
PyQGIS developer cookbook

Version 3.4

QGIS Project

mars 15, 2020

1	Introduction	1
1.1	Scripter dans la console Python	2
1.2	Extensions Python	2
1.3	Exécuter du code python quand QGIS démarre	3
1.4	Applications Python	3
1.5	Notes techniques sur PyQt et SIP	6
2	Chargement de projets	7
3	Chargement de couches	9
3.1	Couches vectorielles	9
3.2	Couches raster	12
3.3	QgsProject instance	13
4	Utiliser des couches raster	15
4.1	Détails d'une couche	15
4.2	Moteur de rendu	16
4.3	Interrogation des données	17
5	Utilisation de couches vectorielles	19
5.1	Récupérer les informations relatives aux attributs	20
5.2	Itérer sur une couche vecteur	20
5.3	Sélection des entités	21
5.4	Modifier des couches vecteur	23
5.5	Utilisation des index spatiaux	26
5.6	Creating Vector Layers	27
5.7	Apparence (Symbologie) des couches vecteur	30
5.8	Sujets complémentaires	38
6	Manipulation de la géométrie	41
6.1	Construction de géométrie	41
6.2	Accéder à la Géométrie	42
6.3	Prédicats et opérations géométriques	43
7	Support de projections	45
7.1	Système de coordonnées de référence	45
7.2	Transformation de CRS	46
8	Using the Map Canvas	49
8.1	Intégrer un canevas de carte	50
8.2	Contour d'édition et symboles de sommets	50
8.3	Utiliser les outils cartographiques avec le canevas	51

8.4	Ecrire des outils cartographiques personnalisés	52
8.5	Ecrire des éléments de canevas de carte personnalisés	54
9	Rendu cartographique et Impression	55
9.1	Rendu simple	55
9.2	Rendu des couches ayant différents SCR	56
9.3	Output using print layout	56
10	Expressions, Filtrage et Calcul de valeurs	59
10.1	Analyse syntaxique d'expressions	60
10.2	Évaluation des expressions	60
10.3	Gestion des erreurs dans une expression	62
11	Lecture et sauvegarde de configurations	63
12	Communiquer avec l'utilisateur	65
12.1	Showing messages. The QgsMessageBar class	65
12.2	Afficher la progression	67
12.3	Journal	68
13	Infrastructure d'authentification	69
13.1	Introduction	70
13.2	Glossaire	70
13.3	QgsAuthManager le point d'entrée	70
13.4	Adapt plugins to use Authentication infrastructure	73
13.5	Authentication GUIs	74
14	Tasks - doing heavy work in the background	77
14.1	Introduction	77
14.2	Exemples	78
15	Développer des extensions Python	83
15.1	Structuring Python Plugins	83
15.2	Code Snippets	91
15.3	Using Plugin Layers	92
15.4	IDE settings for writing and debugging plugins	93
15.5	Releasing your plugin	99
16	Créer une extension avec Processing	103
17	Bibliothèque d'analyse de réseau	105
17.1	Information générale	105
17.2	Construire un graphe	106
17.3	Analyse de graphe	107
18	Extensions Python pour QGIS Server	113
18.1	Architecture des extensions de filtre serveur	114
18.2	Déclencher une exception depuis une extension	115
18.3	Écriture d'une extension serveur	115
18.4	Extension de contrôle d'accès	119
19	Cheat sheet for PyQGIS	123
19.1	Interface utilisateur	123
19.2	Réglages	123
19.3	Barres d'outils	123
19.4	Menus	124
19.5	Canevas	124
19.6	Couches	124
19.7	Table des matières	128
19.8	Table des matières (avancé)	128

19.9 Traitement algorithmes	130
19.10 Décorateurs	131
19.11 Sources	132

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 un bon aperçu des principales fonctionnalités.

- *Scripter dans la console Python*
- *Extensions Python*
- *Exécuter du code python quand QGIS démarre*
 - *Le fichier : `startup.py`*
 - *La variable d'environnement `PYQGIS_STARTUP`*
- *Applications Python*
 - *Utiliser PyQGIS dans des scripts indépendants*
 - *Utiliser PyQGIS dans une application personnalisée*
 - *Exécuter des applications personnalisées*
- *Notes techniques sur PyQt et SIP*

Le support de Python a été introduit pour la première fois dans QGIS 0.9. Il y a de nombreuses façons d'utiliser du code python dans QGIS (elles sont traitées en détail dans les sections suivantes):

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

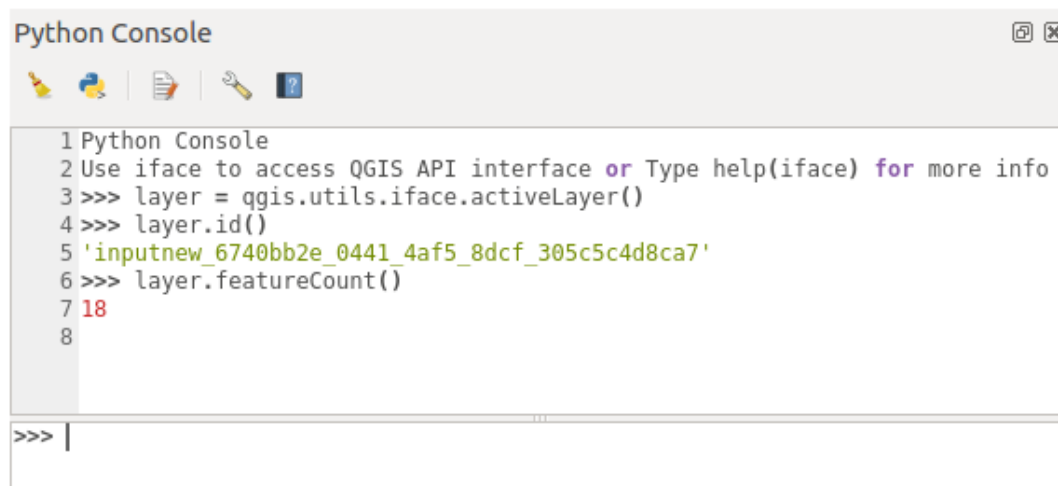
Des liaisons python sont également disponibles pour QGIS Server, ce qui permet de mettre en œuvre des extensions (see *Extensions Python pour QGIS Server*) et des liaisons qui peuvent être intégrées pour intégrer QGIS Server dans une application Python.

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

Une bonne méthode pour apprendre à réaliser des tâches classiques est de télécharger des extensions existantes depuis le *dépôt d'extensions* <<https://plugins.qgis.org/>> puis d'examiner leur code.

1.1 Scripter dans la console Python

QGIS fournit une console Python intégrée Python console pour créer des scripts. La console peut être ouverte grâce au menu: *Extensions* → *Console Python*:



```
Python Console
1 Python Console
2 Use iface to access QGIS API interface or Type help(iface) for more info
3 >>> layer = qgis.utils.iface.activeLayer()
4 >>> layer.id()
5 'inputnew_6740bb2e_0441_4af5_8dcf_305c5c4d8ca7'
6 >>> layer.featureCount()
7 18
8
>>> |
```

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 vectorielle, afficher le décompte 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 composants de l'application QGIS.

Pour simplifier la vie de l'utilisateur, les déclarations suivantes sont exécutées quand la console est ouverte (Dans le futur, il sera possible de définir plus de commandes).

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

Pour ceux qui utilisent fréquemment la console, il peut-être utile de configurer un raccourci clavier pour ouvrir la console (dans *Settings* → *Keyboard shortcuts...*).

1.2 Extensions Python

Il est possible d'étendre les fonctionnalités de QGIS en utilisant des extensions. Les extensions peuvent être écrites en Python. Les principaux avantages par rapport aux extensions C++ sont la simplicité de déploiement (pas de compilation pour chaque plateforme) et une plus grande simplicité de développement.

De nombreuses extensions couvrant de nombreuses fonctionnalités ont été écrites depuis l'introduction du support de Python. L'installateur d'extensions permet facilement aux utilisateurs de récupérer, mettre à jour et supprimer des extensions python. Voir la page des [extensions python](#) pour plus d'information sur les extensions et le développement d'extensions.

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

Note: Des extensions python sont également disponibles pour QGIS server. Voir la page [Extensions Python pour QGIS Server](#) pour plus de détails.

1.3 Exécuter du code python quand QGIS démarre

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

1. Créer un script `startup.py`
2. Définir la variable d'environnement `PYQGIS_STARTUP` sur un fichier python

1.3.1 Le fichier : `startup.py`

A chaque démarrage de QGIS, le répertoire python du profil utilisateur

- Linux: `.local/share/QGIS/QGIS3`
- Windows: `AppData\Roaming\QGIS\QGIS3`
- macOS: `Library/Application Support/QGIS/QGIS3`

est exploré à la recherche d'un fichier nommé `startup.py`. Si ce fichier existe, il est exécuté par l'interpréteur python embarqué.

Note: Le chemin par défaut dépend du système d'exploitation. Pour trouver le chemin qui fonctionne chez vous, ouvrez la console python et exécutez `QStandardPaths.standardLocations(QStandardPaths.AppDataLocation)` pour voir la liste des répertoires par défaut.

1.3.2 La variable d'environnement `PYQGIS_STARTUP`

Vous pouvez exécuter du code python juste avant la fin de l'initialisation de QGIS en définissant la variable d'environnement `PYQGIS_STARTUP` avec le chemin d'un fichier python existant.

Ce code va être exécuté avant la fin de l'initialisation de QGIS. Cette méthode est très utile pour nettoyer le chemin `sys.path`, qui peut être pollué par d'autres chemins, ou pour isoler ou charger un environnement initial sans recourir à un environnement virtuel, par exemple homebrew ou MacPorts sur macOS.

1.4 Applications Python

Il est souvent pratique de créer des scripts pour automatiser des processus. 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, ... Le module `qgis.gui` fournit 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 doivent être configurés pour trouver les ressources QGIS, comme les informations sur les projections et les fournisseurs de données pour lire des couches vecteurs ou raster. Les ressources QGIS sont initialisées en ajoutant quelques lignes au début de votre application ou de votre script. Le code pour initialiser QGIS pour des applications sur mesure ou des scripts autonomes est similaire. Des exemples sont fournis ci-dessous.

Note: 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.

# Finally, exitQgis() is called to remove the
# provider and layer registries from memory

qgs.exitQgis()
```

Nous commençons par importer le module `qgis.core` puis nous configurons le chemin de préfixe. Le chemin de 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 `setPrefixPath` est mis à `True`, ce qui contrôle si les chemins par défaut sont utilisés.

Le chemin d'installation de QGIS varie selon les plateformes; le moyen le plus simple pour trouver celle qui correspond à votre système est d'utiliser la console *Scripter dans la console Python* depuis QGIS et de vérifier la sortie de la commande `QgsApplication.prefixPath()`.

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'allons pas utiliser une interface graphique puisque nous écrivons un script autonome. `QgsApplication` étant configuré, nous chargeons les fournisseurs de données de QGIS et le registre de couches via la méthode `qgs.initQgis()`. QGIS étant initialisé, nous sommes prêts à écrire le reste de notre script. Pour finir, 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 the path to the 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. We need
# this 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.

# Finally, exitQgis() is called 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 chercher 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
- sur macOS: **export PYTHONPATH=**<qgispath>/Contents/Resources/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 peut être inconnu du système d'exploitation auquel cas vous auriez de nouveau une erreur d'import (le message peut varier selon le système):

```
>>> import qgis.core
ImportError: libqgis_core.so.3.2.0: 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>\bin;C:<qgispath>\apps<qgisrelease>\bin;%PATH% où <qgisrelease> devra être remplacé avec le type de release que vous ciblez (ex. qgis-ltr, qgis, qgis-dev)

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 rechercher les 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 pourrait être 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 MacOS, mais sous Linux laisser l'installation de QGIS à l'utilisateur via son gestionnaire de paquets .

1.5 Notes techniques sur PyQt et SIP

Nous avons choisi Python car c'est un des langages les plus adaptés pour la création de scripts. Les liaisons (bindings) PyQGIS sur QGIS3 dépendent de SIP et PyQt5. 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 liaisons Python pour Qt (PyQt) sont opérées via SIP, ce qui permet une intégration parfaite de PyQGIS avec PyQt.

Chargement de projets

Vous avez parfois besoin de charger un projet existant depuis une extension ou (le plus souvent) lorsque vous développez une application QGIS autonome (voir : :ref:'pythonapplications').

Pour charger un projet dans l'application QGIS courante, vous devez créer une instance de la classe `QgsProject`. C'est un objet singleton, ce qui vous impose d'utiliser sa méthode `instance()`. Vous pouvez appeler sa méthode `read()`, en lui passant le chemin du projet à charger:

```
# If you are not inside a QGIS console you first need to import
# qgis and PyQt classes you will use in this script as shown below:
from qgis.core import QgsProject
# 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())
'/home/user/projects/my_qgis_project.qgs'
# Load another project
project.read('/home/user/projects/my_other_qgis_project.qgs')
print(project.fileName())
'/home/user/projects/my_other_qgis_project.qgs'
```

Si vous avez besoin de réaliser des modifications du projet (par exemple ajouter ou supprimer des couches) et de sauver vos changements, appelez la méthode `<qgis.core.QgsProject.write>` de votre instance de projet. La méthode `write()` accepte également un chemin en option pour sauvegarder le projet à un nouvel emplacement:

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

Les deux fonctions `read()` et `write()` renvoie une valeur booléenne qui peut être utilisée pour vérifier que l'opération a réussi.

Note: Si vous codez une application QGIS autonome, afin de synchroniser le projet chargé avec le canevas il vous faut 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('/home/user/projects/my_other_qgis_project.qgs')
```

Chargement de couches

Le fragment de code sur cette page nécessite les importations suivantes:

```
import os # This is is needed in the pyqgis console also
from qgis.core import (
    QgsVectorLayer
)
```

- Couches vectorielles
- Couches raster
- QgsProject instance

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 créer une instance de couche vectorielle, spécifier l'identifiant de la source de données de la couche, le nom pour la couche et le nom du fournisseur de données:

```
# get the path to the shapefile e.g. /home/project/data/ports.shp
path_to_ports_layer = os.path.join(QgsProject.instance().homePath(), "data", "ports
↪", "ports.shp")

# The format is:
# vlayer = QgsVectorLayer(data_source, layer_name, provider_name)

vlayer = QgsVectorLayer(path_to_ports_layer, "Ports layer", "ogr")
if not vlayer.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.

Pour une couche vectorielle geopackage:

```
# get the path to a geopackage e.g. /home/project/data/data.gpkg
path_to_gpkg = os.path.join(QgsProject.instance().homePath(), "data", "data.gpkg")
# append the layername part
gpkg_places_layer = path_to_gpkg + "|layername=places"
# e.g. gpkg_places_layer = "/home/project/data/data.gpkg/layername=places"
vlayer = QgsVectorLayer(gpkg_places_layer, "Places layer", "ogr")
if not vlayer.isValid():
    print("Layer failed to load!")
```

La méthode la plus rapide pour ouvrir et afficher un couche vectorielle dans QGIS est `addVectorLayer()` de la classe `QgisInterface`:

```
vlayer = iface.addVectorLayer(path_to_ports_layer, "Ports layer", "ogr")
if not vlayer:
    print("Layer failed to load!")
```

Cela crée une nouvelle couche et l'ajoute au projet QGIS courant (la faisant apparaître dans la liste des couches) en une seule étape. Cette fonction renvoie une instance de la couche ou `None` 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:

- la bibliothèque OGR (shapefile et de nombreux autres formats de fichiers) — La source de données est le chemin vers le fichier:

- pour un shapefile:

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

- pour un dxf (notez les options internes dans l'uri de la source de données) :

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

- Base de données PostGIS - La source de données est une chaîne de caractères data source contenant toute l'information nécessaire pour établir une connexion avec la base PostgreSQL.

La classe `QgsDataSourceUri` peut générer cette chaîne de caractères pour vous. Notez que QGIS doit être compilé avec le support postgresQL, faute de quoi le fournisseur ne sera pas disponible:

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

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

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.

- fichiers CSV et autres fichiers avec délimiteurs — Pour ouvrir un fichier avec des points virgules comme séparateurs, contenant des champs « x » pour les coordonnées X et « y » pour les coordonnées Y, vous devriez faire ceci:

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


Note: La chaîne de texte du fournisseur de données est structuré comme une URL, le chemin doit ainsi 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 Coordonnées de Référence. Par exemple :

```
uri = "file:///some/path/file.csv?delimiter={}&crs=epsg:4723&wktField={}".
↳format(";", "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 spatialite — De manière 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 URI et utilise le fournisseur de données WFS

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

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

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

3.2 Couches raster

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

```
# get the path to a tif file e.g. /home/project/data/srtm.tif
path_to_tif = os.path.join(QgsProject.instance().homePath(), "data", "srtm.tif")
rlayer = QgsRasterLayer(path_to_tif, "SRTM layer name")
if not rlayer.isValid():
    print("Layer failed to load!")
```

To load a raster from a geopackage:

```
# get the path to a geopackage e.g. /home/project/data/data.gpkg
path_to_gpkg = os.path.join(QgsProject.instance().homePath(), "data", "data.gpkg")
# gpkg_raster_layer = "GPKG:/home/project/data/data.gpkg:srtm"
gpkg_raster_layer = "GPKG:" + path_to_gpkg + ":srtm"

rlayer = QgsRasterLayer(gpkg_raster_layer, "layer name you like", "gdal")

if not rlayer.isValid():
    print("Layer failed to load!")
```

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

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

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

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

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

Here is a description of the parameters that the WCS URI can contain:

WCS URI is composed of **key=value** pairs separated by `&`. It is the same format like query string in URL, encoded the same way. `QgsDataSourceUri` should be used to construct the URI to ensure that special characters are encoded properly.

- **url** (required) : WCS Server URL. Do not use VERSION in URL, because each version of WCS is using different parameter name for **GetCapabilities** version, see param version.
- **identifiant** (required) : Coverage name
- **time** (optional) : time position or time period (beginPosition/endPosition[/timeResolution])
- **format** (optional) : Supported format name. Default is the first supported format with tif in name or the first supported format.
- **crs** (optional) : CRS in form AUTHORITY:ID, e.g. EPSG:4326. Default is EPSG:4326 if supported or the first supported CRS.
- **username** (optional) : Username for basic authentication.
- **password** (optional) : Password for basic authentication.
- **IgnoreGetMapUrl** (optional, hack) : If specified (set to 1), ignore GetCoverage URL advertised by GetCapabilities. May be necessary if a server is not configured properly.

- **InvertAxisOrientation** (optional, hack) : If specified (set to 1), switch axis in GetCoverage request. May be necessary for geographic CRS if a server is using wrong axis order.
- **IgnoreAxisOrientation** (optional, hack) : If specified (set to 1), do not invert axis orientation according to WCS standard for geographic CRS.
- **cache** (optional) : cache load control, as described in `QNetworkRequest::CacheLoadControl`, but request is resend as `PreferCache` if failed with `AlwaysCache`. Allowed values: `AlwaysCache`, `PreferCache`, `PreferNetwork`, `AlwaysNetwork`. Default is `AlwaysCache`.

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:4326'
rlayer = QgsRasterLayer(urlWithParams, 'some layer name', 'wms')
if not rlayer.isValid():
    print("Layer failed to load!")
```

3.3 QgsProject instance

If you would like to use the opened layers for rendering, do not forget to add them to the `QgsProject` instance. The `QgsProject` instance takes ownership of layers and they can be later accessed from any part of the application by their unique ID. When the layer is removed from the project, it gets deleted, too. Layers can be removed by the user in the QGIS interface, or via Python using the `removeMapLayer()` method.

