



# **PyQGIS 3.10 developer cookbook**

**QGIS Project**

**déc. 09, 2020**



---

# Table des matières

---

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Scripter dans la console Python	2
1.2	Extensions Python	2
1.3	Exécuter du code python quand QGIS démarre	3
1.3.1	Le fichier <code>startup.py</code>	3
1.3.2	La variable d'environnement <code>PYQGIS_STARTUP</code>	3
1.4	Applications Python	3
1.4.1	Utiliser PyQGIS dans des scripts indépendants	4
1.4.2	Utiliser PyQGIS dans une application personnalisée	4
1.4.3	Exécuter des applications personnalisées	5
1.5	Notes techniques sur PyQt et SIP	6
<b>2</b>	<b>Chargement de projets</b>	<b>7</b>
2.1	Résoudre les mauvais chemins	8
<b>3</b>	<b>Chargement de couches</b>	<b>9</b>
3.1	Couches vectorielles	9
3.2	Couches raster	12
3.3	QgsProject instance	14
<b>4</b>	<b>Accéder à la table des matières (TOC)</b>	<b>17</b>
4.1	The QgsProject class	17
4.2	QgsLayerTreeGroup class	18
<b>5</b>	<b>Utiliser des couches raster</b>	<b>21</b>
5.1	Détails d'une couche	21
5.2	Moteur de rendu	22
5.2.1	Rasters mono-bande	22
5.2.2	Rasters multi-bandes	23
5.3	Interrogation des données	23
<b>6</b>	<b>Utilisation de couches vectorielles</b>	<b>25</b>
6.1	Récupérer les informations relatives aux attributs	26
6.2	Itérer sur une couche vecteur	26
6.3	Sélection des entités	27
6.3.1	Accès aux attributs	28
6.3.2	Itérer sur une sélection d'entités	28
6.3.3	Itérer sur un sous-ensemble d'entités	28
6.4	Modifier des couches vecteur	29
6.4.1	Ajout d'Entités	30
6.4.2	Suppression d'Entités	30

6.4.3	Modifier des Entités . . . . .	31
6.4.4	Modifier des couches vecteur à l'aide d'un tampon d'édition . . . . .	31
6.4.5	Ajout et Suppression de Champs . . . . .	32
6.5	Utilisation des index spatiaux . . . . .	33
6.6	Création de couches vecteur . . . . .	34
6.6.1	A partir d'une instance de <code>QgsVectorFileWriter</code> . . . . .	34
6.6.2	Directement à partir des entites . . . . .	36
6.6.3	Depuis une instance de <code>QgsVectorLayer</code> . . . . .	36
6.7	Apparence (Symbologie) des couches vecteur . . . . .	38
6.7.1	Moteur de rendu à symbole unique . . . . .	38
6.7.2	Moteur de rendu à symboles catégorisés . . . . .	39
6.7.3	Moteur de rendu à symboles gradués . . . . .	40
6.7.4	Travailler avec les symboles . . . . .	41
6.7.5	Créer ses propres moteurs de rendu . . . . .	44
6.8	Sujets complémentaires . . . . .	46
<b>7</b>	<b>Manipulation de la géométrie</b> . . . . .	<b>47</b>
7.1	Construction de géométrie . . . . .	47
7.2	Accéder à la Géométrie . . . . .	48
7.3	Prédicats et opérations géométriques . . . . .	49
<b>8</b>	<b>Support de projections</b> . . . . .	<b>53</b>
8.1	Système de coordonnées de référence . . . . .	53
8.2	Transformation de SCR . . . . .	54
<b>9</b>	<b>Utilisation du canevas de carte</b> . . . . .	<b>57</b>
9.1	Intégrer un canevas de carte . . . . .	58
9.2	Contour d'édition et symboles de sommets . . . . .	58
9.3	Utiliser les outils cartographiques avec le canevas . . . . .	59
9.4	Ecrire des outils cartographiques personnalisés . . . . .	61
9.5	Ecrire des éléments de canevas de carte personnalisés . . . . .	62
<b>10</b>	<b>Rendu cartographique et Impression</b> . . . . .	<b>65</b>
10.1	Rendu simple . . . . .	65
10.2	Rendu des couches ayant différents SCR . . . . .	66
10.3	Sortie en utilisant la mise en page . . . . .	66
10.3.1	Exporter la mise en page . . . . .	67
10.3.2	Exporter un atlas . . . . .	68
<b>11</b>	<b>Expressions, Filtrage et Calcul de valeurs</b> . . . . .	<b>69</b>
11.1	Analyse syntaxique d'expressions . . . . .	70
11.2	Évaluation des expressions . . . . .	70
11.2.1	Expressions basiques . . . . .	70
11.2.2	Expressions avec entités . . . . .	70
11.2.3	Filtrer une couche à l'aide d'expressions . . . . .	72
11.3	Gestion des erreurs dans une expression . . . . .	72
<b>12</b>	<b>Lecture et sauvegarde de configurations</b> . . . . .	<b>73</b>
<b>13</b>	<b>Communiquer avec l'utilisateur</b> . . . . .	<b>77</b>
13.1	Afficher des messages. La classe <code>QgsMessageBar</code> . . . . .	77
13.2	Afficher la progression . . . . .	80
13.3	Journal . . . . .	80
13.3.1	<code>QgsMessageLog</code> . . . . .	81
13.3.2	Le python intégré dans le module de journalisation . . . . .	81
<b>14</b>	<b>Infrastructure d'authentification</b> . . . . .	<b>83</b>
14.1	Introduction . . . . .	83
14.2	Glossaire . . . . .	84

14.3	QgsAuthManager le point d'entrée . . . . .	84
14.3.1	Initier le gestionnaire et définir le mot de passe principal . . . . .	84
14.3.2	Remplir authdb avec une nouvelle entrée de configuration d'authentification . . . . .	85
14.3.3	Supprimer une entrée de l'authdb . . . . .	86
14.3.4	Laissez l'extension authcfg à QgsAuthManager . . . . .	86
14.4	Adapter les plugins pour utiliser l'infrastructure d'authentification . . . . .	87
14.5	Interfaces d'authentification . . . . .	88
14.5.1	Fenêtre de sélection des identifiants . . . . .	88
14.5.2	Authentication Editor GUI . . . . .	89
14.5.3	Authorities Editor GUI . . . . .	89
<b>15</b>	<b>Tâches - faire un gros travail en arrière-plan</b>	<b>93</b>
15.1	Introduction . . . . .	93
15.2	Exemples . . . . .	94
15.2.1	Extension de QgsTask . . . . .	94
15.2.2	Tâche de la fonction . . . . .	96
15.2.3	Tâche à partir d'un algorithme de traitement . . . . .	98
<b>16</b>	<b>Développer des extensions Python</b>	<b>99</b>
16.1	Structurer les plugins Python . . . . .	99
16.1.1	Ecriture d'un plugin . . . . .	100
16.1.2	Contenu de l'extension . . . . .	101
16.1.3	Documentation . . . . .	105
16.1.4	Traduction . . . . .	105
16.1.5	Conseils et Astuces . . . . .	107
16.2	Extraits de code . . . . .	108
16.2.1	Comment appeler une méthode par un raccourci clavier . . . . .	108
16.2.2	Comment basculer les couches . . . . .	108
16.2.3	Comment accéder à la table d'attributs des entites sélectionnées . . . . .	109
16.2.4	Interface pour le plugin dans le dialogue des options . . . . .	109
16.3	Utilisation de la classe QgsPluginLayer . . . . .	110
16.3.1	Héritage de la classe QgsPluginLayer . . . . .	110
16.4	Paramètres de l'IDE pour l'écriture et le débogage de plugins . . . . .	112
16.4.1	Plugins utiles pour écrire des plugins Python . . . . .	112
16.4.2	Une note sur la configuration de votre IDE sous Linux et Windows . . . . .	112
16.4.3	Débogage à l'aide de l'IDE Pyscripter (Windows) . . . . .	113
16.4.4	Débogage à l'aide d'Eclipse et PyDev . . . . .	113
16.4.5	Débogage avec PyCharm sur Ubuntu avec QGIS compilé . . . . .	117
16.4.6	Débogage à l'aide de PDB . . . . .	119
16.5	Déblocage de votre plugin . . . . .	119
16.5.1	Métadonnées et noms . . . . .	120
16.5.2	Code et aide . . . . .	120
16.5.3	Dépôt officiel des extensions QGIS . . . . .	120
<b>17</b>	<b>Créer une extension avec Processing</b>	<b>123</b>
17.1	Créer à partir de zéro . . . . .	123
17.2	Mise à jour d'un plugin . . . . .	124
<b>18</b>	<b>Bibliothèque d'analyse de réseau</b>	<b>127</b>
18.1	Information générale . . . . .	127
18.2	Construire un graphe . . . . .	127
18.3	Analyse de graphe . . . . .	130
18.3.1	Trouver les chemins les plus courts . . . . .	132
18.3.2	Surfaces de disponibilité . . . . .	134
<b>19</b>	<b>QGIS server et Python</b>	<b>137</b>
19.1	Introduction . . . . .	137
19.2	Principes de base de l'API du serveur . . . . .	138
19.3	Autonome ou intégré . . . . .	138

19.4	Plugins de serveur . . . . .	139
19.4.1	Plugins pour filtres de serveur . . . . .	139
19.4.2	Services personnalisés . . . . .	147
19.4.3	API personnalisées . . . . .	148
<b>20</b>	<b>Fiche d'information sur PyQGIS</b>	<b>151</b>
20.1	Interface utilisateur . . . . .	151
20.2	Réglages . . . . .	151
20.3	Barres d'outils . . . . .	152
20.4	Menus . . . . .	152
20.5	Canevas . . . . .	152
20.6	Couches . . . . .	153
20.7	Table des matières . . . . .	156
20.8	Table des matières (avancé) . . . . .	156
20.9	Traitement algorithmes . . . . .	160
20.10	Décorateurs . . . . .	160
20.11	Composeur . . . . .	162
20.12	Sources . . . . .	162

# CHAPITRE 1

---

## Introduction

---

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 algorithmes de traitement
- Créer des fonctions pour des expressions dans QGIS
- créer des applications personnalisées basées sur l'API QGIS

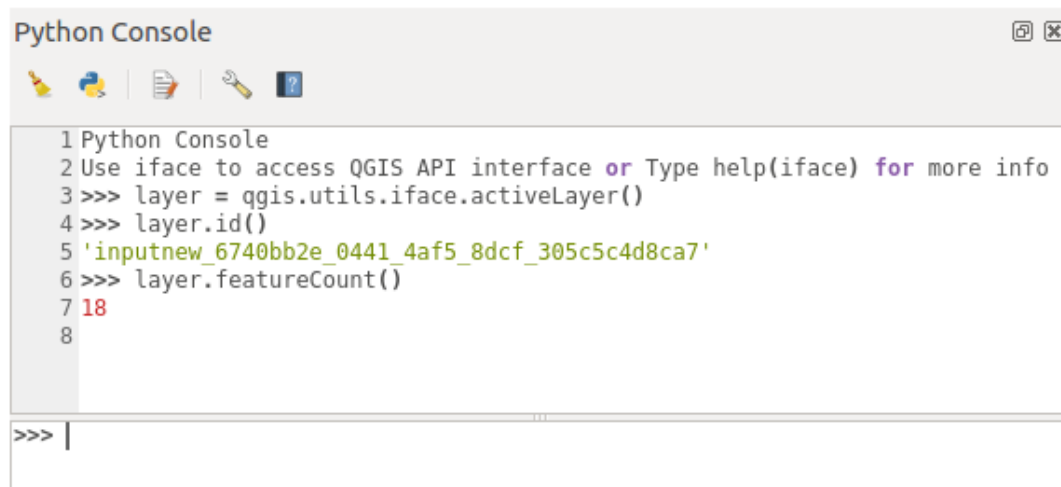
Des liaisons python sont également disponibles pour QGIS Server, ce qui permet de mettre en œuvre des extensions (see *QGIS server et Python*) et des liaisons qui peuvent être intégrées pour intégrer QGIS Server dans une application Python.

Une **API QGIS** référençant et décrivant les classes de la bibliothèque QGIS :`pyqgis` : est disponible. *L'API Python* (`pyqgis`) <> est quasiment identique à l'API C++.

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* :



```
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


>>> |
```

Fig. 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 future, 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 *Préférences*  *Raccourcis clavier...*).

## 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 *QGIS server et Python* 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 :

```

1 from qgis.core import *
2
3 # Supply path to qgis install location
4 QgsApplication.setPrefixPath("/path/to/qgis/installation", True)
5
6 # Create a reference to the QgsApplication. Setting the
7 # second argument to False disables the GUI.
8 qgs = QgsApplication([], False)
9
10 # Load providers
11 qgs.initQgis()
12
13 # Write your code here to load some layers, use processing
14 # algorithms, etc.
15
16 # Finally, exitQgis() is called to remove the
17 # provider and layer registries from memory
18 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 de la méthode `setPrefixPath` est mis à `True`, indiquant que les chemins par défaut sont à utiliser.

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.

```

1 from qgis.core import *
2
3 # Supply the path to the qgis install location
4 QgsApplication.setPrefixPath("/path/to/qgis/installation", True)
5
6 # Create a reference to the QgsApplication.
7 # Setting the second argument to True enables the GUI. We need
8 # this since this is a custom application.
9
10 qgs = QgsApplication([], True)
11
12 # load providers
13 qgs.initQgis()
14
15 # Write your code here to load some layers, use processing
16 # algorithms, etc.

```

(suite sur la page suivante)

(suite de la page précédente)

```

17
18 # Finally, exitQgis() is called to remove the
19 # provider and layer registries from memory
20 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.

Les extraits de code sur cette page nécessitent les importations suivantes si vous êtes en dehors de la console pyqgis :

```
1 from qgis.core import (  
2     QgsProject,  
3     QgsPathResolver  
4 )  
5  
6 from qgis.gui import (  
7     QgsLayerTreeMapCanvasBridge,  
8 )
```

---

## 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 *Applications Python*).

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 :

```
1 # If you are not inside a QGIS console you first need to import
2 # qgis and PyQt classes you will use in this script as shown below:
3 from qgis.core import QgsProject
4 # Get the project instance
5 project = QgsProject.instance()
6 # Print the current project file name (might be empty in case no projects have
7 # been loaded)
8 # print(project.fileName())
9
10 # Load another project
11 import os
12 print(os.getcwd())
13 project.read('testdata/01_project.qgs')
14 print(project.fileName())
```

```
...
testdata/01_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 `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('testdata/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('testdata/my_new_qgis_project.qgs')
```

```
...
```

## 2.1 Résoudre les mauvais chemins

Il peut arriver que des couches chargées dans le projet soient déplacées vers un autre endroit. Lorsque le projet est rechargé, tous les chemins des couches sont brisés.

La classe `QgsPathResolver` avec la classe `setPathPreprocessor()` permet de définir une fonction de préprocesseur de chemin personnalisé, qui permet de manipuler les chemins et les sources de données avant de les résoudre en références de fichiers ou sources de couches.

La fonction de traitement doit accepter un argument de type chaîne de caractères unique (représentant le chemin d'accès au fichier ou la source de données d'origine) et renvoyer une version traitée de ce chemin.

La fonction de pré-processeur de chemin est appelée **avant** tout gestionnaire de mauvaise couche.

Quelques cas d'utilisation :

1. remplacer un chemin qui n'est plus bon :

```
def my_processor(path):
    return path.replace('c:/Users/ClintBarton/Documents/Projects', 'x:/
↳Projects/')

QgsPathResolver.setPathPreprocessor(my_processor)
```

2. remplacer une adresse d'hôte de base de données par une nouvelle :

```
def my_processor(path):
    return path.replace('host=10.1.1.115', 'host=10.1.1.116')

QgsPathResolver.setPathPreprocessor(my_processor)
```

3. remplacer les ids stockés dans la base de données par de nouveaux ids :

```
1 def my_processor(path):
2     path= path.replace("user='gis_team'", "user='team_awesome'")
3     path = path.replace("password='cats'", "password='g7as!m*')")
4     return path
5
6 QgsPathResolver.setPathPreprocessor(my_processor)
```

---

## Chargement de couches

---

Les extraits de code sur cette page nécessitent 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 et ajouter une instance de couche vecteur au projet, il faut spécifier l'identifiant de la source de données de la couche, le nom de la couche et le nom du provider :

```
1 # get the path to the shapefile e.g. /home/project/data/ports.shp
2 path_to_airports_layer = "testdata/airports.shp"
3
4 # The format is:
5 # vlayer = QgsVectorLayer(data_source, layer_name, provider_name)
6
7 vlayer = QgsVectorLayer(path_to_airports_layer, "Airports layer", "ogr")
8 if not vlayer.isValid():
9     print("Layer failed to load!")
10 else:
11     QgsProject.instance().addMapLayer(vlayer)
```

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 :

```

1 # get the path to a geopackage e.g. /usr/share/qgis/resources/data/world_map.gpkg
2 path_to_gpkg = os.path.join(QgsApplication.pkgDataPath(), "resources", "data",
   ↪ "world_map.gpkg")
3 # append the layername part
4 gpkg_countries_layer = path_to_gpkg + "|layername=countries"
5 # e.g. gpkg_places_layer = "/usr/share/qgis/resources/data/world_map.
   ↪ gpkg|layername=countries"
6 vlayer = QgsVectorLayer(gpkg_countries_layer, "Countries layer", "ogr")
7 if not vlayer.isValid():
8     print("Layer failed to load!")
9 else:
10    QgsProject.instance().addMapLayer(vlayer)

```

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_airports_layer, "Airports 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("testdata/airports.shp", "layer_name_you_like",
   ↪ "ogr")
QgsProject.instance().addMapLayer(vlayer)

```

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

```

uri = "testdata/sample.dxf|layername=entities|geometrytype=Polygon"
vlayer = QgsVectorLayer(uri, "layer_name_you_like", "ogr")
QgsProject.instance().addMapLayer(vlayer)

```

- 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 :

```

1 uri = QgsDataSourceUri()
2 # set host name, port, database name, username and password
3 uri.setConnection("localhost", "5432", "dbname", "johny", "xxx")
4 # set database schema, table name, geometry column and optionally
5 # subset (WHERE clause)
6 uri.setDataSource("public", "roads", "the_geom", "cityid = 2643", "primary_
   ↪ key_field")
7
8 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 = "file://{}/testdata/delimited_xy.csv?delimiter={}&xField={}&yField={}".
↳format(os.getcwd(), ";", "x", "y")
vlayer = QgsVectorLayer(uri, "layer name you like", "delimitedtext")
QgsProject.instance().addMapLayer(vlayer)
```

**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 = "testdata/layers.gpx?type=track"
vlayer = QgsVectorLayer(uri, "layer name you like", "gpx")
QgsProject.instance().addMapLayer(vlayer)
```

- 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 :

```
1 uri = QgsDataSourceUri()
2 uri.setDatabase('/home/martin/test-2.3.sqlite')
3 schema = ''
4 table = 'Towns'
5 geom_column = 'Geometry'
6 uri.setDataSource(schema, table, geom_column)
7
8 display_name = 'Towns'
9 vlayer = QgsVectorLayer(uri.uri(), display_name, 'spatialite')
10 QgsProject.instance().addMapLayer(vlayer)
```

- 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")
QgsProject.instance().addMapLayer(vlayer)
```

- Connexion WFS : la connexion est définie avec un URI et en utilisant le fournisseur WFS :

```
uri = "https://demo.geo-solutions.it/geoserver/ows?service=WFS&version=1.1.0&
↳request=GetFeature&typename=geosolutions:regioni"
vlayer = QgsVectorLayer(uri, "my wfs layer", "WFS")
QgsProject.instance().addMapLayer(vlayer)
```

L'uri peut être créée en utilisant la bibliothèque standard `urllib` :

```
1 import urllib
2
3 params = {
4     'service': 'WFS',
5     'version': '1.1.0',
6     'request': 'GetFeature',
7     'typename': 'geosolutions:regioni',
8     'srsname': "EPSG:4326"
```

(suite sur la page suivante)

(suite de la page précédente)

```

9 }
10 uri2 = 'https://demo.geo-solutions.it/geoserver/ows?' + urllib.parse.
↳urllib.parse.urlencode(params))

```

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

```

1 uri = "https://demo.geo-solutions.it/geoserver/ows?service=WFS&version=1.1.0&
↳request=GetFeature&typename=geosolutions:regioni"
2 provider_options = QgsDataProvider.ProviderOptions()
3 # Use project's transform context
4 provider_options.transformContext = QgsProject.instance().transformContext()
5 vlayer.setDataSource(uri, "layer name you like", "WFS", provider_options)
6 QgsProject.instance().addMapLayer(vlayer)

```

## 3.2 Couches raster

Pour accéder aux fichiers raster, la bibliothèque GDAL est utilisée. Elle prend en charge un large éventail de formats de fichiers. Si vous avez des difficultés à ouvrir certains fichiers, vérifiez si votre GDAL prend en charge le format en question (tous les formats ne sont pas disponibles par défaut). Pour charger un raster à partir d'un fichier, indiquez son nom de fichier et son nom d'affichage :

```

1 # get the path to a tif file e.g. /home/project/data/srtm.tif
2 path_to_tif = "qgis-projects/python_cookbook/data/srtm.tif"
3 rlayer = QgsRasterLayer(path_to_tif, "SRTM layer name")
4 if not rlayer.isValid():
5     print("Layer failed to load!")

```

Pour charger un raster à partir d'un géopackage :

```

1 # get the path to a geopackage e.g. /home/project/data/data.gpkg
2 path_to_gpkg = os.path.join(os.getcwd(), "testdata", "sublayers.gpkg")
3 # gpkg_raster_layer = "GPKG:/home/project/data/data.gpkg:srtm"
4 gpkg_raster_layer = "GPKG:" + path_to_gpkg + ":srtm"
5
6 rlayer = QgsRasterLayer(gpkg_raster_layer, "layer name you like", "gdal")
7
8 if not rlayer.isValid():
9     print("Layer failed to load!")

```

Tout comme les couches vecteur, les couches raster peuvent être chargées en utilisant la fonction `addRasterLayer` de l'objet `QgisInterface` :

