois la configuration du chemin faite, nous sauvegardons une référence 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()
```

Maintenant, vous pouvez travailler avec l'API de QGIS — charger des couches et effectuer des traitements ou lancer une interface graphique avec un canevas de carte. Les possibilités sont infinies :-)

### 1.4.3 Exécuter des applications personnalisées

Vous devrez indiquer au système où trouver les bibliothèques 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
```

Ceci peut être corrigé en définissant la variable d'environnement `PYTHONPATH`. Dans les commandes suivantes, `qgispath` doit être remplacé par le réel chemin d'accès au dossier d'installation de QGIS:

- sur Linux: **export PYTHONPATH=/qgispath/share/qgis/python**

- sur Windows: **set PYTHONPATH=c:\qgispath\python**

Le chemin vers les modules PyQGIS est maintenant connu. Néanmoins, ils dépendent des bibliothèques `qgis_core` et `qgis_gui` (les modules Python qui servent d'encapsulation). Le chemin vers ces bibliothèques est inconnu du système d'exploitation et vous allez encore récupérer une erreur d'import (le message peut varier selon le système):

```
>>> 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:

- sur Linux: **export LD\_LIBRARY\_PATH=/qgispath/lib**
- sur Windows: **set PATH=C:\qgispath;%PATH%**

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

- Imposer à l'utilisateur d'installer QGIS sur la plate-forme avant d'installer l'application. L'installateur de l'application devrait s'occuper des emplacements par défaut des bibliothèques QGIS et permettre à l'utilisateur de préciser un chemin si ce dernier n'est pas trouvé. Cette approche a l'avantage d'être plus simple mais elle impose plus d'actions à l'utilisateur.
- 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.



---

## Chargement de projets

---

Parfois, vous avez besoin de charger un projet existant depuis une extension ou (plus courant) depuis une application tierce en python (voir: *Applications Python*).

Pour charger un projet dans l'application QGIS en cours, il vous faut une instance() de la classe QgsProject dont vous appelez la méthode read() en lui passant l'objet QFileInfo qui contient le chemin du projet à charger:

```
# 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'
```

Si vous souhaitez apporter des modifications au projet (par exemple ajouter ou supprimer des couches) et enregistrer vos changements, vous pouvez appeler la méthode write() de votre instance de projet. La méthode write() accepte également une variable optionnelle QFileInfo qui permet de spécifier le chemin de sauvegarde du projet:

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

Les fonctions read() et write() renvoient un booléen que vous pouvez utiliser pour vérifier si l'opération a réussi ou pas.

---

**Note:** Si vous écrivez une application QGIS indépendante, pour synchroniser le projet en cours avec le canevas, vous devez instancier QgsLayerTreeMapCanvasBridge comme dans l'exemple ci-dessous:

```
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'))
```

---



---

## Chargement de couches

---

- Couches vectorielles
- Couches raster
- Registre de couches cartographiques

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

Pour charger une couche vectorielle, spécifiez l'identifiant de la source de données de la couche, un nom pour la couche et le nom du fournisseur:

```
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.

La façon la plus rapide d'ouvrir et d'afficher une couche vectorielle avec QGIS est la fonction `addVectorLayer` de `QgisInterface`:

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

Une nouvelle couche est créée et ajoutée en une seule étape au registre de couches cartographiques (la faisant apparaître dans la liste des couches). La fonction fournit l'instance de la couche ou *Aucune* si la couche n'a pas pu être chargée.

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

- librairie OGR (shapefiles et beaucoup d'autres formats) — la source de données est le chemin d'accès au fichier:

- pour les shp:

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

- pour les dxf (Voir les options internes données dans les données sources de l'url):

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

- Base de données PostGIS — la source des données est une chaîne de caractères contenant toutes les informations nécessaires pour créer une connexion à la base de données PostGIS. La classe `QgsDataSourceURI` peut générer cette chaîne pour vous. Veuillez noter que QGIS doit être compilé avec le support Postgres sinon ce fournisseur ne sera pas disponible :

```
uri = QgsDataSourceURI()
# set host name, port, database name, username and password
uri.setConnection("localhost", "5432", "dbname", "johnny", "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 et autres fichiers texte avec délimiteurs — pour ouvrir un fichier ayant un point-virgule comme délimiteur, et un champ “x” pour la coordonnée X et un champ “y” pour la coordonnée Y vous utiliserez quelque chose comme ceci :

```
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")
```

- Base de données SpatialLite — De façon similaire aux bases de données PostGIS, `QgsDataSourceURI` peut être utilisé pour générer l'identifiant de la source de données :

```
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 créée 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:** Vous pouvez changer la source des données d'une couche existante en appelant `setDataSource()` de l'instance `QgsVectorLayer`. comme dans l'exemple suivant:

```

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

```

---

## 3.2 Couches raster

Pour l'accès aux rasters, la librairie GDAL est utilisée. Elle supporte une grande variété de formats de fichier. Si vous avez des problèmes pour ouvrir un fichier, vérifiez si votre GDAL supporte ce type particulier de fichier (tous les formats ne sont pas disponibles par défaut). Pour ouvrir un raster à partir d'un fichier, spécifiez son nom de fichier et de base :

```

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

```

De la même façon que pour les couches vectorielles, les couches rasters peuvent être ouvertes en utilisant la fonction `addRasterLayer` de la classe `QgisInterface`:

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

Une nouvelle couche est créée et ajoutée en une seule étape au registre de couches cartographiques (la faisant apparaître dans la liste des couches).

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("identifiant", layer_name)
rlayer = QgsRasterLayer(str(uri.encodedUri()), 'my wcs layer', 'wcs')

```

le détail des paramètres URI peut être trouvé dans la [documentation des pilotes](#)

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 Registre de couches cartographiques

Si vous souhaitez utiliser les couches ouvertes pour faire un rendu, n'oubliez pas de les ajouter au registre de couches cartographiques. Ce registre prend possession des couches et elles peuvent être utilisées ultérieurement depuis n'importe quelle partie de l'application en utilisant leur identifiant unique. Lorsqu'une couche est supprimée du registre de couches cartographiques, elle est également supprimée.

En Ajoutant une couche au registre :

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

Les couches sont automatiquement supprimées lorsque vous quittez, mais si vous souhaitez directement supprimer la couche, utilisez :

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

Pour obtenir la liste des couches chargées et leurs identifiant, utilisez :

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

---

## Utiliser des couches raster

---

- Détails d'une couche
- Moteur de rendu
  - Rasters mono-bande
  - Rasters multi-bandes
- Rafraîchir les couches
- Interrogation des données

Cette section liste différentes opérations réalisables avec des couches raster.

### 4.1 Détails d'une couche

Une couche raster est constituée d'une ou plusieurs bandes raster — on la qualifie de mono-bande ou multi-bande. Une bande représente une matrice de valeurs. Les images en couleurs (par ex: photos aériennes) sont des rasters qui disposent de bandes rouge, vert et bleu. Les couches mono-bande représentent soit des variables continues (par ex: élévation) soit des variables discrètes (par ex: utilisation du sol). Dans certains cas, une couche raster comporte une palette et les valeurs du raster se réfèrent aux couleurs stockées dans la palette :

```

rlayer.width(), rlayer.height()
(812, 301)
rlayer.extent()
<qgis._core.QgsRectangle object at 0x00000000F8A2048>
rlayer.extent().toString()
u'12.095833,48.552777 : 18.863888,51.056944'
rlayer.rasterType()
2 # 0 = GrayOrUndefined (single band), 1 = Palette (single band), 2 = Multiband
rlayer.bandCount()
3
rlayer.metadata()
u'<p class="glossy">Driver:</p>...'
rlayer.hasPyramids()
False

```

### 4.2 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.

Pour interroger le moteur de rendu actif

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

Pour paramétrer un moteur de rendu, utilisez la méthode `setRenderer()` de `QgsRasterLayer`. Il existe plusieurs classes de moteur de rendu (dérivées de `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-couleur.

Les sections qui suivent expliquent comment interroger et modifier le style de représentation de la couche. Une fois que les changements ont été effectués, vous pouvez forcer la mise à jour du canevas de carte avec *Rafraîchir les couches*.

**A FAIRE :** Améliorations du contraste, transparence (pas de donnée), valeur maximale/minimale indiquée par l'utilisateur, statistiques sur la bande

## 4.2.1 Rasters mono-bande

Disons que nous souhaitons afficher notre couche raster (à un seule bande) avec des couleurs s'étalant du vert au jaune (pour une plage de valeur de pixel de 0 à 255). Nous allons d'abord préparer un objet `QgsRasterShader` et configurer sa fonction de shader:

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

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:

- linéaire (`INTERPOLATED`): les couleurs résultent d'une interpolation linéaire des entrées de couleur de la carte qui sont en dessous et au dessus de la valeur du pixel actuel.
- discret (`DISCRETE`): la couleur est utilisée depuis l'entrée de la carte de couleur avec une valeur supérieure ou égale.
- exact (`EXACT`): la couleur n'est pas interpolée. Seuls les pixels dont la valeur équivaut aux entrées de la carte de couleur sont représentés.

Dans un second temps, nous associons le shader avec la couche raster:

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

Le nombre 1 dans le code ci-dessus est le numéro de bande (les bandes de raster sont indexées à partir de un).

## 4.2.2 Rasters multi-bandes

Par défaut, QGIS calque les trois premières bandes aux valeurs rouge, vert et bleue pour créer l'image en couleur (style de représentation `MultiBandColor`). Dans certains cas, vous voudrez modifier ce paramétrage. Le code qui suit intervertit les bandes rouge (1) et verte (2)

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

Dans le cas où seule une bande est nécessaire pour la visualisation du raster, le style mono-bande peut être sélectionné, soit en niveaux de gris, soit en pseudo-couleur.

## 4.3 Rafraîchir les couches

Si vous modifiez la symbologie d'une couche et voulez que l'utilisateur en voit le résultat immédiatement, appelez ces méthodes

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

Le premier appel s'assurera que l'image en cache des couches rendues est effacé dans le cas où le cache est activé. Cette fonctionnalité est disponible depuis QGIS 1.4 et elle n'existait pas dans les versions précédentes. Pour s'assurer que le code fonctionne avec toutes les versions de QGIS, vérifions que la méthode existe.

---

**Note:** Cette méthode est obsolète à partir de QGIS 2.18.0 et produira un avertissement. Il suffit d'appeler “`triggerRepaint()`”.

---

La deuxième commande émet un signal forçant l'actualisation de tout canevas de carte contenant la couche.

Avec les couches raster WMS, ces commandes ne fonctionnent pas. Dans ce cas, vous devez le faire explicitement

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

Dans le cas où vous avez modifié la symbologie de la couche (consulter les sections sur les couches vectorielles et rasters pour voir comment faire), vous voudriez sans doute forcer QGIS à mettre à jour la symbologie de la couche dans la légende (panneau Couches). Cela peut être réalisé comme suit (`iface` est une instance de la classe `QgisInterface`)

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

## 4.4 Interrogation des données

Pour récupérer la valeur des bandes d'une couche raster en un point donné

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

La méthode `results` renvoie dans ce cas un dictionnaire où les index de bandes correspondent aux clefs et les valeurs de bandes aux valeurs.

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



---

## Utilisation de couches vectorielles

---

- Récupérer les informations relatives aux attributs
- Sélection des entités
- Itérer sur une couche vecteur
  - Accès aux attributs
  - Itérer sur une sélection d'entités
  - Itérer sur un sous-ensemble d'entités
- Modifier des couches vecteur
  - Ajout d'Entités
  - Suppression d'Entités
  - Modifier des Entités
  - Ajout et Suppression de Champs
- Modifier des couches vecteur à l'aide d'un tampon d'édition
- Utilisation des index spatiaux
- Ecrire dans des couches vecteur
- Fournisseur de données en mémoire
- Apparence (Symbologie) des couches vecteur
  - Moteur de rendu à symbole unique
  - Moteur de rendu à symboles catégorisés
  - Moteur de rendu à symboles gradués
  - Travailler avec les symboles
    - \* Travailler avec des couches de symboles
    - \* Créer des types personnalisés de couches de symbole
  - Créer ses propres moteurs de rendu
- Sujets complémentaires

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

### 5.1 Récupérer les informations relatives aux attributs

Vous pouvez récupérer les informations relatives aux champs associés à une couche vectorielle en appelant `pendingFields()` sur une instance `QgsVectorLayer`

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

---

**Note:** À partir de QGIS 2.12, il existe aussi une: `func: fields()` dans: `class: QgsVectorLayer` qui est un alias de: `func: pendingFields()`.

---

## 5.2 Sélection des entités

Dans QGIS Desktop, les entités peuvent être sélectionnées de plusieurs façons : l'utilisateur peut cliquer sur l'entité, dessiner un rectangle sur le canevas de carte ou utiliser une expression filtrante. Les entités sélectionnées sont généralement identifiées par une couleur différente (jaune par défaut) pour attirer l'attention de l'utilisateur sur la sélection. Il peut parfois être utile de programmer la sélection des entités et la couleur par défaut.

Pour changer la couleur de sélection vous pouvez utiliser la méthode `setSelectionColor()` de `QgsMapCanvas` comme montré dans l'exemple suivant

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

Pour ajouter des entités à la liste des entités sélectionnées pour une couche donnée, vous pouvez appeler `setSelectedFeatures()` en lui passant la liste de l'identifiant des entités