Adding a layer to the current project is done using the `addMapLayer()` method:

```
QgsProject.instance().addMapLayer(rlayer)
```

To add a layer at an absolute position:

```
# first add the layer without showing it
QgsProject.instance().addMapLayer(rlayer, False)
# obtain the layer tree of the top-level group in the project
layerTree = iface.layerTreeCanvasBridge().rootGroup()
# the position is a number starting from 0, with -1 an alias for the end
layerTree.insertChildNode(-1, QgsLayerTreeLayer(rlayer))
```

If you want to delete the layer use the `removeMapLayer()` method:

```
# QgsProject.instance().removeMapLayer(layer_id)
QgsProject.instance().removeMapLayer(rlayer.id())
```

In the above code, the layer id is passed (you can get it calling the `id()` method of the layer), but you can also pass the layer object itself.

For a list of loaded layers and layer ids, use the `mapLayers()` method:

```
QgsProject.instance().mapLayers()
```

Utiliser des couches raster

Avertissement: *Despite our constant efforts, information beyond this line may not be updated for QGIS 3. Refer to <https://qgis.org/pyqgis/master> for the python API documentation or, give a hand to update the chapters you know about. Thanks.*

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

The code snippets on this page needs the following imports if you're outside the pyqgis console:

```
from qgis.core import (  
    QgsRasterLayer,  
    QgsColorRampShader,  
    QgsSingleBandPseudoColorRenderer  
)
```

4.1 Détails d'une couche

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

The following code assumes `rlayer` is a `QgsRasterLayer` object.

```
rlayer = QgsProject.instance().mapLayersByName('srtm')[0]  
# get the resolution of the raster in layer unit  
rlayer.width(), rlayer.height()
```

```
(919, 619)
# get the extent of the layer as QgsRectangle
rlayer.extent()
<QgsRectangle: 20.06856808199999875 -34.27001076999999896, 20.83945284300000012 -
↳33.750775007000000144>
# get the extent of the layer as Strings
rlayer.extent().toString()
'20.0685680819999988,-34.2700107699999990 : 20.8394528430000001,-33.
↳75077500700000014'
# get the raster type: 0 = GrayOrUndefined (single band), 1 = Palette (single_
↳band), 2 = Multiband
rlayer.rasterType()
0
# get the total band count of the raster
rlayer.bandCount()
1
# get all the available metadata as a QgsLayerMetadata object
rlayer.metadata()
'<qgis._core.QgsLayerMetadata object at 0x13711d558>'
```

4.2 Moteur de rendu

When a raster layer is loaded, it gets a default renderer based on its type. It can be altered either in the layer properties or programmatically.

To query the current renderer:

```
rlayer.renderer()
<qgis._core.QgsSingleBandGrayRenderer object at 0x7f471c1da8a0>
rlayer.renderer().type()
'singlebandgray'
```

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

- `QgsMultiBandColorRenderer`
- `QgsPalettedRasterRenderer`
- `QgsSingleBandColorDataRenderer`
- `QgsSingleBandGrayRenderer`
- `QgsSingleBandPseudoColorRenderer`

Single band raster layers can be drawn either in gray colors (low values = black, high values = white) or with a pseudocolor algorithm that assigns colors to the values. Single band rasters with a palette can also be drawn using the palette. Multiband layers are typically drawn by mapping the bands to RGB colors. Another possibility is to use just one band for drawing.

4.2.1 Rasters mono-bande

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

```
fcn = QgsColorRampShader()
fcn.setColorRampType(QgsColorRampShader.Interpolated)
lst = [ QgsColorRampShader.ColorRampItem(0, QColor(0,255,0)),
        QgsColorRampShader.ColorRampItem(255, QColor(255,255,0)) ]
```

```
fcn.setColorRampItemList(lst)
shader = QgsRasterShader()
shader.setRasterShaderFunction(fcn)
```

The shader maps the colors as specified by its color map. The color map is provided as a list of pixel values with associated colors. There are three modes of interpolation:

- **linear** (`Interpolated`): the color is linearly interpolated from the color map entries above and below the pixel value
- **discrete** (`Discrete`): the color is taken from the closest color map entry with equal or higher value
- **exact** (`Exact`): the color is not interpolated, only pixels with values equal to color map entries will be drawn

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

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

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

Finally we have to use the `triggerRepaint` method to see the results:

```
rlayer.triggerRepaint()
```

4.2.2 Rasters multi-bandes

By default, QGIS maps the first three bands to red, green and blue to create a color image (this is the `MultiBandColor` drawing style). In some cases you might want to override these setting. The following code interchanges red band (1) and green band (2):

```
rlayer_multi = QgsProject.instance().mapLayersByName('multiband')[0]
rlayer_multi.renderer().setGreenBand(1)
rlayer_multi.renderer().setRedBand(2)
```

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

We have to use `triggerRepaint` to update the map and see the result:

```
rlayer_multi.triggerRepaint()
```

4.3 Interrogation des données

Raster values can be queried using the `sample` method of the `QgsRasterDataProvider` class. You have to specify a `QgsPointXY` and the band number of the raster layer you want to query. The method returns a tuple with the value and `True` or `False` depending on the results:

```
val, res = rlayer.dataProvider().sample(QgsPointXY(20.50, -34), 1)
```

Another method to query raster values is using the `identify` method that returns a `QgsRasterIdentifyResult` object.

```
ident = rlayer.dataProvider().identify(QgsPointXY(20.5, -34), QgsRaster.
↳ IdentifyFormatValue)

if ident.isValid():
    print(ident.results())
```

In this case, the `results` method returns a dictionary, with band indices as keys, and band values as values. For instance, something like `{1: 323.0}`

Utilisation de couches vectorielles

- *Récupérer les informations relatives aux attributs*
- *Itérer sur une couche vecteur*
- *Sélection des entités*
 - *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*
 - *Modifier des couches vecteur à l'aide d'un tampon d'édition*
 - *Ajout et Suppression de Champs*
- *Utilisation des index spatiaux*
- *Creating Vector Layers*
 - *From an instance of `QgsVectorFileWriter`*
 - *Directly from features*
 - *From an instance of `QgsVectorLayer`*
- *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.

Most work here is based on the methods of the `QgsVectorLayer` class.

5.1 Récupérer les informations relatives aux attributs

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

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

5.2 Itérer sur une couche vecteur

Iterating over the features in a vector layer is one of the most common tasks. Below is an example of the simple basic code to perform this task and showing some information about each feature. The `layer` variable is assumed to have a `QgsVectorLayer` object.

```
layer = iface.activeLayer()
features = layer.getFeatures()

for feature in features:
    # retrieve every feature with its geometry and attributes
    print("Feature ID: ", feature.id())
    # fetch geometry
    # show some information about the feature geometry
    geom = feature.geometry()
    geomSingleType = QgsWkbTypes.isSingleType(geom.wkbType())
    if geom.type() == QgsWkbTypes.PointGeometry:
        # the geometry type can be of single or multi type
        if geomSingleType:
            x = geom.asPoint()
            print("Point: ", x)
        else:
            x = geom.asMultiPoint()
            print("MultiPoint: ", x)
    elif geom.type() == QgsWkbTypes.LineGeometry:
        if geomSingleType:
            x = geom.asPolyline()
            print("Line: ", x, "length: ", geom.length())
        else:
            x = geom.asMultiPolyline()
            print("MultiLine: ", x, "length: ", geom.length())
    elif geom.type() == QgsWkbTypes.PolygonGeometry:
        if geomSingleType:
            x = geom.asPolygon()
            print("Polygon: ", x, "Area: ", geom.area())
        else:
            x = geom.asMultiPolygon()
            print("MultiPolygon: ", x, "Area: ", geom.area())
    else:
```

```

    print("Unknown or invalid geometry")
    # fetch attributes
    attrs = feature.attributes()
    # attrs is a list. It contains all the attribute values of this feature
    print(attrs)

```

5.3 Sélection des entités

In QGIS desktop, features can be selected in different ways: the user can click on a feature, draw a rectangle on the map canvas or use an expression filter. Selected features are normally highlighted in a different color (default is yellow) to draw user's attention on the selection.

Sometimes it can be useful to programmatically select features or to change the default color.

To select all the features, the `selectAll()` method can be used:

```

# Get the active layer (must be a vector layer)
layer = iface.activeLayer()
layer.selectAll()

```

To select using an expression, use the `selectByExpression()` method:

```

# Assumes that the active layer is points.shp file from the QGIS test suite
# (Class (string) and Heading (number) are attributes in points.shp)
layer = iface.activeLayer()
layer.selectByExpression('"Class"=\'B52\' and "Heading" > 10 and "Heading" <70',
↳QgsVectorLayer.SetSelection)

```

To change the selection color you can use `setSelectionColor()` method of `QgsMapCanvas` as shown in the following example:

```

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

```

To add features to the selected features list for a given layer, you can call `select()` passing to it the list of features IDs:

```

selected_fid = []

# Get the first feature id from the layer
for feature in layer.getFeatures():
    selected_fid.append(feature.id())
    break

# Add these features to the selected list
layer.select(selected_fid)

```

To clear the selection:

```

layer.removeSelection()

```

5.3.1 Accès aux attributs

Attributes can be referred to by their name:

```

print(feature['name'])

```

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

```
print(feature[0])
```

5.3.2 Itérer sur une sélection d'entités

If you only need selected features, you can use the `selectedFeatures()` method from the vector layer:

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

5.3.3 Itérer sur un sous-ensemble d'entités

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

```
areaOfInterest = QgsRectangle(450290,400520, 450750,400780)
request = QgsFeatureRequest().setFilterRect(areaOfInterest)

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

For the sake of speed, the intersection is often done only using feature's bounding box. There is however a flag `ExactIntersect` that makes sure that only intersecting features will be returned:

```
request = QgsFeatureRequest().setFilterRect(areaOfInterest).
↳setFlags(QgsFeatureRequest.ExactIntersect)
```

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

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

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

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

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

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

```
# Only return selected fields to increase the "speed" of the request
request.setSubsetOfAttributes([0,2])

# More user friendly version
request.setSubsetOfAttributes(['name','id'],layer.fields())

# Don't return geometry objects to increase the "speed" of the request
request.setFlags(QgsFeatureRequest.NoGeometry)

# Fetch only the feature with id 45
```

```
request.setFilterFid(45)

# The options may be chained
request.setFilterRect(areaOfInterest).setFlags(QgsFeatureRequest.NoGeometry).
↳setFilterFid(45).setSubsetOfAttributes([0,2])
```

5.4 Modifier des couches vecteur

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

```
caps = layer.dataProvider().capabilities()
# Check if a particular capability is supported:
if caps & QgsVectorDataProvider.DeleteFeatures:
    print('The layer supports DeleteFeatures')
```

For a list of all available capabilities, please refer to the [API Documentation of QgsVectorDataProvider](#).

To print layer's capabilities textual description in a comma separated list you can use `capabilitiesString()` as in the following example:

```
caps_string = layer.dataProvider().capabilitiesString()
# Print:
# 'Add Features, Delete Features, Change Attribute Values, Add Attributes,
# Delete Attributes, Rename Attributes, Fast Access to Features at ID,
# Presimplify Geometries, Presimplify Geometries with Validity Check,
# Transactions, Curved Geometries'
```

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.triggerRepaint()
else:
    iface.mapCanvas().refresh()
```

5.4.1 Ajout d'Entités

Create some `QgsFeature` instances and pass a list of them to provider's `addFeatures()` method. It will return two values: result (true/false) and list of added features (their ID is set by the data store).

To set up the attributes of the feature, you can either initialize the feature passing a `QgsFields` object (you can obtain that from the `fields()` method of the vector layer) or call `initAttributes()` passing the number of fields you want to be added.

```
if caps & QgsVectorDataProvider.AddFeatures:
    feat = QgsFeature(layer.fields())
```

```
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.fromPointXY(QgsPointXY(123, 456)))
(res, outFeats) = layer.dataProvider().addFeatures([feat])
```

5.4.2 Suppression d'Entités

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

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

5.4.3 Modifier des Entités

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

```
fid = 100 # ID of the feature we will modify

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

if caps & QgsVectorDataProvider.ChangeGeometries:
    geom = QgsGeometry.fromPointXY(QgsPointXY(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 useful methods to edit geometries (translate, insert or move vertex, etc.).

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

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

The methods are similar to the ones we have seen in the provider, but they are called on the `QgsVectorLayer` object instead.

For these methods to work, the layer must be in editing mode. To start the editing mode, use the `startEditing()` method. To stop editing, use the `commitChanges()` or `rollBack()` methods. The first one will commit all your changes to the data source, while the second one will discard them and will not modify the data source at all.

To find out whether a layer is in editing mode, use the `isEditable()` method.

Here you have some examples that demonstrate how to use these editing methods.

```

from qgis.PyQt.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 a given value
layer.changeAttributeValue(fid, fieldIndex, value)

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

```

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

Here is how you can use the undo functionality:

```

layer.beginEditCommand("Feature triangulation")

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

if problem_occurred:
    layer.destroyEditCommand()
    return

# ... more editing ...

layer.endEditCommand()

```

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

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 = next(layer.getFeatures())
    feat[0] = 5
    layer.updateFeature(feat)

```

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

5.4.5 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 qgis.PyQt.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()
```

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*.

5.5 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:

- create spatial index using the `QgsSpatialIndex()` class:

```
index = QgsSpatialIndex()
```

- add features to index — index takes `QgsFeature` object and adds it to the internal data structure. You can create the object manually or use one from a previous call to the provider's `getFeatures()` method.

```
index.insertFeature(feats)
```

- 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(QgsPointXY(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))
```


5.6 Creating Vector Layers

There are several ways to generate a vector layer dataset:

- the `QgsVectorFileWriter` class: A convenient class for writing vector files to disk, using either a static call to `writeAsVectorFormat()` which saves the whole vector layer or creating an instance of the class and issue calls to `addFeature()`. This class supports all the vector formats that OGR supports (GeoPackage, Shapefile, GeoJSON, KML and others).
- the `QgsVectorLayer` class: instantiates a data provider that interprets the supplied path (url) of the data source to connect to and access the data. It can be used to create temporary, memory-based layers (`memory`) and connect to OGR datasets (`ogr`), databases (`postgres`, `spatialite`, `mysql`, `mssql`) and more (`wfs`, `gpx`, `delimitedtext`...).

5.6.1 From an instance of `QgsVectorFileWriter`

```
# Write to a GeoPackage (default)
error = QgsVectorFileWriter.writeAsVectorFormat(layer,
                                                "/path/to/folder/my_data",
                                                "")

if error[0] == QgsVectorFileWriter.NoError:
    print("success!")
```

```
# Write to an ESRI Shapefile format dataset using UTF-8 text encoding
error = QgsVectorFileWriter.writeAsVectorFormat(layer,
                                                "/path/to/folder/my_esridata",
                                                "UTF-8",
                                                driverName="ESRI Shapefile")

if error[0] == QgsVectorFileWriter.NoError:
    print("success again!")
```

The third (mandatory) parameter specifies output text encoding. Only some drivers need this for correct operation - Shapefile is one of them (other drivers will ignore this parameter). Specifying the correct encoding is important if you are using international (non US-ASCII) characters.

```
# Write to an ESRI GDB file
opts = QgsVectorFileWriter.SaveVectorOptions()
opts.driverName = "FileGDB"
# if no geometry
opts.overrideGeometryType = QgsWkbTypes.NullGeometry
opts.actionOnExistingFile = QgsVectorFileWriter.ActionOnExistingFile.
↔CreateOrOverwriteLayer
opts.layerName = 'my_new_layer_name'
error = QgsVectorFileWriter.writeAsVectorFormat(layer=vlayer,
                                                fileName=gdb_path,
                                                options=opts)

if error[0] == QgsVectorFileWriter.NoError:
    print("success!")
else:
    print(error)
```

You can also convert fields to make them compatible with different formats by using the `FieldValueConverter`. For example, to convert array variable types (e.g. in Postgres) to a text type, you can do the following:

```
LIST_FIELD_NAME = 'xxxx'

class ESRIValueConverter(QgsVectorFileWriter.FieldValueConverter):

    def __init__(self, layer, list_field):
```

```

QgsVectorFileWriter.FieldValueConverter.__init__(self)
    self.layer = layer
    self.list_field_idx = self.layer.fields().indexOfName(list_field)

def convert(self, fieldIdxInLayer, value):
    if fieldIdxInLayer == self.list_field_idx:
        return QgsListFieldFormatter().representValue(layer=vlayer,
                                                    fieldIndex=self.list_field_idx,
                                                    config={},
                                                    cache=None,
                                                    value=value)

    else:
        return value

def fieldDefinition(self, field):
    idx = self.layer.fields().indexOfName(field.name())
    if idx == self.list_field_idx:
        return QgsField(LIST_FIELD_NAME, QVariant.String)
    else:
        return self.layer.fields()[idx]

converter = ESRIValueConverter(vlayer, LIST_FIELD_NAME)
#opts is a QgsVectorFileWriter.SaveVectorOptions as above
opts.fieldValueConverter = converter

```

A destination CRS may also be specified — if a valid instance of `QgsCoordinateReferenceSystem` is passed as the fourth parameter, the layer is transformed to that CRS.

For valid driver names please call the `supportedFiltersAndFormats` method or consult the `supported formats by OGR` — you should pass the value in the « Code » column as the driver name.

Optionally you can set whether to export only selected features, pass further driver-specific options for creation or tell the writer not to create attributes... There are a number of other (optional) parameters; see the `QgsVectorFileWriter` documentation for details.

5.6.2 Directly from features

```

from qgis.PyQt.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 WKBTYPENUM enum
5. layer's spatial reference (instance of
   QgsCoordinateReferenceSystem) - optional
6. driver name for the output file """

writer = QgsVectorFileWriter("my_shapes.shp", "UTF-8", fields, QgsWkbTypes.Point,
↪driverName="ESRI Shapefile")

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

# add a feature

```

```

fet = QgsFeature()

fet.setGeometry(QgsGeometry.fromPointXY(QgsPointXY(10,10)))
fet.setAttributes([1, "text"])
writer.addFeature(fet)

# delete the writer to flush features to disk
del writer

```

5.6.3 From an instance of QgsVectorLayer

Among all the data providers supported by the `QgsVectorLayer` class, let's focus on the memory-based layers. Memory provider is intended to be used mainly by plugin or 3rd party app developers. It does not store data on disk, allowing developers to use it as a fast backend for some temporary layers.