```
iface.addRasterLayer(path_to_tif, "layer name you like")
```

Cela crée une nouvelle couche et l'ajoute au projet en cours (en la faisant apparaître dans la liste des couches) en une seule étape.

Pour charger un raster PostGIS :

Les raster PostGIS, similaires aux vecteurs PostGIS, peuvent être ajoutées à un projet à l'aide d'une chaîne URI. Il est efficace de créer un dictionnaire de chaînes pour les paramètres de connexion à la base de données. Le dictionnaire est ensuite chargé dans un URI vide, avant d'ajouter le raster. Notez qu'il convient d'utiliser `None` lorsque l'on souhaite laisser le paramètre vide :

```

1 uri_config = {#
2 # a dictionary of database parameters
3 'dbname':'gis_db', # The PostgreSQL database to connect to.
4 'host':'localhost', # The host IP address or localhost.
5 'port':'5432', # The port to connect on.
6 'sslmode':'disable', # The SSL/TLS mode. Options: allow, disable, prefer,
↳require, verify-ca, verify-full
7 # user and password are not needed if stored in the authcfg or service
8 'user':None, # The PostgreSQL user name, also accepts the new WFS
↳provider naming.
9 'password':None, # The PostgreSQL password for the user.
10 'service':None, # The PostgreSQL service to be used for connection to the
↳database.
11 'authcfg':'QconfigId', # The QGIS authentication database ID holding connection
↳details.
12 # table and raster column details
13 'schema':'public', # The database schema that the table is located in.
14 'table':'my_rasters', # The database table to be loaded.
15 'column':'rast', # raster column in PostGIS table
16 'mode':'2', # GDAL 'mode' parameter, 2 union raster tiles, 1 separate
↳tiles (may require user input)
17 'sql':None, # An SQL WHERE clause.
18 'key':None, # A key column from the table.
19 'srid':None, # A string designating the SRID of the coordinate
↳reference system.
20 'estimatedmetadata':'False', # A boolean value telling if the metadata is
↳estimated.
21 'type':None, # A WKT string designating the WKB Type.
22 'selectatid':None, # Set to True to disable selection by feature ID.
23 'options':None, # other PostgreSQL connection options not in this list.
24 'connect_timeout':None,
25 'hostaddr':None,
26 'driver':None,
27 'tty':None,
28 'requiresssl':None,
29 'krbsrvname':None,
30 'gsslib':None,
31 }
32 # configure the URI string with the dictionary
33 uri = QgsDataSourceUri()
34 for param in uri_config:
35     if uri_config[param] != None:
36         uri.setParam(param, uri_config[param]) # add parameters to the URI
37
38 # the raster can now be loaded into the project using the URI string and GDAL data
↳provider
39 rlayer = iface.addRasterLayer('PG: ' + uri.uri(False), "raster layer name", "gdal")

```

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

```

layer_name = 'nurc:mosaic'
uri = "https://demo.geo-solutions.it/geoserver/ows?identifiant={}".format(layer_
↳name)
rlayer = QgsRasterLayer(uri, 'my wcs layer', 'wcs')

```

Voici une description des paramètres que l'URI du WCS peut contenir :

L'URI du WCS est composé de paires **clé=valeur** séparées par « & ». C'est le même format que la chaîne de requête dans l'URL, encodée de la même manière. `QgsDataSourceUri` doit être utilisé pour construire l'URI afin de s'assurer que les caractères spéciaux sont encodés correctement.

- **url** (obligatoire) : URL du serveur WCS. Ne pas utiliser **VERSION** dans l'URL, car chaque version de WCS utilise un nom de paramètre différent pour la version **GetCapabilities**, voir la version param.

- **identifier** (obligatoire) : Nom de la couverture
- **time** (facultatif) : position temporelle ou période de temps (beginPosition/endPosition [/timeResolution])
- **format** (facultatif) : Nom du format supporté. La valeur par défaut est le premier format supporté avec tif dans le nom ou le premier format supporté.
- **crs** (facultatif) : CRS sous la forme AUTHORITY :ID, par exemple EPSG :4326. La valeur par défaut est EPSG :4326 si elle est prise en charge ou le premier CRS pris en charge.
- **username** (facultatif) : Nom d'utilisateur pour l'authentification de base.
- **password** (facultatif) : Mot de passe pour l'authentification de base.
- **IgnoreGetMapUrl** (facultatif, hack) : Si spécifié (défini à 1), ignorer l'URL de GetCoverage annoncée par GetCapabilities. Peut être nécessaire si un serveur n'est pas configuré correctement.
- **InvertAxisOrientation** (optionnel, hack) : Si spécifié (défini à 1), changer d'axe dans la demande GetCoverage. Peut être nécessaire pour les CRS géographiques si un serveur utilise un mauvais ordre d'axes.
- **IgnoreAxisOrientation** (optionnel, hack) : Si spécifié (défini à 1), n'inversez pas l'orientation des axes selon la norme WCS pour les CRS géographiques.
- **cache** (facultatif) : contrôle de la charge du cache, comme décrit dans QNetworkRequest::CacheLoadControl, mais la requête est renvoyée en tant que PreferCache si elle a échoué avec AlwaysCache. Valeurs autorisées : AlwaysCache, PreferCache, PreferNetwork, AlwaysNetwork. La valeur par défaut est 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 = "crs=EPSG:4326&format=image/png&layers=tasmania&styles&url=https://
↳demo.geo-solutions.it/geoserver/ows"
rlayer = QgsRasterLayer(urlWithParams, 'some layer name', 'wms')
if not rlayer.isValid():
    print("Layer failed to load!")
```

### 3.3 QgsProject instance

Si vous souhaitez utiliser les couches ouvertes pour le rendu, n'oubliez pas de les ajouter à l'instance `QgsProject`. L'instance `QgsProject` prend la propriété des couches et celles-ci peuvent être accessibles ultérieurement depuis n'importe quelle partie de l'application par leur identifiant unique. Lorsque la couche est retirée du projet, elle est également supprimée. Les couches peuvent être supprimées par l'utilisateur dans l'interface QGIS, ou via Python en utilisant la méthode `removeMapLayer()`.

L'ajout d'une couche au projet actuel se fait à l'aide de la méthode `addMapLayer()` :

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

Pour ajouter une couche à une position absolue :

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

Si vous voulez supprimer la couche, utilisez la méthode `removeMapLayer()` :

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

Dans le code ci-dessus, l'identifiant de la couche est passé (vous pouvez l'obtenir en appelant la méthode `id()` de la couche), mais vous pouvez aussi passer l'objet de la couche lui-même.

Pour obtenir une liste des couches chargées et de leurs identifiants, utilisez la méthode `mapLayers()` :

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

Les extraits de code sur cette page nécessitent les importations suivantes si vous êtes en dehors de la console pyqgis :

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



---

## Accéder à la table des matières (TOC)

---

- *The QgsProject class*
- *QgsLayerTreeGroup class*

Vous pouvez utiliser différentes classes pour accéder à toutes les couches chargées dans la table des matières et les utiliser pour récupérer des informations :

- `QgsProject`
- `QgsLayerTreeGroup`

### 4.1 The QgsProject class

Vous pouvez utiliser `QgsProject` pour récupérer des informations sur la table des matières et toutes les couches chargées.

Vous devez créer une instance de `QgsProject` et utiliser ses méthodes pour obtenir les couches chargées.

La principale méthode est `mapLayers()`. Elle retournera un dictionnaire des couches chargées :

```
layers = QgsProject.instance().mapLayers()
print(layers)
```

```
{'countries_89ae1b0f_f41b_4f42_bca4_caf55ddbe4b6': <QgsMapLayer: 'countries' (ogr)>
↔}
```

Les clés du dictionnaire sont les ids uniques des couches, tandis que les valeurs sont les objets correspondants.

Il est désormais facile d'obtenir toute autre information sur les couches :

```
1 # list of layer names using list comprehension
2 l = [layer.name() for layer in QgsProject.instance().mapLayers().values()]
3 # dictionary with key = layer name and value = layer object
4 layers_list = {}
5 for l in QgsProject.instance().mapLayers().values():
6     layers_list[l.name()] = l
7
8 print(layers_list)
```

```
{'countries': <QgsMapLayer: 'countries' (ogr)>}
```

Vous pouvez également interroger la TOC en utilisant le nom de la couche :

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

---

**Note :** Une liste avec toutes les couches correspondantes est retournée, donc nous indexons avec [0] pour obtenir la première couche avec ce nom.

---

## 4.2 QgsLayerTreeGroup class

L'arbre à couches est une structure arborescente classique constituée de nœuds. Il existe actuellement deux types de nœuds : les nœuds de groupe (`QgsLayerTreeGroup`) et les nœuds de couche (`QgsLayerTreeLayer`).

---

**Note :** Pour plus d'informations, vous pouvez lire ces messages de Martin Dobias sur son blog : [Part 1](#) [Part 2](#) [Part 3](#)

---

On peut accéder facilement à l'arbre des couches du projet avec la méthode `layerTreeRoot()` de la classe `QgsProject` :

```
root = QgsProject.instance().layerTreeRoot()
```

`root` est un nœud de groupe et a des *enfants* :

```
root.children()
```

Une liste des enfants directs est renvoyée. Les enfants du sous-groupe doivent être consultés par leur propre parent direct.

Nous pouvons récupérer un des enfants :

```
child0 = root.children()[0]
print(child0)
```

```
<qgis._core.QgsLayerTreeLayer object at 0x7f1e1ea54168>
```

Les couches peuvent également être récupérées en utilisant leur `id` (unique) :

```
ids = root.findLayerIds()
# access the first layer of the ids list
root.findLayer(ids[0])
```

Et les groupes peuvent également être recherchés en utilisant leurs noms :

```
root.findGroup('Group Name')
```

`QgsLayerTreeGroup` a de nombreuses autres méthodes utiles qui peuvent être utilisées pour obtenir plus d'informations sur la TOC :

```
# list of all the checked layers in the TOC
checked_layers = root.checkedLayers()
print(checked_layers)
```

```
[<QgsMapLayer: 'countries' (ogr)>]
```

Maintenant, ajoutons quelques couches à l'arbre des couches du projet. Il y a deux façons de le faire :



### 1. Ajout explicite en utilisant les fonctions `addLayer()` ou `insertLayer()` :

```

1 # create a temporary layer
2 layer1 = QgsVectorLayer("path_to_layer", "Layer 1", "memory")
3 # add the layer to the legend, last position
4 root.addLayer(layer1)
5 # add the layer at given position
6 root.insertLayer(5, layer1)

```

### 2. Ajout implicite : puisque l'arbre des couches du projet est connecté au registre des couches, il suffit d'ajouter une couche au registre des couches de la carte :

```
QgsProject.instance().addMapLayer(layer1)
```

Vous pouvez facilement passer de `QgsVectorLayer` à `QgsLayerTreeLayer` :

```

node_layer = root.findLayer(country_layer.id())
print("Layer node:", node_layer)
print("Map layer:", node_layer.layer())

```

```

Layer node: <qgis_core.QgsLayerTreeLayer object at 0x7fecceb46ca8>
Map layer: <QgsMapLayer: 'countries' (ogr)>

```

Les groupes peuvent être ajoutés avec la méthode `addGroup()`. Dans l'exemple ci-dessous, le premier ajoutera un groupe à la fin de la table des matières, tandis que pour le second, vous pouvez ajouter un autre groupe à l'intérieur d'un groupe existant :

```

node_group1 = root.addGroup('Simple Group')
# add a sub-group to Simple Group
node_subgroup1 = node_group1.addGroup("I'm a sub group")

```

Pour déplacer les nœuds et les groupes, il existe de nombreuses méthodes utiles.

Le déplacement d'un nœud existant se fait en trois étapes :

1. le clonage du nœud existant
2. le déplacement du nœud cloné vers la position souhaitée
3. en supprimant le nœud d'origine

```

1 # clone the group
2 cloned_group1 = node_group1.clone()
3 # move the node (along with sub-groups and layers) to the top
4 root.insertChildNode(0, cloned_group1)
5 # remove the original node
6 root.removeChildNode(node_group1)

```

Il est un peu plus *compliqué* de déplacer une couche dans la légende :

```

1 # get a QgsVectorLayer
2 vl = QgsProject.instance().mapLayersByName("countries")[0]
3 # create a QgsLayerTreeLayer object from vl by its id
4 myvl = root.findLayer(vl.id())
5 # clone the myvl QgsLayerTreeLayer object
6 myvlclone = myvl.clone()
7 # get the parent. If None (layer is not in group) returns ''
8 parent = myvl.parent()
9 # move the cloned layer to the top (0)
10 parent.insertChildNode(0, myvlclone)
11 # remove the original myvl
12 root.removeChildNode(myvl)

```

ou le déplacer vers un groupe existant :

```
1 # get a QgsVectorLayer
2 vl = QgsProject.instance().mapLayersByName("countries")[0]
3 # create a QgsLayerTreeLayer object from vl by its id
4 myvl = root.findLayer(vl.id())
5 # clone the myvl QgsLayerTreeLayer object
6 myvlclone = myvl.clone()
7 # create a new group
8 group1 = root.addGroup("Group1")
9 # get the parent. If None (layer is not in group) returns ''
10 parent = myvl.parent()
11 # move the cloned layer to the top (0)
12 group1.insertChildNode(0, myvlclone)
13 # remove the QgsLayerTreeLayer from its parent
14 parent.removeChildNode(myvl)
```

Quelques autres méthodes qui peuvent être utilisées pour modifier les groupes et les couches :

```
1 node_group1 = root.findGroup("Group1")
2 # change the name of the group
3 node_group1.setName("Group X")
4 node_layer2 = root.findLayer(country_layer.id())
5 # change the name of the layer
6 node_layer2.setName("Layer X")
7 # change the visibility of a layer
8 node_group1.setItemVisibilityChecked(True)
9 node_layer2.setItemVisibilityChecked(False)
10 # expand/collapse the group view
11 node_group1.setExpanded(True)
12 node_group1.setExpanded(False)
```

Les extraits de code sur cette page nécessitent les importations suivantes si vous êtes en dehors de la console pyqgis :

```
1 from qgis.core import (
2     QgsRasterLayer,
3     QgsProject,
4     QgsPointXY,
5     QgsRaster,
6     QgsRasterShader,
7     QgsColorRampShader,
8     QgsSingleBandPseudoColorRenderer,
9     QgsSingleBandColorDataRenderer,
10    QgsSingleBandGrayRenderer,
11 )
12
13 from qgis.PyQt.QtGui import (
14     QColor,
15 )
```

---

## Utiliser des couches raster

---

### 5.1 Détails d'une couche

Une couche raster est constituée d'une ou de plusieurs bandes rasters — appelées raster à bande unique et à bandes multiples. Une bande représente un raster de valeurs. Une image couleur (par exemple une photo aérienne) est un raster composé de bandes rouges, bleues et vertes. Les rasters à bande unique représentent généralement soit des variables continues (par exemple l'altitude), soit des variables discrètes (par exemple l'utilisation des terres). Dans certains cas, une couche raster est accompagnée d'une palette et les valeurs rasters font référence aux couleurs stockées dans la palette.

Le code suivant suppose que `rlayer` est un objet `QgsRasterLayer`.

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

```
919 619
```

```
# get the extent of the layer as QgsRectangle
print(rlayer.extent())
```

```
<QgsRectangle: 20.06856808199999875 -34.27001076999999896, 20.83945284300000012 -
↪33.75077500700000144>
```

```
# get the extent of the layer as Strings
print(rlayer.extent().toString())
```

```
20.0685680819999988,-34.2700107699999990 : 20.8394528430000001,-33.7507750070000014
```

```
# get the raster type: 0 = GrayOrUndefined (single band), 1 = Palette (single
↪band), 2 = Multiband
print(rlayer.rasterType())
```

```
0
```

```
# get the total band count of the raster
print(rlayer.bandCount())
```

```
1
```

```
# get all the available metadata as a QgsLayerMetadata object
print(rlayer.metadata())
```

```
<qgis._core.QgsLayerMetadata object at 0x13711d558>
```

## 5.2 Moteur de rendu

Lorsqu'une couche raster est chargée, elle obtient un rendu par défaut en fonction de son type. Il peut être modifié soit dans les propriétés de la couche, soit par programmation.

Pour interroger le moteur de rendu actuel :

```
print(rlayer.renderer())
```

```
<qgis._core.QgsSingleBandGrayRenderer object at 0x7f471c1da8a0>
```

```
print(rlayer.renderer().type())
```

```
singlebandgray
```

Pour définir un moteur de rendu, utilisez la méthode `setRenderer` de `QgsRasterLayer`. Il existe un certain nombre de classes de rendu (dérivées de la méthode `QgsRasterRenderer`) :

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

Les couches raster à bande unique peuvent être dessinées soit en gris (valeurs basses = noir, valeurs hautes = blanc), soit avec un algorithme de pseudo-couleurs qui attribue des couleurs aux valeurs. Les raster à bande unique avec une palette peuvent également être dessinées à l'aide de la palette. Les couches multibandes sont généralement dessinées en faisant correspondre les bandes à des couleurs RVB. Une autre possibilité est d'utiliser une seule bande pour le dessin .

### 5.2.1 Rasters mono-bande

Disons que nous voulons un rendu d'une seule bande de la couche raster avec des couleurs allant du vert au jaune (correspondant à des valeurs de pixels de 0 à 255). Dans un premier temps, nous allons préparer un objet `QgsRasterShader` et configurer sa fonction de shader :

```
1 fcn = QgsColorRampShader()
2 fcn.setColorRampType(QgsColorRampShader.Interpolated)
3 lst = [ QgsColorRampShader.ColorRampItem(0, QColor(0,255,0)),
4         QgsColorRampShader.ColorRampItem(255, QColor(255,255,0)) ]
5 fcn.setColorRampItemList(lst)
6 shader = QgsRasterShader()
7 shader.setRasterShaderFunction(fcn)
```

Le shader cartographie les couleurs comme le précise sa carte de couleurs. La carte des couleurs est fournie sous la forme d'une liste de valeurs de pixels avec les couleurs associées. Il existe trois modes d'interpolation :

- linéaire (`Interpolated`) : la couleur est interpolée linéairement à partir des entrées de la carte des couleurs au-dessus et au-dessous de la valeur du pixel
- discrete (`Discrete`) : la couleur est tirée de l'entrée de carte de couleur la plus proche ayant une valeur égale ou supérieure
- exact (`Exact`) : la couleur n'est pas interpolée, seuls les pixels dont la valeur est égale aux entrées de la carte de couleur seront dessinés

Dans la deuxième étape, nous associerons ce shader à la couche raster :

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

Le chiffre 1 dans le code ci-dessus est le numéro de la bande (les bandes rasters sont indexées à partir de 1).

Enfin, nous devons utiliser la méthode `triggerRepaint` pour voir les résultats :

```
rlayer.triggerRepaint()
```

## 5.2.2 Rasters multi-bandes

Par défaut, QGIS mappe les trois premières bandes en rouge, vert et bleu pour créer une image en couleur (c'est le style de dessin `MultiBandColor`). Dans certains cas, vous pouvez vouloir passer outre ces paramètres. Le code suivant échange la bande rouge (1) et la bande verte (2) :

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

Dans le cas où une seule bande est nécessaire pour la visualisation du raster, le dessin d'une seule bande peut être choisi, soit en niveaux de gris, soit en pseudo-couleur.

Nous devons utiliser `triggerRepaint` pour mettre à jour la carte et voir le résultat :

```
rlayer_multi.triggerRepaint()
```

## 5.3 Interrogation des données

Les valeurs raster peuvent être interrogées en utilisant la méthode `sample` de la classe `QgsRasterDataProvider`. Vous devez spécifier une classe `QgsPointXY` et le numéro de bande de la couche raster que vous voulez interroger. La méthode retourne un tuple avec la valeur et `True` ou `False` selon les résultats :

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

Une autre méthode pour interroger les valeurs rasters consiste à utiliser la méthode `identify` qui renvoie un objet `QgsRasterIdentifyResult`.

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

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

```
{1: 323.0}
```

Dans ce cas, la méthode `results` renvoie un dictionnaire, avec les index de bande comme clés, et les valeurs de bande comme valeurs. Par exemple, quelque chose comme `{1 : 323.0}`

Les extraits de code sur cette page nécessitent les importations suivantes si vous êtes en dehors de la console `pyqgis` :

```
1 from qgis.core import (  
2     QgsApplication,  
3     QgsDataSourceUri,  
4     QgsCategorizedSymbolRenderer,  
5     QgsClassificationRange,  
6     QgsPointXY,  
7     QgsProject,  
8     QgsExpression,  
9     QgsField,  
10    QgsFields,  
11    QgsFeature,  
12    QgsFeatureRequest,  
13    QgsFeatureRenderer,  
14    QgsGeometry,  
15    QgsGraduatedSymbolRenderer,  
16    QgsMarkerSymbol,  
17    QgsMessageLog,  
18    QgsRectangle,  
19    QgsRendererCategory,  
20    QgsRendererRange,  
21    QgsSymbol,  
22    QgsVectorDataProvider,  
23    QgsVectorLayer,  
24    QgsVectorFileWriter,  
25    QgsWkbTypes,  
26    QgsSpatialIndex,  
27 )  
28  
29 from qgis.core.additions.edit import edit  
30  
31 from qgis.PyQt.QtGui import (  
32     QColor,  
33 )
```

---

## 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
- Création de couches vecteur
  - A partir d'une instance de `QgsVectorFileWriter`
  - Directement à partir des entités
  - Depuis une instance de `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.

La plupart des exemples de cette section sont basés sur des méthodes de la classe `QgsVectorLayer`.