```
# 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()])
```

Pour effacer la sélection, passez simplement une liste vide

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

## 5.3 Itérer sur une couche vecteur

Itérer sur les entités d'une couche vecteur est l'une des tâches les plus courantes. L'exemple ci-dessous est un code basique pour accomplir cette tâche et qui affiche des informations sur chaque entité. La variable `layer` est présumée être un objet `QgsVectorLayer`:

```
iter = layer.getFeatures()
for feature in iter:
    # retrieve every feature with its geometry and attributes
    # fetch geometry
    geom = feature.geometry()
    print "Feature ID %d: " % feature.id()

    # show some information about the feature
    if geom.type() == Qgs.Point:
        x = geom.asPoint()
        print "Point: " + str(x)
    elif geom.type() == Qgs.Line:
        x = geom.asPolyline()
        print "Line: %d points" % len(x)
    elif geom.type() == Qgs.Polygon:
        x = geom.asPolygon()
        numPts = 0
        for ring in x:
            numPts += len(ring)
        print "Polygon: %d rings with %d points" % (len(x), numPts)
    else:
        print "Unknown"

    # fetch attributes
    attrs = feature.attributes()

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



### 5.3.1 Accès aux attributs

Les attributs peuvent être référencés par leur nom.

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

Autrement, les attributs peuvent être référencés par index. Ce sera un peu plus rapide que d'utiliser leur nom. Par exemple, pour obtenir le premier attribut

```
print feature[0]
```

### 5.3.2 Itérer sur une sélection d'entités

Si vous n'avez besoin que des entités sélectionnées, vous pouvez utiliser la méthode `selectedFeatures()` de la couche vectorielle :

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

Une autre option est la méthode `features()` de Processing :

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

Par défaut, elle fera l'itération de toutes les entités de la couche, s'il n'y a pas de sélection, ou autrement, des entités sélectionnées. Notez que ce comportement peut être changé dans les options de Processing pour ignorer les sélections.

### 5.3.3 Itérer sur un sous-ensemble d'entités

Si vous désirez itérer sur un sous-ensemble donné d'entités dans une couche, tel que celles situées dans une zone donnée, vous devez ajouter un objet `QgsFeatureRequest` à la fonction d'appel `getFeatures()`. Voici un exemple:

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

Avec `setLimit()` vous pouvez limiter le nombre de entité requit. Voici un exemple :

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

Si vous avez besoin d'un filtre basé sur les attributs à la place (ou en addition) d'un filtre spatial comme montré dans l'exemple ci-dessus, vous pouvez construire un objet `QgsExpression` et lui passer le constructeur `QgsFeatureRequest`. Par exemple :

```
# 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)
```

Voir *Expressions, Filtrage et Calcul de valeurs* pour les détails sur la syntaxe supportée par: class: '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

Si vous avez besoin uniquement d'un sous-ensemble des attributs ou si vous n'avez pas besoin des informations géométriques, vous pouvez augmenter considérablement la **vitesse** de la requête de fonctionnalités en utilisant l'indicateur `QgsFeatureRequest.NoGeometry` ou en spécifiant un sous-ensemble d'attributs (éventuellement vide) comme montré dans l'exemple ci-dessus.

---

## 5.4 Modifier des couches vecteur

La majorité des fournisseurs de données vecteurs gère l'édition des données. Parfois, ils gèrent uniquement certaines actions d'édition. Utilisez la fonction `capabilities()` pour trouver quelles sont les fonctionnalités gérées:

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

Pour obtenir une liste de toutes les fonctionnalités disponibles, reportez-vous à la «Documentation API de `QgsVectorDataProvider` <<http://qgis.org/api/classQgsVectorDataProvider.html>>'\_

Pour afficher la description textuelle des capacités du calque dans une liste séparée par des virgules, vous pouvez utiliser: `func: capabilitiesString` comme dans l'exemple suivant

```
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

Créez quelques classes: *QgsFeature* et passez une liste de celles-ci au fournisseur: func: 'addFeatures'. Il renverra deux valeurs: result (true / false) et liste des fonctionnalités ajoutées (leur ID est défini par le magasin de données).

Pour configurer les attributs, vous pouvez soit initialiser la fonction en passant par une: class: *QgsFields* instance ou appel: func: 'initAttributes' en passant par le nombre de champs que vous souhaitez ajouter.

```
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

Pour supprimer des entités, il suffit d'indiquer une liste de leur identifiant

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

### 5.4.3 Modifier des Entités

Il est possible de réaliser des changements soit sur la géométrie de l'entité, soit sur ses attributs. L'exemple qui suit modifie d'abord des valeurs d'attributs situés à l'index 0 et 1 puis modifie la géométrie de l'entité:

```
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 committed 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

Lorsque vous modifiez des vecteurs avec l'application QGIS, vous devez d'abord lancer le mode édition pour une couche donnée puis réaliser des modifications et enfin, sauvegarder (ou annuler) vos changements. Tous les changements que vous réalisez ne sont pas écrits tant que vous ne les avez pas validés, il reste alors dans le tampon d'édition en mémoire de la couche. Il est possible d'utiliser cette fonctionnalité en programmation, c'est juste une autre méthode pour éditer une couche vecteur qui complète l'utilisation directe des fournisseurs de données. Utilisez cette option lorsque vous fournissez des outils graphiques pour l'édition car cela permet à l'utilisateur de valider ou d'annuler ainsi que la possibilité de défaire/refaire. Lorsque les changements sont validés, toutes les modifications stockées dans le tampon d'édition sont sauvegardées dans le fournisseur de données.

Pour savoir si une couche est en mode d'édition, utilisez: `func: isEditable` — les fonctions d'édition ne fonctionnent que lorsque le mode d'édition est activé. Utilisation des fonctions d'édition.

```
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)
```

Pour que les actions annuler/refaire fonctionnent correctement, les appels mentionnés plus haut doivent être encapsulés dans des commandes d'annulation. (si vous n'avez pas besoin d'annuler/refaire et que vous voulez envoyer les changements immédiatement, utilisez la méthode plus simple: *editing with data provider*). Voici comment utiliser la fonctionnalité "Annuler":

```
layer.beginEditCommand("Feature triangulation")  
  
# ... call layer's editing methods ...  
  
if problem_occurred:  
    layer.destroyEditCommand()  
    return  
  
# ... more editing ...  
  
layer.endEditCommand()
```

La fonction `beginEditCommand()` crée et “active” une commande interne qui enregistrera les changements effectuée sur la couche vecteur. Lors de l’appel de la fonction `endEditCommand()`, la commande est poussée sur la pile d’annulation et l’utilisateur peut alors cliquer sur les boutons Annuler/Refaire. Au cas où quelque chose tournerait mal lors des changements, la méthode `destroyEditCommand()` supprimera la commande de la pile et annulera tous les changements réalisés depuis que la commande est active.

Pour démarrer le mode d’édition, il y a : `func: startEditing()` méthode, pour arrêter l’édition il y a : `func: commitChanges()` et : `func: rollBack()` — Cependant, normalement vous ne devriez pas avoir besoin de ces méthodes, elle laisser devrait être déclenchée par l’utilisateur.

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)
```

Cela va automatiquement appeler : `func: commitChanges()` à la fin. Si une exception se produit, : `func: rollBack()` fera toutes les modifications. Dans le cas où un problème est rencontré dans : `func: commitChanges()` (quand la méthode renvoie False) a : `class: QgsEditError` exception sera levée.

## 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 répéter 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:

- créez l’index spatial — le code qui suit créé un index vide

```
index = QgsSpatialIndex()
```

- ajouter les entités à l’index – l’index utilise des objets `QgsFeature` et les ajoute dans sa structure de données interne. Vous pouvez créer les objets manuellement ou utiliser ceux qui sont issus de la méthode `nextFeature()` du fournisseur de données:

```
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 Ecrire dans des couches vecteur

Vous pouvez générer des fichiers de couche vecteur en utilisant la classe `QgsVectorFileWriter`. Elle gère tous les formats vecteurs gérés par QGIS (fichier Shape, GeoJSON, KML, etc.).

Il y a deux façons d'exporter une couche vectorielle:

- A partir d'une instance de la classe `QgsVectorLayer`

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

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

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

Le troisième paramètre indique l'encodage du texte en sortie. Seuls certains pilotes ont besoin de ce paramètre pour fonctionner correctement, les fichiers Shape sont dans ce cas. Néanmoins, vous ne devriez pas rencontrer de problèmes tant que vous n'utilisez pas un jeu de caractères international. Le quatrième paramètre que nous avons laissé à `None` peut indiquer un SCR de destination, si une instance valide de `QgsCoordinateReferenceSystem` est utilisée, la couche est transformée dans ce SCR.

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.

- Directement depuis les entités

```
from PyQt4.QtCore import QVariant

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

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

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 Fournisseur de données en mémoire

Le fournisseur de données en mémoire est utilisable principalement par des extensions ou des applications tierces. Il ne stocke pas de données sur disque ce qui permet aux développeurs de l'utiliser comme support rapide pour des couches temporaires.

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

Le fournisseur de données en mémoire gère également l'indexation spatiale qui est activée en appelant la fonction `createSpatialIndex()` du fournisseur. Une fois l'index spatial créé, vous pourrez itérer sur les entités d'emplacements donnés plus rapidement (car il n'est plus nécessaire de traverser toutes les entités mais uniquement celles qui se trouvent dans le rectangle).

Un fournisseur de données en mémoire est créé en indiquant `'memory'` dans la chaîne de fournisseur du constructeur d'un objet `QgsVectorLayer`.

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 coordonnées de référence, des champs et l'indexation. La syntaxe est la suivante:

**crs=définition** Spécifie le système de coordonnées de référence, où `définition` peut être sous n'importe laquelle des formes acceptées par `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.xMinimum(), e.yMinimum(), e.xMaximum(), e.yMaximum()

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

## 5.9 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.

Le moteur de rendu de chaque couche peut être obtenu comme présenté ci-dessous:

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

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

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

Il existe plusieurs types de moteurs de rendu dans la bibliothèque de base de QGIS:

Type	Classe	Description
singleSymbol	QgsSingleSymbolRenderer	Affiche toutes les entités avec le même symbole.
categorizedSymbol	QgsCategorizedSymbolRenderer	Affiche les entités en utilisant un symbole différent pour chaque catégorie.
graduatedSymbol	QgsGraduatedSymbolRenderer	Affiche les entités en utilisant un symbole différent pour chaque plage de valeurs.