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

The memory provider also supports spatial indexing, which is enabled by calling the provider's `createSpatialIndex()` function. Once the spatial index is created you will be able to iterate over features within smaller regions faster (since it's not necessary to traverse all the features, only those in specified rectangle).

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

The constructor also takes a URI defining the geometry type of the layer, one of: "Point", "LineString", "Polygon", "MultiPoint", "MultiLineString", "MultiPolygon" or "None".

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

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

index=yes Spécifie que le fournisseur utilisera un index spatial

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

L'exemple suivant montre une URI intégrant toutes ces options

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

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

```

from qgis.PyQt.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.fromPointXY(QgsPointXY(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 = vl.extent()
print("extent:", e.xMinimum(), e.yMinimum(), e.xMaximum(), e.yMaximum())

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

5.7 Apparence (Symbologie) des couches vecteur

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

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

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

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

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

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

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

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

```
print(QgsApplication.rendererRegistry().renderersList())
# Print:
['nullSymbol',
'singleSymbol',
'categorizedSymbol',
'graduatedSymbol',
'RuleRenderer',
'pointDisplacement',
'pointCluster',
'invertedPolygonRenderer',
```

```
'heatmapRenderer',
'25dRenderer']
```

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 (renderer.dump ())
```

5.7.1 Moteur de rendu à symbole unique

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

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

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

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

```
symbol = QgsMarkerSymbol.createSimple({'name': 'square', 'color': 'red'})
layer.renderer().setSymbol(symbol)
# show the change
layer.triggerRepaint()
```

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

- circle
- square
- cross
- rectangle
- diamond
- pentagon
- triangle
- equilateral_triangle
- star
- regular_star
- arrow
- filled_arrowhead
- x

To get the full list of properties for the first symbol layer of a symbol instance you can follow the example code:

```
print (layer.renderer().symbol().symbolLayers()[0].properties())
# Prints
{'angle': '0',
'color': '0,128,0,255',
'horizontal_anchor_point': '1',
'joinstyle': 'bevel',
'name': 'circle',
'offset': '0,0',
```

```
'offset_map_unit_scale': '0,0',
'offset_unit': 'MM',
'outline_color': '0,0,0,255',
'outline_style': 'solid',
'outline_width': '0',
'outline_width_map_unit_scale': '0,0',
'outline_width_unit': 'MM',
'scale_method': 'area',
'size': '2',
'size_map_unit_scale': '0,0',
'size_unit': 'MM',
'vertical_anchor_point': '1'}
```

Cela peut être utile si vous souhaitez modifier certaines propriétés:

```
# You can alter a single property...
layer.renderer().symbol().symbolLayer(0).setSize(3)
# ... but not all properties are accessible from methods,
# you can also replace the symbol completely:
props = layer.renderer().symbol().symbolLayer(0).properties()
props['color'] = 'yellow'
props['name'] = 'square'
layer.renderer().setSymbol(QgsMarkerSymbol.createSimple(props))
# show the changes
layer.triggerRepaint()
```

5.7.2 Moteur de rendu à symboles catégorisés

When using a categorized renderer, you can query and set the attribute that is used for classification: use the `classAttribute()` and `setClassAttribute()` methods.

Pour obtenir la liste des catégories

```
for cat in renderer.categories():
    print("{}: {} :: {}".format(cat.value(), cat.label(), cat.symbol()))
```

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

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

5.7.3 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 renderer.ranges():
    print("{} - {}: {} {}".format(
        ran.lowerValue(),
        ran.upperValue(),
        ran.label(),
        ran.symbol()
    ))
```

you can again use the `classAttribute` (to find the classification attribute name), `sourceSymbol` and `sourceColorRamp` methods. Additionally there is the `mode` method which determines how the ranges were created: using equal intervals, quantiles or some other method.

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.PyQt import QtGui

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 = QgsSymbol.defaultSymbol(myVectorLayer.geometryType())
mySymbol1.setColor(myColour)
mySymbol1.setOpacity(myOpacity)
myRange1 = QgsRendererRange(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 = QgsSymbol.defaultSymbol(
    myVectorLayer.geometryType())
mySymbol2.setColor(myColour)
mySymbol2.setOpacity(myOpacity)
myRange2 = QgsRendererRange(myMin, myMax, mySymbol2, myLabel)
myRangeList.append(myRange2)
myRenderer = QgsGraduatedSymbolRenderer('', myRangeList)
myRenderer.setMode(QgsGraduatedSymbolRenderer.EqualInterval)
myRenderer.setClassAttribute(myTargetField)

myVectorLayer.setRenderer(myRenderer)
QgsProject.instance().addMapLayer(myVectorLayer)
```

5.7.4 Travailler avec les symboles

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

- `QgsMarkerSymbol` — for point features
- `QgsLineSymbol` — for line features
- `QgsFillSymbol` — for polygon features

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

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

```
for i in range(symbol.symbolLayerCount()):
    lyr = symbol.symbolLayer(i)
    print("{}: {}".format(i, lyr.layerType()))
```

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

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

Travailler avec des couches de symboles

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

You can get a complete list of the types of symbol layers you can create for a given symbol layer class with the following code:

```
from qgis.core import QgsSymbolLayerRegistry
myRegistry = QgsApplication.symbolLayerRegistry()
myMetadata = myRegistry.symbolLayerMetadata("SimpleFill")
for item in myRegistry.symbolLayersForType(QgsSymbol.Marker):
    print(item)
```

Output:

```
EllipseMarker
FilledMarker
FontMarker
GeometryGenerator
SimpleMarker
SvgMarker
VectorField
```

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

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

Créer des types personnalisés de couches de symbole

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

```
from qgis.core import QgsMarkerSymbolLayer
from qgis.PyQt.QtGui import QColor

class FooSymbolLayer(QgsMarkerSymbolLayer):

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

    def layerType(self):
        return "FooMarker"

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



```

def startRender(self, context):
    pass

def stopRender(self, context):
    pass

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

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

```

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

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

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

```

from qgis.gui import QgsSymbolLayerWidget

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

        self.layer = None

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

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

    def symbolLayer(self):
        return self.layer

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

```

This widget can be embedded into the symbol properties dialog. When the symbol layer type is selected in symbol

properties dialog, it creates an instance of the symbol layer and an instance of the symbol layer widget. Then it calls the `setSymbolLayer` method to assign the symbol layer to the widget. In that method the widget should update the UI to reflect the attributes of the symbol layer. The `symbolLayer` method is used to retrieve the symbol layer again by the properties dialog to use it for the symbol.

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

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.

```
from qgis.core import QgsSymbol, QgsSymbolLayerAbstractMetadata, \
↳QgsSymbolLayerRegistry

class FooSymbolLayerMetadata(QgsSymbolLayerAbstractMetadata):

    def __init__(self):
        QgsSymbolLayerAbstractMetadata.__init__(self, "FooMarker", QgsSymbol.Marker)

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

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

QgsApplication.symbolLayerRegistry().addSymbolLayerType(FooSymbolLayerMetadata())
```

You should pass layer type (the same as returned by the layer) and symbol type (marker/line/fill) to the constructor of the parent class. The `createSymbolLayer()` method takes care of creating an instance of symbol layer with attributes specified in the `props` dictionary. And there is the `createSymbolLayerWidget()` method which returns the settings widget for this symbol layer type.

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

5.7.5 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
from qgis.core import QgsWkbTypes, QgsSymbol, QgsFeatureRenderer

class RandomRenderer(QgsFeatureRenderer):
    def __init__(self, syms=None):
        QgsFeatureRenderer.__init__(self, "RandomRenderer")
        self.syms = syms if syms else [QgsSymbol.defaultSymbol(QgsWkbTypes.
↳geometryType(QgsWkbTypes.Point))]

    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)

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

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

    def renderer(self):
        return self.r

```

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

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

```

from qgis.gui import QgsRendererWidget, QgsColorButton

class RandomRendererWidget(QgsRendererWidget):
    def __init__(self, layer, style, renderer):
        QgsRendererWidget.__init__(self, layer, style)
        if renderer is None or renderer.type() != "RandomRenderer":
            self.r = RandomRenderer()
        else:
            self.r = renderer
        # setup UI
        self.btn1 = QgsColorButton()
        self.btn1.setColor(self.r.syms[0].color())
        self.vbox = QVBoxLayout()

```

```

self.vbox.addWidget(self.btn1)
self.setLayout(self.vbox)
self.connect(self.btn1, SIGNAL("clicked()"), self.setColor1)

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

def renderer(self):
    return self.r

```

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

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

```

from qgis.core import QgsRendererAbstractMetadata, QgsRendererRegistry,
↳ QgsApplication

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

    def createRenderer(self, element):
        return RandomRenderer()

    def createRendererWidget(self, layer, style, renderer):
        return RandomRendererWidget(layer, style, renderer)

QgsApplication.rendererRegistry().addRenderer(RandomRendererMetadata())

```

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

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

```

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

```

The icon can also be associated at any later time using the `setIcon` method of the metadata class. The icon can be loaded from a file (as shown above) or can be loaded from a Qt resource (PyQt5 includes .qrc compiler for Python).

5.8 Sujets complémentaires

A FAIRE :

- création/modification des symboles

- working with style (`QgsStyle`)
- working with color ramps (`QgsColorRamp`)
- 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*

The code snippets on this page need the following imports if you're outside the pyqgis console:

```
from qgis.core import (  
    QgsGeometry,  
    QgsPoint,  
    QgsPointXY,  
    QgsWkbTypes,  
    QgsProject,  
    QgsFeatureRequest,  
    QgsDistanceArea  
)
```

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

PyQGIS provides several options for creating a geometry:

- à partir des coordonnées

```
gPnt = QgsGeometry.fromPointXY(QgsPointXY(1,1))
print(gPnt)
gLine = QgsGeometry.fromPolyline([QgsPoint(1, 1), QgsPoint(2, 2)])
print(gLine)
gPolygon = QgsGeometry.fromPolygonXY([[QgsPointXY(1, 1),
    QgsPointXY(2, 2), QgsPointXY(2, 1)]]))
print(gPolygon)
```

Coordinates are given using `QgsPoint` class or `QgsPointXY` class. The difference between these classes is that `QgsPoint` supports M and Z dimensions.

A Polyline (Linestring) is represented by a list of points.

A Polygon is represented by a list of linear rings (i.e. closed linestrings). The first ring is the outer ring (boundary), optional subsequent rings are holes in the polygon. Note that unlike some programs, QGIS will close the ring for you so there is no need to duplicate the first point as the last.

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)

```
geom = QgsGeometry.fromWkt("POINT(3 4)")
print(geom)
```

- depuis un Well-Known-Binary (WKB)

```
g = QgsGeometry()
wkb = bytes.fromhex("0101000000000000000000004540000000000001440")
g.fromWkb(wkb)

# print WKT representation of the geometry
print(g.asWkt())
```

6.2 Accéder à la Géométrie

First, you should find out the geometry type. The `wkbType()` method is the one to use. It returns a value from the `QgsWkbTypes.Type` enumeration.

```
if gPnt.wkbType() == QgsWkbTypes.Point:
    print(gPnt.wkbType())
    # output: 1 for Point
if gLine.wkbType() == QgsWkbTypes.LineString:
    print(gLine.wkbType())
if gPolygon.wkbType() == QgsWkbTypes.Polygon:
    print(gPolygon.wkbType())
    # output: 3 for Polygon
```

As an alternative, one can use the `type()` method which returns a value from the `QgsWkbTypes.GeometryType` enumeration.

You can use the `displayString()` function to get a human readable geometry type.

```
print(QgsWkbTypes.displayString(gPnt.wkbType()))
# output: 'Point'
print(QgsWkbTypes.displayString(gLine.wkbType()))
# output: 'LineString'
print(QgsWkbTypes.displayString(gPolygon.wkbType()))
# output: 'Polygon'
```



```
Point
LineString
Polygon
```

There is also a helper function `isMultipart()` to find out whether a geometry is multipart or not.

To extract information from a geometry there are accessor functions for every vector type. Here's an example on how to use these accessors:

```
print(gPnt.asPoint())
# output: <QgsPointXY: POINT(1 1)>
print(gLine.asPolyline())
# output: [<QgsPointXY: POINT(1 1)>, <QgsPointXY: POINT(2 2)>]
print(gPolygon.asPolygon())
# output: [[<QgsPointXY: POINT(1 1)>, <QgsPointXY: POINT(2 2)>, <QgsPointXY:
↵POINT(2 1)>, <QgsPointXY: POINT(1 1)>]]
```

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

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

6.3 Prédicats et opérations géométriques

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

Let's see an example that combines iterating over the features in a given layer and performing some geometric computations based on their geometries. The below code will compute and print the area and perimeter of each country in the `countries` layer within our tutorial QGIS project.

The following code assumes `layer` is a `QgsVectorLayer` object that has Polygon feature type.

```
# let's access the 'countries' layer
layer = QgsProject.instance().mapLayersByName('countries')[0]

# let's filter for countries that begin with Z, then get their features
query = '"name" LIKE \'Z%\''
features = layer.getFeatures(QgsFeatureRequest().setFilterExpression(query))

# now loop through the features, perform geometry computation and print the results
for f in features:
    geom = f.geometry()
    name = f.attribute('NAME')
    print(name)
    print('Area: ', geom.area())
    print('Perimeter: ', geom.length())
```

Now you have calculated and printed the areas and perimeters of the geometries. You may however quickly notice that the values are strange. That is because areas and perimeters don't take CRS into account when computed using the `area()` and `length()` methods from the `QgsGeometry` class. For a more powerful area and distance calculation, the `QgsDistanceArea` class can be used, which can perform ellipsoid based calculations:

The following code assumes `layer` is a `QgsVectorLayer` object that has Polygon feature type.

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

```
layer = QgsProject.instance().mapLayersByName('countries')[0]

# let's filter for countries that begin with Z, then get their features
query = '"name" LIKE \'Z%\''
features = layer.getFeatures(QgsFeatureRequest().setFilterExpression(query))

for f in features:
    geom = f.geometry()
    name = f.attribute('NAME')
    print(name)
    print("Perimeter (m):", d.measurePerimeter(geom))
    print("Area (m2):", d.measureArea(geom))

# let's calculate and print the area again, but this time in square kilometers
print("Area (km2):", d.convertAreaMeasurement(d.measureArea(geom), QgsUnitTypes.
↵AreaSquareKilometers))
```

Alternatively, you may want to know the distance and bearing between two points.

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

# Let's create two points.
# Santa claus is a workaholic and needs a summer break,
# lets see how far is Tenerife from his home
santa = QgsPointXY(25.847899, 66.543456)
tenerife = QgsPointXY(-16.5735, 28.0443)

print("Distance in meters: ", d.measureLine(santa, tenerife))
```

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.

- Distance and area using the `QgsDistanceArea` class: [Distance matrix algorithm](#)
- [Lines to polygons algorithm](#)

Support de projections

- *Système de coordonnées de référence*
- *Transformation de CRS*

If you're outside the pyqgis console, the code snippets on this page need the following imports:

```
from qgis.core import (QgsCoordinateReferenceSystem,
                       QgsCoordinateTransform,
                       QgsProject,
                       QgsPointXY,
                       )
```

7.1 Système de coordonnées de référence

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

- spécifier le SCR par son ID

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

QGIS utilise trois identifiants différents pour chaque système de référence:

- `InternalCrsId` — Identifié utilisé dans la base de données interne de QGIS.
- `PostgisCrsId` — Identifiant utilisé dans les bases de données PostGIS
- `EpsgCrsId` — Identifiant assigné par l'organisation EPSG.

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

- 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)
assert crs.isValid()
```

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

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

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

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

Accessing spatial reference system information:

```
crs = QgsCoordinateReferenceSystem(4326)

print("QGIS CRS ID:", crs.srsid())
print("PostGIS SRID:", crs.postgisSrid())
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.isGeographic())
# check type of map units in this CRS (values defined in Qgis::units enum)
print("Map units:", crs.mapUnits())
```

Render:

```
QGIS CRS ID: 3452
PostGIS SRID: 4326
Description: WGS 84
Projection Acronym: longlat
Ellipsoid Acronym: WGS84
Proj4 String: +proj=longlat +datum=WGS84 +no_defs
Is geographic: True
Map units: 6
```

7.2 Transformation de CRS

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

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

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

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

Rendu:

```
Transformed point: <QgsPointXY: POINT(832713.79873844375833869 553423.
↔98688333143945783)>
Transformed back: <QgsPointXY: POINT(18 5)>
```

Using the Map Canvas

Avertissement: *Despite our constant efforts, information beyond this line may not be updated for QGIS 3. Refer to <https://qgis.org/pyqgis/master> for the python API documentation or, give a hand to update the chapters you know about. Thanks.*

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

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

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

Whenever the map has been panned, zoomed in/out (or some other action that triggers a refresh), the map is rendered again within the current extent. The layers are rendered to an image (using the `QgsMapRendererJob` class) and that image is displayed on the canvas. The `QgsMapCanvas` class also controls refreshing of the rendered map. Besides this item which acts as a background, there may be more **map canvas items**.

Typical map canvas items are rubber bands (used for measuring, vector editing etc.) or vertex markers. The canvas items are usually used to give visual feedback for map tools, for example, when creating a new polygon, the map tool creates a rubber band canvas item that shows the current shape of the polygon. All map canvas items are subclasses of `QgsMapCanvasItem` which adds some more functionality to the basic `QGraphicsItem` objects.

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

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

8.1 Intégrer un canevas de carte

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

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

This produces a standalone window with map canvas. It can be also embedded into an existing widget or window. When using `.ui` files and Qt Designer, place a `QWidget` on the form and promote it to a new class: set `QgsMapCanvas` as class name and set `qgis.gui` as header file. The `pyuic5` utility will take care of it. This is a very convenient way of embedding the canvas. The other possibility is to manually write the code to construct map canvas and other widgets (as children of a main window or dialog) and create a layout.

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

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

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

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

```
path_to_ports_layer = os.path.join(QgsProject.instance().homePath(),
                                   "data", "ports", "ports.shp")

vlayer = QgsVectorLayer(path_to_ports_layer, "Ports layer", "ogr")
if not vlayer.isValid():
    print("Layer failed to load!")

# add layer to the registry
QgsProject.instance().addMapLayer(vlayer)

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

# set the map canvas layer set
canvas.setLayers([vlayer])
```

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

8.2 Contour d'édition et symboles de sommets

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

Pour afficher une polyligne:

```
r = QgsRubberBand(canvas, False) # False = not a polygon
points = [QgsPoint(-100, 45), QgsPoint(10, 60), QgsPoint(120, 45)]
r.setToGeometry(QgsGeometry.fromPolyline(points), None)
```

Pour afficher un polygone:

```
r = QgsRubberBand(canvas, True) # True = a polygon
points = [[QgsPointXY(-100, 35), QgsPointXY(10, 50), QgsPointXY(120, 35)]]
r.setToGeometry(QgsGeometry.fromPolygonXY(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)
```

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

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

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

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

You can use the vertex marker like this:

```
m = QgsVertexMarker(canvas)
m.setCenter(QgsPointXY(10, 40))
```

This will draw a red cross on position **[10,45]**. It is possible to customize the icon type, size, color and pen width

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

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

8.3 Utiliser les outils cartographiques avec le canevas

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

```
from qgis.gui import *
from qgis.PyQt.QtWidgets import QAction, QMainWindow
from qgis.PyQt.QtCore import Qt

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.setLayers([layer])

    self.setCentralWidget(self.canvas)

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

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

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

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

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

    self.pan()

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

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

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

```

You can try the above code in the Python console editor. To invoke the canvas window, add the following lines to instantiate the `MyWnd` class. They will render the currently selected layer on the newly created canvas

```

w = MyWnd(iface.activeLayer())
w.show()

```

8.4 Ecrire des outils cartographiques personnalisés

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

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

Voici un exemple d'outil cartographique qui permet de définir une emprise rectangulaire en cliquant et en déplaçant

la souris sur le canevas. Lorsque le rectangle est dessiné, il exporte les coordonnées de ses limites dans la console. On utilise des éléments de contour d'édition décrits auparavant pour afficher le rectangle sélectionné au fur et à mesure de son dessin.