## 6.1 Récupérer les informations relatives aux attributs

Vous pouvez récupérer les informations associées aux champs d'une couche vecteur en appelant la méthode `fields()` d'un objet `QgsVectorLayer` :

```
vlayer = QgsVectorLayer("testdata/airports.shp", "airports", "ogr")
for field in vlayer.fields():
    print(field.name(), field.typeName())
```

```
1 ID Integer64
2 fk_region Integer64
3 ELEV Real
4 NAME String
5 USE String
```

Les méthodes `displayField()` et `mapTipTemplate()` de la classe `QgsVectorLayer` fournissent des informations sur le champ et le modèle utilisés dans l'onglet maptips.

Lorsque vous chargez une couche vecteur, un champ est toujours choisi par QGIS comme « Nom d'affichage », alors que le « maptip HTML » est vide par défaut. Avec ces méthodes, vous pouvez facilement obtenir les deux :

```
vlayer = QgsVectorLayer("testdata/airports.shp", "airports", "ogr")
print(vlayer.displayField())
```

```
NAME
```

**Note :** Si vous changez le nom d'affichage d'un champ en une expression, vous devez utiliser `displayExpression()` au lieu de `displayField()`.

## 6.2 Itérer sur une couche vecteur

Parcourir les enregistrements d'une couche vecteur est l'une des tâches les plus basiques. L'exemple de code ci-dessous vous montre comment le faire pour montrer quelques informations de chaque enregistrement. Ici, la variable `layer` doit être une instance de l'objet `QgsVectorLayer`.

```
1 # "layer" is a QgsVectorLayer instance
2 layer = iface.activeLayer()
3 features = layer.getFeatures()
4
5 for feature in features:
6     # retrieve every feature with its geometry and attributes
7     print("Feature ID: ", feature.id())
8     # fetch geometry
9     # show some information about the feature geometry
10    geom = feature.geometry()
11    geomSingleType = QgsWkbTypes.isSingleType(geom.wkbType())
12    if geom.type() == QgsWkbTypes.PointGeometry:
13        # the geometry type can be of single or multi type
14        if geomSingleType:
15            x = geom.asPoint()
16            print("Point: ", x)
17        else:
18            x = geom.asMultiPoint()
19            print("MultiPoint: ", x)
20    elif geom.type() == QgsWkbTypes.LineGeometry:
21        if geomSingleType:
```

(suite sur la page suivante)



(suite de la page précédente)

```

22     x = geom.asPolyline()
23     print("Line: ", x, "length: ", geom.length())
24     else:
25         x = geom.asMultiPolyline()
26         print("MultiLine: ", x, "length: ", geom.length())
27     elif geom.type() == QgsWkbTypes.PolygonGeometry:
28         if geomSingleType:
29             x = geom.asPolygon()
30             print("Polygon: ", x, "Area: ", geom.area())
31         else:
32             x = geom.asMultiPolygon()
33             print("MultiPolygon: ", x, "Area: ", geom.area())
34     else:
35         print("Unknown or invalid geometry")
36     # fetch attributes
37     attrs = feature.attributes()
38     # attrs is a list. It contains all the attribute values of this feature
39     print(attrs)
40     # for this test only print the first feature
41     break

```

```

Feature ID: 1
Point: <QgsPointXY: POINT(7 45)>
[1, 'First feature']

```

## 6.3 Sélection des entités

Dans QGIS Desktop, vous pouvez sélectionner des enregistrements de différentes façons : vous pouvez cliquer sur un objet, dessiner un rectangle sur la carte ou encore filtrer une couche avec des expressions. Les objets sélectionnés apparaissent normalement en surbrillance dans une couleur différente (le jaune par défaut) pour permettre à l'utilisateur de les distinguer.

Parfois, il peut être utile de sélectionner des entités à l'aide de code ou de changer la couleur de sélection par défaut. pour sélectionner toutes les entités d'une couche vecteur, vous pouvez utiliser la méthode `selectAll()`

```

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

```

Pour sélectionner des entités à l'aide d'une expression, utilisez la méthode `selectByExpression()`

```

# 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)

```

Pour changer la couleur par défaut des objets sélectionnés, vous pouvez utiliser la méthode `setSelectionColor()` de la classe `QgsMapCanvas`, comme montré dans l'exemple suivant :

```

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

```

Pour ajouter des entités à celles déjà sélectionnées vous pouvez appeler la méthode `select()` en lui passant une liste d'ID d'entités :

```

1 selected_fid = []
2

```

(suite sur la page suivante)

(suite de la page précédente)

```
3 # Get the first feature id from the layer
4 for feature in layer.getFeatures():
5     selected_fid.append(feature.id())
6     break
7
8 # Add these features to the selected list
9 layer.select(selected_fid)
```

Pour vider votre sélection :

```
layer.removeSelection()
```

### 6.3.1 Accès aux attributs

Les attributs peuvent être invoqués par leur nom :

```
print(feature['name'])
```

```
First feature
```

D'une autre façon, les attributs peuvent être invoqués par le ID. Cette méthode est légèrement plus rapide que d'utiliser le nom. Par exemple, pour récupérer le second attribut :

```
print(feature[1])
```

```
First feature
```

### 6.3.2 Itérer sur une sélection d'entités

Si vous avez besoin uniquement de sélectionner des entités, vous pouvez utiliser la méthode `selectedFeatures()` depuis une couche vecteur :

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

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

Si vous voulez parcourir un sous-ensemble d'enregistrements d'une couche, par exemple sur une zone donnée, vous devez ajouter une classe `QgsFeatureRequest` à l'appel de la méthode `getFeatures()`. Par exemple :

```
1 areaOfInterest = QgsRectangle(450290,400520, 450750,400780)
2
3 request = QgsFeatureRequest().setFilterRect(areaOfInterest)
4
5 for feature in layer.getFeatures(request):
6     # do whatever you need with the feature
7     pass
```

Pour des questions de rapidité, la recherche d'intersection est souvent réalisée à partir du rectangle d'encombrement minimum des objets. Il existe toutefois l'option `ExactIntersect` qui permet de s'assurer que seul les objets intersectés soit renvoyés.

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

Avec la méthode `setLimit()`, vous pouvez limiter le nombre d'entités sélectionnées. Par exemple :

```
request = QgsFeatureRequest()
request.setLimit(2)
for feature in layer.getFeatures(request):
    print(feature)
```

```
<qgis._core.QgsFeature object at 0x7f9b78590948>
```

Si vous avez besoin d'un filtre basé sur des attributs à la place (ou en plus) d'un filtre spatial, comme montré dans les exemples précédents, vous pouvez construire un objet `QgsExpression` et le passer à la classe constructeur `QgsFeatureRequest` :

```
# 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)
```

Vous pouvez vous référer à la section [Expressions, Filtrage et Calcul de valeurs](#) pour plus de détails sur la syntaxe employée dans `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.

```
1 # Only return selected fields to increase the "speed" of the request
2 request.setSubsetOfAttributes([0,2])
3
4 # More user friendly version
5 request.setSubsetOfAttributes(['name','id'],layer.fields())
6
7 # Don't return geometry objects to increase the "speed" of the request
8 request.setFlags(QgsFeatureRequest.NoGeometry)
9
10 # Fetch only the feature with id 45
11 request.setFilterFid(45)
12
13 # The options may be chained
14 request.setFilterRect(areaOfInterest).setFlags(QgsFeatureRequest.NoGeometry).
    ↳setFilterFid(45).setSubsetOfAttributes([0,2])
```

## 6.4 Modifier des couches vecteur

La plupart des sources vectorielles supportent les fonctions d'édition. Parfois, ces possibilités sont limitées. Vous pouvez utiliser la fonction `capabilities()` pour voir quelles fonctionnalités sont supportées.

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

```
The layer supports DeleteFeatures
```

Pour une liste complète des possibilités, merci de se référer à la documentation de l'API `QgsVectorDataProvider`.

Pour afficher les possibilités d'édition d'une couche dans une liste séparé par des virgules, vous pouvez utiliser la méthode `capabilitiesString()` comme montré dans l'exemple suivant :

```
1 caps_string = layer.dataProvider().capabilitiesString()
2 # Print:
3 # 'Add Features, Delete Features, Change Attribute Values, Add Attributes,
4 # Delete Attributes, Rename Attributes, Fast Access to Features at ID,
5 # Presimplify Geometries, Presimplify Geometries with Validity Check,
6 # 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

```
1 # If caching is enabled, a simple canvas refresh might not be sufficient
2 # to trigger a redraw and you must clear the cached image for the layer
3 if iface.mapCanvas().isCachingEnabled():
4     layer.triggerRepaint()
5 else:
6     iface.mapCanvas().refresh()
```

### 6.4.1 Ajout d'Entités

Créez quelques instances de `QgsFeature` et passez une liste de celles-ci à la méthode `addFeatures()` du provider. Elle retournera deux valeurs : le résultat (`true/false`) et la liste des entites ajoutées (leur ID est défini par le data store).

Pour configurer les attributs d'entite, vous pouvez soit initialiser la fonctionnalité en passant un objet `QgsFields` (vous pouvez obtenir cela par la méthode `fields()` de la couche vecteur) ou appeler `initAttributes()` en passant le nombre de champs que vous voulez ajouter.

```
1 if caps & QgsVectorDataProvider.AddFeatures:
2     feat = QgsFeature(layer.fields())
3     feat.setAttributes([0, 'hello'])
4     # Or set a single attribute by key or by index:
5     feat.setAttribute('name', 'hello')
6     feat.setAttribute(0, 'hello')
7     feat.setGeometry(QgsGeometry.fromPointXY(QgsPointXY(123, 456)))
8     (res, outFeats) = layer.dataProvider().addFeatures([feat])
```

### 6.4.2 Suppression d'Entités

Pour supprimer certaines entites, il suffit de fournir une liste de leurs identifiants.

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

### 6.4.3 Modifier des Entités

Il est possible de modifier la géométrie des entités ou de changer certains attributs. L'exemple suivant modifie d'abord les valeurs des attributs avec les index 0 et 1, puis la géométrie de l'entité.

```

1 fid = 100  # ID of the feature we will modify
2
3 if caps & QgsVectorDataProvider.ChangeAttributeValues:
4     attrs = { 0 : "hello", 1 : 123 }
5     layer.dataProvider().changeAttributeValues({ fid : attrs })
6
7 if caps & QgsVectorDataProvider.ChangeGeometries:
8     geom = QgsGeometry.fromPointXY(QgsPointXY(111,222))
9     layer.dataProvider().changeGeometryValues({ fid : geom })

```

---

#### Astuce : Favoriser la classe `QgsVectorLayerEditUtils` pour les modifications de géométrie uniquement

Si vous avez seulement besoin de modifier des géométries, vous pouvez utiliser la classe `QgsVectorLayerEditUtils` qui fournit quelques méthodes utiles pour modifier des géométries (traduire, insérer ou déplacer un sommet, etc.).

---

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

Lorsque vous éditez des vecteurs dans l'application QGIS, vous devez d'abord lancer le mode d'édition pour une couche particulière, puis faire quelques modifications et enfin valider (ou annuler) les changements. Toutes les modifications que vous faites ne sont pas écrites avant que vous les validiez — elles restent dans le tampon d'édition en mémoire de la couche. Il est possible d'utiliser cette fonctionnalité également par programmation — c'est juste une autre méthode d'édition des couches vecteur qui complète l'utilisation directe des fournisseurs de données. Utilisez cette option lorsque vous fournissez des outils d'interface graphique pour l'édition de la couche vecteur, car cela permet à l'utilisateur de décider s'il veut valider/rétablir et d'utiliser les fonctions annuler/rétablir. Lorsque les modifications sont validées, toutes les modifications du tampon d'édition sont enregistrées dans le fournisseur de données.

Les méthodes sont similaires à celles que nous avons vues dans le fournisseur, mais elles sont appelées sur l'objet `QgsVectorLayer` à la place.

Pour que ces méthodes fonctionnent, la couche doit être en mode édition. Pour lancer le mode d'édition, utilisez la méthode `startEditing()`. Pour arrêter l'édition, utilisez les méthodes `commitChanges()` ou `rollback()`. La première va valider tous vos changements à la source de données, tandis que la seconde va les rejeter et ne modifiera pas du tout la source de données.

Pour savoir si une couche est en mode édition, utilisez la méthode `isEditable()`.

Vous trouverez ici quelques exemples qui montrent comment utiliser ces méthodes de mise à jour.

```

1 from qgis.PyQt.QtCore import QVariant
2
3 feat1 = feat2 = QgsFeature(layer.fields())
4 fid = 99
5 feat1.setId(fid)
6
7 # add two features (QgsFeature instances)
8 layer.addFeatures([feat1, feat2])
9 # delete a feature with specified ID
10 layer.deleteFeature(fid)
11
12 # set new geometry (QgsGeometry instance) for a feature
13 geometry = QgsGeometry.fromWkt("POINT(7 45)")
14 layer.changeGeometry(fid, geometry)

```

(suite sur la page suivante)

(suite de la page précédente)

```

15 # update an attribute with given field index (int) to a given value
16 fieldIndex = 1
17 value = 'My new name'
18 layer.changeAttributeValue(fid, fieldIndex, value)
19
20 # add new field
21 layer.addAttribute(QgsField("mytext", QVariant.String))
22 # remove a field
23 layer.deleteAttribute(fieldIndex)

```

Pour que l'annulation/rétablissement fonctionne correctement, les appels mentionnés ci-dessus doivent être enveloppés dans des commandes d'annulation. (Si vous ne vous souciez pas de undo/redo et que vous voulez que les modifications soient enregistrées immédiatement, alors vous aurez un travail plus facile en utilisant *editing with data provider*.)

Voici comment vous pouvez utiliser la fonction d'annulation :

```

1 layer.beginEditCommand("Feature triangulation")
2
3 # ... call layer's editing methods ...
4
5 if problem_occurred:
6     layer.destroyEditCommand()
7     # ... tell the user that there was a problem
8     # and return
9
10 # ... more editing ...
11
12 layer.endEditCommand()

```

La méthode `beginEditCommand()` créera une commande interne « active » et enregistrera les changements ultérieurs dans la couche vecteur. Avec l'appel à `endEditCommand()` la commande est poussée sur la pile d'annulation et l'utilisateur pourra annuler/refaire depuis l'interface graphique. Au cas où quelque chose se serait mal passé lors des modifications, la méthode `destroyEditCommand()` supprimera la commande et annulera toutes les modifications effectuées alors que cette commande était 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)

```

Cela appellera automatiquement `commitChanges()` à la fin. Si une exception se produit, il appellera `rollback()` tous les changements. Si un problème est rencontré dans `commitChanges()` (lorsque la méthode retourne `False`), une exception `QgsEditError` sera levée.

## 6.4.5 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.

```

1 from qgis.PyQt.QtCore import QVariant
2
3 if caps & QgsVectorDataProvider.AddAttributes:
4     res = layer.dataProvider().addAttributes(
5         [QgsField("mytext", QVariant.String),
6          QgsField("myint", QVariant.Int)])
7

```

(suite sur la page suivante)

(suite de la page précédente)

```

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

1 # Alternate methods for removing fields
2 # first create temporary fields to be removed (f1-3)
3 layer.dataProvider().addAttributes([QgsField("f1",QVariant.Int),QgsField("f2",
4     ↪QVariant.Int),QgsField("f3",QVariant.Int)])
5 layer.updateFields()
6 count=layer.fields().count() # count of layer fields
7 ind_list=list((count-3, count-2)) # create list
8 # remove a single field with an index
9 layer.dataProvider().deleteAttributes([count-1])
10
11 # remove multiple fields with a list of indices
12 layer.dataProvider().deleteAttributes(ind_list)

```

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 : Sauvegarder directement les modifications en utilisant « with » based command

En utilisant `with edit(layer)` : les modifications seront automatiquement validées en appelant `commitChanges()` à la fin. Si une exception se produit, il fera `rollback()` tous les changements. Voir *Modifier des couches vecteur à l'aide d'un tampon d'édition*.

---

## 6.5 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 :

- créer un index spatial en utilisant la classe `QgsSpatialIndex()` :

```
index = QgsSpatialIndex()
```

- ajouter un index aux entités — index takes `QgsFeature` et l'ajoute à la structure interne des données. Vous pouvez créer l'objet manuellement ou utiliser celui d'un appel précédent à la méthode du provider `getFeatures()`

```
index.addFeature(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 :

```
1 # returns array of feature IDs of five nearest features
2 nearest = index.nearestNeighbor(QgsPointXY(25.4, 12.7), 5)
3
4 # returns array of IDs of features which intersect the rectangle
5 intersect = index.intersects(QgsRectangle(22.5, 15.3, 23.1, 17.2))
```

## 6.6 Création de couches vecteur

Il existe plusieurs façons de générer un jeu de données de couche vecteur :

- la classe `QgsVectorFileWriter` class : Une classe pratique pour écrire des fichiers vecteur sur le disque, en utilisant soit un appel statique à `writeAsVectorFormat()` qui sauvegarde toute la couche vecteur, soit en créant une instance de la classe et en lançant des appels à `addFeature()`. Cette classe supporte tous les formats vecteur qu'OGR supporte (GeoPackage, Shapefile, GeoJSON, KML et autres).
- la classe `QgsVectorLayer` : instancie un provider de données qui interprète le chemin d'accès (url) fourni de la source de données pour se connecter et accéder aux données. Elle peut être utilisée pour créer des couches temporaires en mémoire (memory) et se connecter à des ensembles de données OGR (ogr), des bases de données (postgres, spatialite, mysql, mssql) et plus encore (wfs, gpx, delimitedtext...).

### 6.6.1 A partir d'une instance de `QgsVectorFileWriter`

```
1 # SaveVectorOptions contains many settings for the writer process
2 save_options = QgsVectorFileWriter.SaveVectorOptions()
3 transform_context = QgsProject.instance().transformContext()
4 # Write to a GeoPackage (default)
5 error = QgsVectorFileWriter.writeAsVectorFormatV2(layer,
6                                                    "testdata/my_new_file.gpkg",
7                                                    transform_context,
8                                                    save_options)
9 if error[0] == QgsVectorFileWriter.NoError:
10     print("success!")
11 else:
12     print(error)
```

```
1 # Write to an ESRI Shapefile format dataset using UTF-8 text encoding
2 save_options = QgsVectorFileWriter.SaveVectorOptions()
3 save_options.driverName = "ESRI Shapefile"
4 save_options.fileEncoding = "UTF-8"
5 transform_context = QgsProject.instance().transformContext()
6 error = QgsVectorFileWriter.writeAsVectorFormatV2(layer,
7                                                    "testdata/my_new_shapefile",
8                                                    transform_context,
9                                                    save_options)
10 if error[0] == QgsVectorFileWriter.NoError:
11     print("success again!")
12 else:
13     print(error)
```

```
1 # Write to an ESRI GDB file
2 save_options = QgsVectorFileWriter.SaveVectorOptions()
3 save_options.driverName = "FileGDB"
4 # if no geometry
5 save_options.overrideGeometryType = QgsWkbTypes.Unknown
6 save_options.actionOnExistingFile = QgsVectorFileWriter.CreateOrOverwriteLayer
```

(suite sur la page suivante)



(suite de la page précédente)

```

7 save_options.layerName = 'my_new_layer_name'
8 transform_context = QgsProject.instance().transformContext()
9 gdb_path = "testdata/my_example.gdb"
10 error = QgsVectorFileWriter.writeAsVectorFormatV2(layer,
11                                             gdb_path,
12                                             transform_context,
13                                             save_options)
14 if error[0] == QgsVectorFileWriter.NoError:
15     print("success!")
16 else:
17     print(error)

```

Vous pouvez également convertir des champs pour les rendre compatibles avec différents formats en utilisant la classe `FieldValueConverter`. Par exemple, pour convertir des types de variables de tableau (par exemple dans Postgres) en un type de texte, vous pouvez faire ce qui suit

```

1 LIST_FIELD_NAME = 'xxxx'
2
3 class ESRIValueConverter(QgsVectorFileWriter.FieldValueConverter):
4
5     def __init__(self, layer, list_field):
6         QgsVectorFileWriter.FieldValueConverter.__init__(self)
7         self.layer = layer
8         self.list_field_idx = self.layer.fields().indexOfName(list_field)
9
10    def convert(self, fieldIdxInLayer, value):
11        if fieldIdxInLayer == self.list_field_idx:
12            return QgsListFieldFormatter().representValue(layer=vlayer,
13                                                         fieldIndex=self.list_field_idx,
14                                                         config={},
15                                                         cache=None,
16                                                         value=value)
17
18        else:
19            return value
20
21    def fieldDefinition(self, field):
22        idx = self.layer.fields().indexOfName(field.name())
23        if idx == self.list_field_idx:
24            return QgsField(LIST_FIELD_NAME, QVariant.String)
25        else:
26            return self.layer.fields()[idx]
27
28 converter = ESRIValueConverter(vlayer, LIST_FIELD_NAME)
29 opts = QgsVectorFileWriter.SaveVectorOptions()
30 opts.fieldValueConverter = converter

```

Un CRS de destination peut également être spécifié — si une instance valide de `QgsCoordinateReferenceSystem` est passée comme quatrième paramètre, la couche est transformée en ce CRS.

Pour des noms de driver valides, veuillez appeler la méthode `supportedFiltersAndFormats` ou consulter les [formats supportés par OGR](#) — vous devez passer la valeur dans la colonne « Code » comme nom de driver

Vous pouvez choisir d'exporter uniquement certaines entites, de passer des options de création spécifiques au driver ou de demander au writer de ne pas créer d'attributs... Il existe un certain nombre d'autres paramètres (optionnels); voir la documentation `QgsVectorFileWriter` pour plus de détails.

## 6.6.2 Directement à partir des entites