Des moteurs de rendu personnalisés peuvent également être disponibles et vous ne pouvez donc pas supposer qu'il n'y a que ces types. Vous pouvez interroger le singleton `QgsRendererV2Registry` pour savoir quels sont les moteurs de rendu disponibles :

```
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

Vous pouvez obtenir le symbole utilisé pour le rendu en appelant la méthode `symbol()` et le modifier avec la méthode `setSymbol()` (pour les développeurs C++, le moteur de rendu devient propriétaire du symbole).

Vous pouvez changer le symbole utilisé par une couche vectorielle donnée en appelant `setSymbol()` en lui passant l'instance du symbole requis. Les symboles pour les couches de *points*, *lignes* et *polygones* peuvent être créés en appelant la fonction `createSimple()` des classes correspondantes `QgsMarkerSymbolV2`, `QgsLineSymbolV2` et `QgsFillSymbolV2`.



Le dictionnaire passé à `createSimple()` définit les propriétés de style du symbole.

Par exemple, vous pouvez remplacer le symbole utilisé par une couche `** point **` particulière en appelant: `func: setSymbol()` passer une instance d'une classe: `QgsMarkerSymbolV2` comme dans l'exemple de code suivant

```
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

Pour obtenir la liste complète des propriétés de la première couche de symbole d'une instance de symbole, vous pouvez suivre l'exemple de code:

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

Vous pouvez interroger et indiquer le nom de l'attribut qui sera utilisé pour la classification en utilisant les méthodes `classAttribute()` et `setClassAttribute()`.

Pour obtenir la liste des catégories

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

Où `value()` est la valeur utilisée pour la discrimination entre les catégories, `label()` est un texte utilisé pour la description des catégories et la méthode `symbol()` renvoie le symbole associé.

Le moteur de rendu stocke généralement le symbole original et la rampe de couleur qui ont été utilisés pour la classification. On peut les obtenir par les méthodes `sourceColorRamp()` and `sourceSymbol()`.

## 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

Pour la représentation des symboles, il existe la classe de base `QgsSymbolV2` qui est dérivée en trois sous-classes:

- `QgsMarkerSymbolV2` — pour les entités ponctuelles.
- `QgsLineSymbolV2` — pour les entités linéaires.
- `QgsFillSymbolV2` — pour les entités polygonales.

**Chaque symbole est constitué d'une ou plusieurs couche de symboles** (classes dérivées de `QgsSymbolLayerV2`). Les couches de symboles font le rendu, la classe du symbole sert seulement de conteneur pour les couches de symbole.

Il est possible d'explorer une instance de symbole (récupérée depuis un moteur de rendu): la méthode `type()` indique s'il s'agit d'un symbole de marqueur, de ligne ou remplissage. Il existe une méthode `dump()` qui renvoie une brève description du symbole. Pour obtenir la liste des couches de symbole:

```

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

```

Pour trouver la couleur du symbole, utilisez la méthode `color()` et la méthode `setColor()` pour la changer. Avec les symboles de marqueurs vous pouvez également interroger la taille et la rotation du symbole à l'aide des méthodes `size()` et `angle()`. Pour les symboles de ligne, la méthode `width()` renvoie la largeur de la ligne.

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

### Travailler avec des couches de symboles

Comme évoqué auparavant, les couches de symboles (sous-classe de `QgsSymbolLayerV2`) déterminent l'apparence des entités. Il existe plusieurs couches de symboles de base pour l'utilisation courante. Il est possible d'implémenter de nouveaux types de symboles et de personnaliser l'affichage des entités. La méthode `layerType()` identifie uniquement la classe de la couche de symboles. Celles qui sont présentes par défaut sont les types `SimpleMarker`, `SimpleLine` et `SimpleFill`.

Vous pouvez obtenir une liste complète des types de couches de symbole pour une classe donnée de symbole de la manière suivante:

```

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

```

Sortie

```

EllipseMarker
FontMarker
SimpleMarker

```

SvgMarker  
VectorField

la classe `QgsSymbolLayerV2Registry` gère une base de données de tous les types de symboles de couche disponibles.

Pour accéder à la donnée de la couche de symbole, utilisez la méthode `properties()` qui retourne un dictionnaire des propriétés définissant l'apparence du symbole. Chaque type de couche de symbole comporte un jeu de propriétés. Il existe également des méthodes génériques `color()`, `size()`, `angle()`, `width()` accompagnées de leur équivalent d'attribution de valeur. La taille et l'angle sont disponibles uniquement pour les couches de symbole de marqueurs et la largeur, pour les couches de symbole de ligne.

## 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)
```

La méthode `layerType()` détermine le nom de la couche de symbole. Elle doit être unique parmi toutes les couches de symbole déjà existantes. Des propriétés sont utilisées pour la persistance des attributs. La méthode `clone()` renvoie une copie de la couche de symbole avec des attributs complètement identiques. Enfin, il reste les méthodes de rendu: `startRender()` est appelée avant le rendu de la première entité, `stopRender()` lorsque le rendu est terminé. La méthode `renderPoint()` s'occupe du rendu. Les coordonnées du ou des point(s) sont déjà transformées dans le SCR de sortie.

Pour les polygones et les polygones, la seule différence est la méthode de rendu: vous utiliserez `renderPolyline()` qui reçoit une liste de lignes et resp. `renderPolygon()` qui reçoit une liste de points pour définir l'enveloppe extérieure en premier paramètre et une liste des trous (ou `None`) dans le deuxième paramètre.

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 crée 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 symbole.

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())
```

Vous devez renseigner le type de couche (la même renvoyée par la couche) et le type de symbole (marker/line/fill) au constructeur de la classe parent. La méthode `createSymbolLayer()` s'occupe de créer l'instance d'une couche de symbole avec les attributs indiqués dans le dictionnaire *props*. (Attention, les clefs sont des instances

QString et non des objets Python "str"). Et il existe également la méthode `createSymbolLayerWidget()` qui renvoie l'interface de paramétrage pour ce type de couche de symbole.

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.defaultSymbol(QGis.Circle)]

    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)
```

Le constructeur de la classe parente `QgsFeatureRendererV2` nécessite un nom de moteur de rendu (qui doit être unique parmi tous les moteurs de rendu). La méthode `symbolForFeature()` est celle qui décide du symbole qui sera utilisé pour une entité particulière. `startRender()` et `stopRender()` gèrent l'initialisation et la finalisation du rendu des symboles. La méthode `usedAttributes()` renvoie une liste des noms de champs dont a besoin le moteur de rendu. Enfin la fonction `clone()` renvoie une copie du moteur de rendu.

Comme avec les couches de symbole, il est possible d'attacher une interface graphique pour la configuration du moteur de rendu. Elle doit être dérivée de la classe `QgsRendererV2Widget`. L'exemple qui suit crée un bouton qui permet à l'utilisateur de paramétrer le symbole du premier symbole.

```
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

```

Le constructeur reçoit les instances de la couche active (`QgsVectorLayer`), le style global (`QgsStyleV2`) ainsi que le moteur de rendu courant. S'il n'y a pas de moteur de rendu ou si le moteur de rendu est d'un type différent, il sera remplacé par notre nouveau moteur de rendu, sinon, le moteur de rendu actuel (qui dispose déjà du bon type). Le contenu de l'interface doit être mis à jour pour refléter l'état du moteur de rendu. Lorsque la boîte de dialogue du moteur de rendu est acceptée, la méthode `renderer()` de l'interface est appelée pour récupérer le moteur de rendu actuel, qui sera affecté à la couche.

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()` crée 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.

Pour associer une icône au moteur de rendu, vous pouvez en déclarer une dans le constructeur de `QgsRendererV2AbstractMetadata` dans le troisième (optionnel) argument. La fonction `__init__()` du constructeur de la classe de base de `RandomRendererMetadata` devient alors:

```

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

```

L'icône peut être également associée à n'importe quel moment en utilisant la méthode `setIcon()` de la classe de métadonnées. L'icône peut être chargée depuis un fichier (comme montré ci-dessus) ou peut être chargée depuis une ressource Qt (PyQt4 inclut un compilateur Python de fichiers .qrc).

## 5.10 Sujets complémentaires

### A FAIRE :

- création/modification des symboles
- travailler avec le style (`QgsStyleV2`)
- Travailler avec les palettes de couleurs (`QgsVectorColorRampV2`)

- Rendu basé sur des règles (voir [this blogpost](#))
- Explorer les couches de symboles et les registres de rendus



---

## Manipulation de la géométrie

---

- Construction de géométrie
- Accéder à la Géométrie
- Prédicats et opérations géométriques

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

Il existe plusieurs options pour créer une géométrie:

- 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)]])
```

Les coordonnées sont indiquées à l'aide de la classe `QgsPoint`.

La polyligne (Linestring) est représentée par une succession de points. Le polygone est représenté par une succession de polylignes en anneaux (c'est-à-dire des polylignes fermées). Le premier anneau représente l'anneau externe (la limite), les potentiels anneaux ultérieures sont des trous dans le polygone.

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 = '010100000000000000000045400000000000001440'.decode('hex')
>>> g.fromWkb(wkb)
>>> g.exportToWkt()
'Point (42 5)'
```

## 6.2 Accéder à la Géométrie

Vous devriez d'abord trouver le type de la géométrie en utilisant la méthode `wkbType()`. Elle renvoie la valeur depuis l'énumération `Qgis.WkbType`

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

Une autre alternative réside dans l'utilisation de la méthode `type()` qui renvoie une valeur de la liste `Qgis.GeometryType`. Il existe également une fonction `isMultipart()` pour vous aider à déterminer si une géométrie est multi-parties ou non.

Pour extraire l'information d'une géométrie il existe des fonctions d'accès pour chaque type de vecteur. Voici comment utiliser ces accès:

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

---

**Note:** Les tuples (x,y) ne sont pas de vrais tuples, ce sont des objets `QgsPoint`, leurs valeurs sont accessibles avec les fonctions `x()` et `y()`.

---

Pour les géométries multi-parties, il y a des fonctions accesseurs similaires: `asMultiPoint()`, `asMultiPolyline()`, `asMultiPolygon()`.

## 6.3 Prédicats et opérations géométriques

QGIS utilise la bibliothèque GEOS pour les opérations avancées de géométrie telles que les prédicats (`contains()`, `intersects()`, ...) et les opérations d'ensemble (`union()`, `difference()`, ...). QGIS peut également réaliser des calculs sur les propriétés géométriques des géométries comme la surface (dans le cas des polygones) ou la longueur (polygones et lignes).

Voici un exemple succinct qui combine l'itération sur les entités d'une couche donnée et des calculs géométriques sur leur géométrie.

```
# 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()
```

Les calculs de surfaces et de périmètres ne prennent pas en compte le SCR lorsque qu'on utilise les méthodes de la classe `QgsGeometry`. Pour réaliser des calculs plus précis sur les surfaces et les distances, la classe `QgsDistanceArea` peut être utilisée. Si les projections sont désactivées, les calculs seront faits en mode planaire, sinon ils tiendront compte de l'ellipsoïde.

```
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.