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

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

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

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

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

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

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

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

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

        return QgsRectangle(self.startPoint, self.endPoint)
```

```
def deactivate(self):
    QgsMapTool.deactivate(self)
    self.deactivated.emit()
```

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

The code snippets on this page needs the following imports:

```
import os
```

- *Rendu simple*
- *Rendu des couches ayant différents SCR*
- *Output using print layout*
 - *Exporting the layout*
 - *Exporting a layout atlas*

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

9.1 Rendu simple

The rendering is done creating a `QgsMapSettings` object to define the rendering options, and then constructing a `QgsMapRendererJob` with those options. The latter is then used to create the resulting image.

Here's an example:

```
image_location = os.path.join(QgsProject.instance().homePath(), "render.png")

# e.g. vlayer = iface.activeLayer()
vlayer = QgsProject.instance().mapLayersByName("countries")[0]
options = QgsMapSettings()
options.setLayers([vlayer])
options.setBackgroundColor(QColor(255, 255, 255))
options.setOutputSize(QSize(800, 600))
options.setExtent(vlayer.extent())

render = QgsMapRendererParallelJob(options)

def finished():
```

```

img = render.renderedImage()
# save the image; e.g. img.save("/Users/myuser/render.png", "png")
img.save(image_location, "png")
print("saved")

render.finished.connect(finished)

render.start()

```

9.2 Rendu des couches ayant différents SCR

If you have more than one layer and they have a different CRS, the simple example above will probably not work: to get the right values from the extent calculations you have to explicitly set the destination CRS

```

settings.setLayers(layers)
render.setDestinationCrs(layers[0].crs())

```

9.3 Output using print layout

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

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

The central class of the layout is the `QgsLayout` class, which is derived from the Qt `QGraphicsScene` class. Let us create an instance of it:

```

p = QgsProject()
layout = QgsLayout(p)
layout.initializeDefaults()

```

Now we can add various elements (map, label, ...) to the layout. All these objects are represented by classes that inherit from the base `QgsLayoutItem` class.

Here's a description of some of the main layout items that can be added to a layout.

- **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

```

map = QgsLayoutItemMap(layout)
layout.addItem(map)

```

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

```

label = QgsLayoutItemLabel(layout)
label.setText("Hello world")
label.adjustSizeToText()
layout.addItem(label)

```

- **légende**

```

legend = QgsLayoutItemLegend(layout)
legend.setLinkedMap(map) # map is an instance of QgsLayoutItemMap
layout.addItem(legend)

```

- Échelle graphique

```

item = QgsLayoutItemScaleBar(layout)
item.setStyle('Numeric') # optionally modify the style
item.setLinkedMap(map) # map is an instance of QgsLayoutItemMap
item.applyDefaultSize()
layout.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))

polygonItem = QgsLayoutItemPolygon(polygon, layout)
layout.addItem(polygonItem)

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

symbol = QgsFillSymbol.createSimple(props)
polygonItem.setSymbol(symbol)

```

- table

Once an item is added to the layout, it can be moved and resized:

```

item.attemptMove(QgsLayoutPoint(1.4, 1.8, QgsUnitTypes.LayoutCentimeters))
item.attemptResize(QgsLayoutSize(2.8, 2.2, QgsUnitTypes.LayoutCentimeters))

```

A frame is drawn around each item by default. You can remove it as follows:

```

# for a composer label
label.setFrameEnabled(False)

```

Besides creating the layout items by hand, QGIS has support for layout templates which are essentially compositions with all their items saved to a .qpt file (with XML syntax).

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

9.3.1 Exporting the layout

To export a layout, the `QgsLayoutExporter` class must be used.

```
base_path = os.path.join(QgsProject.instance().homePath())
pdf_path = os.path.join(base_path, "output.pdf")

exporter = QgsLayoutExporter(layout)
exporter.exportToPdf(pdf_path, QgsLayoutExporter.PdfExportSettings())
```

Use the `exportToImage()` in case you want to export to an image instead of a PDF file.

9.3.2 Exporting a layout atlas

If you want to export all pages from a layout that has the atlas option configured and enabled, you need to use the `atlas()` method in the exporter (`QgsLayoutExporter`) with small adjustments. In the following example, the pages are exported to PNG images:

```
exporter.exportToImage(layout.atlas(), base_path, 'png', QgsLayoutExporter.
↳ImageExportSettings())
```

Notice that the outputs will be saved in the base path folder, using the output filename expression configured on atlas.

Le fragment de code sur cette page nécessite les importations suivantes:

```
from qgis.core import (
    edit,
    QgsExpression,
    QgsExpressionContext,
    QgsFeature,
    QgsFeatureRequest,
    QgsField,
    QgsFields,
    QgsVectorLayer,
    QgsPointXY,
    QgsGeometry,
    QgsProject,
    QgsExpressionContextUtils
)
```

Expressions, Filtrage et Calcul de valeurs

- *Analyse syntaxique d'expressions*
- *Évaluation des expressions*
 - *Expressions basiques*
 - *Expressions avec entités*
 - *Filtrer une couche à l'aide d'expressions*
- *Gestion des erreurs dans une expression*

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

The following example shows how to check if a given expression can be parsed correctly:

```
exp = QgsExpression('1 + 1 = 2')
assert(not exp.hasParserError())

exp = QgsExpression('1 + 1 = ')
assert(exp.hasParserError())

assert(exp.parserErrorMessage() == '\nsyntax error, unexpected $end')
```

10.2 Évaluation des expressions

Expressions can be used in different contexts, for example to filter features or to compute new field values. In any case, the expression has to be evaluated. That means that its value is computed by performing the specified computational steps, which can range from simple arithmetic to aggregate expressions.

10.2.1 Expressions basiques

Cette expression basique est évaluée à 1, signifiant « vrai »:

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

10.2.2 Expressions avec entités

To evaluate an expression against a feature, a `QgsExpressionContext` object has to be created and passed to the evaluate function in order to allow the expression to access the feature's field values.

The following example shows how to create a feature with a field called « Column » and how to add this feature to the expression context.

```

fields = QgsFields()
field = QgsField('Column')
fields.append(field)
feature = QgsFeature()
feature.setFields(fields)
feature.setAttribute(0, 99)

exp = QgsExpression('"Column"')
context = QgsExpressionContext()
context.setFeature(feature)
assert (exp.evaluate(context) == 99)

```

The following is a more complete example of how to use expressions in the context of a vector layer, in order to compute new field values:

```

from qgis.PyQt.QtCore import QVariant

# create a vector layer
vl = QgsVectorLayer("Point", "Companies", "memory")
pr = vl.dataProvider()
pr.addAttributes([QgsField("Name", QVariant.String),
                  QgsField("Employees", QVariant.Int),
                  QgsField("Revenue", QVariant.Double),
                  QgsField("Rev. per employee", QVariant.Double),
                  QgsField("Sum", QVariant.Double),
                  QgsField("Fun", QVariant.Double)])
vl.updateFields()

# add data to the first three fields
my_data = [
    {'x': 0, 'y': 0, 'name': 'ABC', 'emp': 10, 'rev': 100.1},
    {'x': 1, 'y': 1, 'name': 'DEF', 'emp': 2, 'rev': 50.5},
    {'x': 5, 'y': 5, 'name': 'GHI', 'emp': 100, 'rev': 725.9}]

for rec in my_data:
    f = QgsFeature()
    pt = QgsPointXY(rec['x'], rec['y'])
    f.setGeometry(QgsGeometry.fromPointXY(pt))
    f.setAttributes([rec['name'], rec['emp'], rec['rev']])
    pr.addFeature(f)

vl.updateExtents()
QgsProject.instance().addMapLayer(vl)

# The first expression computes the revenue per employee.
# The second one computes the sum of all revenue values in the layer.
# The final third expression doesn't really make sense but illustrates
# the fact that we can use a wide range of expression functions, such
# as area and buffer in our expressions:
expression1 = QgsExpression('"Revenue"/"Employees"')
expression2 = QgsExpression('sum("Revenue")')
expression3 = QgsExpression('area(buffer($geometry, "Employees"))')

# QgsExpressionContextUtils.globalProjectLayerScopes() is a convenience
# function that adds the global, project, and layer scopes all at once.
# Alternatively, those scopes can also be added manually. In any case,
# it is important to always go from "most generic" to "most specific"
# scope, i.e. from global to project to layer
context = QgsExpressionContext()
context.appendScopes(QgsExpressionContextUtils.globalProjectLayerScopes(vl))

with edit(vl):

```

```

for f in vl.getFeatures():
    context.setFeature(f)
    f['Rev. per employee'] = expression1.evaluate(context)
    f['Sum'] = expression2.evaluate(context)
    f['Fun'] = expression3.evaluate(context)
    vl.updateFeature(f)

print( f['Sum'])

```

10.2.3 Filtrer une couche à l'aide d'expressions

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

```

layer = QgsVectorLayer("Point?field=Test:integer",
                      "addfeat", "memory")

layer.startEditing()

for i in range(10):
    feature = QgsFeature()
    feature.setAttributes([i])
    assert(layer.addFeature(feature))
layer.commitChanges()

expression = 'Test >= 3'
request = QgsFeatureRequest().setFilterExpression(expression)

matches = 0
for f in layer.getFeatures(request):
    matches += 1

assert(matches == 7)

```

10.3 Gestion des erreurs dans une expression

Les erreurs liées à une expression peuvent se révéler lors de l'analyse de l'expression ou de son évaluation :

```

exp = QgsExpression("1 + 1 = 2")
if exp.hasParserError():
    raise Exception(exp.parserErrorString())

value = exp.evaluate()
if exp.hasEvalError():
    raise ValueError(exp.evalErrorString())

```

Le fragment de code sur cette page nécessite les importations suivantes:

```

from qgis.core import (
    QgsProject,
    QgsSettings,
    QgsVectorLayer
)

```

Lecture et sauvegarde de configurations

Avertissement: *Despite our constant efforts, information beyond this line may not be updated for QGIS 3. Refer to <https://qgis.org/pyqgis/master> for the python API documentation or, give a hand to update the chapters you know about. Thanks.*

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 identifier différents types de paramètres :

- **global settings** — they are bound to the user at a particular machine. QGIS itself stores a lot of global settings, for example, main window size or default snapping tolerance. Settings are handled using the `QgsSettings` class, through for example the `setValue()` and `value()` methods.

Ci-après un exemple d'utilisation de ces méthodes.

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

def read():
    s = QgsSettings()
    mytext = s.value("myplugin/mytext", "default text")
    myint = s.value("myplugin/myint", 123)
    myreal = s.value("myplugin/myreal", 2.71)
    nonexistent = s.value("myplugin/nonexistent", None)
    print(mytext)
    print(myint)
    print(myreal)
    print(nonexistent)
```

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

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

Ci-après 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 (returns a tuple with the value, and a status boolean
# which communicates whether the value retrieved could be converted to
# its type, in these cases a string, an integer, a double and a boolean
# respectively)

mytext, type_conversion_ok = proj.readEntry("myplugin",
                                           "mytext",
                                           "default text")
myint, type_conversion_ok = proj.readNumEntry("myplugin",
                                              "myint",
                                              123)
mydouble, type_conversion_ok = proj.readDoubleEntry("myplugin",
                                                    "mydouble",
                                                    123)
mybool, type_conversion_ok = proj.readBoolEntry("myplugin",
                                                "mybool",
                                                123)
```

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

- **map layer settings** — these settings are related to a particular instance of a map layer with a project. They are *not* connected with underlying data source of a layer, so if you create two map layer instances of one shapefile, they will not share the settings. The settings are stored inside the project file, so if the user opens the project again, the layer-related settings will be there again. The value for a given setting is retrieved using the `customProperty()` method, and can be set using the `setCustomProperty()` one.

```
vlayer = QgsVectorLayer()
# save a value
vlayer.setCustomProperty("mytext", "hello world")

# read the value again (returning "default text" if not found)
mytext = vlayer.customProperty("mytext", "default text")
```

Communiquer avec l'utilisateur

- *Showing messages. The QgsMessageBar class*
- *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 Showing messages. The QgsMessageBar class

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

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

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

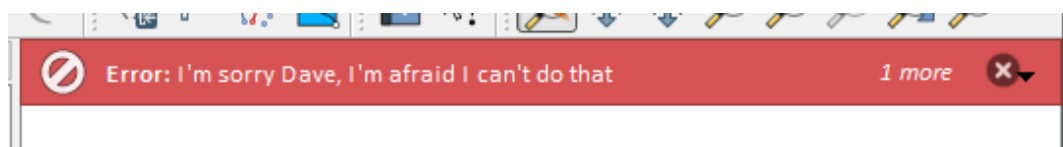


Figure 12.1: Barre de message de QGIS

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

```
iface.messageBar().pushMessage("Oops", "The plugin is not working as it should",
    level=QgsMessageBar.Critical, duration=3)
```

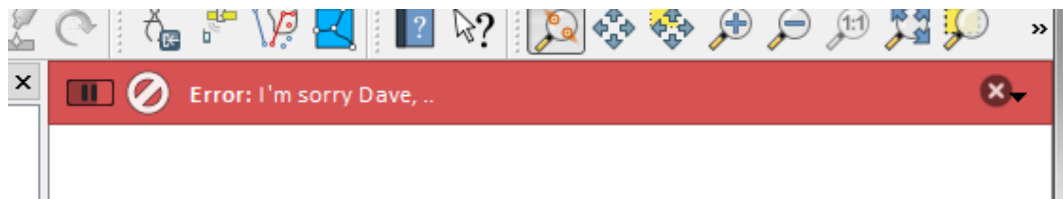


Figure 12.2: Barre de message de Qgis avec décompte

The examples above show an error bar, but the `level` parameter can be used to creating warning messages or info messages, using the `Qgis.MessageLevel` enumeration. You can use up to 4 different levels:

0. Info
1. Warning
2. Critical
3. Success

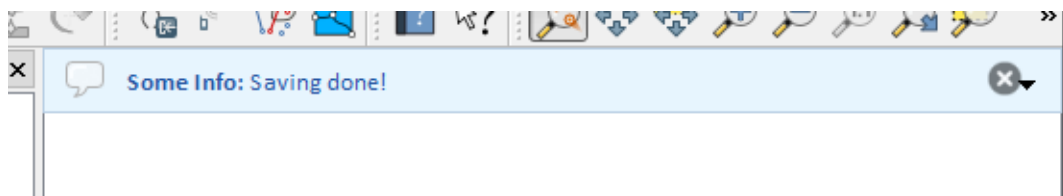


Figure 12.3: Barre de message QGis (info)

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, Qgis.Warning)
```

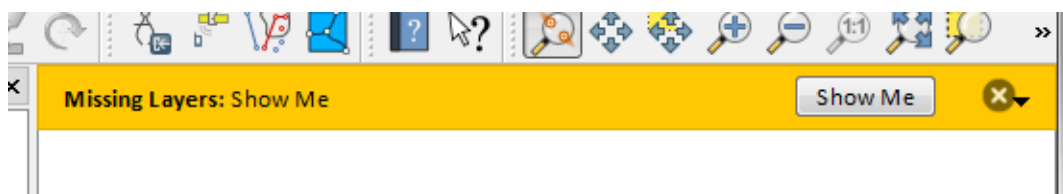


Figure 12.4: Barre de message QGis avec un bouton

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

```
class MyDialog(QDialog):
    def __init__(self):
        QDialog.__init__(self)
        self.bar = QgsMessageBar()
        self.bar.setSizePolicy(QSizePolicy.Minimum, QSizePolicy.Fixed)
        self.setLayout(QGridLayout())
        self.layout().setContentsMargins(0, 0, 0, 0)
```



```

self.buttonbox = QDialogButtonBox(QDialogButtonBox.Ok)
self.buttonbox.accepted.connect(self.run)
self.layout().addWidget(self.buttonbox, 0, 0, 2, 1)
self.layout().addWidget(self.bar, 0, 0, 1, 1)
def run(self):
    self.bar.pushMessage("Hello", "World", level=Qgis.Info)

myDlg = MyDialog()
myDlg.show()

```

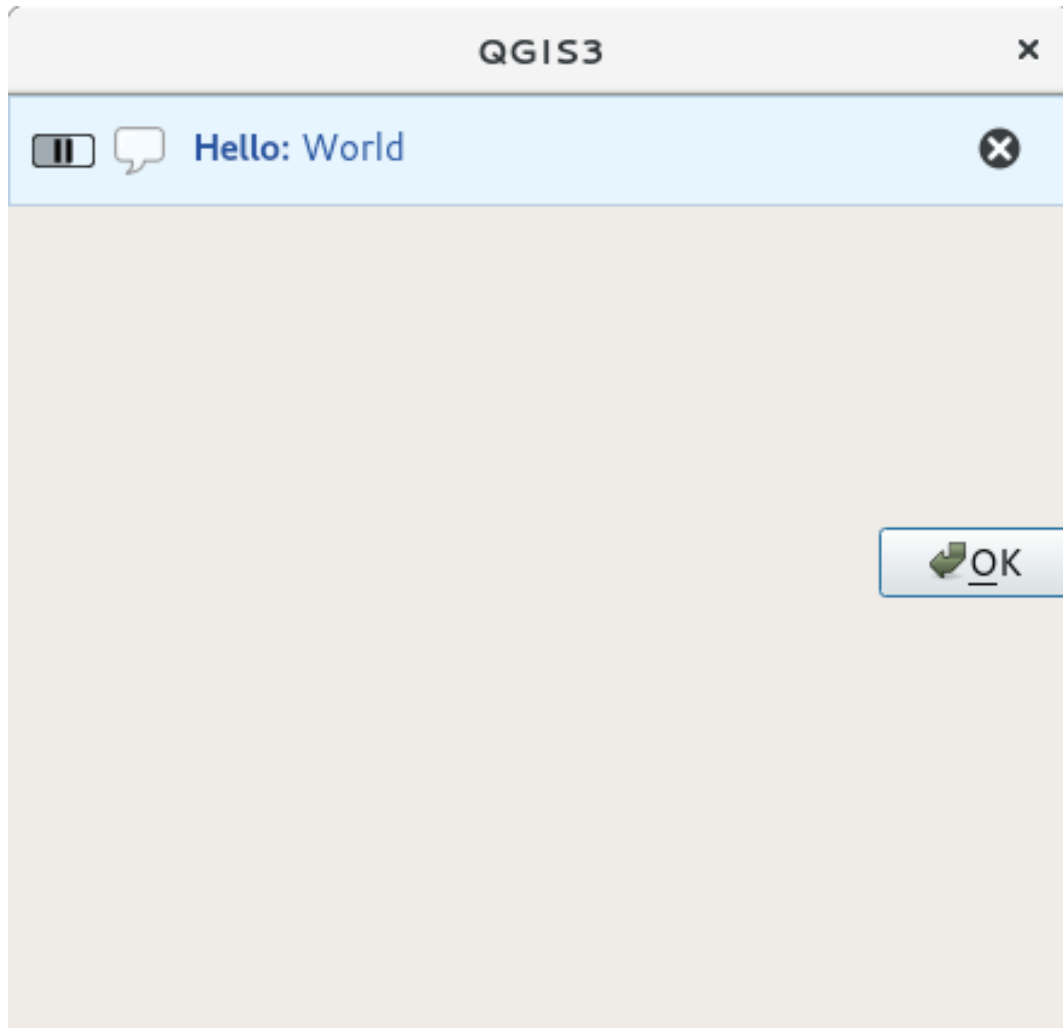


Figure 12.5: Barre de message QGIS avec une boîte de dialogue personnalisée

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 qgis.PyQt.QtWidgets import QProgressBar
from qgis.PyQt.QtCore import *
progressMessageBar = iface.messageBar().createMessage("Doing something boring...")
progress = QProgressBar()
progress.setMaximum(10)

```