```

1  from qgis.PyQt.QtCore import QVariant
2
3  # define fields for feature attributes. A QgsFields object is needed
4  fields = QgsFields()
5  fields.append(QgsField("first", QVariant.Int))
6  fields.append(QgsField("second", QVariant.String))
7
8  """ create an instance of vector file writer, which will create the vector file.
9  Arguments:
10 1. path to new file (will fail if exists already)
11 2. field map
12 3. geometry type - from WKBTYPe enum
13 4. layer's spatial reference (instance of
14   QgsCoordinateReferenceSystem)
15 5. coordinate transform context
16 6. save options (driver name for the output file, encoding etc.)
17 """
18
19 crs = QgsProject.instance().crs()
20 transform_context = QgsProject.instance().transformContext()
21 save_options = QgsVectorFileWriter.SaveVectorOptions()
22 save_options.driverName = "ESRI Shapefile"
23 save_options.fileEncoding = "UTF-8"
24
25 writer = QgsVectorFileWriter.create(
26     "testdata/my_new_shapefile.shp",
27     fields,
28     QgsWkbTypes.Point,
29     crs,
30     transform_context,
31     save_options
32 )
33
34 if writer.hasError() != QgsVectorFileWriter.NoError:
35     print("Error when creating shapefile: ", writer.errorMessage())
36
37 # add a feature
38 fet = QgsFeature()
39
40 fet.setGeometry(QgsGeometry.fromPointXY(QgsPointXY(10,10)))
41 fet.setAttribute(1, "text")
42 writer.addFeature(fet)
43
44 # delete the writer to flush features to disk
45 del writer

```

## 6.6.3 Depuis une instance de QgsVectorLayer

Parmi tous les provider de données supportés par la classe `QgsVectorLayer`, concentrons-nous sur les couches en memoire. Le provider de memoire est destiné à être utilisé principalement par les développeurs de plugins ou d'applications tierces. Il ne stocke pas de données sur le disque, ce qui permet aux développeurs de l'utiliser comme un backend rapide pour certaines couches temporaires.

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

Le provider de memoire prend également en charge l'indexation spatiale, qui est activée en appelant la fonction `createSpatialIndex()` du provider. Une fois l'index spatial créé, vous pourrez itérer plus rapidement sur des éléments situés dans des régions plus petites (puisque'il n'est pas nécessaire de parcourir toutes les entites, seulement celles qui se trouvent dans un rectangle spécifié).

Un provider mémoire est créé en passant "memory" comme chaîne de provider au constructeur `QgsVectorLayer`.

Le constructeur prend également une URI définissant le type de géométrie de la couche, l'un des : Point », « LineString », « Polygon », « MultiPoint », « MultiLineString », « MultiPolygon » ou « none ».

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

**crs=définition** Spécifie le système de référence de coordonnées, où la définition peut être l'une des formes acceptées par `QgsCoordinateReferenceSystem.createFromString`

**index=yes** Spécifie que le fournisseur utilisera un index spatial

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

L'exemple suivant montre une URI intégrant toutes ces options

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

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

```
1 from qgis.PyQt.QtCore import QVariant
2
3 # create layer
4 vl = QgsVectorLayer("Point", "temporary_points", "memory")
5 pr = vl.dataProvider()
6
7 # add fields
8 pr.addAttributes([QgsField("name", QVariant.String),
9                   QgsField("age",   QVariant.Int),
10                  QgsField("size",  QVariant.Double)])
11 vl.updateFields() # tell the vector layer to fetch changes from the provider
12
13 # add a feature
14 fet = QgsFeature()
15 fet.setGeometry(QgsGeometry.fromPointXY(QgsPointXY(10,10)))
16 fet.setAttributes(["Johny", 2, 0.3])
17 pr.addFeatures([fet])
18
19 # update layer's extent when new features have been added
20 # because change of extent in provider is not propagated to the layer
21 vl.updateExtents()
```

Finalement, vérifions que tout s'est bien déroulé

```
1 # show some stats
2 print("fields:", len(pr.fields()))
3 print("features:", pr.featureCount())
4 e = vl.extent()
5 print("extent:", e.xMinimum(), e.yMinimum(), e.xMaximum(), e.yMaximum())
6
7 # iterate over features
8 features = vl.getFeatures()
9 for fet in features:
10     print("F:", fet.id(), fet.attributes(), fet.geometry().asPoint())
```

```
fields: 3
features: 1
extent: 10.0 10.0 10.0 10.0
F: 1 ['Johny', 2, 0.3] <QgsPointXY: POINT(10 10)>
```

## 6.7 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.

Le rendu pour une couche donnée peut être obtenu comme indiqué ci-dessous :

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

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

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

```
Type: singleSymbol
```

Il y a plusieurs types de moteurs de rendu connus disponibles dans core centrale de QGIS :

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

Il peut aussi y avoir des types de rendus personnalisés, alors ne supposez jamais qu'il n'y a que ces types. Vous pouvez interroger la classe `QgsRendererRegistry` de l'application pour connaître les moteurs de rendu actuellement disponibles :

```
print(QgsApplication.rendererRegistry().renderersList())
```

```
['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 :

```
renderer.dump()
```

```
SINGLE: MARKER SYMBOL (1 layers) color 190,207,80,255
```

### 6.7.1 Moteur de rendu à symbole unique

Vous pouvez obtenir le symbole utilisé pour le rendu en appelant la méthode `symbol()` et le changer avec la méthode `setSymbol()` (note pour les développeurs C++ : le moteur de rendu prend la propriété du symbole.)

Vous pouvez changer le symbole utilisé par une couche vecteur particulière en appelant `setSymbol()` en passant une instance du symbole approprié. Les symboles des couches *point*, *line* et *polygon* peuvent être créés en appelant la fonction `createSimple()` des classes correspondantes `QgsMarkerSymbol`, `QgsLineSymbol` et `QgsFillSymbol`.

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

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

```

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

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

```
print(layer.renderer().symbol().symbolLayers()[0].properties())
```

```

{'angle': '0', 'color': '255,0,0,255', 'horizontal_anchor_point': '1', 'joinstyle
↪': 'bevel', 'name': 'square', 'offset': '0,0', 'offset_map_unit_scale': '3x:0,0,
↪0,0,0,0', 'offset_unit': 'MM', 'outline_color': '35,35,35,255', 'outline_style':
↪'solid', 'outline_width': '0', 'outline_width_map_unit_scale': '3x:0,0,0,0,0,0',
↪'outline_width_unit': 'MM', 'scale_method': 'diameter', 'size': '2', 'size_map_
↪unit_scale': '3x:0,0,0,0,0,0', 'size_unit': 'MM', 'vertical_anchor_point': '1'}

```

Cela peut être utile si vous souhaitez modifier certaines propriétés :

```

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

```

## 6.7.2 Moteur de rendu à symboles catégorisés

Lorsque vous utilisez un moteur de rendu catégorisé, vous pouvez interroger et définir l'attribut qui est utilisé pour la classification : utilisez les méthodes `classAttribute()` et `setClassAttribute()`.

Pour obtenir la liste des catégories

```

1 categorized_renderer = QgsCategorizedSymbolRenderer()
2 # Add a few categories
3 cat1 = QgsRendererCategory('1', QgsMarkerSymbol(), 'category 1')
4 cat2 = QgsRendererCategory('2', QgsMarkerSymbol(), 'category 2')
5 categorized_renderer.addCategory(cat1)
6 categorized_renderer.addCategory(cat2)
7

```

(suite sur la page suivante)

(suite de la page précédente)

```

8 for cat in categorized_renderer.categories():
9     print("{}: {} :: {}".format(cat.value(), cat.label(), cat.symbol()))

```

```

1: category 1 :: <qgis._core.QgsMarkerSymbol object at 0x7f378ffcd9d8>
2: category 2 :: <qgis._core.QgsMarkerSymbol object at 0x7f378ffcd9d8>

```

Où `value()` est la valeur utilisée pour la discrimination entre les catégories, `label()` est un texte utilisé pour la description de la catégorie et `symbol()` la méthode retourne le symbole attribué.

Le moteur de rendu stocke généralement aussi le symbole original et la rampe de couleur qui ont été utilisés pour la classification : `sourceColorRamp()` et `sourceSymbol()`.

### 6.7.3 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 :

```

1 graduated_renderer = QgsGraduatedSymbolRenderer()
2 # Add a few categories
3 graduated_renderer.addClassRange(QgsRendererRange(QgsClassificationRange('class 0-
  ↳100', 0, 100), QgsMarkerSymbol()))
4 graduated_renderer.addClassRange(QgsRendererRange(QgsClassificationRange('class_
  ↳101-200', 101, 200), QgsMarkerSymbol()))
5
6 for ran in graduated_renderer.ranges():
7     print("{} - {}: {} {}".format(
8         ran.lowerValue(),
9         ran.upperValue(),
10        ran.label(),
11        ran.symbol()
12    ))

```

```

0.0 - 100.0: class 0-100 <qgis._core.QgsMarkerSymbol object at 0x7f8bad281b88>
101.0 - 200.0: class 101-200 <qgis._core.QgsMarkerSymbol object at 0x7f8bad281b88>

```

vous pouvez à nouveau utiliser les méthodes `classAttribute` (pour trouver le nom de l'attribut de classification), `sourceSymbol` et `sourceColorRamp`. Il existe également la méthode `mode` qui détermine comment les plages ont été créées : en utilisant des intervalles égaux, des quantiles ou une autre méthode.

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

```

1 from qgis.PyQt import QtGui
2
3 myVectorLayer = QgsVectorLayer("testdata/airports.shp", "airports", "ogr")
4 myTargetField = 'ELEV'
5 myRangeList = []
6 myOpacity = 1
7 # Make our first symbol and range...
8 myMin = 0.0
9 myMax = 50.0
10 myLabel = 'Group 1'
11 myColour = QtGui.QColor('#ffee00')
12 mySymbol1 = QgsSymbol.defaultSymbol(myVectorLayer.geometryType())
13 mySymbol1.setColor(myColour)
14 mySymbol1.setOpacity(myOpacity)

```

(suite sur la page suivante)

(suite de la page précédente)

```

15 myRange1 = QgsRendererRange(myMin, myMax, mySymbol1, myLabel)
16 myRangeList.append(myRange1)
17 #now make another symbol and range...
18 myMin = 50.1
19 myMax = 100
20 myLabel = 'Group 2'
21 myColour = QtGui.QColor('#00eeff')
22 mySymbol2 = QgsSymbol.defaultSymbol(
23     myVectorLayer.geometryType())
24 mySymbol2.setColor(myColour)
25 mySymbol2.setOpacity(myOpacity)
26 myRange2 = QgsRendererRange(myMin, myMax, mySymbol2, myLabel)
27 myRangeList.append(myRange2)
28 myRenderer = QgsGraduatedSymbolRenderer('', myRangeList)
29 myClassificationMethod = QgsApplication.classificationMethodRegistry().method(
30     ↪"EqualInterval")
31 myRenderer.setClassificationMethod(myClassificationMethod)
32 myRenderer.setClassAttribute(myTargetField)
33 myVectorLayer.setRenderer(myRenderer)

```

## 6.7.4 Travailler avec les symboles

Pour la représentation des symboles, il y a `QgsSymbol` classe de base avec trois classes dérivées :

- `QgsMarkerSymbol` — pour entite point
- `QgsLineSymbol` — pour les entites de ligne
- `QgsFillSymbol` — pour les entites de type polygone.

**Chaque symbole est constitué d'une ou plusieurs couches de symboles** (classes dérivées de `QgsSymbolLayer`). Les couches de symboles font le rendu réel, la classe de symbole elle-même sert uniquement de conteneur pour les couches de symboles.

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

```

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

```

```
0: SimpleMarker
```

Pour connaître la couleur d'un symbole, utilisez la méthode `color` et `setColor` pour changer sa couleur. Avec les symboles de marqueurs, vous pouvez également demander la taille et la rotation des symboles avec les méthodes `size` et `angle`. Pour les symboles de ligne, la méthode `width` renvoie la largeur de la ligne.

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

## Travailler avec des couches de symboles

Comme indiqué précédemment, les couches de symboles (sous-classes de `QgsSymbolLayer`) déterminent l'apparence des entités. Il existe plusieurs classes de couches de symboles de base pour un usage général. Il est possible d'implémenter de nouveaux types de couches de symboles et donc de personnaliser arbitrairement la façon dont les entités seront rendues. La méthode `layerType()` identifie uniquement la classe de couche de symboles — les types de couches de symboles de base et par défaut sont « SimpleMarker », « SimpleLine » et « SimpleFill ».

Vous pouvez obtenir une liste complète des types de couches de symboles que vous pouvez créer pour une classe de couches de symboles donnée avec le code suivant :

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

```
1 EllipseMarker
2 FilledMarker
3 FontMarker
4 GeometryGenerator
5 RasterMarker
6 SimpleMarker
7 SvgMarker
8 VectorField
```

La classe `QgsSymbolLayerRegistry` gère une base de données de tous les types de couches de symboles disponibles.

Pour accéder aux données de la couche de symboles, utilisez sa méthode `properties()` qui renvoie un dictionnaire de valeurs clés des propriétés qui déterminent l'apparence. Chaque type de couche de symboles possède un ensemble spécifique de propriétés qu'il utilise. En outre, il existe des méthodes génériques `color`, `size`, `angle` et `width`, avec leurs homologues. Bien entendu, la taille et l'angle ne sont disponibles que pour les couches de symboles de marqueurs et la largeur pour les couches de symboles de lignes.

## 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.

```
1 from qgis.core import QgsMarkerSymbolLayer
2 from qgis.PyQt.QtGui import QColor
3
4 class FooSymbolLayer(QgsMarkerSymbolLayer):
5
6     def __init__(self, radius=4.0):
7         QgsMarkerSymbolLayer.__init__(self)
8         self.radius = radius
9         self.color = QColor(255,0,0)
10
11    def layerType(self):
12        return "FooMarker"
13
14    def properties(self):
15        return { "radius" : str(self.radius) }
16
17    def startRender(self, context):
18        pass
19
```

(suite sur la page suivante)



(suite de la page précédente)

```

20 def stopRender(self, context):
21     pass
22
23 def renderPoint(self, point, context):
24     # Rendering depends on whether the symbol is selected (QGIS >= 1.5)
25     color = context.selectionColor() if context.selected() else self.color
26     p = context.renderContext().painter()
27     p.setPen(color)
28     p.drawEllipse(point, self.radius, self.radius)
29
30 def clone(self):
31     return FooSymbolLayer(self.radius)

```

La méthode `layerType` détermine le nom de la couche de symboles ; il doit être unique parmi toutes les couches de symboles. La méthode `properties` est utilisée pour la persistance des attributs. La méthode `clone` doit renvoyer une copie de la couche de symboles avec tous les attributs exactement identiques. Enfin, il existe des méthodes de rendu : `startRender` est appelé avant de rendre la première entité, `stopRender` lorsque le rendu est terminé, et `renderPoint` est appelé pour effectuer le rendu. Les coordonnées du (des) point(s) sont déjà transformées en coordonnées de sortie.

Pour les polygones et les polygones, la seule différence réside dans la méthode de rendu : vous utiliseriez `renderPolyline` qui reçoit une liste de lignes, tandis que `renderPolygon` qui reçoit une liste de points sur l'anneau extérieur comme premier paramètre et une liste d'anneaux intérieurs (ou None) comme second paramètre.

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

```

1 from qgis.gui import QgsSymbolLayerWidget
2
3 class FooSymbolLayerWidget(QgsSymbolLayerWidget):
4     def __init__(self, parent=None):
5         QgsSymbolLayerWidget.__init__(self, parent)
6
7         self.layer = None
8
9         # setup a simple UI
10        self.label = QLabel("Radius:")
11        self.spinRadius = QDoubleSpinBox()
12        self.hbox = QHBoxLayout()
13        self.hbox.addWidget(self.label)
14        self.hbox.addWidget(self.spinRadius)
15        self.setLayout(self.hbox)
16        self.connect(self.spinRadius, SIGNAL("valueChanged(double)"), \
17                    self.radiusChanged)
18
19        def setSymbolLayer(self, layer):
20            if layer.layerType() != "FooMarker":
21                return
22            self.layer = layer
23            self.spinRadius.setValue(layer.radius)
24
25        def symbolLayer(self):
26            return self.layer
27
28        def radiusChanged(self, value):
29            self.layer.radius = value
30            self.emit(SIGNAL("changed()"))

```

Ce widget peut être intégré dans le dialogue des propriétés des symboles. Lorsque le type de couche de symboles est sélectionné dans le dialogue des propriétés des symboles, il crée une instance de la couche de symboles et une instance du widget de couche de symboles. Ensuite, il appelle la méthode `setSymbolLayer` pour assigner la couche de

symboles au widget. Dans cette méthode, le widget doit mettre à jour l'interface utilisateur pour refléter les attributs de la couche de symboles. La méthode `symbolLayer` est utilisée pour récupérer la couche de symbole à nouveau par le dialogue des propriétés pour l'utiliser pour le symbole.

À chaque changement d'attribut, le widget doit émettre le signal `changed()` pour permettre au dialogue des propriétés de mettre à jour l'aperçu du symbole.

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

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

```

1 from qgis.core import QgsSymbol, QgsSymbolLayerAbstractMetadata, \
  ↳QgsSymbolLayerRegistry
2
3 class FooSymbolLayerMetadata(QgsSymbolLayerAbstractMetadata):
4
5     def __init__(self):
6         super().__init__("FooMarker", "My new Foo marker", QgsSymbol.Marker)
7
8     def createSymbolLayer(self, props):
9         radius = float(props["radius"]) if "radius" in props else 4.0
10        return FooSymbolLayer(radius)
11
12 QgsApplication.symbolLayerRegistry().addSymbolLayerType(FooSymbolLayerMetadata())

```

Vous devez passer le type de couche (le même que celui renvoyé par la couche) et le type de symbole (marqueur/ligne/remplissage) au constructeur de la classe mère. La méthode `createSymbolLayer()` prend soin de créer une instance de couche de symboles avec les attributs spécifiés dans le dictionnaire `props`. Et il y a la méthode `createSymbolLayerWidget()` qui renvoie le widget de paramétrage pour ce type de couche de symbole.

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

## 6.7.5 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.

```

1 import random
2 from qgis.core import QgsWkbTypes, QgsSymbol, QgsFeatureRenderer
3
4
5 class RandomRenderer(QgsFeatureRenderer):
6     def __init__(self, syms=None):
7         super().__init__("RandomRenderer")
8         self.syms = syms if syms else [
9             QgsSymbol.defaultSymbol(QgsWkbTypes.geometryType(QgsWkbTypes.Point)),
10            QgsSymbol.defaultSymbol(QgsWkbTypes.geometryType(QgsWkbTypes.Point))
11        ]
12
13    def symbolForFeature(self, feature, context):
14        return random.choice(self.syms)
15
16    def startRender(self, context, fields):
17        super().startRender(context, fields)
18        for s in self.syms:

```

(suite sur la page suivante)

(suite de la page précédente)

```

19     s.startRender(context, fields)
20
21     def stopRender(self, context):
22         super().stopRender(context)
23         for s in self.syms:
24             s.stopRender(context)
25
26     def usedAttributes(self, context):
27         return []
28
29     def clone(self):
30         return RandomRenderer(self.syms)

```

Le constructeur de la classe parent `QgsFeatureRenderer` a besoin d'un nom de moteur de rendu (qui doit être unique parmi les moteurs de rendu). La méthode `symbolForFeature` est celle qui décide quel symbole sera utilisé pour une fonctionnalité particulière. `startRender` et `stopRender` s'occupent de l'initialisation/finalisation du rendu des symboles. La méthode `usedAttributes` peut renvoyer une liste de noms de champs que le moteur de rendu s'attend à voir présents. Enfin, la méthode `clone` devrait retourner une copie du moteur de rendu.

Comme pour les couches de symboles, il est possible d'attacher une interface graphique pour la configuration du moteur de rendu. Elle doit être dérivée de `QgsRendererWidget`. L'exemple de code suivant crée un bouton qui permet à l'utilisateur de définir le premier symbole

```

1  from qgis.gui import QgsRendererWidget, QgsColorButton
2
3
4  class RandomRendererWidget(QgsRendererWidget):
5      def __init__(self, layer, style, renderer):
6          super().__init__(layer, style)
7          if renderer is None or renderer.type() != "RandomRenderer":
8              self.r = RandomRenderer()
9          else:
10             self.r = renderer
11             # setup UI
12             self.btn1 = QgsColorButton()
13             self.btn1.setColor(self.r.syms[0].color())
14             self.vbox = QVBoxLayout()
15             self.vbox.addWidget(self.btn1)
16             self.setLayout(self.vbox)
17             self.btn1.colorChanged.connect(self.setColor1)
18
19     def setColor1(self):
20         color = self.btn1.color()
21         if not color.isValid(): return
22         self.r.syms[0].setColor(color)
23
24     def renderer(self):
25         return self.r

```

Le constructeur reçoit des instances de la couche active (`QgsVectorLayer`), du style global (`QgsStyle`) et du moteur de rendu courant. S'il n'y a pas de moteur de rendu ou si le moteur de rendu a un type différent, il sera remplacé par notre nouveau moteur de rendu, sinon nous utiliserons le moteur de rendu actuel (qui a déjà le type dont nous avons besoin). Le contenu du widget doit être mis à jour pour montrer l'état actuel du moteur de rendu. Lorsque le dialogue du moteur de rendu est accepté, la méthode `renderer` du widget est appelée pour obtenir le moteur de rendu actuel — il sera affecté au calque.

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` :

```

1 from qgis.core import (
2     QgsRendererAbstractMetadata,
3     QgsRendererRegistry,
4     QgsApplication
5 )
6
7 class RandomRendererMetadata(QgsRendererAbstractMetadata):
8
9     def __init__(self):
10         super().__init__("RandomRenderer", "Random renderer")
11
12     def createRenderer(self, element):
13         return RandomRenderer()
14
15     def createRendererWidget(self, layer, style, renderer):
16         return RandomRendererWidget(layer, style, renderer)
17
18 QgsApplication.rendererRegistry().addRenderer(RandomRendererMetadata())

```

De même que pour les couches de symboles, le constructeur de métadonnées attend le nom du moteur de rendu, le nom visible pour les utilisateurs et éventuellement le nom de l'icône du moteur de rendu. La méthode `createRenderer` passe une instance `QDomElement` qui peut être utilisée pour restaurer l'état du moteur de rendu à partir de l'arbre DOM. La méthode `createRendererWidget` crée le widget de configuration. Il n'a pas besoin d'être présent ou peut renvoyer « None » si le moteur de rendu n'est pas fourni avec l'interface graphique.

Pour associer une icône au moteur de rendu, vous pouvez l'assigner dans le constructeur `QgsRendererAbstractMetadata` comme troisième argument (facultatif) — le constructeur de la classe de base dans la fonction `RandomRendererMetadata __init__()` devient

```

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

```

L'icône peut également être associée à tout moment par la suite en utilisant la méthode `setIcon` de la classe de métadonnées. L'icône peut être chargée à partir d'un fichier (comme indiqué ci-dessus) ou à partir d'une ressource Qt (PyQt5 comprend le compilateur `.qrc` pour Python).

## 6.8 Sujets complémentaires

### A FAIRE :

- création/modification des symboles
- travailler avec le style (`QgsStyle`)
- travailler avec des rampes de couleur (`QgsColorRamp`)
- Explorer les couches de symboles et les registres de rendus

Les extraits de code sur cette page nécessitent les importations suivantes si vous êtes en dehors de la console pyqgis :

```

1 from qgis.core import (
2     QgsGeometry,
3     QgsPoint,
4     QgsPointXY,
5     QgsWkbTypes,
6     QgsProject,
7     QgsFeatureRequest,
8     QgsVectorLayer,
9     QgsDistanceArea,
10    QgsUnitTypes,
11 )

```

---

## Manipulation de la géométrie

---

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

Les points, les linestring et les polygones qui représentent une entité spatiale sont communément appelés géométries. Dans QGIS, ils sont représentés par la classe `QgsGeometry`

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.

La description et les spécifications de toutes les constructions et relations géométriques possibles sont disponibles dans le document « OGC Simple Feature Access Standards » <<https://www.opengeospatial.org/standards/sfa>>\_ pour des détails avancés.

### 7.1 Construction de géométrie

PyQGIS fournit plusieurs options pour créer une géométrie :

- à partir des coordonnées

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

Les coordonnées sont données en utilisant la classe `QgsPoint` ou la classe `QgsPointXY`. La différence entre ces classes est que la classe `QgsPoint` supporte les dimensions M et Z.

Une polyligne (Linestring) est représentée par une liste de points.

Un Polygone est représenté par une liste d'anneaux linéaires (c'est-à-dire des chaînes de caractères fermées). Le premier anneau est l'anneau extérieur (limite), les anneaux suivants facultatifs sont des trous dans le polygone. Notez que contrairement à certains programmes, QGIS fermera l'anneau pour vous, il n'est donc pas nécessaire de dupliquer le premier point comme le dernier.

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)

```
1 g = QgsGeometry()
2 wkb = bytes.fromhex("0101000000000000000000000045400000000000001440")
3 g.fromWkb(wkb)
4
5 # print WKT representation of the geometry
6 print(g.asWkt())
```

## 7.2 Accéder à la Géométrie

Tout d'abord, vous devez trouver le type de géométrie. La méthode `wkbType()` est celle à utiliser. Elle renvoie une valeur provenant de l'énumération `QgsWkbTypes.Type`.