Vous pouvez retrouver plus d'information dans les sources suivantes :

- Modification de la géométrie: [Algorithme de Reprojection](#)
- Distance et surface en utilisant la classe `QgsDistanceArea` class: [Algorithme Matrice des Distances](#)
- **'Algorithme de conversion de multi-parties en partie unique'** [https://raw.githubusercontent.com/qgis/QGIS/release-2\\_18/python/plugins/processing/algs/qgis/MultipartToSingleparts.py](https://raw.githubusercontent.com/qgis/QGIS/release-2_18/python/plugins/processing/algs/qgis/MultipartToSingleparts.py)>'



---

## Support de projections

---

- Système de coordonnées de référence
- Projections

### 7.1 Système de coordonnées de référence

Les systèmes de coordonnées de référence (SCR) sont encapsulés par la classe `QgsCoordinateReferenceSystem`. Les instances de cette classe peuvent être créées de différentes façons :

- 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",0.017453292519943295], '
      AXIS["Longitude",EAST], AXIS["Latitude",NORTH]]'
crs = QgsCoordinateReferenceSystem(wkt)
```

- créer des SCR invalides et ensuite utiliser l'une des fonctions `create*()` pour l'initialiser. Dans l'exemple suivant, nous utilisons la chaîne Proj4 pour initialiser la projection

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

Il faut vérifier (via une recherche dans la base de données) si la création du SCR a réussi: la fonction `isValid()` doit retourner `True`.

Veillez noter que pour initialiser des systèmes de références spatiales, QGIS doit rechercher les valeurs appropriées dans sa base de données interne `srs.db`. Ainsi, lorsque vous créez une application QGIS indépendante, vous devez en définir les chemins par défaut correctement avec la fonction `QgsApplication.setPrefixPath()` sinon l'application ne pourra pas retrouver la base de données des projections. Si vous utilisez les commandes depuis la console Python de QGIS ou si vous développez une extension, vous n'avez pas à vous en préoccuper: tout est déjà géré pour vous.

Accéder à l'information sur le système de référence spatiale

```
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

Vous pouvez réaliser des transformations entre deux systèmes de références spatiales différents en utilisant la classe `QgsCoordinateTransform`. Le moyen le plus simple de l'utiliser est de créer un SCR source et un autre cible, puis de construire une instance de la classe `QgsCoordinateTransform` avec. Ensuite, appelez répétitivement la fonction `transform()` pour lancer la transformation. Par défaut, la transformation va de la source vers la cible mais elle peut également être lancée en sens inverse.

```
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
```

---

## Utiliser le Canevas de carte

---

- Intégrer un canevas de carte
- Utiliser les outils cartographiques avec le canevas
- Contour d'édition et symboles de sommets
- Ecrire des outils cartographiques personnalisés
- Ecrire des éléments de canevas de carte personnalisés

L'objet canevas de carte est probablement l'objet le plus important de QGIS car c'est lui qui permet d'afficher la carte composée par la superposition des couches et d'interagir avec les cartes et les couches. Le canevas montre toujours une partie de la carte définie dans l'emprise courante du canevas. L'interaction est réalisée par l'utilisation d'**outils cartographiques**. Ces outils permettent: le déplacement, le zoom, l'identification des couches, les mesures, l'édition vectorielle, etc. Comme dans les autres logiciels graphiques, il y a toujours un outil actif et l'utilisateur peut migrer d'un outil à l'autre.

Le canevas de carte est implémenté dans la classe `QgsMapCanvas` du module `qgis.gui`. L'implémentation se base sur l'environnement de la Vue Graphique de Qt. Cette structure fournit généralement une surface ainsi qu'une vue où les objets graphiques personnalisés sont placés et avec lesquels l'utilisateur peut interagir. Nous assumerons que vous connaissez suffisamment Qt pour comprendre les concepts de scène graphique, de vue et d'éléments. Si ce n'est pas le cas, assurez-vous de lire [aperçu rapide de l'environnement](#).

Lorsque la carte a été déplacée, zoomée (ou qu'un évènement a déclenché son rafraichissement), la carte est redessinée dans l'emprise courante. Les couches sont rendues dans une image (en utilisant la classe `QgsMapRenderer`) et cette image est ensuite affichée dans le canevas. L'objet graphique (en termes de structure de vue graphique Qt) responsable de l'affichage de la carte est la classe `QgsMapCanvasMap`. Elle contrôle également le rafraichissement de la carte rendue. En plus de cet objet qui fait office d'arrière plan, il peut y avoir plusieurs **objets de canevas de carte**. Typiquement, il peut exister des contours d'édition (utilisés pour faire des mesures, pour éditer des vecteurs, etc.) ou des symboles de sommets. Les objets du canevas sont généralement utilisés pour donner un retour visuel des outils de cartographie, par exemple, lorsqu'on crée un polygone, l'outil cartographique crée un contour d'édition qui affiche la forme actualisée du polygone. Tous les objets de canevas sont des sous-classes de `QgsMapCanvasItem` qui ajoute des fonctionnalités aux objets de la classe basique `QGraphicsItem`.

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

- le canevas de carte — pour visualiser la carte
- des objets de canevas — objets additionnels qui peuvent être affichés dans le canevas de carte
- les outils cartographiques — pour interagir avec le canevas de carte

### 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()
```

Ce code crée une fenêtre indépendante avec un canevas de carte. Il peut également être intégré dans un objet ou une fenêtre existant. Lorsque vous utilisez des fichiers .ui avec Qt Designer, placez un `QWidget` dans le formulaire et transformez-le en une nouvelle classe. Utilisez `QgsMapCanvas` en tant que nom de classe et utilisez `qgis.gui` comme fichier d'en-tête. L'utilitaire `pyuic4` le prendra en compte. C'est un moyen assez pratique pour intégrer un canevas. L'autre possibilité est d'écrire du code qui construira le canevas et les autres objets (comme fils de la fenêtre principale ou d'une boîte de dialogue) et de créer la mise en page.

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

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

(Au cas où vous vous poseriez la question, `Qt` vient du module `PyQt4.QtCore` et `Qt.white` est une des instances prédéfinies de `QColor`.)

Il est maintenant temps d'ajouter des couches cartographiques. Nous allons d'abord ouvrir une couche et l'ajouter au registre de couches cartographiques. Ensuite, nous définirons l'emprise du canevas de carte et établirons la liste des couches pour le canevas

```
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

L'exemple qui suit construit une fenêtre contenant un canevas de carte et des outils cartographiques sommaires pour se déplacer dans la carte et zoomer. Les actions sont créées pour l'activation de chaque outil: le déplacement est réalisé avec la classe `QgsMapToolPan`, le zoom avec une paire d'instances de la classe `QgsMapToolZoom`. Les actions sont paramétrées pour pouvoir être cochées et sont assignées ensuite aux outils pour gérer automatiquement l'état activé/désactivé des actions. Lorsqu'un outil cartographique est activé, son action est paramétrée comme sélectionnée et l'action du précédent outil cartographique est désélectionnée. Les outils cartographiques sont activés par la méthode `setMapTool()`.

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

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

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

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



```

self.setCentralWidget(self.canvas)

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

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

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

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

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

self.pan()

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

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

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

```

Vous pouvez insérer le code ci-dessus dans un fichier, par exemple `mywnd.py` et l'exécuter dans la console Python de QGIS. Ce code mettra la couche actuellement sélectionnée dans un canevas de carte nouvellement créé

```

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

```

Assurez-vous juste que le fichier `mywnd.py` est répertorié dans les chemins d'accès de Python (`sys.path`). Si ce n'est pas le cas, vous pouvez simplement l'y ajouter: `sys.path.insert(0, '/my/path')` — autrement, la déclaration d'import échouera, faute de trouver le module.

## 8.3 Contour d'édition et symboles de sommets

Utilisez les éléments du canevas de carte pour afficher des données supplémentaires au-dessus de la carte dans le canevas. Il est possible de créer ses propres classes d'éléments de canevas (traité ci-dessous) mais il existe deux classes d'éléments par défaut très utiles : `QgsRubberBand` pour dessiner des poli-lignes ou des polygones et `QgsVertexMarker` pour dessiner des points. Elles utilisent toutes les deux des coordonnées cartographiques et la forme est donc déplacée/ajustée automatiquement lorsque le canevas est déplacé ou zoomé.

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)
```

Veillez noter que les points d'un polygone ne sont pas stockés dans une liste. En fait, il s'agit d'une liste d'anneaux contenant les anneaux linéaires du polygone: 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)
```

Les objets de canevas sont liés à la scène du canevas. Pour les cacher temporairement (et les afficher plus tard), utilisez les fonctions `hide()` et `show()`. Pour supprimer complètement un objet, vous devez le retirer de la scène du canevas:

```
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).

Un contour d'édition peut être utilisé pour dessiner des points mais la classe `QgsVertexMarker` est plus appropriée pour ce travail (la classe `QgsRubberBand` se contentera de dessiner un rectangle autour du point désiré). Comment utiliser un symbole de sommet:

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

Le code ci-dessus dessinera une croix rouge à la position [0,0]. Il est possible de personnaliser le type d'icône, la taille, la couleur et la taille du crayon:

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

Pour cacher temporairement des symboles de sommet et les supprimer du canevas, on peut utiliser les mêmes techniques que pour les contours d'édition.

## 8.4 Ecrire des outils cartographiques personnalisés

Vous pouvez écrire vos propres outils pour implémenter un comportement personnalisé aux actions lancées par les utilisateurs sur le canevas.

Les outils de carte doivent hériter de la classe `QgsMapTool` ou de toute autre classe dérivée et être sélectionnés comme outils actifs dans le canevas en utilisant la méthode `setMapTool()` que nous avons déjà rencontrée.

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, Qgs.Polygon)
self.rubberBand.setColor(Qt.red)
self.rubberBand.setWidth(1)
self.reset()

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

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

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

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

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

def showRect(self, startPoint, endPoint):
    self.rubberBand.reset(Qgs.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)
```

---

## Rendu cartographique et Impression

---

- Rendu simple
- Rendu des couches ayant différents SCR
- Sortie utilisant un compositeur de carte
  - Sortie vers une image raster
  - Export en PDF

Il existe deux approches pour effectuer un rendu de données en entrée dans une carte: soit on utilise une méthode rapide avec la classe `QgsMapRenderer`, soit on produit une sortie plus élaborée en utilisant la classe `QgsComposition` et ses dérivés.

### 9.1 Rendu simple

Rendu de quelques couches en utilisant `QgsMapRenderer`: créer le périphérique d'affichage (`QImage`, `QPainter` etc.), paramétrer un jeu de couches, l'étendue, la taille de la sortie et faire le rendu:

```
# 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

S'il y a plusieurs couches et qu'elles ont des SCR différents, l'exemple simple ci-dessus ne fonctionnera probablement pas : pour obtenir les bonnes valeurs des calculs de l'emprise vous devez définir explicitement le SCR de la destination et activer la reprojection à la volée comme montré dans l'exemple ci-dessous (seule la configuration du rendu est décrite)

```
...
# 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 Sortie utilisant un compositeur de carte

Le compositeur de carte est un outil très pratique lorsqu'on veut réaliser une sortie plus complexe que le rendu présenté ci-dessus. En utilisant le compositeur, il est possible de créer des mises en page de cartes complexes en utilisant des vues de cartes, des étiquettes, des légendes, des tables ainsi que d'autres éléments qui sont généralement présents dans les cartes papier. Les mises en page peuvent ensuite être exportées en PDF, dans des images raster ou directement imprimées.

Le compositeur est composé d'un ensemble de classes. Elles appartiennent toutes à la bibliothèque core. L'application QGIS dispose d'une interface graphique dédiée pour le placement des éléments mais celle-ci n'est pas disponible dans la bibliothèque graphique. Si vous n'êtes pas familier de la [structure de Vue Graphique de Qt](#), nous vous encourageons à lire cette documentation car le compositeur est basé dessus. Consultez aussi [La documentation Python sur l'implémentation de QGraphicView](#).