```

progress.setAlignment(Qt.AlignLeft|Qt.AlignVCenter)
progressMessageBar.layout().addWidget(progress)
iface.messageBar().pushWidget(progressMessageBar, Qgis.Info)

for i in range(10):
    time.sleep(1)
    progress.setValue(i + 1)

iface.messageBar().clearWidgets()

```

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

```

vlayer = QgsProject.instance().mapLayersByName("countries")[0]

count = vlayer.featureCount()
features = vlayer.getFeatures()

for i, feature in enumerate(features):
    # do something time-consuming here
    print('') # printing should give enough time to present the progress

    percent = i / float(count) * 100
    # iface.mainWindow().statusBar().showMessage("Processed {} %".
↳format(int(percent)))
    iface.statusBarIface().showMessage("Processed {} %".format(int(percent)))

iface.statusBarIface().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
↳', level=Qgis.Info)
QgsMessageLog.logMessage("Your plugin code might have some problems", level=Qgis.
↳Warning)
QgsMessageLog.logMessage("Your plugin code has crashed!", level=Qgis.Critical)

```

Avertissement: Use of the Python `print` statement is unsafe to do in any code which may be multithreaded. This includes **expression functions**, **renderers**, **symbol layers** and **Processing algorithms** (amongst others). In these cases you should always use thread safe classes (`QgsLogger` or `QgsMessageLog`) instead.

Note: You can see the output of the `QgsMessageLog` in the `log_message_panel`

Note:

- `QgsLogger` is for messages for debugging / developers (i.e. you suspect they are triggered by some broken code)
 - `QgsMessageLog` is for messages to investigate issues by sysadmins (e.g. to help a sysadmin to fix configurations)
-

Infrastructure d'authentification

- *Introduction*
- *Glossaire*
- *QgsAuthManager le point d'entrée*
 - *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*
 - *Retirer l'entrée de authdb*
 - *Leave authcfg expansion to QgsAuthManager*
 - * *Exemples PKI avec d'autres fournisseurs de données*
- *Adapt plugins to use Authentication infrastructure*
- *Authentication GUIs*
 - *Fenêtre de sélection des identifiants*
 - *Authentication Editor GUI*
 - *Authorities Editor GUI*

Avertissement: *Despite our constant efforts, information beyond this line may not be updated for QGIS 3. Refer to <https://qgis.org/pyqgis/master> for the python API documentation or; give a hand to update the chapters you know about. Thanks.*

13.1 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.

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](#).

13.2 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 `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*

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

Configuration de l'authentification *Configuration de l'authentification*

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.

13.3 QgsAuthManager le point d'entrée

The `QgsAuthManager` singleton is the entry point to use the credentials stored in the QGIS encrypted *Authentication DB*, i.e. the `qgis-auth.db` file under the active user profile folder.

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

13.3.1 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" )

```

13.3.2 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("https://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()

```

Avertissement: 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.

13.3.3 Retirer l'entrée de 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" )

```

13.3.4 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 the `certIdentity()` method supports the following list of providers:

```

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

```

For example, to access a WMS service using stored credentials identified with `authcfg = 'fmls770'`, we just have to use the `authcfg` in the data source URL like in the following snippet:

```

authCfg = 'fmls770'
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.

Avertissement: The 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, built using the `QgsDataSourceURI` class, is used to set a 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.

Exemples PKI avec d'autres fournisseurs de données

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

13.4 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

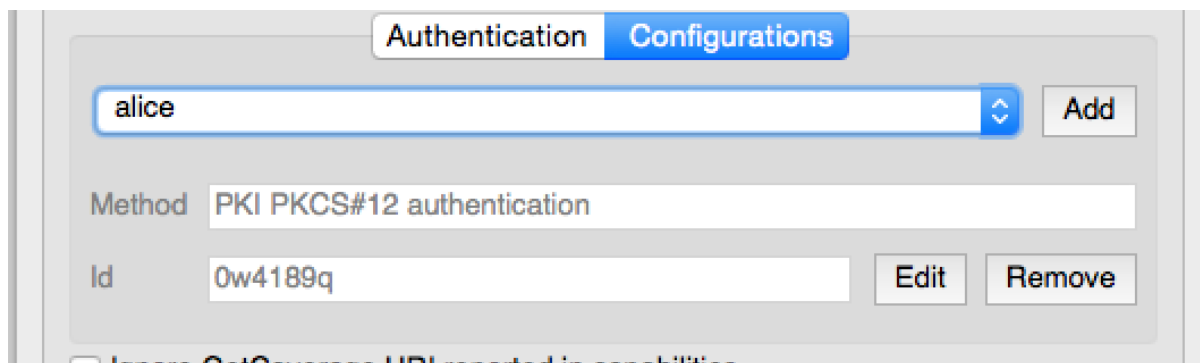
```

13.5 Authentication GUIs

Ce paragraphe liste les fenêtres utiles à l'intégration d'une infrastructure d'authentification dans des interfaces personnalisées.

13.5.1 Fenêtre de sélection des identifiants

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* <*qgis.gui.QgsAuthConfigSelect*>.



and can be used as in the following snippet:

```
# create the instance of the QgsAuthConfigSelect GUI hierarchically linked to
# the widget referred with `parent`
gui = QgsAuthConfigSelect( parent, "postgres" )
# add the above created gui in a new tab of the interface where the
# GUI has to be integrated
tabGui.insertTab( 1, gui, "Configurations" )
```

The above example is taken from the QGIS source [code](#). The second parameter of the GUI constructor refers to data provider type. The parameter is used to restrict the compatible *Authentication Methods* with the specified provider.

13.5.2 Authentication Editor GUI

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

and can be used as in the following snippet:

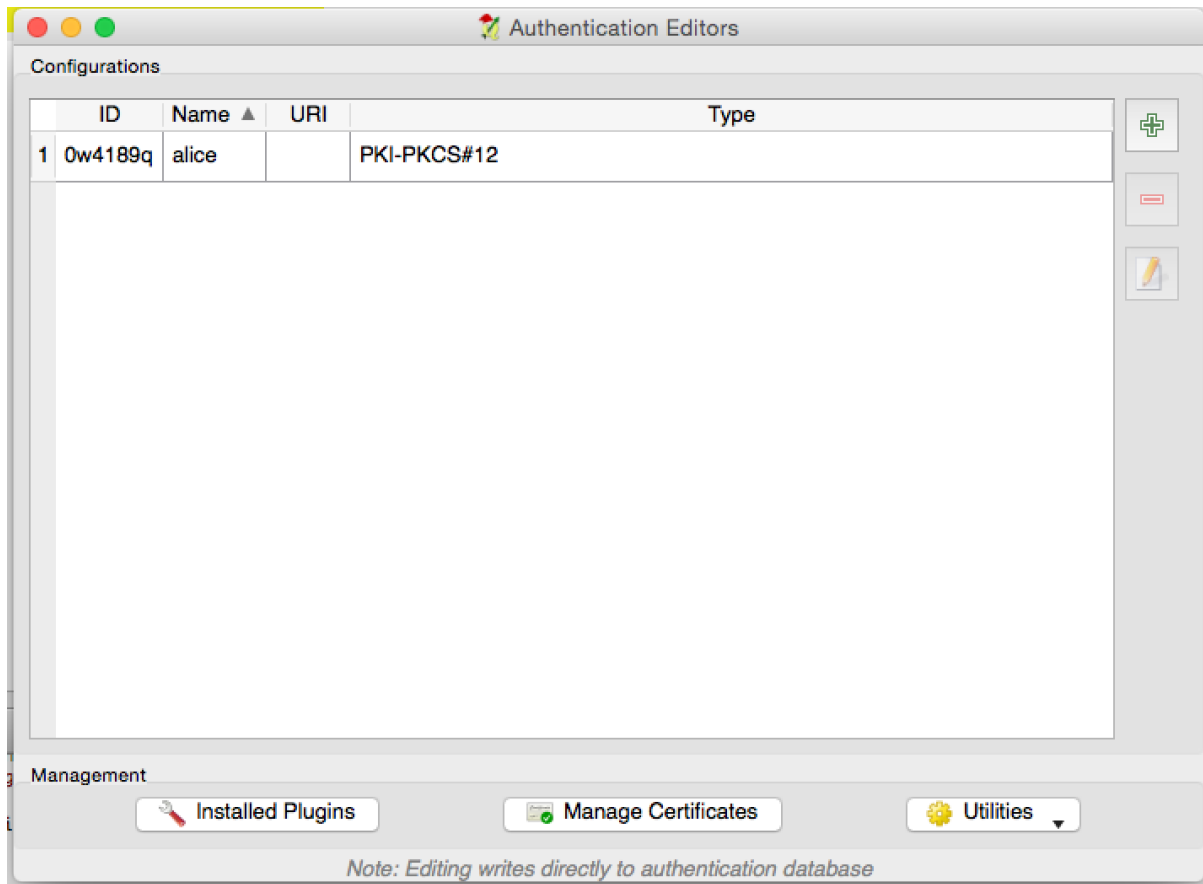
```
# create the instance of the QgsAuthEditorWidgets GUI hierarchically linked to
# the widget referred with `parent`
gui = QgsAuthEditorWidgets( parent )
gui.show()
```

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

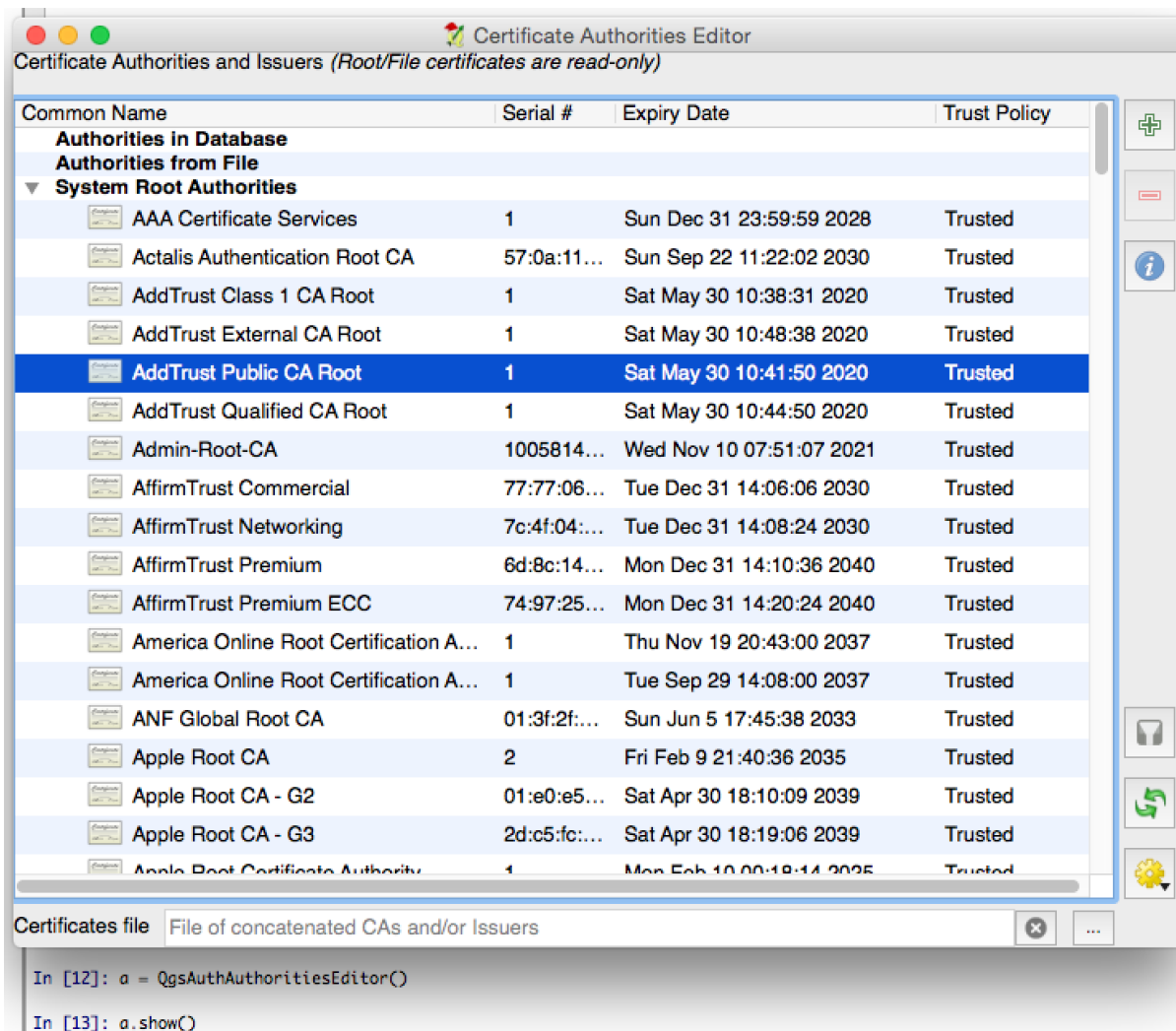
13.5.3 Authorities Editor GUI

A GUI used to manage only authorities is managed by the *QgsAuthAuthoritiesEditor* <*qgis.gui.QgsAuthAuthoritiesEditor*> class.

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()
```



Tasks - doing heavy work in the background

14.1 Introduction

Background processing using threads is a way to maintain a responsive user interface when heavy processing is going on. Tasks can be used to achieve threading in QGIS.

A task (`QgsTask`) is a container for the code to be performed in the background, and the task manager (`QgsTaskManager`) is used to control the running of the tasks. These classes simplify background processing in QGIS by providing mechanisms for signaling, progress reporting and access to the status for background processes. Tasks can be grouped using subtasks.

The global task manager (found with `QgsApplication.taskManager()`) is normally used. This means that your tasks may not be the only tasks that are controlled by the task manager.

There are several ways to create a QGIS task:

- Create your own task by extending `QgsTask`

```
class SpecialisedTask(QgsTask):
```

- Create a task from a function

```
QgsTask.fromFunction('heavy function', heavyFunction,  
                    onfinished=workdone)
```

- Create a task from a processing algorithm

```
QgsProcessingAlgRunnerTask('native:buffer', params, context,  
                            feedback)
```

Avertissement: Any background task (regardless of how it is created) must NEVER perform any GUI based operations, such as creating new widgets or interacting with existing widgets. Qt widgets must only be accessed or modified from the main thread. Attempting to use them from background threads will result in crashes.

Dependencies between tasks can be described using the `addSubTask` function of `QgsTask`. When a dependency is stated, the task manager will automatically determine how these dependencies will be executed. Whenever possible dependencies will be executed in parallel in order to satisfy them as quickly as possible. If a task

on which another task depends is canceled, the dependent task will also be canceled. Circular dependencies can make deadlocks possible, so be careful.

If a task depends on a layer being available, this can be stated using the `setDependentLayers` function of `QgsTask`. If a layer on which a task depends is not available, the task will be canceled.

Once the task has been created it can be scheduled for running using the `addTask` function of the task manager. Adding a task to the manager automatically transfers ownership of that task to the manager, and the manager will cleanup and delete tasks after they have executed. The scheduling of the tasks is influenced by the task priority, which is set in `addTask`.

The status of tasks can be monitored using `QgsTask` and `QgsTaskManager` signals and functions.

14.2 Exemples

14.2.1 Extending QgsTask

In this example `RandomIntegerSumTask` extends `QgsTask` and will generate 100 random integers between 0 and 500 during a specified period of time. If the random number is 42, the task is aborted and an exception is raised. Several instances of `RandomIntegerSumTask` (with subtasks) are generated and added to the task manager, demonstrating two types of dependencies.