```
1 if gPnt.wkbType() == QgsWkbTypes.Point:
2     print(gPnt.wkbType())
3     # output: 1 for Point
4 if gLine.wkbType() == QgsWkbTypes.LineString:
5     print(gLine.wkbType())
6     # output: 2 for LineString
7 if gPolygon.wkbType() == QgsWkbTypes.Polygon:
8     print(gPolygon.wkbType())
9     # output: 3 for Polygon
```

Comme alternative, on peut utiliser la méthode `type()` qui retourne une valeur de l'énumération `QgsWkbTypes.GeometryType`.

Vous pouvez utiliser la fonction `displayString()` pour obtenir un type de géométrie lisible par l'homme.

```
1 print(QgsWkbTypes.displayString(gPnt.wkbType()))
2 # output: 'Point'
3 print(QgsWkbTypes.displayString(gLine.wkbType()))
4 # output: 'LineString'
5 print(QgsWkbTypes.displayString(gPolygon.wkbType()))
6 # output: 'Polygon'
```

```
Point
LineString
Polygon
```

Il existe également une fonction d'aide `isMultipart()` pour savoir si une géométrie est multipartite ou non.

Pour extraire des informations d'une géométrie, il existe des fonctions d'accès pour chaque type de vecteur. Voici un exemple d'utilisation de ces accesseurs :

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

(suite sur la page suivante)

(suite de la page précédente)

```

5 print(gPolygon.asPolygon())
6 # output: [[<QgsPointXY: POINT(1 1)>, <QgsPointXY: POINT(2 2)>, <QgsPointXY:
↳POINT(2 1)>, <QgsPointXY: POINT(1 1)>]]

```

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

Pour les géométries en plusieurs parties, il existe des fonctions d'accès similaires : `asMultiPoint()`, `asMultiPolyline()` et `asMultiPolygon()`.

## 7.3 Prédicats et opérations géométriques

QGIS utilise la bibliothèque GEOS pour des opérations de géométrie avancées telles que les prédicats de géométrie (`contains()`, `intersects()`, ...) et les opérations de prédicats (`combine()`, `difference()`, ...). Il peut également calculer les propriétés géométriques des géométries, telles que l'aire (dans le cas des polygones) ou les longueurs (pour les polygones et les lignes).

Voyons un exemple qui combine l'itération sur les entités d'une couche donnée et l'exécution de certains calculs géométriques basés sur leurs géométries. Le code ci-dessous permet de calculer et d'imprimer la superficie et le périmètre de chaque pays dans la couche `pays` dans le cadre de notre projet tutoriel QGIS.

Le code suivant suppose que `layer` est un objet `QgsVectorLayer` qui a le type d'entité `Polygon`.

```

1 # let's access the 'countries' layer
2 layer = QgsProject.instance().mapLayersByName('countries')[0]
3
4 # let's filter for countries that begin with Z, then get their features
5 query = '"name" LIKE \'Zu%\''
6 features = layer.getFeatures(QgsFeatureRequest().setFilterExpression(query))
7
8 # now loop through the features, perform geometry computation and print the results
9 for f in features:
10     geom = f.geometry()
11     name = f.attribute('NAME')
12     print(name)
13     print('Area: ', geom.area())
14     print('Perimeter: ', geom.length())

```

```

1 Zubin Potok
2 Area: 0.040717371293465573
3 Perimeter: 0.9406133328077781
4 Zulia
5 Area: 3.708060762610232
6 Perimeter: 17.172123598311487
7 Zuid-Holland
8 Area: 0.4204687950359031
9 Perimeter: 4.098878517120812
10 Zug
11 Area: 0.027573510374275363
12 Perimeter: 0.7756605461489624

```

Vous avez maintenant calculé et imprimé les surfaces et les périmètres des géométries. Vous pouvez cependant rapidement remarquer que les valeurs sont étranges. C'est parce que les surfaces et les périmètres ne prennent pas en compte le CRS lorsqu'ils sont calculés à l'aide des méthodes `area()` et `length()` de la classe `QgsGeometry`. Pour un calcul de surface et de distance plus puissant, la classe `QgsDistanceArea` peut être utilisée, ce qui permet d'effectuer des calculs basés sur des ellipsoïdes :

Le code suivant suppose que `layer` est un objet `QgsVectorLayer` qui a le type d'entité `Polygon`.

```
1 d = QgsDistanceArea()
2 d.setEllipsoid('WGS84')
3
4 layer = QgsProject.instance().mapLayersByName('countries')[0]
5
6 # let's filter for countries that begin with Z, then get their features
7 query = '"name" LIKE \'Zu%\''
8 features = layer.getFeatures(QgsFeatureRequest().setFilterExpression(query))
9
10 for f in features:
11     geom = f.geometry()
12     name = f.attribute('NAME')
13     print(name)
14     print("Perimeter (m):", d.measurePerimeter(geom))
15     print("Area (m2):", d.measureArea(geom))
16
17 # let's calculate and print the area again, but this time in square kilometers
18 print("Area (km2):", d.convertAreaMeasurement(d.measureArea(geom), QgsUnitTypes.
↪AreaSquareKilometers))
```

```
1 Zubin Potok
2 Perimeter (m): 87581.40256396442
3 Area (m2): 369302069.18814206
4 Area (km2): 369.30206918814207
5 Zulia
6 Perimeter (m): 1891227.0945423362
7 Area (m2): 44973645460.19726
8 Area (km2): 44973.64546019726
9 Zuid-Holland
10 Perimeter (m): 331941.8000214341
11 Area (m2): 3217213408.4100943
12 Area (km2): 3217.213408410094
13 Zug
14 Perimeter (m): 67440.22483063207
15 Area (m2): 232457391.52097562
16 Area (km2): 232.45739152097562
```

Vous pouvez également vouloir connaître la distance et le bearing entre deux points.

```
1 d = QgsDistanceArea()
2 d.setEllipsoid('WGS84')
3
4 # Let's create two points.
5 # Santa claus is a workaholic and needs a summer break,
6 # lets see how far is Tenerife from his home
7 santa = QgsPointXY(25.847899, 66.543456)
8 tenerife = QgsPointXY(-16.5735, 28.0443)
9
10 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 et surface à l'aide de la classe QgsDistanceArea : algorithme Matrice des distances](#)
- [algorithme Lignes vers polygones](#)

Les extraits de code sur cette page nécessitent les importations suivantes si vous êtes en dehors de la console pyqgis :

```
1 from qgis.core import (
2     QgsCoordinateReferenceSystem,
3     QgsCoordinateTransform,
4     QgsProject,
5     QgsPointXY,
```

(suite sur la page suivante)



(suite de la page précédente)

6

)



## 8.1 Système de coordonnées de référence

Les systèmes de référence de coordonnées (CRS) sont encapsulés par la classe `QgsCoordinateReferenceSystem` class. Les instances de cette classe peuvent être créées de plusieurs manières différentes :

- spécifier le SCR par son ID

```
# EPSG 4326 is allocated for WGS84
crs = QgsCoordinateReferenceSystem("EPSG:4326")
assert crs.isValid()
```

QGIS prend en charge différents identificateurs CRS dans les formats suivants :

- EPSG:<code> — ID attribué par l'organisation EPSG - traité avec `createFromOgcWms()`
- POSTGIS:<srid> — ID utilisé dans les bases de données PostGIS - traité avec `createFromSrid()`
- INTERNAL:<srsid> — ID utilisé dans la base de données interne QGIS - géré avec `createFromSrsId()`
- PROJ:<proj> - géré avec `createFromProj()`
- WKT:<wkt> - géré avec `createFromWkt()`

Si aucun préfixe n'est spécifié, la définition de WKT est supposée.

- spécifier le SCR par son Well-Known-Text (WKT)

```
1 wkt = 'GEOGCS["WGS84", DATUM["WGS84", SPHEROID["WGS84", 6378137.0, 298.
   ↳2572223563]], ' \
2     'PRIMEM["Greenwich", 0.0], UNIT["degree",0.017453292519943295], ' \
3     'AXIS["Longitude",EAST], AXIS["Latitude",NORTH]]'
4 crs = QgsCoordinateReferenceSystem(wkt)
5 assert crs.isValid()
```

- créer un CRS invalide et utiliser ensuite l'une des fonctions « créer\* » pour l'initialiser. Dans l'exemple suivant, nous utilisons une chaîne Proj pour initialiser la projection.

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

Il est sage de vérifier si la création (c'est-à-dire la consultation dans la base de données) du CRS a réussi : `isValid()` doit retourner `True`.

Notez que pour l'initialisation des systèmes de référence spatiale, QGIS doit rechercher les valeurs appropriées dans sa base de données interne : file `:srs.db`. Ainsi, dans le cas où vous créez une application indépendante, vous devez définir les chemins correctement avec `QgsApplication.setPrefixPath()`, sinon la base de données ne sera pas trouvée. Si vous exécutez les commandes depuis la console Python de QGIS ou si vous développez un plugin, vous ne vous en souciez pas : tout est déjà configuré pour vous.

Accès aux informations du système de référence spatiale :

```
1 crs = QgsCoordinateReferenceSystem("EPSG:4326")
2
3 print("QGIS CRS ID:", crs.srsid())
4 print("PostGIS SRID:", crs.postgisSrid())
5 print("Description:", crs.description())
6 print("Projection Acronym:", crs.projectionAcronym())
7 print("Ellipsoïd Acronym:", crs.ellipsoidAcronym())
8 print("Proj String:", crs.toProj())
9 # check whether it's geographic or projected coordinate system
10 print("Is geographic:", crs.isGeographic())
11 # check type of map units in this CRS (values defined in Qgis::units enum)
12 print("Map units:", crs.mapUnits())
```

Rendu :

```
1 QGIS CRS ID: 3452
2 PostGIS SRID: 4326
3 Description: WGS 84
4 Projection Acronym: longlat
5 Ellipsoïd Acronym: WGS84
6 Proj String: +proj=longlat +datum=WGS84 +no_defs
7 Is geographic: True
8 Map units: 6
```

## 8.2 Transformation de SCR

Vous pouvez effectuer une transformation entre différents systèmes de références spatiaux en utilisant la classe `QgsCoordinateTransform`. La façon la plus simple de l'utiliser est de créer un CRS source et un CRS destination et de construire une instance de `QgsCoordinateTransform` avec ceux-ci et le projet en cours. Ensuite, il suffit d'appeler plusieurs fois la fonction `transform()` pour effectuer la transformation. Par défaut, elle effectue une transformation directe, mais elle peut également effectuer une transformation inverse.

```
1 crsSrc = QgsCoordinateReferenceSystem("EPSG:4326") # WGS 84
2 crsDest = QgsCoordinateReferenceSystem("EPSG:32633") # WGS 84 / UTM zone 33N
3 transformContext = QgsProject.instance().transformContext()
4 xform = QgsCoordinateTransform(crsSrc, crsDest, transformContext)
5
6 # forward transformation: src -> dest
7 pt1 = xform.transform(QgsPointXY(18,5))
8 print("Transformed point:", pt1)
9
10 # inverse transformation: dest -> src
11 pt2 = xform.transform(pt1, QgsCoordinateTransform.ReverseTransform)
12 print("Transformed back:", pt2)
```

Rendu :

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

Les extraits de code sur cette page nécessitent les importations suivantes si vous êtes en dehors de la console pyqgis :

```
1 from qgis.PyQt.QtGui import (  
2     QColor,  
3 )  
4  
5 from qgis.PyQt.QtCore import Qt, QRectF  
6  
7 from qgis.core import (  
8     QgsVectorLayer,  
9     QgsPoint,  
10    QgsPointXY,  
11    QgsProject,  
12    QgsGeometry,  
13    QgsMapRendererJob,  
14 )  
15  
16 from qgis.gui import (  
17    QgsMapCanvas,  
18    QgsVertexMarker,  
19    QgsMapCanvasItem,  
20    QgsRubberBand,  
21 )
```



---

## Utilisation du canevas de carte

---

- *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*

Le widget de canevas de carte est probablement le widget le plus important dans QGIS car il affiche la carte composée de couches superposées et permet l'interaction avec la carte et les couches. Le canevas montre toujours une partie de la carte définie par l'étendue courante de la carte. L'interaction est réalisée grâce aux **outils cartographiques** : il existe des outils pour se déplacer, zoomer, identifier les couches, mesurer, éditer des vecteurs... Comme pour les autres programmes graphiques, il y a toujours un outil actif et l'utilisateur peut passer d'un outil disponible à l'autre.

Le canevas de carte est implémenté par la classe `QgsMapCanvas` dans le module `qgis.gui`. L'implémentation est basée sur le framework de vue graphique de Qt. Ce framework fournit en général une surface et une vue où des items graphiques personnalisés peuvent être placés et avec lesquels l'utilisateur peut interagir. Nous supposons que vous suffisamment familier avec Qt pour comprendre les concepts de scène graphique, de vue et d'items. Si ce n'est pas le cas, veuillez lire la [documentation du framework](#).

A chaque fois que la carte a été déplacée, zoomée (ou toute autre action déclenchant un rafraîchissement), la carte est rendue de nouveau dans l'étendue courante. Les couches sont rendues sous forme d'image (en utilisant la classe `QgsMapRendererJob`) et cette image est affichée sur la canevas. La classe `QgsMapCanvas` contrôle aussi le rafraîchissement de la carte rendue. En plus de cet item qui joue le rôle d'arrière-plan, il peut y avoir d'autres **items de canevas de carte**.

Les éléments typiques d'un canevas de carte sont les élastiques (utilisés pour la mesure, l'édition vecteur, etc.) ou les marqueurs de sommet. Les éléments de canevas sont généralement utilisés pour donner un retour visuel aux outils cartographiques. Par exemple, lors de la création d'un nouveau polygone, l'outil cartographique crée un élément de canevas élastique qui montre la forme actuelle du polygone. Tous les éléments du canevas de carte sont des sous-classes de `QgsMapCanvasItem` qui ajoute quelques fonctionnalités supplémentaires aux objets de base de `QGraphicsItem`.

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

- le canevas de carte — pour visualiser la carte
- éléments du canevas de la carte — éléments supplémentaires pouvant être affichés sur le canevas de la carte
- outils cartographiques — pour l'interaction avec le canevas de la carte

## 9.1 Intégrer un canevas de carte

Map canvas est un widget comme tous les autres widgets Qt, donc l'utiliser est aussi simple que de le créer et de le montrer.

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

On obtient ainsi une fenêtre autonome avec un canevas de carte. Elle peut également être intégrée dans un widget ou une fenêtre existante. Lorsque vous utilisez les fichiers `.ui` et Qt Designer, placez un `QWidget` sur le formulaire et promouvez-le dans une nouvelle classe : définissez `QgsMapCanvas` comme nom de classe et définissez `qgis.gui` comme fichier d'en-tête. L'utilitaire « `pyuic5` » s'occupera de tout cela. C'est une façon très pratique d'intégrer le canevas. L'autre possibilité est d'écrire manuellement le code pour construire le canevas de la carte et d'autres widgets (comme les enfants d'une fenêtre principale ou d'un dialogue) et créer une mise en page.

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

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

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

Il est maintenant temps d'ajouter quelques couches cartographiques. Nous allons d'abord ouvrir une couche et l'ajouter au projet en cours. Ensuite, nous définirons l'étendue du canevas et la liste des couches pour le canevas.

```
1 vlayer = QgsVectorLayer('testdata/airports.shp', "Airports layer", "ogr")
2 if not vlayer.isValid():
3     print("Layer failed to load!")
4
5 # add layer to the registry
6 QgsProject.instance().addMapLayer(vlayer)
7
8 # set extent to the extent of our layer
9 canvas.setExtent(vlayer.extent())
10
11 # set the map canvas layer set
12 canvas.setLayers([vlayer])
```

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

## 9.2 Contour d'édition et symboles de sommets

Pour afficher des données supplémentaires en haut de la carte dans le canevas, utilisez les éléments du canevas de la carte. Il est possible de créer des classes d'éléments de canevas personnalisés (voir ci-dessous), mais il existe deux classes d'éléments de canevas utiles pour des raisons de commodité : `QgsRubberBand` pour dessiner des polygones ou des polygones, et `QgsVertexMarker` pour dessiner des points. Ils fonctionnent tous les deux avec des coordonnées cartographiques, de sorte que la forme est déplacée ou mise à l'échelle automatiquement lorsque le canevas fait l'objet d'un panoramique ou d'un zoom.

Pour montrer 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)
```

Les éléments du canevas sont liés à la scène du canevas. Pour les cacher temporairement (et les montrer à nouveau), utilisez la combinaison `hide()` et `show()`. Pour supprimer complètement l'élément, vous devez le retirer de la scène du canevas

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

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

L'élastique peut également être utilisé pour dessiner des points, mais la classe `QgsVertexMarker` est mieux adaptée pour cela (`QgsRubberBand` ne dessinerait qu'un rectangle autour du point désiré).

Vous pouvez utiliser le marqueur de sommet comme ceci :

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

Cela permettra de tracer une croix rouge sur la position [10,45]. Il est possible de personnaliser le type d'icône, la taille, la couleur et la largeur du stylo

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

Pour cacher temporairement les marqueurs de vertex et les retirer du canevas, utilisez les mêmes méthodes que pour les élastiques.

## 9.3 Utiliser les outils cartographiques avec le canevas

L'exemple suivant construit une fenêtre qui contient un canevas de carte et des outils cartographiques de base pour le panoramique et le zoom. Des actions sont créées pour l'activation de chaque outil : le panoramique est effectué avec une paire d'instances `QgsMapToolPan`, le zoom avant/arrière avec une paire d'instances `QgsMapToolZoom`. Les actions sont définies comme vérifiables et ensuite assignées aux outils pour permettre la gestion automatique de l'état vérifié/décoché des actions – quand un outil de carte est activé, son action est marquée comme sélectionnée et l'action de l'outil de carte précédent est désélectionnée. Les outils cartographiques sont activés en utilisant la méthode `setMapTool()`.