La classe principale du compositeur est `QgsComposition` qui est dérivée de la classe `QGraphicsScene`. Créons-en une:

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

Veuillez noter que la composition prend une instance de `QgsMapRenderer` en argument. Ce code est utilisable au sein de l'application QGIS et il utilisera le moteur de rendu de la carte depuis le canevas de carte. La composition utilise plusieurs paramètres du moteur de rendu de carte, principalement le jeu par défaut des couches et l'emprise actuelle. Lorsqu'on utilise le compositeur dans une application autonome, vous pouvez créer votre instance de moteur de rendu de carte de la même manière que précédemment et la passer à la composition.

Il est possible d'ajouter plusieurs éléments (carte, étiquette, etc.) à la composition. Ces éléments doivent hériter de la classe `QgsComposerItem`. Les éléments actuellement gérés sont les suivants:

- 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**

Par défaut, les nouveaux éléments du compositeur ont une position nulle (bord supérieur gauche de la page) et une taille à zéro. La position et la taille sont toujours mesurées en millimètres

```
# 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)
```

Par défaut, un cadre est dessiné autour de chaque élément. Voici comment le supprimer:

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

En plus de créer les éléments du compositeur à la main, QGIS gère des modèles de composition qui sont des compositions dont l'ensemble des objets est enregistré dans un fichier .qpt (syntaxe XML). Malheureusement, cette fonctionnalité n'est pas encore disponible dans l'API.

Une fois la composition prête (les éléments de compositeur ont été créés et ajoutés à la composition), nous pouvons en générer une sortie raster et/ou vecteur.

Les paramètres d'impression par défaut sont une taille de page en A4 et une résolution de 300dpi. Vous pouvez les changer si nécessaire. La taille du papier est paramétrée en millimètres

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

### 9.3.1 Sortie vers une image raster

Le code qui suit montre comment effectuer le rendu d'une composition dans une image raster:

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

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

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

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

### 9.3.2 Export en PDF

Le code qui suite effectue un rendu d'une composition dans un fichier PDF:

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

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



---

## Expressions, Filtrage et Calcul de valeurs

---

- Analyse syntaxique d'expressions
- Évaluation des expressions
  - Expressions basiques
  - Expressions avec entités
  - Gestion des erreurs
- Exemples

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 précedence 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

L'exemple suivant évaluera l'expression renseignée sur une entité. "Colonne" est le nom du champ de la couche.

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

Vous pouvez aussi utiliser `QgsExpression.prepare()` si vous avez besoin de vérifier plus d'une entité. Utiliser `QgsExpression.prepare()` accélérera le temps d'évaluation.

```
>>> 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"
```



---

## 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.

Nous pouvons faire des différences entre différents types de paramètres :

- **Paramètres globaux** — ils sont liés à l'utilisateur d'une machine en particulier. QGIS enregistre lui-même un certain nombre de variables globales, par exemple, la taille de la fenêtre principale ou la tolérance d'accrochage par défaut. Cette fonctionnalité est fournie directement par la bibliothèque Qt grâce à la classe `QSettings`. Par défaut, cette classe enregistre les paramètres dans l'environnement "natif" du système d'exploitation — base de registre (sous Windows), fichier `.plist` (sous macOS) ou fichier `.ini` (sous Unix). La [documentation QSettings](#) est complète et nous ne présenterons donc qu'un cas simple :

```
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)
```

Le second paramètre de la méthode `value()` est optionnel et indique la valeur par défaut lorsqu'aucune valeur n'existe pour le paramètre nommé.

- **Paramètres du projet** — ils varient suivant les différents projets et sont de fait reliés au fichier de projet. On y trouve par exemple la couleur de fond du canevas de cartes ou le système de référence de coordonnées (SCR) de destination. Un fond blanc est WGS84 peuvent convenir à un projet, un fond jaune et une projection UTM seront plus adaptés à un autre projet. Voici un exemple d'utilisation:

```
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]
```

Comme vous pouvez le constater, la méthode `writeEntry()` est utilisée pour tous les types de données mais il existe plusieurs méthodes pour lire la valeur du paramètre et la méthode qui correspond doit être utilisée selon le type de données.

- **Paramètres de couche cartographique** — ces paramètres sont liés à une instance particulière de couche cartographique au sein d'un projet. Ils ne sont *pas* connectés à la source de données sous-jacente d'une couche. Si vous créez deux instances de couche à partir d'un fichier Shape, elles ne partageront pas leurs paramètres. Les paramètres sont stockés dans le fichier de projet de manière à ce que lorsque l'utilisateur ouvre à nouveau le projet, les paramètres liés à la couche sont encore présents. Cette fonctionnalité a été ajoutée à QGIS 1.4. L'API est similaire à celle de la classe `QSettings`: elle récupère les données et renvoie des instances de la classe `QVariant`:

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

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

---

## Communiquer avec l'utilisateur

---

- Afficher des messages: La classe `QgsMessageBar`
- Afficher la progression
- Journal

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 Afficher des messages: La classe `QgsMessageBar`

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=QgsMe
```

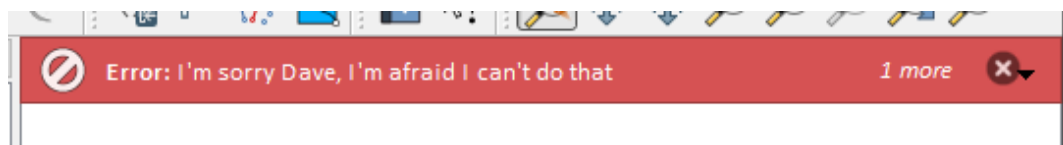


Figure 12.1: Barre de message de QGIS

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

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

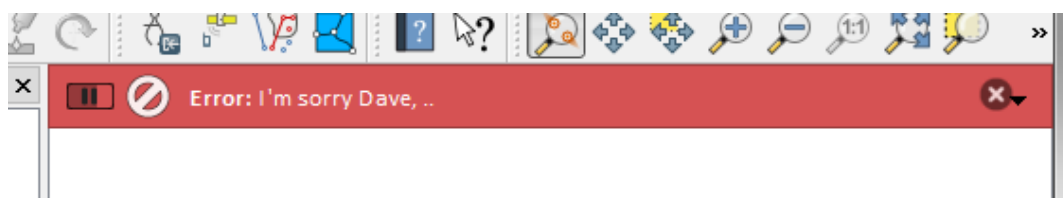


Figure 12.2: Barre de message de Qgis avec décompte

Les exemples ci-dessus montrent une barre d'erreur. Le paramètre `level` peut être utilisé pour créer des messages d'avertissement ou d'information en utilisant respectivement les constantes `QgsMessageBar.WARNING` ou `QgsMessageBar.INFO`.

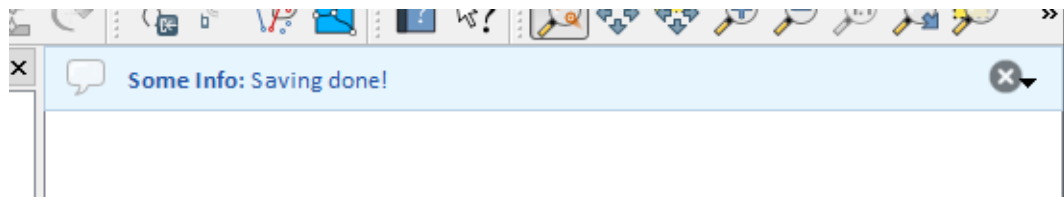


Figure 12.3: Barre de message QGis (info)

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)
```

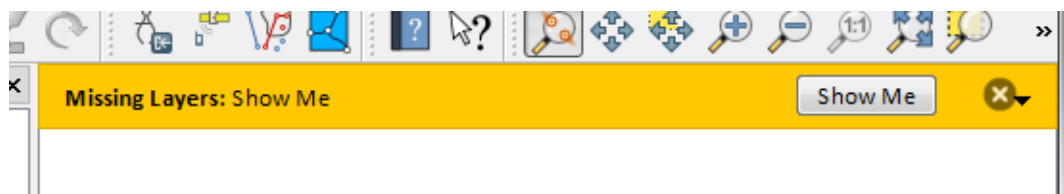


Figure 12.4: Barre de message QGis avec un bouton

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 a 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 utiliser dans la console.

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



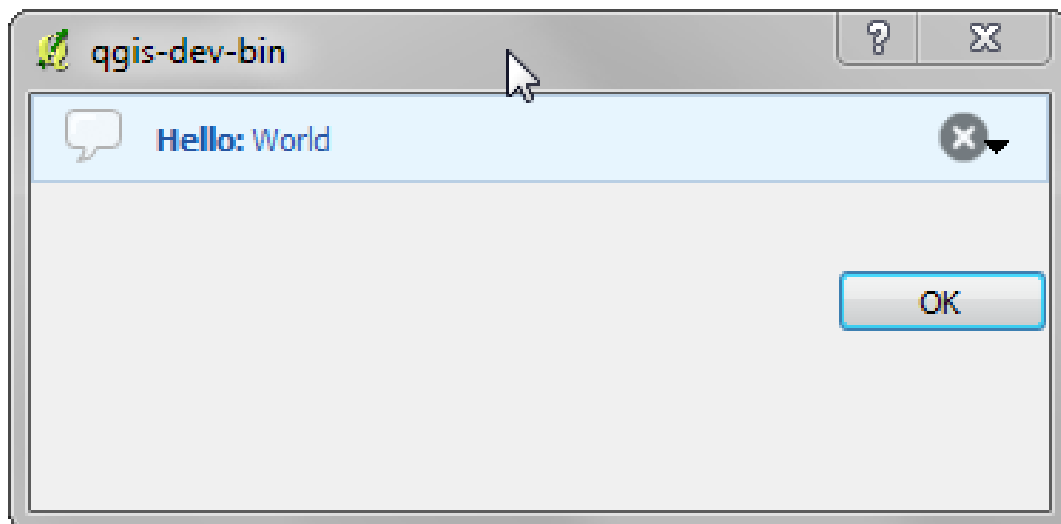


Figure 12.5: Barre de message QGIS avec une boîte de dialogue personnalisée

```

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()

```

Vous pouvez également utiliser la barre d'état incorporée pour afficher une progression, comme dans l'exemple suivant

```

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

```



---

## Développer des extensions Python

---

- Écriture d'une extension
  - Fichiers de l'extension
- Contenu de l'extension
  - Métadonnées de l'extension
  - `__init__.py`
  - `mainPlugin.py`
  - Fichier de ressources
- Documentation
- Traduction
  - Prérequis logiciels
  - Fichiers et répertoire
    - \* Fichier `.pro`
    - \* fichier `.ts`
    - \* fichier `.qm`
  - Translate using Makefile
  - Charger le plugin

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  $\geq 1.8.0$ . 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
<code>name</code>	Vrai	texte court contenant le nom de l’extension
<code>qgisMinimumVersion</code>	Vrai	version minimum de QGIS en notation par points
<code>qgisMaximumVersion</code>	Faux	version maximum de QGIS en notation par points
<code>description</code>	Vrai	un texte court qui décrit l’extension. Le HTML n’est pas autorisé
<code>about</code>	Vrai	un texte long qui décrit l’extension en détail, pas de HTML autorisé
<code>version</code>	Vrai	texte court avec le numéro de version par points
<code>author</code>	Vrai	nom de l’auteur
<code>email</code>	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
<code>changelog</code>	Faux	texte, peut être multi-lignes, pas de HTML autorisé
<code>experimental</code>	Faux	indicateur booléen, <i>Vrai</i> ou <i>Faux</i>
<code>deprecated</code>	Faux	indicateur booléen, <i>Vrai</i> ou <i>Faux</i> , s’applique à l’extension entière et pas simplement à la version chargée
<code>tags</code>	Faux	liste séparée par une virgule, les espaces sont autorisés à l’intérieur des balises individuelles
<code>homepage</code>	Faux	une URL valide pointant vers la page d’accueil de l’extension
<code>repository</code>	Vrai	une URL valide pour le dépôt du code source
<code>tracker</code>	Faux	une URL valide pour les billets et rapports de bugs
<code>icon</code>	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)
<code>category</code>	Faux	soit <i>Raster</i> , <i>Vector</i> , <i>Database</i> ou <i>Web</i>

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

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

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

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

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

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

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

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

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

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

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`.