```
import random
from time import sleep

from qgis.core import (
    QgsApplication, QgsTask, QgsMessageLog,
)

MESSAGE_CATEGORY = 'RandomIntegerSumTask'

class RandomIntegerSumTask(QgsTask):
    """This shows how to subclass QgsTask"""
    def __init__(self, description, duration):
        super().__init__(description, QgsTask.CanCancel)
        self.duration = duration
        self.total = 0
        self.iterations = 0
        self.exception = None
    def run(self):
        """Here you implement your heavy lifting.
        Should periodically test for isCanceled() to gracefully
        abort.
        This method MUST return True or False.
        Raising exceptions will crash QGIS, so we handle them
        internally and raise them in self.finished
        """
        QgsMessageLog.logMessage('Started task "{}".format(
            self.description()),
            MESSAGE_CATEGORY, QgsInfo)
        wait_time = self.duration / 100
        for i in range(100):
            sleep(wait_time)
            # use setProgress to report progress
            self.setProgress(i)
            arandominteger = random.randint(0, 500)
            self.total += arandominteger
            self.iterations += 1
            # check isCanceled() to handle cancellation
            if self.isCanceled():
                return False
```

```

    # simulate exceptions to show how to abort task
    if arandominteger == 42:
        # DO NOT raise Exception('bad value!')
        # this would crash QGIS
        self.exception = Exception('bad value!')
        return False
    return True
def finished(self, result):
    """
    This function is automatically called when the task has
    completed (successfully or not).
    You implement finished() to do whatever follow-up stuff
    should happen after the task is complete.
    finished is always called from the main thread, so it's safe
    to do GUI operations and raise Python exceptions here.
    result is the return value from self.run.
    """
    if result:
        QgsMessageLog.logMessage(
            'Task "{name}" completed\n' \
            'Total: {total} (with {iterations} \
            'iterations)'.format(
                name=self.description(),
                total=self.total,
                iterations=self.iterations),
            MESSAGE_CATEGORY, Qgis.Success)
    else:
        if self.exception is None:
            QgsMessageLog.logMessage(
                'Task "{name}" not successful but without \
                'exception (probably the task was manually \
                'canceled by the user)'.format(
                    name=self.description()),
                MESSAGE_CATEGORY, Qgis.Warning)
        else:
            QgsMessageLog.logMessage(
                'Task "{name}" Exception: {exception}'.format(
                    name=self.description(),
                    exception=self.exception),
                MESSAGE_CATEGORY, Qgis.Critical)
            raise self.exception
def cancel(self):
    QgsMessageLog.logMessage(
        'Task "{name}" was canceled'.format(
            name=self.description()),
        MESSAGE_CATEGORY, Qgis.Info)
    super().cancel()

longtask = RandomIntegerSumTask('waste cpu long', 20)
shorttask = RandomIntegerSumTask('waste cpu short', 10)
minitask = RandomIntegerSumTask('waste cpu mini', 5)
shortsubtask = RandomIntegerSumTask('waste cpu subtask short', 5)
longsubtask = RandomIntegerSumTask('waste cpu subtask long', 10)
shortestsubtask = RandomIntegerSumTask('waste cpu subtask shortest', 4)

# Add a subtask (shortsubtask) to shorttask that must run after
# minitask and longtask has finished
shorttask.addSubTask(shortsubtask, [minitask, longtask])
# Add a subtask (longsubtask) to longtask that must be run
# before the parent task
longtask.addSubTask(longsubtask, [], QgsTask.ParentDependsOnSubTask)
# Add a subtask (shortestsubtask) to longtask

```

```

longtask.addSubTask(shortestsubtask)

QgsApplication.taskManager().addTask(longtask)
QgsApplication.taskManager().addTask(shorttask)
QgsApplication.taskManager().addTask(minitask)

```

14.2.2 Task from function

Create a task from a function (`doSomething` in this example). The first parameter of the function will hold the `QgsTask` for the function. An important (named) parameter is `on_finished`, that specifies a function that will be called when the task has completed. The `doSomething` function in this example has an additional named parameter `wait_time`.

```

import random
from time import sleep

MESSAGE_CATEGORY = 'TaskFromFunction'

def doSomething(task, wait_time):
    """
    Raises an exception to abort the task.
    Returns a result if success.
    The result will be passed, together with the exception (None in
    the case of success), to the on_finished method.
    If there is an exception, there will be no result.
    """
    QgsMessageLog.logMessage('Started task {}'.format(task.description()),
                             MESSAGE_CATEGORY, QgsInfo)

    wait_time = wait_time / 100
    total = 0
    iterations = 0
    for i in range(100):
        sleep(wait_time)
        # use task.setProgress to report progress
        task.setProgress(i)
        arandominteger = random.randint(0, 500)
        total += arandominteger
        iterations += 1
        # check task.isCanceled() to handle cancellation
        if task.isCanceled():
            stopped(task)
            return None
        # raise an exception to abort the task
        if arandominteger == 42:
            raise Exception('bad value!')
    return {'total': total, 'iterations': iterations,
            'task': task.description()}

def stopped(task):
    QgsMessageLog.logMessage(
        'Task "{name}" was canceled'.format(
            name=task.description()),
        MESSAGE_CATEGORY, QgsInfo)

def completed(exception, result=None):
    """This is called when doSomething is finished.
    Exception is not None if doSomething raises an exception.
    result is the return value of doSomething."""
    if exception is None:
        if result is None:
            QgsMessageLog.logMessage(

```

```

        'Completed with no exception and no result '\
        '(probably manually canceled by the user)',
        MESSAGE_CATEGORY, Qgis.Warning)
    else:
        QgsMessageLog.logMessage(
            'Task {name} completed\n'
            'Total: {total} ( with {iterations} '
            'iterations)'.format(
                name=result['task'],
                total=result['total'],
                iterations=result['iterations']),
            MESSAGE_CATEGORY, Qgis.Info)
    else:
        QgsMessageLog.logMessage("Exception: {}".format(exception),
            MESSAGE_CATEGORY, Qgis.Critical)
    raise exception

# Create a few tasks
task1 = QgsTask.fromFunction('Waste cpu 1', doSomething,
                             on_finished=completed, wait_time=4)
task2 = QgsTask.fromFunction('Waste cpu 2', doSomething,
                             on_finished=completed, wait_time=3)
QgsApplication.taskManager().addTask(task1)
QgsApplication.taskManager().addTask(task2)

```

14.2.3 Task from a processing algorithm

Create a task that uses the algorithm `qgis:randompointsinextent` to generate 50000 random points inside a specified extent. The result is added to the project in a safe way.

```

from functools import partial
from qgis.core import (QgsTaskManager, QgsMessageLog,
                      QgsProcessingAlgRunnerTask, QgsApplication,
                      QgsProcessingContext, QgsProcessingFeedback,
                      QgsProject)

MESSAGE_CATEGORY = 'AlgRunnerTask'

def task_finished(context, successful, results):
    if not successful:
        QgsMessageLog.logMessage('Task finished unsuccessfully',
            MESSAGE_CATEGORY, Qgis.Warning)
    output_layer = context.getMapLayer(results['OUTPUT'])
    # because getMapLayer doesn't transfer ownership, the layer will
    # be deleted when context goes out of scope and you'll get a
    # crash.
    # takeMapLayer transfers ownership so it's then safe to add it
    # to the project and give the project ownership.
    if output_layer and output_layer.isValid():
        QgsProject.instance().addMapLayer(
            context.takeResultLayer(output_layer.id()))

alg = QgsApplication.processingRegistry().algorithmById(
    'qgis:randompointsinextent')

context = QgsProcessingContext()
feedback = QgsProcessingFeedback()
params = {
    'EXTENT': '0.0,10.0,40,50 [EPSG:4326]',
    'MIN_DISTANCE': 0.0,
    'POINTS_NUMBER': 50000,
    'TARGET_CRS': 'EPSG:4326',
}

```

```
'OUTPUT': 'memory:My random points'
}
task = QgsProcessingAlgRunnerTask(alg, params, context, feedback)
task.executed.connect(partial(task_finished, context))
QgsApplication.taskManager().addTask(task)
```

Lire également : <https://www.opengis.ch/2018/06/22/threads-in-pyqgis3/>.

15.1 Structuring Python Plugins

- *Writing a plugin*
 - *Plugin files*
- *Plugin content*
 - *Plugin metadata*
 - *__init__.py*
 - *mainPlugin.py*
 - *Resource File*
- *Documentation*
- *Translation*
 - *Software requirements*
 - *Files and directory*
 - * *.pro file*
 - * *.ts file*
 - * *.qm file*
 - *Translate using Makefile*
 - *Load the plugin*
- *Tips and Tricks*
 - *Plugin Reloader*
 - *Accessing Plugins*
 - *Log Messages*
 - *Share your plugin*

In order to create a plugin, here are some steps to follow:

1. *Idea*: Have an idea about what you want to do with your new QGIS plugin. Why do you do it? What problem do you want to solve? Is there already another plugin for that problem?
2. *Create files*: The essentials: a starting point `__init__.py`; fill in the *Plugin metadata* `metadata.txt`. Then implement your own design. A main Python plugin body e.g. `mainplugin.py`. Probably a form in Qt Designer `form.ui`, with its `resources.qrc`.
3. *Write code*: Write the code inside the `mainplugin.py`
4. *Test*: Close and re-open QGIS and import your plugin again. Check if everything is OK.
5. *Publish*: Publish your plugin in QGIS repository or make your own repository as an « arsenal » of personal « GIS weapons ».

15.1.1 Writing a plugin

Since the introduction of Python plugins in QGIS, a number of plugins have appeared. The QGIS team maintains an *Official Python plugin repository*. You can use their source to learn more about programming with PyQGIS or find out whether you are duplicating development effort.

Plugin files

Here's the directory structure of our example plugin

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

What is the meaning of the files:

- `__init__.py` = The starting point of the plugin. It has to have the `classFactory()` method and may have any other initialisation code.
- `mainPlugin.py` = The main working code of the plugin. Contains all the information about the actions of the plugin and the main code.
- `resources.qrc` = The .xml document created by Qt Designer. Contains relative paths to resources of the forms.
- `resources.py` = The translation of the .qrc file described above to Python.
- `form.ui` = The GUI created by Qt Designer.
- `form.py` = The translation of the form.ui described above to Python.
- `metadata.txt` = Contains general info, version, name and some other metadata used by plugins website and plugin infrastructure.

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

There is a QGIS plugin called [Plugin Builder 3](#) that creates a plugin template for QGIS and doesn't require an internet connection. This is the recommended option, as it produces 3.x compatible sources.

Avertissement: If you plan to upload the plugin to the *Official Python plugin repository* you must check that your plugin follows some additional rules, required for plugin *Validation*

15.1.2 Plugin content

Here you can find information and examples about what to add in each of the files in the file structure described above.

Plugin metadata

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

Note: All metadata must be in UTF-8 encoding.

Metadata name	Re-quired	Notes
name	True	a short string containing the name of the plugin
qgisMinimumVersion	True	dotted notation of minimum QGIS version
qgisMaximumVersion	False	dotted notation of maximum QGIS version
description	True	short text which describes the plugin, no HTML allowed
about	True	longer text which describes the plugin in details, no HTML allowed
version	True	short string with the version dotted notation
author	True	author name
email	True	email of the author, only shown on the website to logged in users, but visible in the Plugin Manager after the plugin is installed
changelog	False	string, can be multiline, no HTML allowed
experimental	False	boolean flag, <i>True</i> or <i>False</i>
deprecated	False	boolean flag, <i>True</i> or <i>False</i> , applies to the whole plugin and not just to the uploaded version
tags	False	comma separated list, spaces are allowed inside individual tags
homepage	False	a valid URL pointing to the homepage of your plugin
repository	True	a valid URL for the source code repository
tracker	False	a valid URL for tickets and bug reports
icon	False	a file name or a relative path (relative to the base folder of the plugin's compressed package) of a web friendly image (PNG, JPEG)
category	False	one of <i>Raster</i> , <i>Vector</i> , <i>Database</i> and <i>Web</i>

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

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

```
category=Raster
```

Note: If `qgisMaximumVersion` is empty, it will be automatically set to the major version plus `.99` when uploaded to the *Official Python plugin repository*.

An example for this metadata.txt

```
; the next section is mandatory

[general]
name=HelloWorld
email=me@example.com
author=Just Me
qgisMinimumVersion=3.0
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=https://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=3.99
```

`__init__.py`

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

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

## any other initialisation needed
```

mainPlugin.py

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

```

from qgis.PyQt.QtGui import *
from qgis.PyQt.QtWidgets import *

# initialize Qt resources from file resources.py
from . 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")
        self.action.triggered.connect(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
        self.iface.mapCanvas().renderComplete.connect(self.renderTest)

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

        # disconnect form signal of the canvas
        self.iface.mapCanvas().renderComplete.disconnect(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!")

```

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

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

In the above example, `addPluginToMenu` is used. This will add the corresponding menu action to the *Plugins* menu. Alternative methods exist to add the action to a different menu. Here is a list of those methods:

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

- `addPluginToWebMenu()`

All of them have the same syntax as the `addPluginToMenu` method.

Adding your plugin menu to one of those predefined method is recommended to keep consistency in how plugin entries are organized. However, you can add your custom menu group directly to the menu bar, as the next example demonstrates:

```
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")
    self.action.triggered.connect(self.run)
    self.menu.addAction(self.action)

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

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

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

Resource File

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

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

It is good to use a prefix that will not collide with other plugins or any parts of QGIS, otherwise you might get resources you did not want. Now you just need to generate a Python file that will contain the resources. It's done with `pyrcc5` command:

```
pyrcc5 -o resources.py resources.qrc
```

Note: In Windows environments, attempting to run the `pyrcc5` from Command Prompt or Powershell will probably result in the error « Windows cannot access the specified device, path, or file [...] ». The easiest solution is probably to use the OSGeo4W Shell but if you are comfortable modifying the `PATH` environment variable or specifying the path to the executable explicitly you should be able to find it at `<Your QGIS Install Directory>\bin\pyrcc5.exe`.

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

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

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

15.1.3 Documentation

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

The `showPluginHelp()` function looks for help files in the same directory as the calling module. It will look for, in turn, `index-ll_cc.html`, `index-ll.html`, `index-en.html`, `index-en_us.html` and `index.html`, displaying whichever it finds first. Here `ll_cc` is the QGIS locale. This allows multiple translations of the documentation to be included with the plugin.

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

15.1.4 Translation

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

Software requirements

The easiest way to create and manage all the translation files is to install [Qt Linguist](#). In a Debian-based GNU/Linux environment you can install it typing:

```
sudo apt install qttools5-dev-tools
```

Files and directory

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

All the translation files have to be within this directory.

.pro file

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

In this `.pro` file you have to specify all the files and forms you want to translate. This file is used to set up the localization files and variables. A possible project file, matching the structure of our *example plugin*:

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

Your plugin might follow a more complex structure, and it might be distributed across several files. If this is the case, keep in mind that `pylupdate5`, the program we use to read the `.pro` file and update the translatable string, does not expand wild card characters, so you need to place every file explicitly in the `.pro` file. Your project file might then look like something like this:

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

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

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

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

.ts file

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

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

```
pylupdate5 your_plugin.pro
```

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

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

.qm file

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

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

```
lrelease your_plugin.ts
```

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

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 **Qt 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.

Load the plugin

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

You should see your plugin in the correct language.

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

15.1.5 Tips and Tricks

Plugin Reloader

During development of your plugin you will frequently need to reload it in QGIS for testing. This is very easy using the Plugin Reloader plugin. You can find it as an experimental plugin with the Plugin Manager.

Accessing Plugins

You can access all the classes of installed plugins from within QGIS using python, which can be handy for debugging purposes.:

```
my_plugin = qgis.utils.plugins['My Plugin']
```

Log Messages

Plugins have their own tab within the `log_message_panel`.

Share your plugin

QGIS is hosting hundreds of plugins in the plugin repository. Consider sharing yours! It will extend the possibilities of QGIS and people will be able to learn from your code. All hosted plugins can be found and installed from within QGIS with the Plugin Manager.

Information and requirements are here: plugins.qgis.org.

15.2 Code Snippets

- *How to call a method by a key shortcut*
- *How to toggle Layers*
- *How to access attribute table of selected features*

This section features code snippets to facilitate plugin development.

15.2.1 How to call a method by a key shortcut

In the plug-in add to the `initGui()`

```
self.key_action = QAction("Test Plugin", self.iface.mainWindow())
self.iface.registerMainWindowAction(self.key_action, "Ctrl+I") # action triggered_
↳by Ctrl+I
self.iface.addPluginToMenu("&Test plugins", self.key_action)
self.key_action.triggered.connect(self.key_action_triggered)
```

To `unload()` add

```
self.iface.unregisterMainWindowAction(self.key_action)
```

The method that is called when CTRL+I is pressed

```
def key_action_triggered(self):
    QMessageBox.information(self.iface.mainWindow(), "Ok", "You pressed Ctrl+I")
```

15.2.2 How to toggle Layers

There is an API to access layers in the legend. Here is an example that toggles the visibility of the active layer

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

15.2.3 How to access attribute table of selected features

```
def change_value(value):
    """Change the value in the second column for all selected features.

    :param value: The new value.
    """
    layer = iface.activeLayer()
    if layer:
        count_selected = layer.selectedFeatureCount()
        if count_selected > 0:
            layer.startEditing()
            id_features = layer.selectedFeatureIds()
            for i in id_features:
                layer.changeAttributeValue(i, 1, value) # 1 being the second column
            layer.commitChanges()
        else:
            iface.messageBar().pushCritical("Error",
                "Please select at least one feature from current layer")
    else:
        iface.messageBar().pushCritical("Error", "Please select a layer")
```

The method requires one parameter (the new value for the second field of the selected feature(s)) and can be called by

```
changeValue(50)
```

15.3 Using Plugin Layers

Avertissement: *Despite our constant efforts, information beyond this line may not be updated for QGIS 3. Refer to <https://qgis.org/pyqgis/master> for the python API documentation or, give a hand to update the chapters you know about. Thanks.*

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.

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

15.3.1 Subclassing QgsPluginLayer

Below is an example of a minimal QgsPluginLayer implementation. It is an excerpt of the [Watermark example plugin](#)

```
class WatermarkPluginLayer(QgsPluginLayer):
    LAYER_TYPE="watermark"

    def __init__(self):
        QgsPluginLayer.__init__(self, WatermarkPluginLayer.LAYER_TYPE, "Watermark_
↪plugin_layer")
        self.setValid(True)

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

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

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

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

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

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

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

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

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

15.4 IDE settings for writing and debugging plugins

Avertissement: *Despite our constant efforts, information beyond this line may not be updated for QGIS 3. Refer to <https://qgis.org/pyqgis/master> for the python API documentation or, give a hand to update the chapters you know about. Thanks.*

- *A note on configuring your IDE on Windows*
- *Debugging using Eclipse and PyDev*
 - *Installation*

- *Preparing QGIS*
- *Setting up Eclipse*
- *Configuring the debugger*
- *Making eclipse understand the API*
- *Debugging using PDB*

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

15.4.1 A note on configuring your IDE on Windows

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

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

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

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

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

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

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

To create that batch file, follow these steps:

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

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

15.4.2 Debugging using Eclipse and PyDev

Installation

To use Eclipse, make sure you have installed the following

- Eclipse
- Aptana Studio 3 Plugin or PyDev
- QGIS 2.x

Preparing QGIS

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

- Go to *Plugins* → *Manage and Install plugins...*
- Search for *Remote Debug* (at the moment it's still experimental, so enable experimental plugins under the *Options* tab in case it does not show up). Install it.
- Search for *Plugin reloader* and install it as well. This will let you reload a plugin instead of having to close and restart QGIS to have the plugin reloaded.