```
1 from qgis.gui import *
2 from qgis.PyQt.QtWidgets import QAction, QMainWindow
3 from qgis.PyQt.QtCore import Qt
4
5 class MyWnd(QMainWindow):
6     def __init__(self, layer):
7         QMainWindow.__init__(self)
8
```

(suite sur la page suivante)

```

9     self.canvas = QgsMapCanvas()
10    self.canvas.setCanvasColor(Qt.white)
11
12    self.canvas.setExtent(layer.extent())
13    self.canvas.setLayers([layer])
14
15    self.setCentralWidget(self.canvas)
16
17    self.actionZoomIn = QAction("Zoom in", self)
18    self.actionZoomOut = QAction("Zoom out", self)
19    self.actionPan = QAction("Pan", self)
20
21    self.actionZoomIn.setCheckable(True)
22    self.actionZoomOut.setCheckable(True)
23    self.actionPan.setCheckable(True)
24
25    self.actionZoomIn.triggered.connect(self.zoomIn)
26    self.actionZoomOut.triggered.connect(self.zoomOut)
27    self.actionPan.triggered.connect(self.pan)
28
29    self.toolbar = self.addToolBar("Canvas actions")
30    self.toolbar.addAction(self.actionZoomIn)
31    self.toolbar.addAction(self.actionZoomOut)
32    self.toolbar.addAction(self.actionPan)
33
34    # create the map tools
35    self.toolPan = QgsMapToolPan(self.canvas)
36    self.toolPan.setAction(self.actionPan)
37    self.toolZoomIn = QgsMapToolZoom(self.canvas, False) # false = in
38    self.toolZoomIn.setAction(self.actionZoomIn)
39    self.toolZoomOut = QgsMapToolZoom(self.canvas, True) # true = out
40    self.toolZoomOut.setAction(self.actionZoomOut)
41
42    self.pan()
43
44    def zoomIn(self):
45        self.canvas.setMapTool(self.toolZoomIn)
46
47    def zoomOut(self):
48        self.canvas.setMapTool(self.toolZoomOut)
49
50    def pan(self):
51        self.canvas.setMapTool(self.toolPan)

```

Vous pouvez essayer le code ci-dessus dans l'éditeur de console Python. Pour invoquer la fenêtre de canevas, ajoutez les lignes suivantes pour instancier la classe « MyWnd ». Elles rendront la couche actuellement sélectionnée sur le canevas nouvellement créé

```

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

```

## 9.4 Ecrire des outils cartographiques personnalisés

Vous pouvez écrire vos outils personnalisés, pour mettre en œuvre un comportement personnalisé aux actions effectuées par les utilisateurs sur le canevas.

Les outils cartographiques doivent hériter de la classe `QgsMapTool`, ou de toute classe dérivée, et être sélectionnés comme outils actifs dans le canevas en utilisant la méthode `setMapTool()` comme nous l'avons déjà vu.

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.

```

1 class RectangleMapTool(QgsMapToolEmitPoint):
2     def __init__(self, canvas):
3         self.canvas = canvas
4         QgsMapToolEmitPoint.__init__(self, self.canvas)
5         self.rubberBand = QgsRubberBand(self.canvas, True)
6         self.rubberBand.setColor(Qt.red)
7         self.rubberBand.setWidth(1)
8         self.reset()
9
10    def reset(self):
11        self.startPoint = self.endPoint = None
12        self.isEmittingPoint = False
13        self.rubberBand.reset(True)
14
15    def canvasPressEvent(self, e):
16        self.startPoint = self.toMapCoordinates(e.pos())
17        self.endPoint = self.startPoint
18        self.isEmittingPoint = True
19        self.showRect(self.startPoint, self.endPoint)
20
21    def canvasReleaseEvent(self, e):
22        self.isEmittingPoint = False
23        r = self.rectangle()
24        if r is not None:
25            print("Rectangle:", r.xMinimum(),
26                  r.yMinimum(), r.xMaximum(), r.yMaximum()
27                  )
28
29    def canvasMoveEvent(self, e):
30        if not self.isEmittingPoint:
31            return
32
33        self.endPoint = self.toMapCoordinates(e.pos())
34        self.showRect(self.startPoint, self.endPoint)
35
36    def showRect(self, startPoint, endPoint):
37        self.rubberBand.reset(QGis.Polygon)
38        if startPoint.x() == endPoint.x() or startPoint.y() == endPoint.y():
39            return
40
41        point1 = QgsPoint(startPoint.x(), startPoint.y())
42        point2 = QgsPoint(startPoint.x(), endPoint.y())
43        point3 = QgsPoint(endPoint.x(), endPoint.y())
44        point4 = QgsPoint(endPoint.x(), startPoint.y())
45
46        self.rubberBand.addPoint(point1, False)
47        self.rubberBand.addPoint(point2, False)
48        self.rubberBand.addPoint(point3, False)
49        self.rubberBand.addPoint(point4, True) # true to update canvas

```

(suite sur la page suivante)

```

50     self.rubberBand.show()
51
52     def rectangle(self):
53         if self.startPoint is None or self.endPoint is None:
54             return None
55         elif (self.startPoint.x() == self.endPoint.x() or \
56               self.startPoint.y() == self.endPoint.y()):
57             return None
58
59         return QgsRectangle(self.startPoint, self.endPoint)
60
61     def deactivate(self):
62         QgsMapTool.deactivate(self)
63         self.deactivated.emit()

```

## 9.5 Ecrire des éléments de canevas de carte personnalisés

Voici un exemple d'un canevas personnalisé qui dessine un cercle :

```

1  class CircleCanvasItem(QgsMapCanvasItem):
2      def __init__(self, canvas):
3          super().__init__(canvas)
4          self.center = QgsPoint(0, 0)
5          self.size = 100
6
7      def setCenter(self, center):
8          self.center = center
9
10     def center(self):
11         return self.center
12
13     def setSize(self, size):
14         self.size = size
15
16     def size(self):
17         return self.size
18
19     def boundingRect(self):
20         return QRectF(self.center.x() - self.size/2,
21                       self.center.y() - self.size/2,
22                       self.center.x() + self.size/2,
23                       self.center.y() + self.size/2)
24
25     def paint(self, painter, option, widget):
26         path = QPainterPath()
27         path.moveTo(self.center.x(), self.center.y());
28         path.arcTo(self.boundingRect(), 0.0, 360.0)
29         painter.fillPath(path, QColor("red"))
30
31
32     # Using the custom item:
33     item = CircleCanvasItem(iface.mapCanvas())
34     item.setCenter(QgsPointXY(200,200))
35     item.setSize(80)

```

Les extraits de code sur cette page nécessitent les importations suivantes :

```
1 import os
2
3 from qgis.core import (
4     QgsGeometry,
5     QgsMapSettings,
6     QgsPrintLayout,
7     QgsMapSettings,
8     QgsMapRendererParallelJob,
9     QgsLayoutItemLabel,
10    QgsLayoutItemLegend,
11    QgsLayoutItemMap,
12    QgsLayoutItemPolygon,
13    QgsLayoutItemScaleBar,
14    QgsLayoutExporter,
15    QgsLayoutItem,
16    QgsLayoutPoint,
17    QgsLayoutSize,
18    QgsUnitTypes,
19    QgsProject,
20    QgsFillSymbol,
21 )
22
23 from qgis.PyQt.QtGui import (
24     QPolygonF,
25     QColor,
26 )
27
28 from qgis.PyQt.QtCore import (
29     QPointF,
30     QRectF,
31     QSize,
32 )
```



---

## Rendu cartographique et Impression

---

- *Rendu simple*
- *Rendu des couches ayant différents SCR*
- *Sortie en utilisant la mise en page*
  - *Exporter la mise en page*
  - *Exporter un atlas*

Il y a généralement deux approches lorsque les données d'entrée doivent être rendues sous forme de carte : soit le faire rapidement en utilisant *QgsMapRendererJob*, soit produire une sortie plus fine en composant la carte avec la classe *QgsLayout*.

### 10.1 Rendu simple

Le rendu est effectué en créant un objet *QgsMapSettings* pour définir les paramètres de rendu, puis en construisant un objet *QgsMapRendererJob* avec ces paramètres. Ce dernier est ensuite utilisé pour créer l'image résultante.

Voici un exemple :

```
1 image_location = os.path.join(QgsProject.instance().homePath(), "render.png")
2
3 vlayer = iface.activeLayer()
4 settings = QgsMapSettings()
5 settings.setLayers([vlayer])
6 settings.setBackgroundColor(QColor(255, 255, 255))
7 settings.setOutputSize(QSize(800, 600))
8 settings.setExtent(vlayer.extent())
9
10 render = QgsMapRendererParallelJob(settings)
11
12 def finished():
13     img = render.renderedImage()
14     # save the image; e.g. img.save("/Users/myuser/render.png", "png")
15     img.save(image_location, "png")
16
17 render.finished.connect(finished)
```

(suite sur la page suivante)

```
18
19 render.start()
```

## 10.2 Rendu des couches ayant différents SCR

Si vous avez plus d'une couche et qu'elles ont un CRS différent, l'exemple simple ci-dessus ne fonctionnera probablement pas : pour obtenir les bonnes valeurs à partir des calculs d'étendue, vous devez définir explicitement le CRS de destination

```
layers = [iface.activeLayer()]
settings.setLayers(layers)
settings.setDestinationCrs(layers[0].crs())
```

## 10.3 Sortie en utilisant la mise en page

La mise en page est un outil très pratique si vous souhaitez obtenir un résultat plus sophistiqué que le simple rendu présenté ci-dessus. Il est possible de créer des mises en page complexes composées de vues de cartes, d'étiquettes, de légendes, de tableaux et d'autres éléments qui sont généralement présents sur les cartes papier. Les mises en page peuvent ensuite être exportées au format PDF, en images raster ou directement imprimées sur une imprimante.

La mise en page consiste en un ensemble de classes. Elles appartiennent toutes à la bibliothèque centrale. L'application QGIS dispose d'une interface graphique pratique pour le placement des éléments, bien qu'elle ne soit pas disponible dans la bibliothèque GUI. Si vous n'êtes pas familiers avec le [Qt Graphics View framework](#), alors vous êtes encouragé à consulter la documentation dès maintenant, car la mise en page est basée sur celui-ci.

La classe centrale de la mise en page est la classe `QgsLayout`, qui est dérivée de la classe `Qt QGraphicsScene`. Créons une instance de celle-ci :

```
project = QgsProject()
layout = QgsPrintLayout(project)
layout.initializeDefaults()
```

Nous pouvons maintenant ajouter divers éléments (carte, étiquette, ...) à la mise en page. Tous ces objets sont représentés par des classes qui héritent de la classe de base `QgsLayoutItem`.

Voici une description de certains des principaux éléments qui peuvent être ajoutés à une mise en page.

- 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



```

1 item = QgsLayoutItemScaleBar(layout)
2 item.setStyle('Numeric') # optionally modify the style
3 item.setLinkedMap(map) # map is an instance of QgsLayoutItemMap
4 item.applyDefaultSize()
5 layout.addItem(item)

```

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

```

1 polygon = QPolygonF()
2 polygon.append(QPointF(0.0, 0.0))
3 polygon.append(QPointF(100.0, 0.0))
4 polygon.append(QPointF(200.0, 100.0))
5 polygon.append(QPointF(100.0, 200.0))
6
7 polygonItem = QgsLayoutItemPolygon(polygon, layout)
8 layout.addItem(polygonItem)
9
10 props = {}
11 props["color"] = "green"
12 props["style"] = "solid"
13 props["style_border"] = "solid"
14 props["color_border"] = "black"
15 props["width_border"] = "10.0"
16 props["joinstyle"] = "miter"
17
18 symbol = QgsFillSymbol.createSimple(props)
19 polygonItem.setSymbol(symbol)

```

- table

Une fois qu'un élément est ajouté à la mise en page, il peut être déplacé et redimensionné :

```

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

```

Par défaut, un cadre est dessiné autour de chaque élément. Vous pouvez le supprimer comme suit

```

# for a composer label
label.setFrameEnabled(False)

```

Outre la création manuelle des éléments de mise en page, QGIS prend en charge les modèles de mise en page qui sont essentiellement des compositions dont tous les éléments sont enregistrés dans un fichier .qpt (avec une syntaxe XML).

Une fois que la composition est prête (les éléments de mise en page ont été créés et ajoutés à la composition), nous pouvons procéder à la production d'une sortie raster et/ou vecteur.

### 10.3.1 Exporter la mise en page

Pour exporter une mise en page, la classe `QgsLayoutExporter` doit être utilisée.

```

1 base_path = os.path.join(QgsProject.instance().homePath())
2 pdf_path = os.path.join(base_path, "output.pdf")
3
4 exporter = QgsLayoutExporter(layout)
5 exporter.exportToPdf(pdf_path, QgsLayoutExporter.PdfExportSettings())

```

Utilisez la méthode `exportToImage()` au cas où vous voudriez exporter vers une image au lieu d'un fichier PDF.

### 10.3.2 Exporter un atlas

Si vous souhaitez exporter toutes les pages d'une mise en page pour laquelle l'option atlas est configurée et activée, vous devez utiliser la méthode `atlas()` dans l'exportateur (`QgsLayoutExporter`) avec de petits ajustements. Dans l'exemple suivant, les pages sont exportées en image PNG :

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

Notez que les sorties seront enregistrées dans le dossier du chemin de base, en utilisant l'expression du nom de fichier de sortie configurée sur l'atlas.

Les extraits de code sur cette page nécessitent les importations suivantes si vous êtes en dehors de la console pyqgis :

```
1 from qgis.core import (  
2     edit,  
3     QgsExpression,  
4     QgsExpressionContext,  
5     QgsFeature,  
6     QgsFeatureRequest,  
7     QgsField,  
8     QgsFields,  
9     QgsVectorLayer,  
10    QgsPointXY,  
11    QgsGeometry,  
12    QgsProject,  
13    QgsExpressionContextUtils  
14 )
```

---

## 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`

## 11.1 Analyse syntaxique d'expressions

L'exemple suivant montre comment vérifier si une expression donnée peut être analysée correctement :

```

1 exp = QgsExpression('1 + 1 = 2')
2 assert(not exp.hasParserError())
3
4 exp = QgsExpression('1 + 1 = ')
5 assert(exp.hasParserError())
6
7 assert(exp.parserErrorString() == '\nsyntax error, unexpected $end')
```

## 11.2 Évaluation des expressions

Les expressions peuvent être utilisées dans différents contextes, par exemple pour filtrer des entités ou pour calculer de nouvelles valeurs de champ. Dans tous les cas, l'expression doit être évaluée. Cela signifie que sa valeur est calculée en effectuant les étapes de calcul spécifiées, qui peuvent aller de l'arithmétique simple aux expressions agrégées.

### 11.2.1 Expressions basiques

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

```

exp = QgsExpression('1 + 1 = 2')
assert(exp.evaluate()) # exp.evaluate() returns 1 and assert() recognizes this as
↳ True
```

### 11.2.2 Expressions avec entités

Pour évaluer une expression par rapport à une fonctionnalité, un objet `QgsExpressionContext` doit être créé et transmis à la fonction d'évaluation afin de permettre à l'expression d'accéder aux valeurs de champ de la fonctionnalité.

L'exemple suivant montre comment créer une entité avec un champ appelé « Colonne » et comment ajouter cette entité au contexte d'expression.

```

1 fields = QgsFields()
2 field = QgsField('Column')
3 fields.append(field)
4 feature = QgsFeature()
5 feature.setFields(fields)
6 feature.setAttribute(0, 99)
7
8 exp = QgsExpression('"Column"')
9 context = QgsExpressionContext()
10 context.setFeature(feature)
11 assert(exp.evaluate(context) == 99)
```

Voici un exemple plus complet de la façon d'utiliser des expressions dans le contexte d'une couche vectorielle, afin de calculer de nouvelles valeurs de champ :

```

1  from qgis.PyQt.QtCore import QVariant
2
3  # create a vector layer
4  vl = QgsVectorLayer("Point", "Companies", "memory")
5  pr = vl.dataProvider()
6  pr.addAttributes([QgsField("Name", QVariant.String),
7                    QgsField("Employees", QVariant.Int),
8                    QgsField("Revenue", QVariant.Double),
9                    QgsField("Rev. per employee", QVariant.Double),
10                   QgsField("Sum", QVariant.Double),
11                   QgsField("Fun", QVariant.Double)])
12 vl.updateFields()
13
14 # add data to the first three fields
15 my_data = [
16     {'x': 0, 'y': 0, 'name': 'ABC', 'emp': 10, 'rev': 100.1},
17     {'x': 1, 'y': 1, 'name': 'DEF', 'emp': 2, 'rev': 50.5},
18     {'x': 5, 'y': 5, 'name': 'GHI', 'emp': 100, 'rev': 725.9}]
19
20 for rec in my_data:
21     f = QgsFeature()
22     pt = QgsPointXY(rec['x'], rec['y'])
23     f.setGeometry(QgsGeometry.fromPointXY(pt))
24     f.setAttributes([rec['name'], rec['emp'], rec['rev']])
25     pr.addFeature(f)
26
27 vl.updateExtents()
28 QgsProject.instance().addMapLayer(vl)
29
30 # The first expression computes the revenue per employee.
31 # The second one computes the sum of all revenue values in the layer.
32 # The final third expression doesn't really make sense but illustrates
33 # the fact that we can use a wide range of expression functions, such
34 # as area and buffer in our expressions:
35 expression1 = QgsExpression('"Revenue"/"Employees"')
36 expression2 = QgsExpression('sum("Revenue")')
37 expression3 = QgsExpression('area(buffer($geometry,"Employees"))')
38
39 # QgsExpressionContextUtils.globalProjectLayerScopes() is a convenience
40 # function that adds the global, project, and layer scopes all at once.
41 # Alternatively, those scopes can also be added manually. In any case,
42 # it is important to always go from "most generic" to "most specific"
43 # scope, i.e. from global to project to layer
44 context = QgsExpressionContext()
45 context.appendScopes(QgsExpressionContextUtils.globalProjectLayerScopes(vl))
46
47 with edit(vl):
48     for f in vl.getFeatures():
49         context.setFeature(f)
50         f['Rev. per employee'] = expression1.evaluate(context)
51         f['Sum'] = expression2.evaluate(context)
52         f['Fun'] = expression3.evaluate(context)
53         vl.updateFeature(f)
54
55 print( f['Sum'])

```

```
876.5
```

## 11.2.3 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.

```
1 layer = QgsVectorLayer("Point?field=Test:integer",
2                       "addfeat", "memory")
3
4 layer.startEditing()
5
6 for i in range(10):
7     feature = QgsFeature()
8     feature.setAttributes([i])
9     assert(layer.addFeature(feature))
10 layer.commitChanges()
11
12 expression = 'Test >= 3'
13 request = QgsFeatureRequest().setFilterExpression(expression)
14
15 matches = 0
16 for f in layer.getFeatures(request):
17     matches += 1
18
19 assert(matches == 7)
```

## 11.3 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 :

```
1 exp = QgsExpression("1 + 1 = 2")
2 if exp.hasParserError():
3     raise Exception(exp.parserErrorString())
4
5 value = exp.evaluate()
6 if exp.hasEvalError():
7     raise ValueError(exp.evalErrorString())
```

Les extraits de code sur cette page nécessitent les importations suivantes si vous êtes en dehors de la console pyqgis :

```
1 from qgis.core import (
2     QgsProject,
3     QgsSettings,
4     QgsVectorLayer
5 )
```

---

## Lecture et sauvegarde de configurations

---

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

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

Nous pouvons identifier différents types de paramètres :

- **réglages globaux** — ils sont liés à l'utilisateur sur une machine particulière. QGIS lui-même stocke de nombreux paramètres globaux, par exemple, la taille de la fenêtre principale ou la tolérance d'accrochage par défaut. Les paramètres sont gérés par la classe `QgsSettings`, par exemple par les méthodes `setValue()` et `value()`.

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

```
1 def store():
2     s = QgsSettings()
3     s.setValue("myplugin/mytext", "hello world")
4     s.setValue("myplugin/myint", 10)
5     s.setValue("myplugin/myreal", 3.14)
6
7 def read():
8     s = QgsSettings()
9     mytext = s.value("myplugin/mytext", "default text")
10    myint = s.value("myplugin/myint", 123)
11    myreal = s.value("myplugin/myreal", 2.71)
12    nonexistent = s.value("myplugin/nonexistent", None)
13    print(mytext)
14    print(myint)
15    print(myreal)
16    print(nonexistent)
```

Le second paramètre de la méthode `value()` est facultatif et spécifie la valeur par défaut qui est retournée s'il n'y a pas de valeur précédente définie pour le nom du paramètre passé.

Pour une méthode permettant de pré-configurer les valeurs par défaut des paramètres globaux via le fichier `global_settings.ini`, voir `deploying_organization` pour plus de détails.

- **Les paramètres du projet** — varient selon les différents projets et sont donc liés à un fichier de projet. La couleur de fond du canevas de la carte ou le système de référence des coordonnées de destination (CRS) en

sont des exemples — le fond blanc et le WGS84 peuvent convenir à un projet, tandis que le fond jaune et la projection UTM conviennent mieux à un autre.

Ci-après un exemple d'utilisation.

```

1 proj = QgsProject.instance()
2
3 # store values
4 proj.writeEntry("myplugin", "mytext", "hello world")
5 proj.writeEntry("myplugin", "myint", 10)
6 proj.writeEntry("myplugin", "mydouble", 0.01)
7 proj.writeEntry("myplugin", "mybool", True)
8
9 # read values (returns a tuple with the value, and a status boolean
10 # which communicates whether the value retrieved could be converted to
11 # its type, in these cases a string, an integer, a double and a boolean
12 # respectively)
13
14 mytext, type_conversion_ok = proj.readEntry("myplugin",
15                                           "mytext",
16                                           "default text")
17 myint, type_conversion_ok = proj.readNumEntry("myplugin",
18                                              "myint",
19                                              123)
20 mydouble, type_conversion_ok = proj.readDoubleEntry("myplugin",
21                                                    "mydouble",
22                                                    123)
23 mybool, type_conversion_ok = proj.readBoolEntry("myplugin",
24                                                "mybool",
25                                                123)

```

Comme vous pouvez le voir, la méthode `writeEntry()` est utilisée pour tous les types de données, mais il existe plusieurs méthodes pour relire la valeur de réglage, et la méthode correspondante doit être sélectionnée pour chaque type de données.

- **paramètres de couche cartographique** — Ces paramètres sont liés à une instance particulière d'une couche cartographique dans un projet. Ils ne sont *pas* liés à la source de données sous-jacente d'une couche, donc si vous créez deux instances de couche cartographique d'un shapefile, elles ne partageront pas les paramètres. Les paramètres sont stockés dans le fichier de projet, donc si l'utilisateur ouvre à nouveau le projet, les paramètres liés à la couche seront à nouveau là. La valeur d'un paramètre donné est récupérée à l'aide de la méthode `customProperty()`, et peut être définie à l'aide de la méthode `setCustomProperty()`.