---

## Infrastructure d'authentification

---

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

### 14.1 Introduction

Les infrastructures d'authentification de la référence utilisateur 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. L'API d'authentification sera modifiée dans 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 'authenticifié'. 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 = 'fmls770'
```

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

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

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

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

## Available Authentication methods

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

1. Basic User and password authentication
2. Identity-Cert Identity certificate authentication
3. PKI-Paths PKI paths authentication
4. PKI-PKCS#12 PKI PKCS#12 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
bundle = QgsPkiBundle.fromPemPaths( "/path/to/alice-cert.pem",
                                   "/path/to/alice-key_w-pass.pem",
                                   "unlock_pwd",
                                   "list_of_CAs_to_bundle" )

assert bundle is not None
assert bundle.isValid()
```

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

### 14.3.3 Remove entry from authdb

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

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

### 14.3.4 Leave authcfg expansion to QgsAuthManager

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

---

**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:

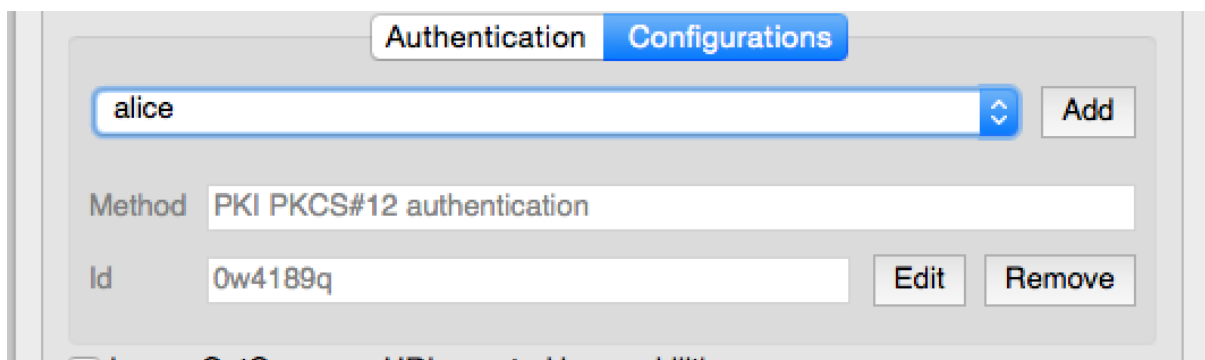
```
http = NetworkAccessManager(authid="my_authCfg", exception_class=My_FailedRequestError)
try:
    response, content = http.request( "my_rest_url" )
except My_FailedRequestError, e:
    # Handle exception
    pass
```

## 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:

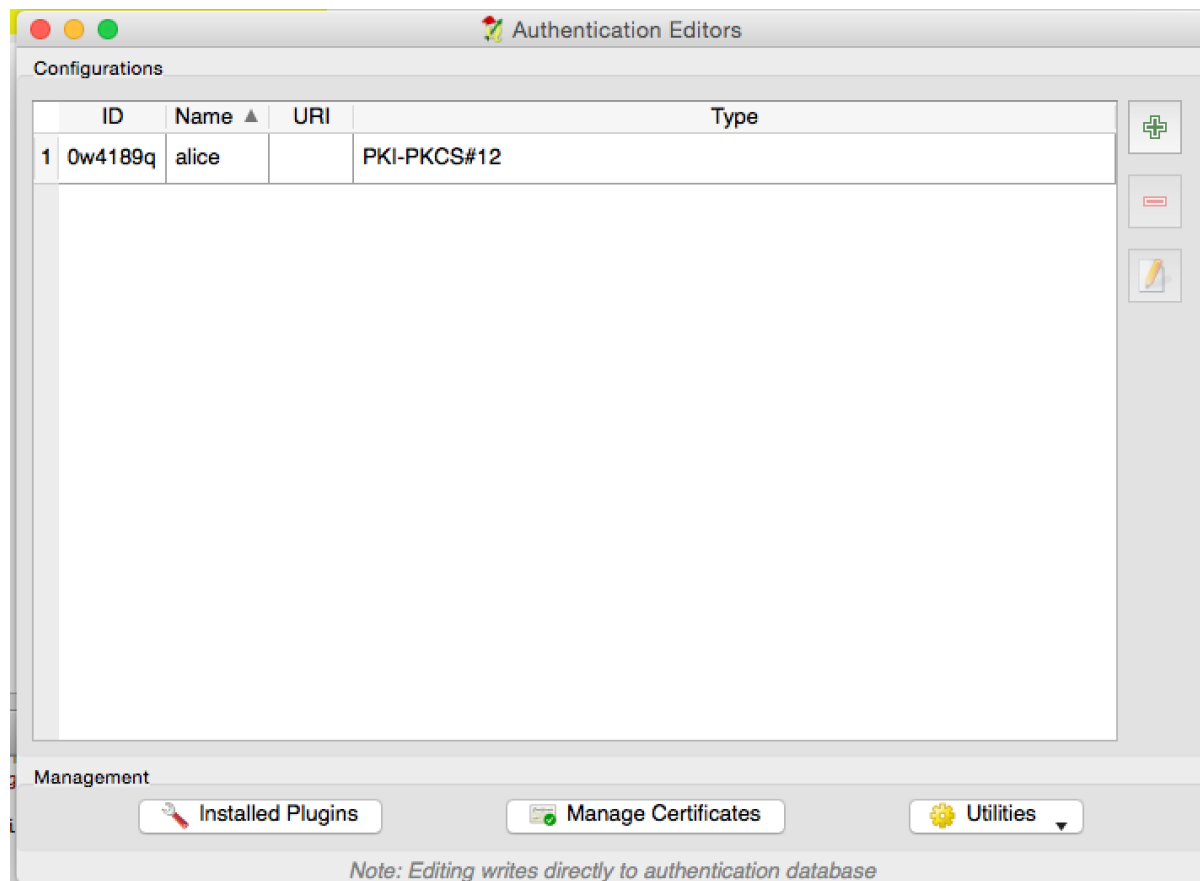
```
# create the instance of the QgsAuthConfigSelect GUI hierarchically linked to
# the widget referred with 'parent'
gui = QgsAuthConfigSelect( parent, "postgres" )
# add the above created gui in a new tab of the interface where the
```

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

L'exemple ci-dessus est tiré du code source de QGIS <<https://github.com/qgis/QGIS/blob/master/src/providers/postgres/qgspgnewco>>. Le second paramètre du constructeur de GUI fait référence au fournisseur de données. Le paramètre est utilisé pour restreindre la compatibilité de la méthode d'authentification avec le fournisseur spécifié.

## 14.5.2 Authentication Editor GUI

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



and can be used as in the following snippet:

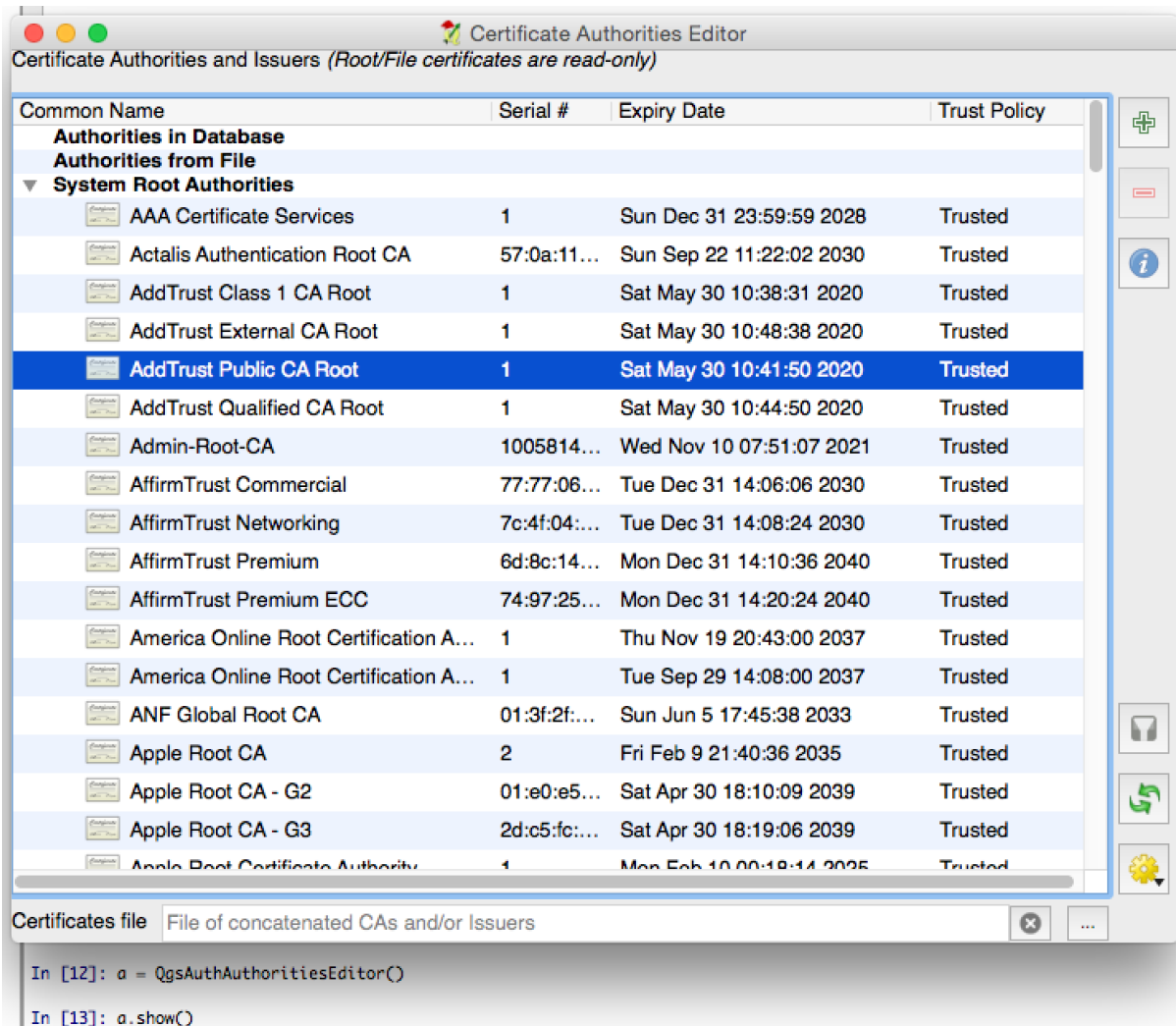
```
# create the instance of the QgsAuthEditorWidgets GUI hierarchically linked to
# the widget referred with 'parent'
gui = QgsAuthConfigSelect( parent )
gui.show()
```

an integrated example can be found in the related [test](#)

## 14.5.3 Authorities Editor GUI

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

and can be used as in the following snippet:



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

---

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

---

- Note sur la configuration de l'EDI sous Windows
- Débogage à l'aide d'Eclipse et PyDev
  - Installation
  - Préparation de QGIS
  - Configuration d'Eclipse
  - Configurer le débogueur
  - Permettre à Eclipse de comprendre l'API
- Débogage à l'aide de PDB

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 Pyscripiter:

- Faites une copie du fichier `qgis-unstable.bat` et renommez-le en `pyscripiter.bat`.
- Ouvrez-le dans un éditeur et supprimez la dernière ligne, celle qui lance QGIS.
- Ajoutez une ligne qui pointe vers l'exécutable de Pyscripiter 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 Pyscripiter 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:\pyscripiter\pyscripiter.exe --python25 --pythondllpath=C:\OSGeo4W\bin
```

Désormais, lorsque vous cliquerez sur ce fichier batch, il lancera Pyscripiter 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.

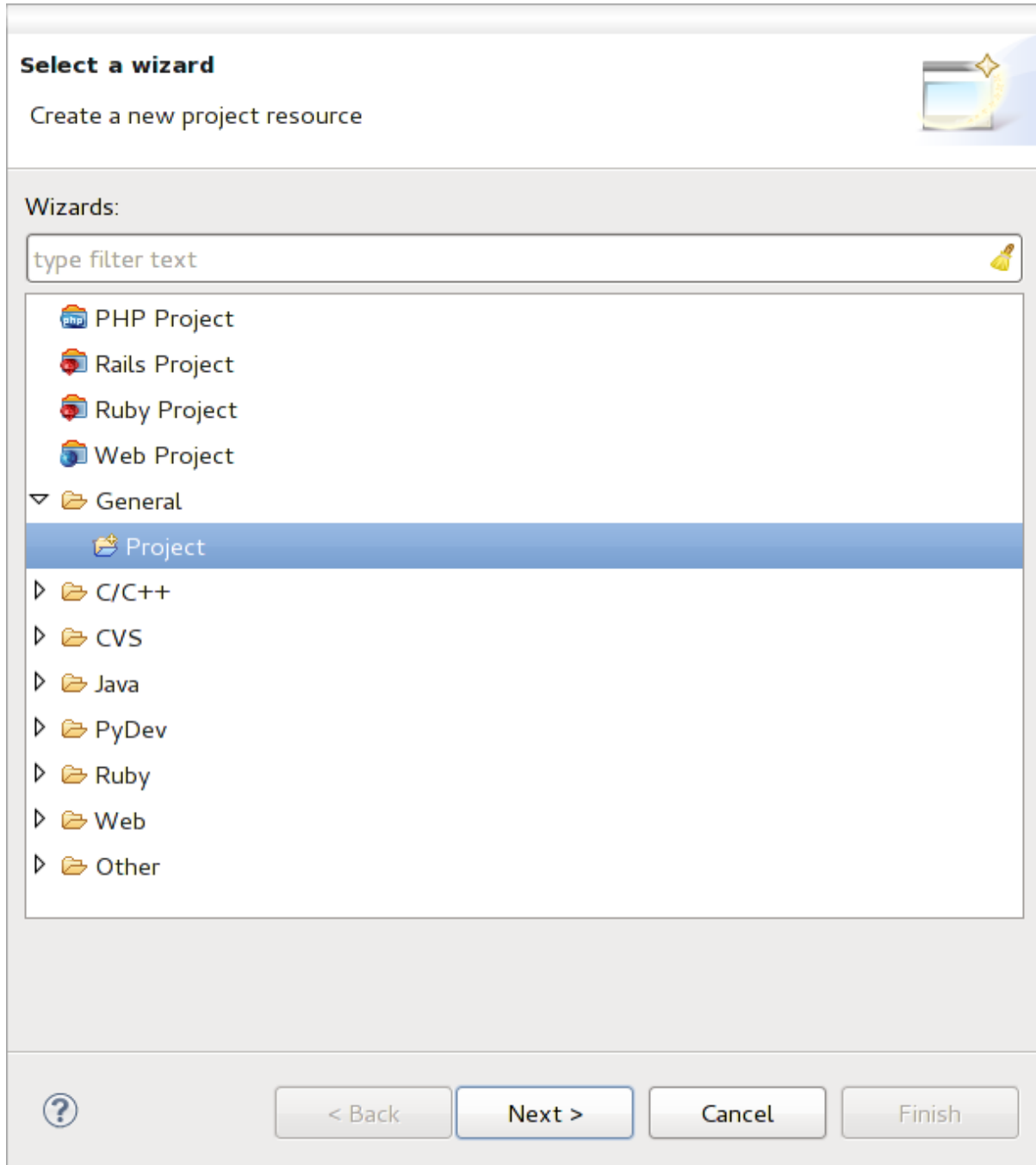


Figure 15.1: Projet Eclipse

## 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).

```

87         self.verticalExaggerationChanged.emit(val)
88
89     def printProfile(self):
90         printer = QPrinter( QPrinter.HighResolution )
91         printer.setOutputFormat( QPrinter.PdfFormat )
92         printer.setPaperSize( QPrinter.A4 )
93         printer.setOrientation( QPrinter.Landscape )
94
95         printPreviewDlg = QPrintPreviewDialog( )
96         printPreviewDlg.paintRequested.connect( self.printRequested )
97
98         printPreviewDlg.exec_()
99
100     @pyqtSlot( QPrinter )
101     def printRequested( self, printer ):
102         self.webView.print_( printer )
103

```

Figure 15.2: 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.

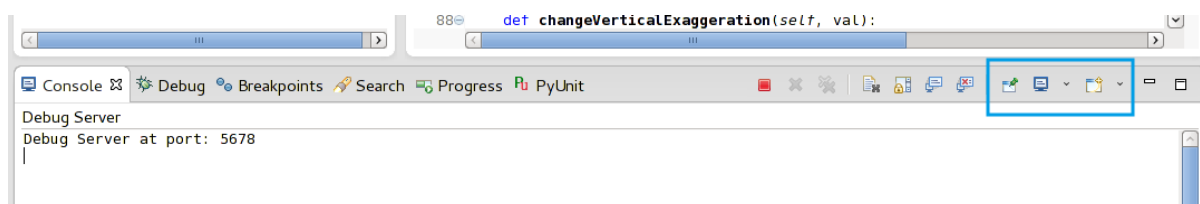


Figure 15.3: Console de débogage de PyDev

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* → *Interpréteur* → *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*.

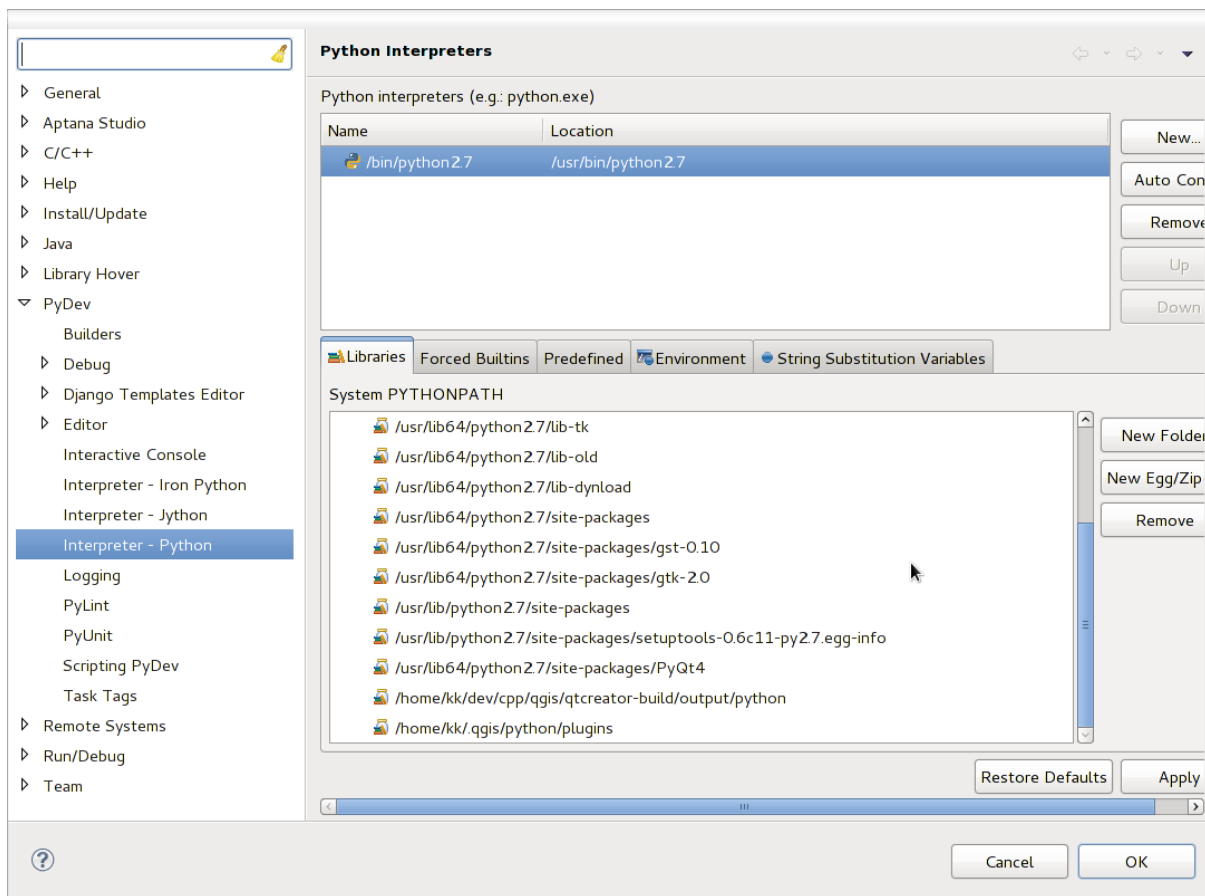


Figure 15.4: Console de débogage de PyDev

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

---

## 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
```

---

## 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)
```



---

## Publier votre extension

---

- Métadonnées et noms
- Code et aide
- Dépôt officiel des extensions Python
  - Permissions
  - Gestion de la confiance
  - Validation
  - Structure d'une 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'emballage 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](#).

Veillez 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, ressources\_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 son 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
```



---

## Extraits de code

---

- Comment appeler une méthode à l'aide d'un raccourci clavier
- Comment activer des couches:
- Comment accéder à la table attributaire des entités sélectionnées

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)
```

---

## Créer une extensions Processing

---

- Créer une extension qui ajoute un fournisseur d'algorithmes
- Créer une extension qui contient un jeu de scripts de traitements

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).

Ce document décrit comment créer un plugin pour ajouter de nouvelles fonctionnalités en tant qu'algorithme de traitement.

Il y a principalement deux façons d'y parvenir :

- Créer une extension qui ajoute un fournisseur d'algorithmes: cette option est plus complexe mais offre beaucoup de flexibilité
- Créer une extension qui contient un jeu de scripts de traitements: la solution la plus simple, il vous faut simplement un jeu (de fichiers) de scripts de traitement.

### 20.1 Créer une extension qui ajoute un fournisseur d'algorithmes

Pour créer un fournisseur d'algorithmes, suivez ces étapes:

- 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 référer pour plus d'informations.

### 20.2 Créer une extension qui contient un jeu de scripts de traitements

Pour créer un jeu de scripts de traitements, suivez ces étapes:

- Créez vos scripts comme décrit dans le livre de recette PyQGIS. Tous les scripts que vous souhaitez devraient être disponibles dans la boîte à outils Processing.

- Dans le groupe *Scripts/Outils* de la boîte à outils Processing, double-cliquez sur l'entrée *Créer une extension de collection de scripts*. Une fenêtre s'affichera où vous pourrez sélectionner les scripts que vous voulez ajouter à l'extension (à partir du jeu disponible dans la boîte à outils) ainsi que certaines informations supplémentaires requises pour les métadonnées de l'extension.
- Cliquez sur OK et l'extension est créée.
- Vous pouvez ajouter des scripts additionnels à l'extension en ajoutant des fichiers de script python dans le répertoire *scripts* dans le répertoire de l'extension concernée.

---

## Bibliothèque d'analyse de réseau

---

- Information générale
- Construire un graphe
- Analyse de graphe
  - Trouver les chemins les plus courts
  - Surfaces de disponibilité

Depuis la révision [ee19294562](#) (QGIS  $\geq$  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 polygones)
- 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 polygones)
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 polygones. Les nœuds des polygones deviendront les sommets du graphe et les segments des polygones 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.

La conversion d'une couche vecteur en graphe est réalisée en utilisant un **Constructeur** de motifs de programmation. Un graphe est construit en utilisant un élément appelé **Directeur**. Pour l'instant, il n'y a qu'un seul Directeur: **QgsLineVectorLayerDirector**. Le directeur créé les paramètres de base qui seront utilisés pour la construction d'un graphe à partir d'une couche vecteur de ligne, utilisée par le Constructeur pour créer le graphe. Pour l'instant, comme pour le cas du directeur, il n'existe qu'un seul constructeur: **QgsGraphBuilder**, qui créé des objets **QgsGraph**. Vous pouvez implémenter vos propres constructeurs qui produiront des graphes compatibles avec des bibliothèques telles que **BGL** ou **NetworkX**.

Pour calculer les propriétés des arcs, une **stratégie** basée sur les motifs de programmation est employée. Pour l'instant, seule la stratégie **QgsDistanceArcProperter** est disponible; elle prend en compte la longueur de la route. Vous pouvez implémenter votre propre stratégie qui utilisera tous les paramètres nécessaires. Par exemple, l'extension **RoadGraph** utilise une stratégie qui détermine le temps de trajet en utilisant les longueurs d'arc et la vitesse à partir d'attributs.

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 bidirectionnelles (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.

Pour obtenir l’arbre du chemin le plus court, utilisez les méthodes `shortestTree()` et `dijkstra()` de la classe `QgsGraphAnalyzer`. Il est recommandé d’utiliser la méthode `dijkstra()` car elle fonctionne plus rapidement et utilise la mémoire de manière plus efficace.

La méthode `shortestTree()` est utile lorsque vous voulez approcher l'arbre du chemin le plus court. Elle crée 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)
```