Setting up Eclipse

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

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

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

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

Configuring the debugger

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

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

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

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

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

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

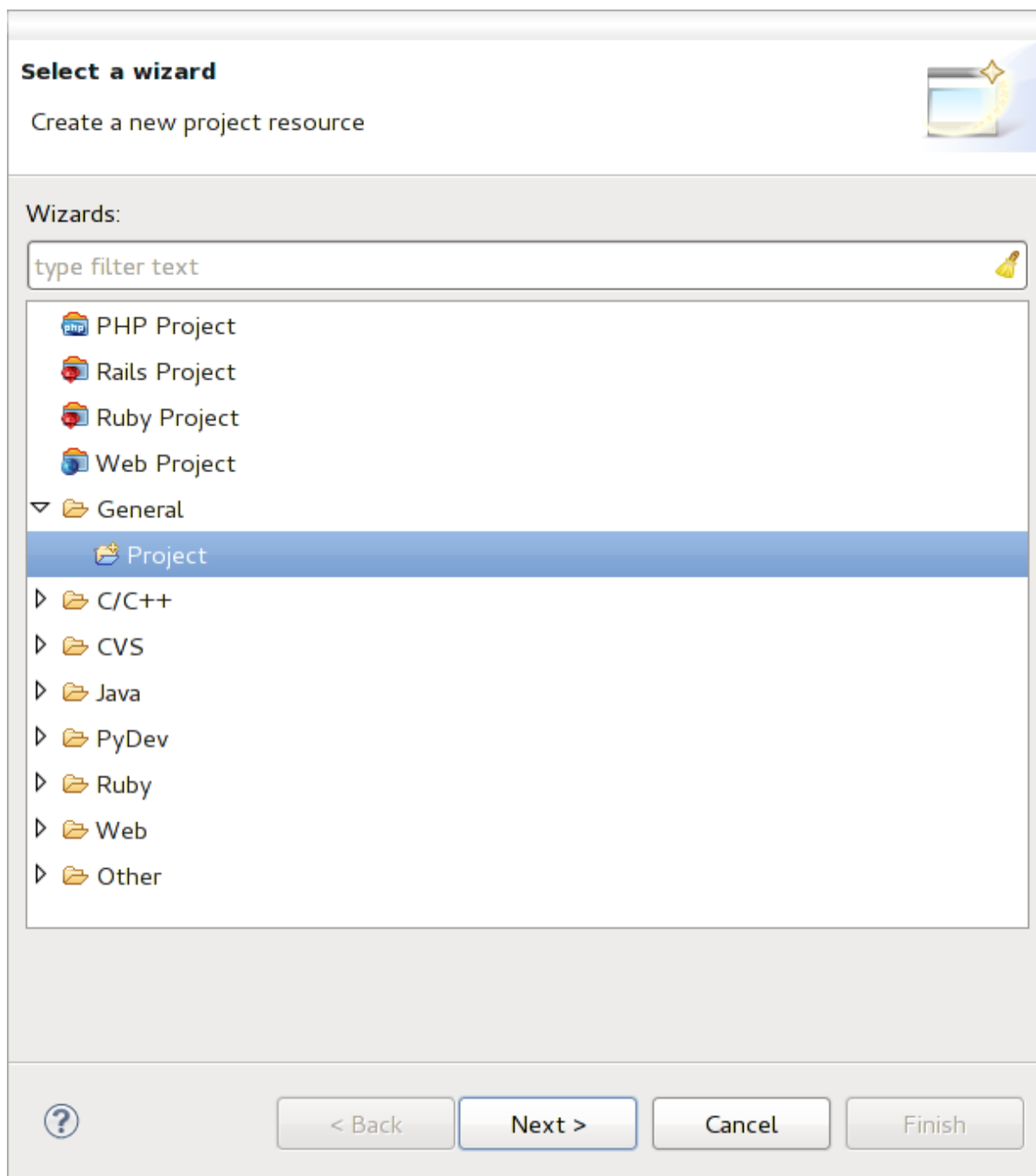


Figure 15.1: Eclipse project

```

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: Breakpoint

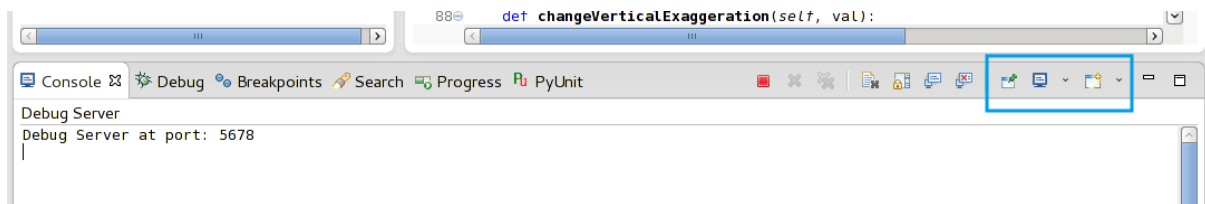


Figure 15.3: PyDev Debug Console

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

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

Making eclipse understand the API

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

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

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

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

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

You should also add your plugins folder here (it is in `python/plugins` under the user profile folder).

Next jump to the *Forced Builtins* tab, click on *New...* and enter `qgis`. This will make Eclipse parse the QGIS API. You probably also want Eclipse to know about the PyQt4 API. Therefore also add PyQt4 as forced builtin. That should probably already be present in your libraries tab.

Click *OK* and you're done.

Note: Every time the QGIS API changes (e.g. if you're compiling QGIS master and the SIP file changed), you should go back to this page and simply click *Apply*. This will let Eclipse parse all the libraries again.

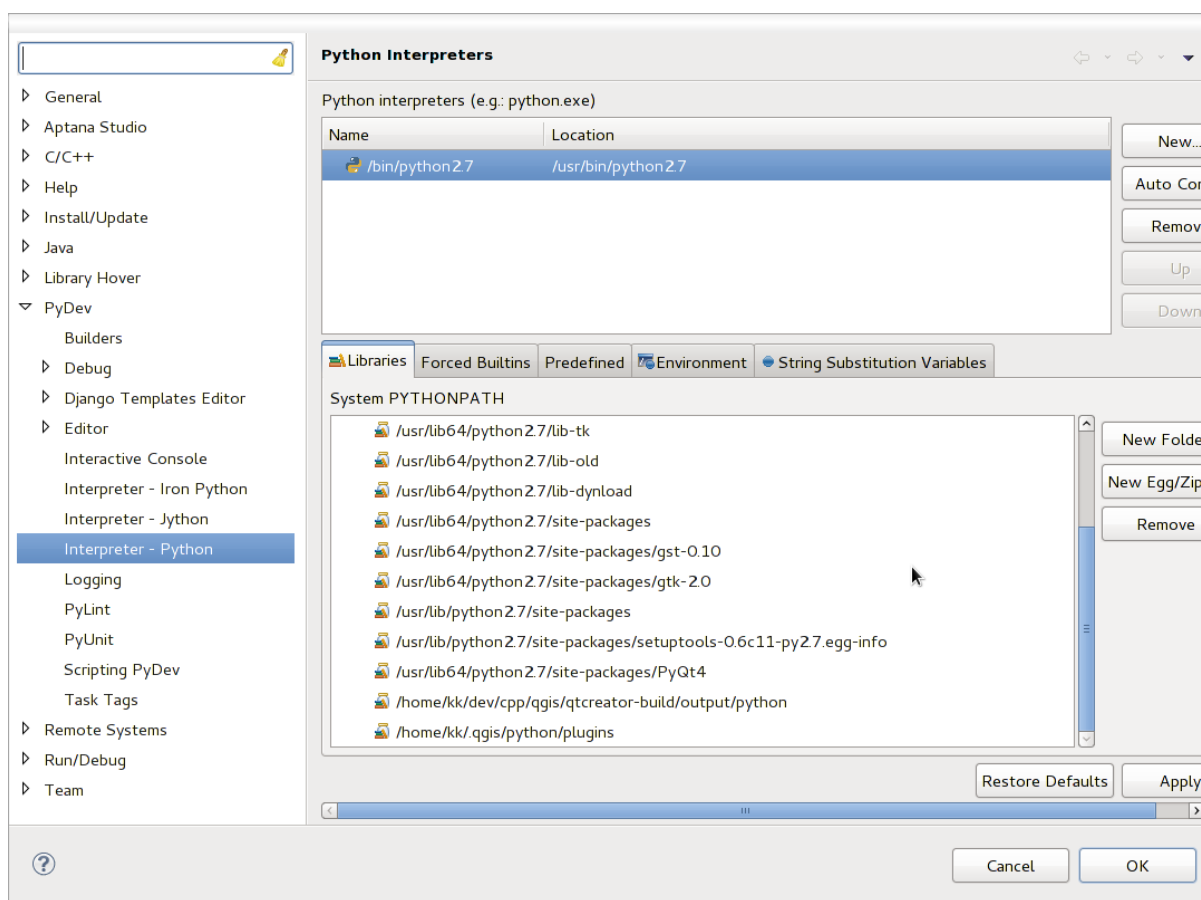


Figure 15.4: PyDev Debug Console

15.4.3 Debugging using PDB

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

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

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

Then run QGIS from the command line.

On Linux do:

```
$ ./Qgis
```

On macOS do:

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

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

TODO: Add testing information

15.5 Releasing your plugin

Avertissement: *Despite our constant efforts, information beyond this line may not be updated for QGIS 3. Refer to <https://qgis.org/pyqgis/master> for the python API documentation or, give a hand to update the chapters you know about. Thanks.*

- *Metadata and names*
- *Code and help*
- *Official Python plugin repository*
 - *Permissions*
 - *Trust management*
 - *Validation*
 - *Plugin structure*

Once your plugin is ready and you think the plugin could be helpful for some people, do not hesitate to upload it to *Official Python plugin repository*. On that page you can also find packaging guidelines about how to prepare the plugin to work well with the plugin installer. Or in case you would like to set up your own plugin repository, create a simple XML file that will list the plugins and their metadata.

Please take special care to the following suggestions:

15.5.1 Metadata and names

- avoid using a name too similar to existing plugins
- if your plugin has a similar functionality to an existing plugin, please explain the differences in the About field, so the user will know which one to use without the need to install and test it

- avoid repeating « plugin » in the name of the plugin itself
- use the description field in metadata for a 1 line description, the About field for more detailed instructions
- include a code repository, a bug tracker, and a home page; this will greatly enhance the possibility of collaboration, and can be done very easily with one of the available web infrastructures (GitHub, GitLab, Bitbucket, etc.)
- choose tags with care: avoid the uninformative ones (e.g. vector) and prefer the ones already used by others (see the plugin website)
- add a proper icon, do not leave the default one; see QGIS interface for a suggestion of the style to be used

15.5.2 Code and help

- do not include generated file (ui_*.py, resources_rc.py, generated help files...) and useless stuff (e.g. .gitignore) in repository
- add the plugin to the appropriate menu (Vector, Raster, Web, Database)
- when appropriate (plugins performing analyses), consider adding the plugin as a subplugin of Processing framework: this will allow users to run it in batch, to integrate it in more complex workflows, and will free you from the burden of designing an interface
- include at least minimal documentation and, if useful for testing and understanding, sample data.

15.5.3 Official Python plugin repository

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

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

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

TODO: Insert a link to the governance document

Permissions

These rules have been implemented in the official plugin repository:

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

Trust management

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

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

Validation

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

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

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

Plugin structure

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

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

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. Elles sont récupérées depuis le dossier `~/ (UserProfile) /python/plugins` et les chemins suivants:

- UNIX/Mac: `(qgis_prefix) /share/qgis/python/plugins`
- Windows: `(qgis_prefix) /python/plugins`

Pour savoir à quoi correspondent `~` et `(UserProfile)`, veuillez vous référer à `core_and_external_plugins`.

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

Créer une extension avec Processing

Suivant le type d'extension que vous voulez développer, il sera parfois plus judicieux d'ajouter sa fonctionnalité sous forme d'un algorithme Processing (ou un ensemble d'algorithmes). Cela vous apportera une meilleure intégration au sein de QGIS, des fonctionnalités supplémentaires (puisque'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).

Pour distribuer ces algorithmes, vous devez créer un nouveau plugin qui les ajoute à la boîte à outils de traitement. Le plugin doit contenir un fournisseur d'algorithmes, qui doit être enregistré lors de l'instanciation du plugin.

Pour créer un plugin à partir de zéro qui contient un fournisseur d'algorithme, vous pouvez suivre ces étapes en utilisant Plugin Builder :

- 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.

Si vous souhaitez ajouter votre extension à Processing, il vous faut@ ajouter un peu de code.

Dans votre fichier `metadata.txt` vous devez ajouter une variable :

```
hasProcessingProvider=yes
```

Au sein du fichier Python qui contient la méthode `initGui` paramétrant votre extension, vous devez adapter quelques lignes comme suit :

```
from qgis.core import QgsApplication
from .processing_provider.provider import Provider

class YourPluginName():

    def __init__(self):
        self.provider = None

    def initProcessing(self):
        self.provider = Provider()
```

```

QgsApplication.processingRegistry().addProvider(self.provider)

def initGui(self):
    self.initProcessing()

def unload(self):
    QgsApplication.processingRegistry().removeProvider(self.provider)

```

You can create a folder `processing_provider` with three files in it:

- `__init__.py` with nothing in it. This is necessary to make a valid Python package.
- `provider.py` which will create the Processing provider and expose your algorithms.

```

from qgis.core import QgsProcessingProvider

from .example_processing_algorithm import ExampleProcessingAlgorithm

class Provider(QgsProcessingProvider):

    def loadAlgorithms(self, *args, **kwargs):
        self.addAlgorithm(ExampleProcessingAlgorithm())
        # add additional algorithms here
        # self.addAlgorithm(MyOtherAlgorithm())

    def id(self, *args, **kwargs):
        """The ID of your plugin, used for identifying the provider.

        This string should be a unique, short, character only string,
        eg "qgis" or "gdal". This string should not be localised.
        """
        return 'yourplugin'

    def name(self, *args, **kwargs):
        """The human friendly name of your plugin in Processing.

        This string should be as short as possible (e.g. "Lastools", not
        "Lastools version 1.0.1 64-bit") and localised.
        """
        return self.tr('Your plugin')

    def icon(self):
        """Should return a QIcon which is used for your provider inside
        the Processing toolbox.
        """
        return QgsProcessingProvider.icon(self)

```

- `example_processing_algorithm.py` which contains the example algorithm file. Copy/paste the content of the script template: https://github.com/qgis/QGIS/blob/release-3_4/python/plugins/processing/script/ScriptTemplate.py

Now you can reload your plugin in QGIS and you should see your example script in the Processing toolbox and modeler.

Bibliothèque d'analyse de réseau

Avertissement: *Despite our constant efforts, information beyond this line may not be updated for QGIS 3. Refer to <https://qgis.org/pyqgis/master> for the python API documentation or, give a hand to update the chapters you know about. Thanks.*

- *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.

17.1 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)

17.2 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.

Converting from a vector layer to the graph is done using the [Builder](#) programming pattern. A graph is constructed using a so-called Director. There is only one Director for now: `QgsLineVectorLayerDirector`. The director sets the basic settings that will be used to construct a graph from a line vector layer, used by the builder to create the graph. Currently, as in the case with the director, only one builder exists: `QgsGraphBuilder`, that creates `QgsGraph` objects. You may want to implement your own builders that will build a graphs compatible with such libraries as [BGL](#) or [NetworkX](#).

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

Il est temps de plonger dans le processus.

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

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

Ensuite, quelques exemples pour créer un directeur

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

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

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

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

Voici la liste complète de la signification de ces paramètres:

- `vl` — couche vecteur utilisée pour construire le graphe
- `directionFieldId` — index du champ de la table d'attribut où est stockée 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)
```

Now we can use the builder, which will create the graph. The `QgsGraphBuilder` class constructor takes several 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])
```

17.3 Analyse de graphe

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

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

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

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

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

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

- `source` — 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)
```

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

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

Here is some very simple code to display the shortest path tree using the graph created with the `shortestTree` method (select linestring layer in *Layers* panel and replace coordinates with your own).

Avertissement: Use this code only as an example, it creates a lot of `QgsRubberBand` objects and may be slow on large datasets.

```
from qgis.core import *
from qgis.gui import *
from qgis.networkanalysis import *
from qgis.PyQt.QtCore import *
from qgis.PyQt.QtGui import *

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

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

graph = builder.graph()

idStart = graph.findVertex(pStart)

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

```

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

```

Same thing but using the `dijkstra` method

```

from qgis.core import *
from qgis.gui import *
from qgis.networkanalysis import *
from qgis.PyQt.QtCore import *
from qgis.PyQt.QtGui 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())

```

17.3.1 Trouver les chemins les plus courts

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

```

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

```

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.

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

```

from qgis.core import *
from qgis.gui import *
from qgis.networkanalysis import *
from qgis.PyQt.QtCore import *
from qgis.PyQt.QtGui import *

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

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

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

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

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

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

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

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

    for pnt in p:
        rb.addPoint(pnt)

```

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

```

from qgis.core import *
from qgis.gui import *
from qgis.networkanalysis import *
from qgis.PyQt.QtCore import *
from qgis.PyQt.QtGui 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)

```

17.3.2 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 surfaces de disponibilité de la caserne de pompiers.

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

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 qgis.core import *
from qgis.gui import *
from qgis.networkanalysis import *
from qgis.PyQt.QtCore import *
from qgis.PyQt.QtGui 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
↳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

Avertissement: *Despite our constant efforts, information beyond this line may not be updated for QGIS 3. Refer to <https://qgis.org/pyqgis/master> for the python API documentation or, give a hand to update the chapters you know about. Thanks.*

- *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`

Python plugins can also run on QGIS Server (see `label_qgisserver`):

- By using the *server interface* (`QgsServerInterface`) a Python plugin running on the server can alter the behavior of existing core services (WMS, WFS etc.).
- With the *server filter interface* (`QgsServerFilter`) you can change the input parameters, change the generated output or even provide new services.
- With the *access control interface* (`QgsAccessControlFilter`) you can apply some access restriction per requests.

18.1 Architecture des extensions de filtre serveur

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

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

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

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.
 - pass request to an instance of `QgsServerInterface`
 - call plugins `requestReady` filters
 - **S'il n'y a pas de réponse**
 - * **Si SERVICE vaut WMS/WFS/WCS.**
 - **Créer un serveur WMS/WFS/WCS.**
 - call server's `executeRequest` and possibly call `sendResponse` plugin filters when streaming output or store the byte stream output and content type in the request handler
 - * call plugins `responseComplete` filters
 - call plugins `sendResponse` filters
 - Demander au gestionnaire d'émettre la réponse.

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

18.1.1 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).

18.1.2 sendResponse

This is called whenever output is sent to **FCGI** `stdout` (and from there, to the client), this is normally done after core services have finished their process and after `responseComplete` hook was called, but in a few cases XML can become so huge that a streaming XML implementation was needed (WFS GetFeature is one of them), in this case, `sendResponse` is called multiple times before the response is complete (and before `responseComplete` is called). The obvious consequence is that `sendResponse` is normally called once but might be exceptionally called multiple times and in that case (and only in that case) it is also called before `responseComplete`.

`sendResponse` is the best place for direct manipulation of core service's output and while `responseComplete` is typically also an option, `sendResponse` is the only viable option in case of streaming services.

18.1.3 responseComplete

This is called once when core services (if hit) finish their process and the request is ready to be sent to the client. As discussed above, this is normally called before `sendResponse` except for streaming services (or other plugin filters) that might have called `sendResponse` earlier.

`responseComplete` is the ideal place to provide new services implementation (WPS or custom services) and to perform direct manipulation of the output coming from core services (for example to add a watermark upon a WMS image).

18.2 Déclencher une exception depuis une extension

Some work has still to be done on this topic: the current implementation can distinguish between handled and unhandled exceptions by setting a `QgsRequestHandler` property to an instance of `QgsMapServiceException`, this way the main C++ code can catch handled python exceptions and ignore unhandled exceptions (or better: log them).

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.

18.3 Écriture d'une extension serveur

A server plugin is a standard QGIS Python plugin as described in *Développer des extensions Python*, that just provides an additional (or alternative) interface: a typical QGIS desktop plugin has access to QGIS application through the `QgisInterface` instance, a server plugin has also access to a `QgsServerInterface`.

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

```
server=True
```

The example plugin discussed here (with many more example filters) is available on github: [QGIS HelloServer Example Plugin](https://github.com/QGIS-HelloServer-Example-Plugin). You could also find more examples at <https://github.com/el Paso/qgis3-server-vagrant/tree/master/resources/web/plugins> or browsing the [QGIS plugins repository](#).

18.3.1 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*
```

18.3.2 __init__.py

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

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

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

18.3.3 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`.