```

1 vlayer = QgsVectorLayer()
2 # save a value
3 vlayer.setCustomProperty("mytext", "hello world")
4
5 # read the value again (returning "default text" if not found)
6 mytext = vlayer.customProperty("mytext", "default text")

```

Les extraits de code sur cette page nécessitent les importations suivantes si vous êtes en dehors de la console pyqgis :

```

1 from qgis.core import (
2     QgsMessageLog,
3     QgsGeometry,
4 )
5
6 from qgis.gui import (
7     QgsMessageBar,
8 )
9
10 from qgis.PyQt.QtWidgets import (
11     QSizePolicy,
12     QPushButton,
13     QDialog,

```

(suite sur la page suivante)



(suite de la page précédente)

```
14     QGridLayout,  
15     QDialogButtonBox,  
16 )
```



## Communiquer avec l'utilisateur

- *Afficher des messages. La classe `QgsMessageBar`*
- *Afficher la progression*
- *Journal*
  - *`QgsMessageLog`*
  - *Le python intégré dans le module de journalisation*

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

### 13.1 Afficher des messages. La classe `QgsMessageBar`

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

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

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

```
Messages(2): Error : I'm sorry Dave, I'm afraid I can't do that
```

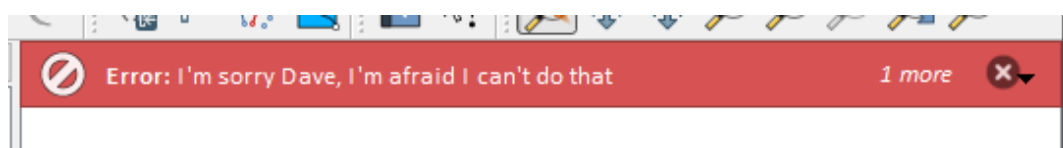


Fig. 13.1 – Barre de message de QGIS

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=Qgis.Critical, duration=3)
```

Messages(2): Oops : The plugin is not working as it should

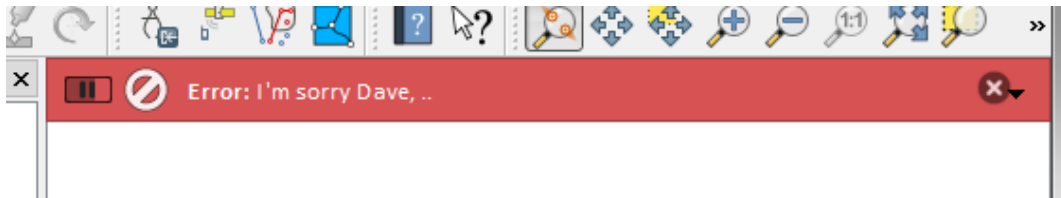


Fig. 13.2 – Barre de message de Qgis avec décompte

Les exemples ci-dessus montrent une barre d’erreur, mais le paramètre `level` peut être utilisé pour créer des messages d’avertissement ou des messages d’information, en utilisant l’énumération `Qgis.MessageLevel`. Vous pouvez utiliser jusqu’à 4 niveaux différents :

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

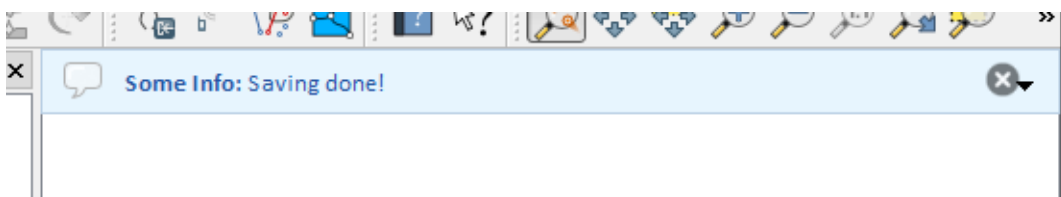


Fig. 13.3 – Barre de message QGis (info)

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

```
1 def showError():
2     pass
3
4 widget = iface.messageBar().createMessage("Missing Layers", "Show Me")
5 button = QPushButton(widget)
6 button.setText("Show Me")
7 button.pressed.connect(showError)
8 widget.layout().addWidget(button)
9 iface.messageBar().pushWidget(widget, Qgis.Warning)
```

Messages(1): Missing Layers : Show Me

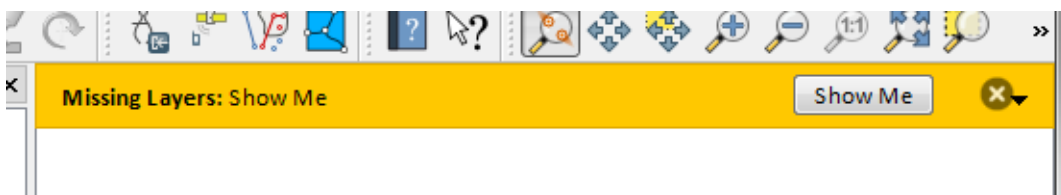


Fig. 13.4 – Barre de message QGis avec un bouton

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

```
1 class MyDialog(QDialog):
2     def __init__(self):
3         QDialog.__init__(self)
4         self.bar = QgsMessageBar()
5         self.bar.setSizePolicy( QSizePolicy.Minimum, QSizePolicy.Fixed )
6         self.setLayout(QGridLayout())
7         self.layout().setContentsMargins(0, 0, 0, 0)
8         self.buttonbox = QDialogButtonBox(QDialogButtonBox.Ok)
9         self.buttonbox.accepted.connect(self.run)
10        self.layout().addWidget(self.buttonbox, 0, 0, 2, 1)
11        self.layout().addWidget(self.bar, 0, 0, 1, 1)
12    def run(self):
13        self.bar.pushMessage("Hello", "World", level=Qgis.Info)
14
15 myDlg = MyDialog()
16 myDlg.show()
```

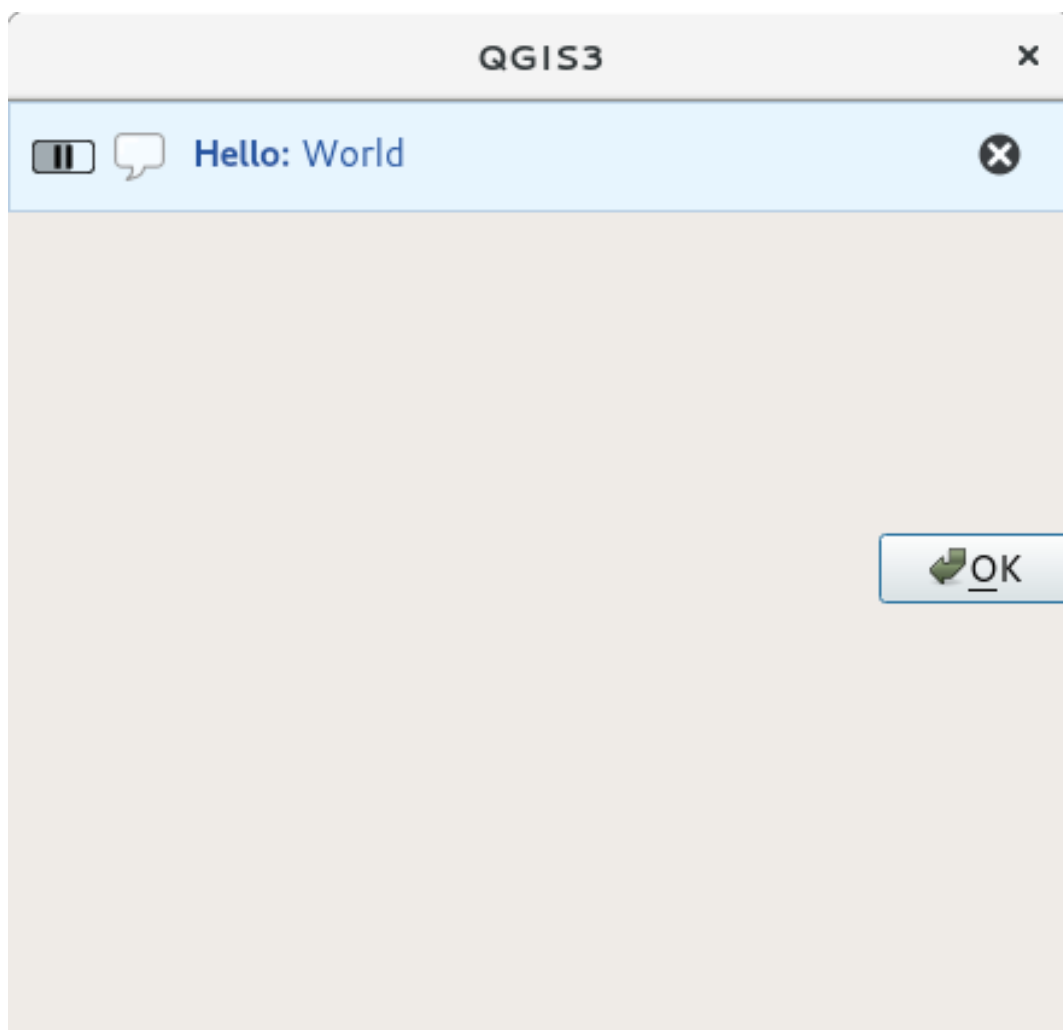


Fig. 13.5 – Barre de message QGis avec une boîte de dialogue personnalisée

## 13.2 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.

```

1 import time
2 from qgis.PyQt.QtWidgets import QProgressBar
3 from qgis.PyQt.QtCore import *
4 progressMessageBar = iface.messageBar().createMessage("Doing something boring...")
5 progress = QProgressBar()
6 progress.setMaximum(10)
7 progress.setAlignment(Qt.AlignLeft|Qt.AlignVCenter)
8 progressMessageBar.layout().addWidget(progress)
9 iface.messageBar().pushWidget(progressMessageBar, Qgis.Info)
10
11 for i in range(10):
12     time.sleep(1)
13     progress.setValue(i + 1)
14
15 iface.messageBar().clearWidgets()

```

```
Messages(0): Doing something boring...
```

Vous pouvez également utiliser la barre d'état intégrée pour signaler les progrès, comme dans l'exemple suivant :

```

1 vlayer = iface.activeLayer()
2
3 count = vlayer.featureCount()
4 features = vlayer.getFeatures()
5
6 for i, feature in enumerate(features):
7     # do something time-consuming here
8     print('.') # printing should give enough time to present the progress
9
10    percent = i / float(count) * 100
11    # iface.mainWindow().statusBar().showMessage("Processed {} %".
↪format(int(percent)))
12    iface.statusBarIface().showMessage("Processed {} %".format(int(percent)))
13
14 iface.statusBarIface().clearMessage()

```

## 13.3 Journal

Trois types de journalisation sont disponibles dans QGIS pour enregistrer et sauvegarder toutes les informations relatives à l'exécution de votre code. Chacun a son emplacement de sortie spécifique. Veuillez utiliser la méthode de journalisation qui convient le mieux à votre situation :

- `QgsMessageLog` est destiné aux messages pour communiquer des problèmes à l'utilisateur. La sortie du `QgsMessageLog` est affichée dans le panneau des messages de log.
- Le module python intégré **logging** est destiné au débogage au niveau de l'API Python QGIS (PyQGIS). Il est recommandé aux développeurs de scripts Python qui ont besoin de déboguer leur code python, par exemple les identifiants d'entités ou les géométries
- `QgsLogger` est destiné aux messages pour le débogage / les développements *QGIS interne* (c'est-à-dire que vous soupçonnez que quelque chose est déclenché par un code défectueux). Les messages ne sont visibles qu'avec les versions de QGIS destinées aux développeurs.

Des exemples pour les différents types d'exploitation forestière sont présentés dans les sections suivantes.

**Avertissement :** L'utilisation de l'instruction Python `print` n'est pas sûre dans tout code qui peut être multithreadé et **ralentit extrêmement l'algorithme**. Cela inclut les **fonctions d'expression**, les **renderers**, les **couches de symboles** et les **algorithmes de traitement** (entre autres). Dans ces cas, vous devriez toujours utiliser le module python **logging** ou les classes thread safe (`QgsLogger` ou `QgsMessageLog`) à la place.

### 13.3.1 QgsMessageLog

```
# 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)
```

```
MyPlugin(0): Your plugin code has been executed correctly
(1): Your plugin code might have some problems
(2): Your plugin code has crashed!
```

**Note :** Vous pouvez voir la sortie du `QgsMessageLog` dans `log_message_panel`.

### 13.3.2 Le python intégré dans le module de journalisation

```
1 import logging
2 formatter = '%(asctime)s - %(name)s - %(levelname)s - %(message)s'
3 logfilename=r'c:\temp\example.log'
4 logging.basicConfig(filename=logfilename, level=logging.DEBUG, format=formatter)
5 logging.info("This logging info text goes into the file")
6 logging.debug("This logging debug text goes into the file as well")
```

La méthode `basicConfig` permet de configurer la configuration de base de l'enregistrement. Dans le code ci-dessus, le nom de fichier, le niveau de journalisation et le format sont définis. Le nom du fichier fait référence à l'endroit où écrire le fichier journal, le niveau de journalisation définit les niveaux de sortie et le format définit le format dans lequel chaque message est sorti.

```
2020-10-08 13:14:42,998 - root - INFO - This logging text goes into the file
2020-10-08 13:14:42,998 - root - DEBUG - This logging debug text goes into the_
→file as well
```

Si vous voulez effacer le fichier journal chaque fois que vous exécutez votre script, vous pouvez faire quelque chose comme

```
if os.path.isfile(logfilename):
    with open(logfilename, 'w') as file:
        pass
```

D'autres ressources sur la façon d'utiliser le système d'exploitation forestière python sont disponibles à l'adresse suivante

- <https://docs.python.org/3/library/logging.html>
- <https://docs.python.org/3/howto/logging.html>
- <https://docs.python.org/3/howto/logging-cookbook.html>

**Avertissement :** Veuillez noter que sans enregistrement dans un fichier en définissant un nom de fichier, l'enregistrement peut être multithreadé, ce qui ralentit fortement la sortie.

Les extraits de code sur cette page nécessitent les importations suivantes si vous êtes en dehors de la console pyqgis :

```
1 from qgis.core import (
2     QgsApplication,
3     QgsRasterLayer,
4     QgsAuthMethodConfig,
5     QgsDataSourceUri,
6     QgsPkiBundle,
7     QgsMessageLog,
8 )
9
10 from qgis.gui import (
11     QgsAuthAuthoritiesEditor,
12     QgsAuthConfigEditor,
13     QgsAuthConfigSelect,
14     QgsAuthSettingsWidget,
15 )
16
17 from qgis.PyQt.QtWidgets import (
18     QWidget,
19     QTabWidget,
20 )
21
22 from qgis.PyQt.QtNetwork import QSslCertificate
```



---

## Infrastructure d'authentification

---

- *Introduction*
- *Glossaire*
- *QgsAuthManager le point d'entrée*
  - *Initier le gestionnaire et définir le mot de passe principal*
  - *Remplir authdb avec une nouvelle entrée de configuration d'authentification*
    - *Méthodes d'authentification disponibles*
    - *Populate Authorities*
    - *Manage PKI bundles with QgsPkiBundle*
  - *Supprimer une entrée de l'authdb*
  - *Laissez l'extension authcfg à QgsAuthManager*
    - *Exemples PKI avec d'autres fournisseurs de données*
- *Adapter les plugins pour utiliser l'infrastructure d'authentification*
- *Interfaces d'authentification*
  - *Fenêtre de sélection des identifiants*
  - *Authentication Editor GUI*
  - *Authorities Editor GUI*

### 14.1 Introduction

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

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

Dans QGIS Desktop, le système d'authentification est régulièrement utilisé par les fournisseurs de données lorsqu'une accréditation est nécessaire pour l'acquisition de certaines ressources, par exemple, lorsqu'une couche nécessite une connexion à une base Postgres.

Il existe aussi quelques widgets de la bibliothèque graphique de QGIS que les développeur de plugin peuvent utiliser pour intégrer facilement le système d'authentification dans leur code :

- `QgsAuthConfigEditor`
- `QgsAuthConfigSelect`
- `QgsAuthSettingsWidget`

Les `tests` dans le code source sont un bon exemple d'infrastructure d'authentification.

**Avertissement :** Du fait de contraintes de sécurité ayant conduit au design de l'infrastructure d'authentification, seules quelques méthodes internes sont exposées en Python.

## 14.2 Glossaire

Voici quelques définitions des principaux éléments étudiés dans ce chapitre.

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

**Base de données d'authentification** Une base de donnée sqlite `qgis-auth.db` cryptée des *Mot de passe principal* où les Configurations d'authentification sont stockées. Par exemple vos noms d'utilisateur et mots de passe, vos certificats et clés personnelles et plus généralement toutes vos méthodes d'authentification

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

**Configuration de l'authentification** Un ensemble de données d'authentification dépendant de la *Méthode d'authentification*. Par exemple, une méthode basique d'authentification enregistre le couple Nom d'utilisateur / Mot de passe.

**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 "authenticité". Chaque méthode est mise à disposition comme une librairie chargée dynamiquement pendant la phase d'initialisation de l'infrastructure d'authentification de QGIS.

## 14.3 QgsAuthManager le point d'entrée

Le singleton `QgsAuthManager` est le point d'entrée pour utiliser les informations d'identification stockées dans la base de données QGIS cryptée *Authentication DB*, c'est-à-dire le fichier `qgis-auth.db` sous le dossier actif `user profile`.

Cette classe s'occupe de l'interaction avec l'utilisateur : en lui demandant de définir un mot de passe maître ou en l'utilisant de manière transparente pour accéder à des informations stockées cryptées.

### 14.3.1 Initier le gestionnaire et définir le mot de passe principal

L'extrait suivant donne un exemple de définition d'un mot de passe principal pour ouvrir l'accès aux paramètres d'authentification. Les commentaires du code sont importants pour comprendre l'extrait.

```

1 authMgr = QgsApplication.authManager()
2
3 # check if QgsAuthManager has already been initialized... a side effect
4 # of the QgsAuthManager.init() is that AuthDbPath is set.
5 # QgsAuthManager.init() is executed during QGIS application init and hence
6 # you do not normally need to call it directly.
7 if authMgr.authenticationDatabasePath():
8     # already initialised => we are inside a QGIS app.
9     if authMgr.masterPasswordIsSet():
10         msg = 'Authentication master password not recognized'
11         assert authMgr.masterPasswordSame("your master password"), msg
12     else:
13         msg = 'Master password could not be set'
14         # The verify parameter check if the hash of the password was
15         # already saved in the authentication db
16         assert authMgr.setMasterPassword("your master password",

```

(suite sur la page suivante)

(suite de la page précédente)

```

17         verify=True), msg
18     else:
19         # outside qgis, e.g. in a testing environment => setup env var before
20         # db init
21         os.environ['QGIS_AUTH_DB_DIR_PATH'] = "/path/where/located/qgis-auth.db"
22         msg = 'Master password could not be set'
23         assert authMgr.setMasterPassword("your master password", True), msg
24         authMgr.init("/path/where/located/qgis-auth.db")

```

### 14.3.2 Remplir authdb avec une nouvelle entrée de configuration d'authentification

Tout justificatif stocké est une instance *Authentication Configuration* de la classe `QgsAuthMethodConfig` à laquelle on accède en utilisant une chaîne unique comme la suivante :

```
authcfg = 'fm1s770'
```

cette chaîne est générée automatiquement lors de la création d'une entrée en utilisant l'API ou l'interface graphique QGIS, mais il peut être utile de la définir manuellement à une valeur connue dans le cas où la configuration doit être partagée (avec des identifiants différents) entre plusieurs utilisateurs au sein d'une organisation.

`QgsAuthMethodConfig` est la classe de base pour tout Méthode d'authentification. Toute méthode d'authentification définit une carte de hachage de configuration où les informations d'authentification seront stockées. Ci-après, un extrait utile pour stocker les informations d'identification du chemin de l'ICP pour un utilisateur hypothétique :

```

1  authMgr = QgsApplication.authManager()
2  # set alice PKI data
3  config = QgsAuthMethodConfig()
4  config.setName("alice")
5  config.setMethod("PKI-Paths")
6  config.setUri("https://example.com")
7  config.setConfig("certpath", "path/to/alice-cert.pem" )
8  config.setConfig("keypath", "path/to/alice-key.pem" )
9  # check if method parameters are correctly set
10 assert config.isValid()
11
12 # register alice data in authdb returning the `authcfg` of the stored
13 # configuration
14 authMgr.storeAuthenticationConfig(config)
15 newAuthCfgId = config.id()
16 assert newAuthCfgId

```

### Méthodes d'authentification disponibles

Les bibliothèques de Méthode d'authentification sont chargées dynamiquement pendant l'initialisation du gestionnaire d'authentification. Les méthodes d'authentification disponibles sont :

1. Basique : authentification avec utilisateur et mot de passe
2. Esri-Token : authentification basée sur le jeton ESRI
3. Identity-Cert : authentification par certificat d'identité
4. OAuth2 : authentification OAuth2
5. PKI-Paths PKI paths authentication
6. PKI-PKCS#12 PKI PKCS#12 authentication

## Populate Authorities

```

1 authMgr = QgsApplication.authManager()
2 # add authorities
3 cacerts = QSslCertificate.fromPath( "/path/to/ca_chains.pem" )
4 assert cacerts is not None
5 # store CA
6 authMgr.storeCertAuthorities(cacerts)
7 # and rebuild CA caches
8 authMgr.rebuildCaCertsCache()
9 authMgr.rebuildTrustedCaCertsCache()

```

## Manage PKI bundles with QgsPkiBundle

La classe `QgsPkiBundle` est une classe de commodité permettant d'emballer des paquets d'ICP composés sur la chaîne `SslCert`, `SslKey` et `CA`. Ci-après, un extrait pour obtenir la protection par mot de passe :

```

1 # add alice cert in case of key with pwd
2 caBundlesList = [] # List of CA bundles
3 bundle = QgsPkiBundle.fromPemPaths( "/path/to/alice-cert.pem",
4                                     "/path/to/alice-key_w-pass.pem",
5                                     "unlock_pwd",
6                                     caBundlesList )
7 assert bundle is not None
8 # You can check bundle validity by calling:
9 # bundle.isValid()

```

Référez-vous à la documentation de la classe `QgsPkiBundle` pour extraire les cert/clés/CA du bundle.