Voici un exemple de code très simple pour afficher l'arbre du chemin le plus court en utilisant un graphe créé avec la méthode `shortestTree()` (sélectionnez la couche de polylignes dans la table des matières et remplacez les coordonnées avec les vôtres). **Attention**, ce code est juste un exemple, il crée de nombreux objets `QgsRubberBand` et il reste très lent sur les jeux de données volumineux.

```
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 polygones 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))
```

---

## Extensions Python pour QGIS Server

---

- Architecture des extensions de filtre serveur
  - requestReady
  - sendResponse
  - responseComplete
- Déclencher une exception depuis une extension
- Écriture d'une extension serveur
  - Fichiers de l'extension
  - `__init__.py`
  - `HelloServer.py`
  - Modifier la couche en entrée
  - Modifier ou remplacer la couche en sortie
- Extension de contrôle d'accès
  - Fichiers de l'extension
  - `__init__.py`
  - `AccessControl.py`
  - `layerFilterExpression`
  - `layerFilterSubsetString`
  - `layerPermissions`
  - `authorizedLayerAttributes`
  - `allowToEdit`
  - `cacheKey`

Les extensions Python peuvent aussi être exécutées avec QGIS Server (voir *label\_qgisserver*): en utilisant l'*interface serveur* (`QgsServerInterface`) une extension Python exécutée sur le serveur peut modifier le comportement des services principaux existants (WMS, WFS etc.).

Avec l'*interface de filtre serveur* (`QgsServerFilter`), nous pouvons modifier les paramètres d'entrée, modifier la sortie ou encore fournir de nouveaux services.

Avec l'*interface de contrôle d'accès* (`QgsAccessControlFilter`) nous pouvons appliquer des restrictions d'accès sur chaque requête.

### 22.1 Architecture des extensions de filtre serveur

Les extensions Python de serveur sont chargées une fois que l'application CGI démarre. Elles enregistrent une ou plusieurs classes `QgsServerFilter` (à partir de ce point, il serait judicieux d'aller consulter [les documentations de l'API des extensions serveur](#)). Chaque filtre doit implémenter au moins une des trois fonctions de rappel:

- `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

Une extension serveur est simplement une extension QGIS en Python, comme décrite dans *Développer des extensions Python*, qui fournit une interface additionnelle (ou alternative): une extension QGIS Desktop typique a accès à l'application à travers l'instance de classe `QgisInterface` mais une extension serveur a également accès à la classe `QgsServerInterface`.

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

```
server=True
```

L'extension d'exemple développée ici (avec d'autres exemple de filtres) est disponible sur github: [Extension exemple QGIS HelloServer](#)

### 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

Ce fichier est requis par le système d'import de Python. QGIS Server impose aussi que ce fichier contienne une fonction `serverClassFactory()` qui est appelée lorsque l'extension est chargée dans QGIS Server. Elle reçoit une référence vers une instance de la classe `QgsServerInterface` et doit renvoyer l'instance de la classe de l'extension. Voici à quoi devrait ressembler le fichier `__init__.py`:

```
# -*- 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 utilisées 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', QgsMess
        else:
            QgsMessageLog.logMessage("FAIL - ParamsFilter.responseComplete", 'plugin', QgsMess
```

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

```
src/core/qgsmessageolog.cpp: 45: (logMessage) [0ms] 2014-12-12T12:39:29 plugin[0] HelloServerServ
src/core/qgsmessageolog.cpp: 45: (logMessage) [1ms] 2014-12-12T12:39:29 Server[0] Server plugin H
src/core/qgsmessageolog.cpp: 45: (logMessage) [0ms] 2014-12-12T12:39:29 Server[0] Server python p
src/mapserver/qgsgetrequesthandler.cpp: 35: (parseInput) [0ms] query string is: SERVICE=HELLO&re
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/qgsmessageolog.cpp: 45: (logMessage) [0ms] 2014-12-12T12:39:29 plugin[0] HelloFilter.req
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 t
src/mapserver/qgshttprequesthandler.cpp: 59: (setHttpResponse) [0ms] Byte array looks good, sett
src/core/qgsmessageolog.cpp: 45: (logMessage) [0ms] 2014-12-12T12:39:29 plugin[0] HelloFilter.resp
src/core/qgsmessageolog.cpp: 45: (logMessage) [0ms] 2014-12-12T12:39:29 plugin[0] SUCCESS - Param
src/core/qgsmessageolog.cpp: 45: (logMessage) [0ms] 2014-12-12T12:39:29 plugin[0] RemoteConsoleFi
src/mapserver/qgshttprequesthandler.cpp: 158: (sendResponse) [0ms] Sending HTTP response
src/core/qgsmessageolog.cpp: 45: (logMessage) [0ms] 2014-12-12T12:39:29 plugin[0] HelloFilter.sen
```

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 pouvez 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 résultats 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

Ce fichier est requis par le système d'import de Python. QGIS Server impose aussi que ce fichier contienne une fonction `serverClassFactory()` qui est appelée lorsque l'extension est chargée dans QGIS Server. Elle reçoit une référence vers une instance de la classe `QgsServerInterface` et doit renvoyer l'instance de la classe de l'extension. Voici à quoi devrait ressembler le fichier `__init__.py`:

```
# -*- 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 capabilities donc pour avoir un cache par rôle vous pouvez retourner le rôle dans cette méthode. Ou retourner `None` pour complètement désactiver le cache.





- 
- API, 1
  - applications personnalisées
    - Exécution en cours, 4
    - Python, 3
  - Authentication config, 72
  - Authentication Configuration, 72
  - Base de données d'authentification, 72
  - Calcul de valeurs, 50
  - Canevas de carte, 40
    - Contour d'édition, 43
    - intégré, 41
    - Marqueurs de sommet, 43
    - Objets de canevas personnalisés, 45
    - Outils cartographiques, 42
    - Outils de personnalisation des cartes, 44
  - canevas de carte
    - architecture, 41
  - Chargement
    - couches OGR, 9
    - couches PostGIS, 9
    - couches raster, 11
    - couches SpatiaLite, 10
    - Couches vecteurs, 9
    - Fichiers de Texte Délimité, 10
    - fichiers GPX, 10
    - géométries MySQL, 10
    - Projets, 7
    - raster WMS, 11
    - Vecteur WFS, 10
  - Console
    - Python, 2
  - couche de symboles
    - création de types personnalisés, 30
    - Fonctionne avec, 29
  - Couche mémoire, 24
  - Couches de l'extension, 84
    - Héritage de QgsPluginLayer, 85
  - couches OGR
    - Chargement, 9
  - couches PostGIS
    - Chargement, 9
  - couches raster
    - Chargement, 11
    - Détails, 13
    - Faire une requête, 15
    - mono-bande, 14
    - Moteur de rendu, 13
    - multi-bande, 14
    - rafraîchissement, 15
    - Raster, 12
  - couches SpatiaLite
    - Chargement, 10
  - Couches vecteurs
    - Éditer, 20
    - Chargement, 9
    - création, 23
    - Symbologie, 26
  - démarrage
    - Python, 1
  - Entités itératives, 18
  - Environnement
    - PYQGIS\_STARTUP, 2
  - Exécution en cours
    - applications personnalisées, 4
  - Expressions, 50
    - Évaluation, 52
    - Analyse, 52
  - Extensions
    - écriture, 62
    - écriture de code, 63
    - accéder aux attributs des entités sélectionnées, 93
    - Afficher des couches, 93
    - Ajout de raccourci, 93
    - Algorithme de traitement, 94
    - Débogage, 78
    - Dépôt officiel des extensions Python, 90
    - développement, 59, 69
    - Documentation, 67
    - extraits de code, 67
    - Fichier de ressources, 66
    - Initialisation, 64
    - Interaction avec l'utilisateur, 56
    - Métadonnées, 63
    - metadata.txt, 107
    - mise en œuvre de l'aide, 67
    - publication, 87
-

- Traduction, 67
- extensions
  - test, 84
- extensions serveur
  - développement, 104
  - metadata.txt, 107
- Faire une requête
  - couches raster, 15
- Fichiers de Texte Délimité
  - Chargement, 10
- fichiers GPX
  - Chargement, 10
- Filtrage, 50
- Géométrie
  - accéder à, 36
  - Construction, 35
  - Gestion, 34
  - Prédicats et opérations, 36
- géométries MySQL
  - Chargement, 10
- impression de carte, 46
- index spatial, 23
- Métadonnées, 107
- métadonnées, 107
- Méthode d'authentification, 72
- metadata.txt, 63, 107
- Mot de passe principal, 71
- Moteur de rendu
  - Personnalisation, 32
- Paramètres
  - couche cartographique, 56
  - Global, 55
  - Lecture, 53
  - Projet, 55
  - Stockage, 53
- Personnalisation
  - Moteur de rendu, 32
- Projections, 40
- Projets
  - Chargement, 7
- PyQGIS
  - Couches vecteurs, 17
- PYQGIS\_STARTUP
  - Environnement, 2
- Python
  - applications personnalisées, 3
  - Console, 2
  - démarrage, 1
  - développer des extensions, 59
  - développer des extensions serveur, 104
  - Extensions, 3
  - Infrastructure d'authentification, 69
  - Scripts autonomes, 3
  - startup.py, 2
  - rafraîchissement
    - couches raster, 15
  - Raster
    - couches raster, 12
  - raster WMS
    - Chargement, 11
  - registre couche cartographique, 11
    - Ajout d'une couche, 12
  - Rendu de carte, 46
    - Simple, 47
  - rendu par symbole gradué, 28
  - rendu par symbole unique, 26
  - rendu par symbologie catégorisée, 27
  - resources.qrc, 66
  - Sélection des entités, 17
  - Scripts autonomes
    - Python, 3
  - Sorties
    - image raster, 50
    - PDF, 50
    - Utiliser le composeur de cartes, 48
  - Symbole
    - Fonctionne avec, 29
  - Symbologie
    - rendu par catégorie de symboles, 27
    - rendu par symbole gradué, 28
    - rendu par symbole unique, 26
  - Système de coordonnées de référence, 39
  - Vecteur WFS
    - Chargement, 10