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

- `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 )

```

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

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

Les exemples suivants montrent quelques cas d'utilisation courants :

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

```

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

```

src/core/qgsmessagelog.cpp: 45: (logMessage) [0ms] 2014-12-12T12:39:29 plugin[0]
↳HelloServerServer - loading filter ParamsFilter
src/core/qgsmessagelog.cpp: 45: (logMessage) [1ms] 2014-12-12T12:39:29 Server[0]
↳Server plugin HelloServer loaded!
src/core/qgsmessagelog.cpp: 45: (logMessage) [0ms] 2014-12-12T12:39:29 Server[0]
↳Server python plugins loaded
src/mapserver/qgsgetrequesthandler.cpp: 35: (parseInput) [0ms] query string is:
↳SERVICE=HELLO&request=GetOutput
src/mapserver/qgshttprequesthandler.cpp: 547: (requestStringToParameterMap) [1ms]
↳inserting pair SERVICE // HELLO into the parameter map
src/mapserver/qgshttprequesthandler.cpp: 547: (requestStringToParameterMap) [0ms]
↳inserting pair REQUEST // GetOutput into the parameter map

```

```

src/mapserver/qgsserverfilter.cpp: 42: (requestReady) [0ms] QgsServerFilter_
↳plugin default requestReady called
src/core/qgsmessagelog.cpp: 45: (logMessage) [0ms] 2014-12-12T12:39:29 plugin[0]_
↳HelloFilter.requestReady
src/mapserver/qgis_map_serv.cpp: 235: (configPath) [0ms] Using default_
↳configuration file path: /home/xxx/apps/bin/admin.sld
src/mapserver/qgshttprequestandler.cpp: 49: (setHttpResponse) [0ms] Checking_
↳byte array is ok to set...
src/mapserver/qgshttprequestandler.cpp: 59: (setHttpResponse) [0ms] Byte array_
↳looks good, setting response...
src/core/qgsmessagelog.cpp: 45: (logMessage) [0ms] 2014-12-12T12:39:29 plugin[0]_
↳HelloFilter.responseComplete
src/core/qgsmessagelog.cpp: 45: (logMessage) [0ms] 2014-12-12T12:39:29 plugin[0]_
↳SUCCESS - ParamsFilter.responseComplete
src/core/qgsmessagelog.cpp: 45: (logMessage) [0ms] 2014-12-12T12:39:29 plugin[0]_
↳RemoteConsoleFilter.responseComplete
src/mapserver/qgshttprequestandler.cpp: 158: (sendResponse) [0ms] Sending HTTP_
↳response
src/core/qgsmessagelog.cpp: 45: (logMessage) [0ms] 2014-12-12T12:39:29 plugin[0]_
↳HelloFilter.sendResponse

```

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).

18.3.5 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 qgis.PyQt.QtCore import *
from qgis.PyQt.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 {}".format(request.infoFormat()), 'plugin', QgsMessageLog.INFO)
            # 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.

18.4 Extension de contrôle d'accès

18.4.1 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*

```

18.4.2 __init__.py

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

```

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

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

```

18.4.3 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.

18.4.4 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 ».

18.4.5 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 ».

18.4.6 layerPermissions

Limiter l'accès à la couche.

Return an object of type `LayerPermissions`, which has the properties:

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

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.

18.4.7 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".

18.4.8 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.

18.4.9 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.

19.1 Interface utilisateur

Change Look & Feel

```
from qgis.PyQt.QtWidgets import QApplication

app = QApplication.instance()
qss_file = open(r"/path/to/style/file.qss").read()
app.setStyleSheet(qss_file)
```

Changer l'icône et le titre

```
from qgis.PyQt.QtGui import QIcon

icon = QIcon(r"/path/to/logo/file.png")
iface.mainWindow().setWindowIcon(icon)
iface.mainWindow().setWindowTitle("My QGIS")
```

19.2 Réglages

Get QSettings list

```
from qgis.PyQt.QtCore import QSettings

qs = QSettings()

for k in sorted(qs.allKeys()):
    print(k)
```

19.3 Barres d'outils

Supprimer une barre d'outils

```
from qgis.utils import iface

toolbar = iface.helpToolBar()
parent = toolbar.parentWidget()
parent.removeToolBar(toolbar)

# and add again
parent.addToolBar(toolbar)
```

Remove actions toolbar

```
actions = iface.attributesToolBar().actions()
iface.attributesToolBar().clear()
iface.attributesToolBar().addAction(actions[4])
iface.attributesToolBar().addAction(actions[3])
```

19.4 Menus

Supprimer menu

```
from qgis.utils import iface

# for example Help Menu
menu = iface.helpMenu()
menubar = menu.parentWidget()
menubar.removeAction(menu.menuAction())

# and add again
menubar.addAction(menu.menuAction())
```

19.5 Canevas

Accéder au canevas

```
from qgis.utils import iface

canvas = iface.mapCanvas()
```

Changer la couleur du canevas

```
from qgis.PyQt.QtCore import Qt

iface.mapCanvas().setCanvasColor(Qt.black)
iface.mapCanvas().refresh()
```

Map Update interval

```
from qgis.PyQt.QtCore import QSettings
# Set milliseconds (150 milliseconds)
QSettings().setValue("/qgis/map_update_interval", 150)
```

19.6 Couches

Ajouter couche vecteur

```

from qgis.utils import iface

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

```

Récupérer la couche active

```
layer = iface.activeLayer()
```

Lister toutes les couches

```

from qgis.core import QgsProject

QgsProject.instance().mapLayers().values()

```

Obtenir le nom des couches

```

layers_names = []
for layer in QgsProject.instance().mapLayers().values():
    layers_names.append(layer.name())

print("layers TOC = {}".format(layers_names))

```

Sinon

```

layers_names = [layer.name() for layer in QgsProject.instance().mapLayers().
↪values()]
print("layers TOC = {}".format(layers_names))

```

Recherche par nom de couche

```

from qgis.core import QgsProject

layer = QgsProject.instance().mapLayersByName("layer name you like")[0]
print(layer.name())

```

Définir la couche active

```

from qgis.core import QgsProject

layer = QgsProject.instance().mapLayersByName("layer name you like")[0]
iface.setActiveLayer(layer)

```

Refresh layer at interval

```

from qgis.core import QgsProject

layer = QgsProject.instance().mapLayersByName("layer name you like")[0]
# Set seconds (5 seconds)
layer.setAutoRefreshInterval(5000)
# Enable auto refresh
layer.setAutoRefreshEnabled(True)

```

Afficher des méthodes

```
dir(layer)
```

Ajouter nouvelle entité avec géométrie

```
from qgis.core import QgsFeature, QgsGeometry

feat = QgsFeature()
geom = QgsGeometry()
feat.setGeometry(geom)
feat.setFields(layer.fields())

iface.openFeatureForm(layer, feat, False)
```

Ajouter nouvelle entité sans géométrie

```
from qgis.core import QgsPointXY

pr = layer.dataProvider()
feat = QgsFeature()
feat.setGeometry(QgsGeometry.fromPointXY(QgsPointXY(10,10)))
pr.addFeatures([feat])
```

Retourner entités

```
for f in layer.getFeatures():
    print(f)
```

Retourner entités sélectionnées

```
for f in layer.selectedFeatures():
    print(f)
```

Retourner l'id des entités sélectionnées

```
selected_ids = layer.selectedFeatureIds()
print(selected_ids)
```

Create a memory layer from selected features Ids

```
from qgis.core import QgsFeatureRequest

memory_layer = layer.materialize(QgsFeatureRequest().setFilterFids(layer.
↳selectedFeatureIds()))
QgsProject.instance().addMapLayer(memory_layer)
```

Retourner géométrie

```
# Point layer
for f in layer.getFeatures():
    geom = f.geometry()
    print('%f, %f' % (geom.asPoint().y(), geom.asPoint().x()))
```

Déplacer géométrie

```
geom.translate(100, 100)
poly.setGeometry(geom)
```

Définir le SCR

```
from qgis.core import QgsProject, QgsCoordinateReferenceSystem

for layer in QgsProject.instance().mapLayers().values():
    layer.setCrs(QgsCoordinateReferenceSystem(4326, QgsCoordinateReferenceSystem.
↳EpsgCrsId))
```

Afficher le SCR

```

from qgis.core import QgsProject

for layer in QgsProject.instance().mapLayers().values():
    crs = layer.crs().authid()
    layer.setName('{} ({}).format(layer.name(), crs))

```

Cacher une colonne du champ

```

from qgis.core import QgsEditorWidgetSetup

def fieldVisibility (layer, fname):
    setup = QgsEditorWidgetSetup('Hidden', {})
    for i, column in enumerate(layer.fields()):
        if column.name() == fname:
            layer.setEditorWidgetSetup(idx, setup)
            break
        else:
            continue

```

Couche depuis le WKT

```

from qgis.core import QgsVectorLayer, QgsFeature, QgsGeometry, QgsProject

layer = QgsVectorLayer('Polygon?crs=epsg:4326', 'Mississippi', 'memory')
pr = layer.dataProvider()
poly = QgsFeature()
geom = QgsGeometry.fromWkt("POLYGON ((-88.82 34.99,-88.0934.89,-88.39 30.34,-89.57,
↳30.18,-89.73 31,-91.63 30.99,-90.8732.37,-91.23 33.44,-90.93 34.23,-90.30 34.99,-
↳88.82 34.99))")
poly.setGeometry(geom)
pr.addFeatures([poly])
layer.updateExtents()
QgsProject.instance().addMapLayers([layer])

```

Charger toutes les couches du GeoPackage

```

from qgis.core import QgsVectorLayer, QgsProject

fileName = "/path/to/gpkg/file.gpkg"
layer = QgsVectorLayer(fileName, "test", "ogr")
subLayers = layer.dataProvider().subLayers()

for subLayer in subLayers:
    name = subLayer.split('!::!')[1]
    uri = "%s|layername=%s" % (fileName, name,)
    # Create layer
    sub_vlayer = QgsVectorLayer(uri, name, 'ogr')
    # Add layer to map
    QgsProject.instance().addMapLayer(sub_vlayer)

```

Charger couche tuile (couche XYZ)

```

from qgis.core import QgsRasterLayer, QgsProject

def loadXYZ(url, name):
    rasterLyr = QgsRasterLayer("type=xyz&url=" + url, name, "wms")
    QgsProject.instance().addMapLayer(rasterLyr)

urlWithParams = 'type=xyz&url=https://a.tile.openstreetmap.org/%7Bz%7D/%7Bx%7D/%7By
↳%7D.png&zmax=19&zmin=0&crs=EPSG3857'
loadXYZ(urlWithParams, 'OpenStreetMap')

```

Enlever toutes les couches

```
QgsProject.instance().removeAllMapLayers()
```

Enlever tout

```
QgsProject.instance().clear()
```

19.7 Table des matières

Accès aux couches cochées

```
from qgis.utils import iface
iface.mapCanvas().layers()
```

Supprimer le menu contextuel

```
ltv = iface.layerTreeView()
mp = ltv.menuProvider()
ltv.setMenuProvider(None)
# Restore
ltv.setMenuProvider(mp)
```

19.8 Table des matières (avancé)

Noeud racine

```
from qgis.core import QgsProject
root = QgsProject.instance().layerTreeRoot()
print (root)
print (root.children())
```

Access the first child node

```
from qgis.core import QgsLayerTreeGroup, QgsLayerTreeLayer, QgsLayerTree
child0 = root.children()[0]
print (child0.name())
print (type(child0))
print (isinstance(child0, QgsLayerTreeLayer))
print (isinstance(child0.parent(), QgsLayerTree))
```

Trouver des groupes et des nœuds

```
from qgis.core import QgsLayerTreeGroup, QgsLayerTreeLayer

def get_group_layers(group):
    print('- group: ' + group.name())
    for child in group.children():
        if isinstance(child, QgsLayerTreeGroup):
            # Recursive call to get nested groups
            get_group_layers(child)
        else:
            print(' - layer: ' + child.name())

root = QgsProject.instance().layerTreeRoot()
```

```

for child in root.children():
    if isinstance(child, QgsLayerTreeGroup):
        get_group_layers(child)
    elif isinstance(child, QgsLayerTreeLayer):
        print ('- layer: ' + child.name())

```

Trouver un groupe à partir du nom

```
print (root.findGroup("My Group"))
```

Ajouter couche

```

from qgis.core import QgsVectorLayer, QgsProject

layer1 = QgsVectorLayer("Point?crs=EPSG:4326", "layer name you like", "memory")
QgsProject.instance().addMapLayer(layer1, False)
node_layer1 = root.addLayer(layer1)

```

Ajouter groupe

```

from qgis.core import QgsLayerTreeGroup

node_group2 = QgsLayerTreeGroup("Group 2")
root.addChildNode(node_group2)

```

Supprimer couche

```
root.removeLayer(layer1)
```

Supprimer groupe

```
root.removeChildNode(node_group2)
```

Déplacer noeud

```

cloned_group1 = node_group.clone()
root.insertChildNode(0, cloned_group1)
root.removeChildNode(node_group)

```

Renommer noeud

```

cloned_group1.setName("Group X")
node_layer1.setName("Layer X")

```

Changer la visibilité

```

print (cloned_group1.isVisible())
cloned_group1.setItemVisibilityChecked(False)
node_layer1.setItemVisibilityChecked(False)

```

Expand node

```

print (cloned_group1.isExpanded())
cloned_group1.setExpanded(False)

```

Hidden node trick

```

from qgis.core import QgsProject

model = iface.layerTreeView().layerTreeModel()
ltv = iface.layerTreeView()
root = QgsProject.instance().layerTreeRoot()

```

```

layer = QgsProject.instance().mapLayersByName('layer name you like')[0]
node=root.findLayer( layer.id())

index = model.node2index( node )
ltv.setRowHidden( index.row(), index.parent(), True )
node.setCustomProperty( 'nodeHidden', 'true' )
ltv.setCurrentIndex(model.node2index(root))

```

Node signals

```

def onWillAddChildren(node, indexFrom, indexTo):
    print ("WILL ADD", node, indexFrom, indexTo)

def onAddedChildren(node, indexFrom, indexTo):
    print ("ADDED", node, indexFrom, indexTo)

root.willAddChildren.connect (onWillAddChildren)
root.addedChildren.connect (onAddedChildren)

```

Créer nouvelle table des matières (TDM)

```

from qgis.core import QgsProject, QgsLayerTreeModel
from qgis.gui import QgsLayerTreeView

root = QgsProject.instance().layerTreeRoot()
model = QgsLayerTreeModel(root)
view = QgsLayerTreeView()
view.setModel(model)
view.show()

```

19.9 Traitement algorithmes

Obtenir liste algorithmes

```

from qgis.core import QgsApplication

for alg in QgsApplication.processingRegistry().algorithms():
    print("{}: {} --> {}".format(alg.provider().name(), alg.name(), alg.
↳displayName()))

```

Sinon

```

def alglist():
    s = ''
    for i in QgsApplication.processingRegistry().algorithms():
        l = i.displayName().ljust(50, "-")
        r = i.id()
        s += '{}---->{}\n'.format(l, r)
    print(s)

```

Obtenir aide algorithmes

Sélection aléatoire

```

import processing

processing.algorithmHelp("qgis:randomselection")

```

Exécuter l’algorithme

Dans cet exemple, le résultat est stocké dans une couche en mémoire temporaire, qui est ajoutée au projet.

```
import processing
result = processing.run("native:buffer", {'INPUT': layer, 'OUTPUT': 'memory:'})
QgsProject.instance().addMapLayer(result['OUTPUT'])
```

Combien d'algorithmes ?

```
from qgis.core import QgsApplication

len(QgsApplication.processingRegistry().algorithms())
```

How many providers are there?

```
from qgis.core import QgsApplication

len(QgsApplication.processingRegistry().providers())
```

How many expressions are there?

```
from qgis.core import QgsExpression

len(QgsExpression.Functions())
```

19.10 Décorateurs

Droits d'auteur

```
from qgis.PyQt.Qt import QTextDocument
from qgis.PyQt.QtGui import QFont

mQFont = "Sans Serif"
mQFontSize = 9
mLabelQString = "© QGIS 2019"
mMarginHorizontal = 0
mMarginVertical = 0
mLabelQColor = "#FF0000"

INCHES_TO_MM = 0.0393700787402 # 1 millimeter = 0.0393700787402 inches
case = 2

def add_copyright(p, text, xOffset, yOffset):
    p.translate( xOffset , yOffset )
    text.drawContents(p)
    p.setWorldTransform( p.worldTransform() )

def _on_render_complete(p):
    deviceHeight = p.device().height() # Get paint device height on which this_
    ↪painter is currently painting
    deviceWidth = p.device().width() # Get paint device width on which this_
    ↪painter is currently painting
    # Create new container for structured rich text
    text = QTextDocument()
    font = QFont()
    font.setFamily(mQFont)
    font.setPointSize(int(mQFontSize))
    text.setDefaultFont(font)
    style = "<style type='text/css'> p {color: " + mLabelQColor + "}</style>"
    text.setHtml( style + "<p>" + mLabelQString + "</p>" )
    # Text Size
```

```
size = text.size()

# RenderMillimeters
pixelsInchX = p.device().logicalDpiX()
pixelsInchY = p.device().logicalDpiY()
xOffset = pixelsInchX * INCHES_TO_MM * int(mMarginHorizontal)
yOffset = pixelsInchY * INCHES_TO_MM * int(mMarginVertical)

# Calculate positions
if case == 0:
    # Top Left
    add_copyright(p, text, xOffset, yOffset)

elif case == 1:
    # Bottom Left
    yOffset = deviceHeight - yOffset - size.height()
    add_copyright(p, text, xOffset, yOffset)

elif case == 2:
    # Top Right
    xOffset = deviceWidth - xOffset - size.width()
    add_copyright(p, text, xOffset, yOffset)

elif case == 3:
    # Bottom Right
    yOffset = deviceHeight - yOffset - size.height()
    xOffset = deviceWidth - xOffset - size.width()
    add_copyright(p, text, xOffset, yOffset)

elif case == 4:
    # Top Center
    xOffset = deviceWidth / 2
    add_copyright(p, text, xOffset, yOffset)

else:
    # Bottom Center
    yOffset = deviceHeight - yOffset - size.height()
    xOffset = deviceWidth / 2
    add_copyright(p, text, xOffset, yOffset)

# Emitted when the canvas has rendered
iface.mapCanvas().renderComplete.connect(_on_render_complete)
# Repaint the canvas map
iface.mapCanvas().refresh()
```

19.11 Sources

- QGIS Python (PyQGIS) API
- QGIS C++ API
- Questions QGIS sur StackOverFlow
- Script de Klas Karlsson
- Boundless lib-qgis-common repository