### 14.3.3 Supprimer une entrée de l'authdb

Nous pouvons supprimer une entrée de *Base de données d'Authentification* en utilisant son identifiant `authcfg` avec le code suivant :

```

authMgr = QgsApplication.authManager()
authMgr.removeAuthenticationConfig( "authCfg_Id_to_remove" )

```

### 14.3.4 Laissez l'extension authcfg à QgsAuthManager

La meilleure façon d'utiliser un *Authentication Config* stocké dans la Base de données d'Authentification est de le référencer avec l'identifiant unique `authcfg`. L'expansion, signifie le convertir d'un identifiant en un ensemble complet de justificatifs d'identité. La meilleure façon d'utiliser la Configuration d'authentification stockée est de la laisser gérée automatiquement par le gestionnaire d'authentification. L'utilisation courante d'une configuration stockée est de se connecter à un service d'authentification comme un WMS ou un WFS ou à une base de données.

**Note :** Tenez compte du fait que tous les provider de données QGIS ne sont pas intégrés à l'infrastructure d'authentification. Chaque méthode d'authentification, dérivée de la classe de base `QgsAuthMethod` et prend en charge un ensemble différent de provider. Par exemple, la méthode `certIdentity()` prend en charge la liste de provider suivante :

```

authM = QgsApplication.authManager()
print(authM.authMethod("Identity-Cert").supportedDataProviders())

```

Exemple de sortie :

```
['ows', 'wfs', 'wcs', 'wms', 'postgres']
```

Par exemple, pour accéder à un service WMS en utilisant des informations d'identification stockées identifiées par `authcfg = 'fm1s770'`, il suffit d'utiliser `authcfg` dans l'URL de la source de données comme dans l'extrait suivant :

```
1 authCfg = 'fm1s770'
2 quri = QgsDataSourceUri()
3 quri.setParam("layers", 'usa:states')
4 quri.setParam("styles", '')
5 quri.setParam("format", 'image/png')
6 quri.setParam("crs", 'EPSG:4326')
7 quri.setParam("dpiMode", '7')
8 quri.setParam("featureCount", '10')
9 quri.setParam("authcfg", authCfg) # <---- here my authCfg url parameter
10 quri.setParam("contextualWMSLegend", '0')
11 quri.setParam("url", 'https://my_auth_enabled_server_ip/wms')
12 rlayer = QgsRasterLayer(str(quri.encodedUri(), "utf-8"), 'states', 'wms')
```

En majuscules, le provider `wms` prendra soin d'étendre le paramètre URI `authcfg` avec l'identifiant juste avant d'établir la connexion HTTP.

**Avertissement :** Le développeur devra laisser l'extension `authcfg` à la `QgsAuthManager`, de cette façon il sera sûr que l'extension ne sera pas faite trop tôt.

Habituellement, une chaîne URI, construite en utilisant la classe `QgsDataSourceURI`, est utilisée pour définir une source de données de la manière suivante :

```
authCfg = 'fm1s770'
quri = QgsDataSourceUri("my WMS uri here")
quri.setParam("authcfg", authCfg)
rlayer = QgsRasterLayer(quri.uri(False), 'states', 'wms')
```

**Note :** Le paramètre `False` est important pour éviter l'expansion complète de l'URI de l'id `authcfg` présent dans l'URI.

## Exemples PKI avec d'autres fournisseurs de données

Un autre exemple peut être lu directement dans les tests QGIS en amont comme dans `test_authmanager_pki_ows` ou `test_authmanager_pki_postgres`.

## 14.4 Adapter les plugins pour utiliser l'infrastructure d'authentification

De nombreux plugins tiers utilisent `httplib2` ou d'autres bibliothèques réseau Python pour gérer les connexions HTTP au lieu de s'intégrer avec `QgsNetworkAccessManager` et l'intégration de l'infrastructure d'authentification correspondante.

Pour faciliter cette intégration, une fonction d'aide Python a été créée, appelée « `NetworkAccessManager` ». Son code se trouve [ici](#).

Cette classe d'aide peut être utilisée comme dans l'extrait suivant :

```

1 http = NetworkAccessManager(authid="my_authCfg", exception_class=My_
  ↳FailedRequestError)
2 try:
3     response, content = http.request( "my_rest_url" )
4 except My_FailedRequestError, e:
5     # Handle exception
6     pass

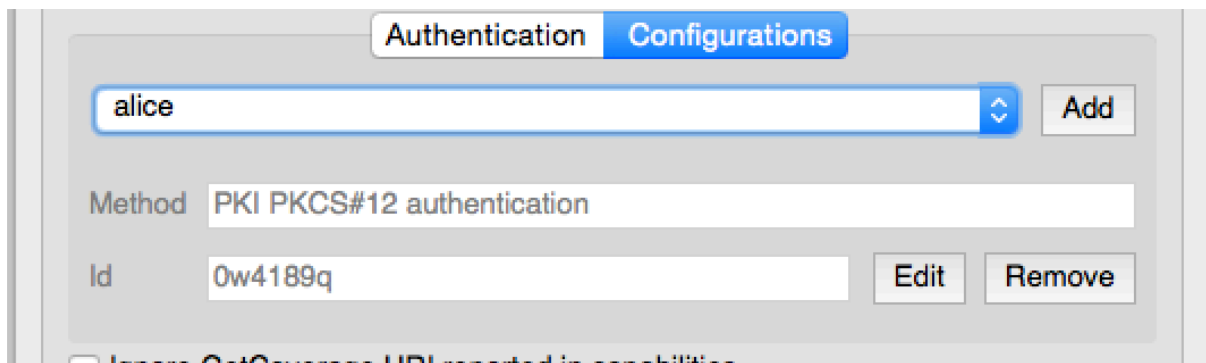
```

## 14.5 Interfaces d'authentification

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

### 14.5.1 Fenêtre de sélection des identifiants

S'il est nécessaire de sélectionner une configuration d'authentification à partir du jeu stocké dans la *Base de données d'authentification*, il est disponible dans la classe d'interface `QgsAuthConfigSelect`.



et peut être utilisé comme dans l'extrait suivant :

```

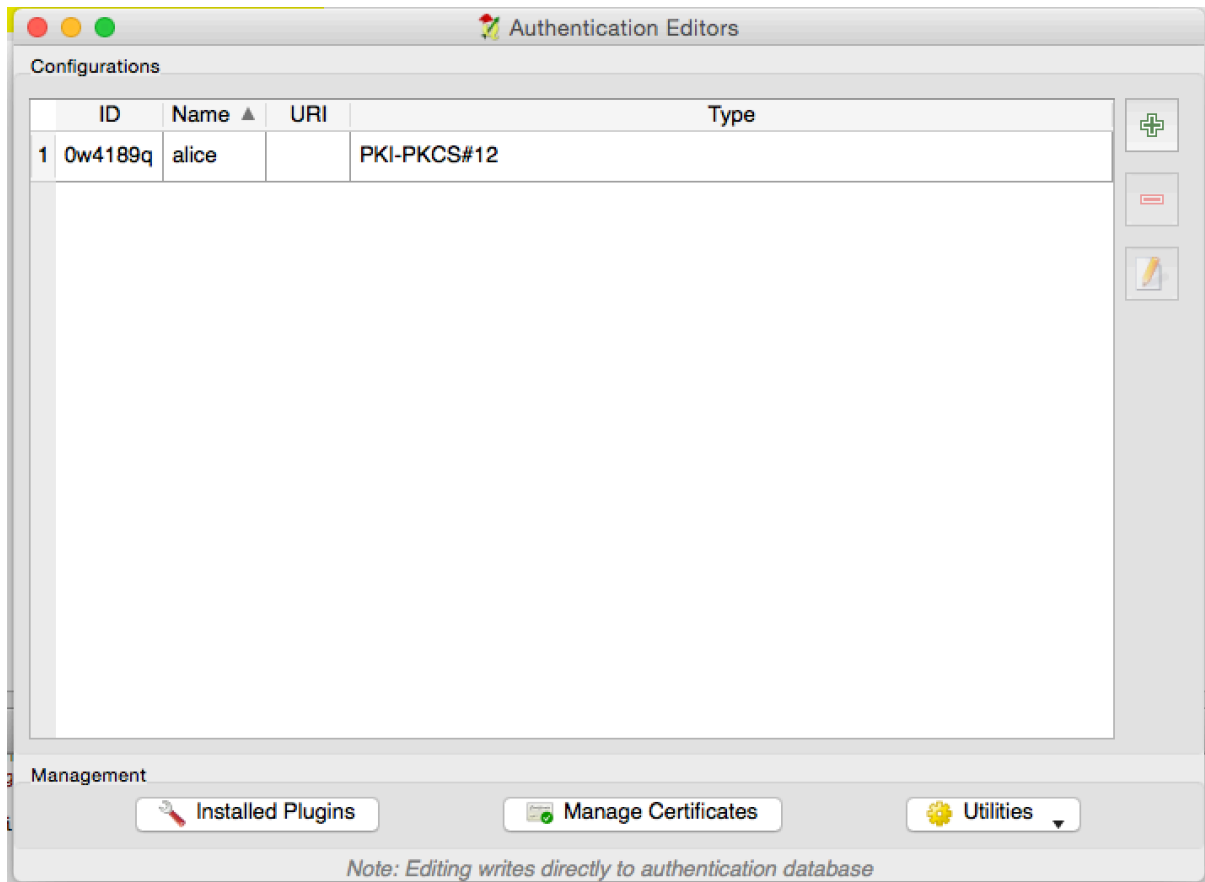
1 # create the instance of the QgsAuthConfigSelect GUI hierarchically linked to
2 # the widget referred with `parent`
3 parent = QWidget() # Your GUI parent widget
4 gui = QgsAuthConfigSelect( parent, "postgres" )
5 # add the above created gui in a new tab of the interface where the
6 # GUI has to be integrated
7 tabGui = QTabWidget()
8 tabGui.insertTab( 1, gui, "Configurations" )

```

L'exemple ci-dessus est tiré du code source de QGIS. Le deuxième paramètre du constructeur de l'interface graphique se réfère au type de fournisseur de données. Ce paramètre est utilisé pour restreindre la compatibilité de la Méthode d'authentification avec le fournisseur spécifié.

## 14.5.2 Authentication Editor GUI

L'interface graphique complète utilisée pour gérer les références, les autorités et pour accéder aux utilitaires d'authentification est gérée par la classe `QgsAuthEditorWidgets`.



et peut être utilisé comme dans l'extrait suivant :

```

1 # create the instance of the QgsAuthEditorWidgets GUI hierarchically linked to
2 # the widget referred with `parent`
3 parent = QWidget() # Your GUI parent widget
4 gui = QgsAuthConfigSelect( parent )
5 gui.show()

```

Un exemple intégré peut être trouvé dans le fichier `test`.

## 14.5.3 Authorities Editor GUI

Une interface graphique utilisée pour gérer uniquement les autorités est gérée par la classe `QgsAuthAuthoritiesEditor`.

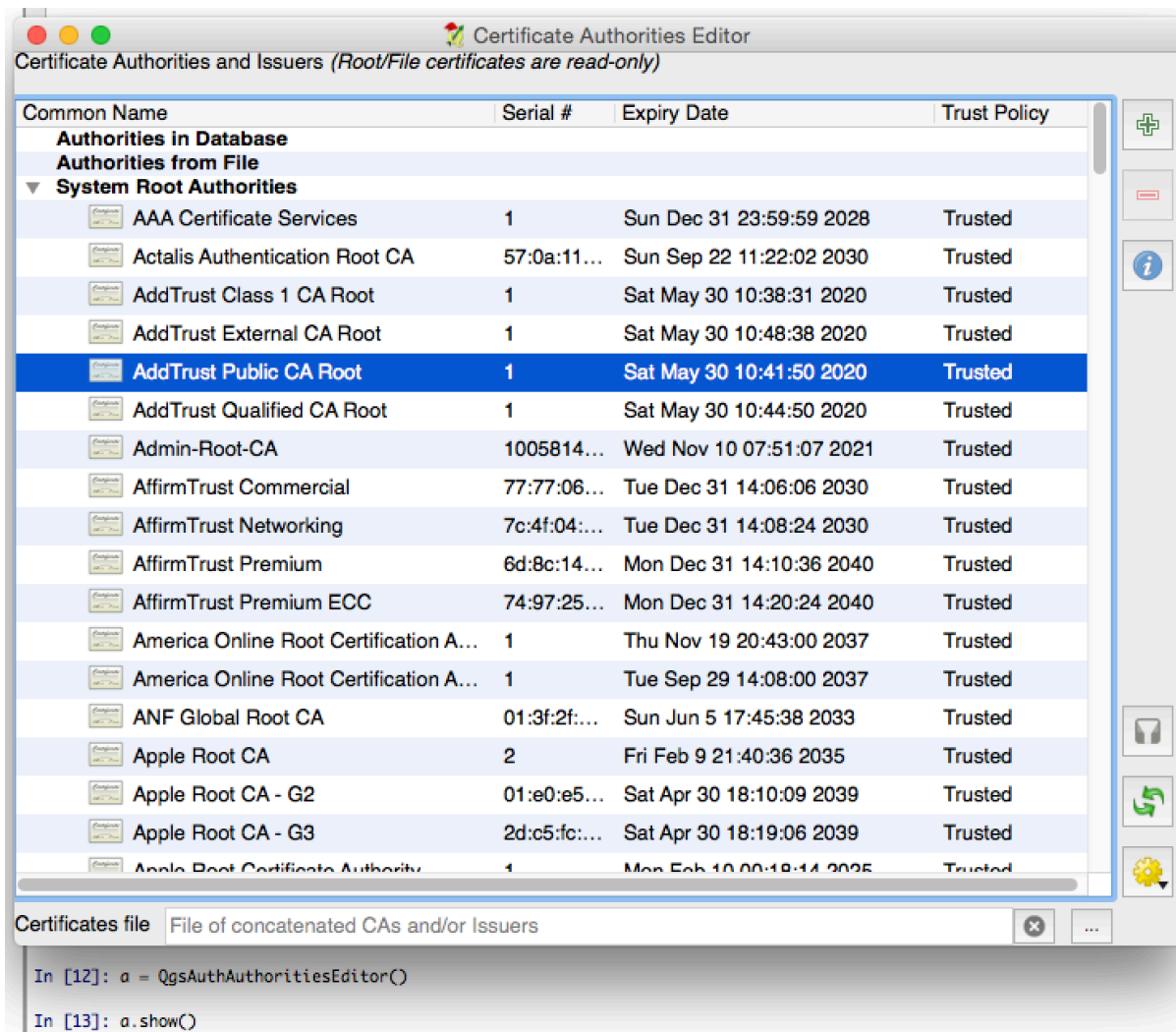
et peut être utilisé comme dans l'extrait suivant :

```

1 # create the instance of the QgsAuthAuthoritiesEditor GUI hierarchically
2 # linked to the widget referred with `parent`
3 parent = QWidget() # Your GUI parent widget
4 gui = QgsAuthAuthoritiesEditor( parent )
5 gui.show()

```

Les extraits de code sur cette page nécessitent les importations suivantes si vous êtes en dehors de la console pyqgis :





```
1 from qgis.core import (  
2     QgsProcessingContext,  
3     QgsTaskManager,  
4     QgsTask,  
5     QgsProcessingAlgRunnerTask,  
6     Qgs,  
7     QgsProcessingFeedback,  
8     QgsApplication,  
9     QgsMessageLog,  
10 )
```



## Tâches - faire un gros travail en arrière-plan

### 15.1 Introduction

Le traitement en arrière-plan à l'aide de threads est un moyen de maintenir une interface utilisateur réactive en cas de traitement lourd. Les tâches peuvent être utilisées pour réaliser des threads dans QGIS.

Une tâche (`QgsTaskManager`) est un conteneur pour le code à exécuter en arrière-plan, et le gestionnaire de tâches (`QgsTaskManager`) est utilisé pour contrôler l'exécution des tâches. Ces classes simplifient le traitement en arrière-plan dans QGIS en fournissant des mécanismes de signalisation, de rapport d'avancement et d'accès à l'état des processus en arrière-plan. Les tâches peuvent être regroupées à l'aide de sous-tâches.

Le gestionnaire de tâches global (trouvé avec `QgsApplication.taskManager()`) est normalement utilisé. Cela signifie que vos tâches peuvent ne pas être les seules à être contrôlées par le gestionnaire de tâches.

Il existe plusieurs façons de créer une tâche QGIS :

- Créez votre propre tâche en étendant `QgsTask`.

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

- Créer une tâche à partir d'une fonction

```
1 def heavyFunction():
2     # Some CPU intensive processing ...
3     pass
4
5 def workdone():
6     # ... do something useful with the results
7     pass
8
9 task = QgsTask.fromFunction('heavy function', heavyFunction,
10                            onfinished=workdone)
```

- Créer une tâche à partir d'un algorithme de traitement

```
1 params = dict()
2 context = QgsProcessingContext()
3 feedback = QgsProcessingFeedback()
4
5 buffer_alg = QgsApplication.instance().processingRegistry().algorithmById(
6     ↪ 'native:buffer')
```

(suite sur la page suivante)

(suite de la page précédente)

```

6 task = QgsProcessingAlgRunnerTask(buffer_alg, params, context,
7                                 feedback)

```

**Avertissement :** Toute tâche en arrière-plan (quelle que soit la façon dont elle est créée) ne doit JAMAIS utiliser un QObject qui vit sur le thread principal, comme l'accès à QgsVectorLayer, QgsProject ou effectuer des opérations basées sur une interface graphique comme la création de nouveaux widgets ou l'interaction avec des widgets existants. Les widgets Qt ne doivent être accessibles ou modifiés que depuis le fil principal. Les données utilisées dans une tâche doivent être copiées avant que la tâche ne soit lancée. Tenter de les utiliser à partir des fils de discussion en arrière-plan entraînera des plantages.

Les dépendances entre les tâches peuvent être décrites en utilisant la fonction `addSubTask` de `QgsTask`. Lorsqu'une dépendance est indiquée, le gestionnaire de tâches détermine automatiquement comment ces dépendances seront exécutées. Dans la mesure du possible, les dépendances seront exécutées en parallèle afin de les satisfaire le plus rapidement possible. Si une tâche dont dépend une autre tâche est annulée, la tâche dépendante sera également annulée. Les dépendances circulaires peuvent rendre possible des blocages, soyez donc prudent.

Si une tâche dépend de la disponibilité d'une couche, cela peut être indiqué en utilisant la fonction `setDependentLayers` de `QgsTask`. Si une couche dont dépend une tâche n'est pas disponible, la tâche sera annulée.

Une fois que la tâche a été créée, elle peut être programmée pour s'exécuter en utilisant la fonction `addTask` du gestionnaire de tâches. L'ajout d'une tâche au gestionnaire transfère automatiquement la propriété de cette tâche au gestionnaire, et le gestionnaire nettoiera et supprimera les tâches après leur exécution. La planification des tâches est influencée par la priorité des tâches, qui est définie dans `addTask`.

L'état des tâches peut être surveillé en utilisant les signaux et fonctions `QgsTask` et `QgsTaskManager`.

## 15.2 Exemples

### 15.2.1 Extension de QgsTask

Dans cet exemple, `RandomIntegerSumTask` étend `QgsTask` et va générer 100 entiers aléatoires entre 0 et 500 pendant une période de temps spécifiée. Si le nombre aléatoire est de 42, la tâche est abandonnée et une exception est levée. Plusieurs instances de `RandomIntegerSumTask` (avec des sous-tâches) sont générées et ajoutées au gestionnaire de tâches, démontrant ainsi deux types de dépendances.

```

1 import random
2 from time import sleep
3
4 from qgis.core import (
5     QgsApplication, QgsTask, QgsMessageLog,
6     )
7
8 MESSAGE_CATEGORY = 'RandomIntegerSumTask'
9
10 class RandomIntegerSumTask(QgsTask):
11     """This shows how to subclass QgsTask"""
12
13     def __init__(self, description, duration):
14         super().__init__(description, QgsTask.CanCancel)
15         self.duration = duration
16         self.total = 0
17         self.iterations = 0
18         self.exception = None
19
20     def run(self):
21         """Here you implement your heavy lifting.

```

(suite sur la page suivante)

(suite de la page précédente)

```

22     Should periodically test for isCanceled() to gracefully
23     abort.
24     This method MUST return True or False.
25     Raising exceptions will crash QGIS, so we handle them
26     internally and raise them in self.finished
27     """
28     QgsMessageLog.logMessage('Started task "{}".format(
29         self.description()),
30         MESSAGE_CATEGORY, Qgis.Info)
31     wait_time = self.duration / 100
32     for i in range(100):
33         sleep(wait_time)
34         # use setProgress to report progress
35         self.setProgress(i)
36         arandominteger = random.randint(0, 500)
37         self.total += arandominteger
38         self.iterations += 1
39         # check isCanceled() to handle cancellation
40         if self.isCanceled():
41             return False
42         # simulate exceptions to show how to abort task
43         if arandominteger == 42:
44             # DO NOT raise Exception('bad value!')
45             # this would crash QGIS
46             self.exception = Exception('bad value!')
47             return False
48     return True
49
50     def finished(self, result):
51         """
52         This function is automatically called when the task has
53         completed (successfully or not).
54         You implement finished() to do whatever follow-up stuff
55         should happen after the task is complete.
56         finished is always called from the main thread, so it's safe
57         to do GUI operations and raise Python exceptions here.
58         result is the return value from self.run.
59         """
60         if result:
61             QgsMessageLog.logMessage(
62                 'RandomTask "{}" completed\n' \
63                 'RandomTotal: {total} (with {iterations} '\
64                 'iterations)'.format(
65                     name=self.description(),
66                     total=self.total,
67                     iterations=self.iterations),
68                 MESSAGE_CATEGORY, Qgis.Success)
69         else:
70             if self.exception is None:
71                 QgsMessageLog.logMessage(
72                     'RandomTask "{}" not successful but without '\
73                     'exception (probably the task was manually '\
74                     'canceled by the user)'.format(
75                         name=self.description()),
76                     MESSAGE_CATEGORY, Qgis.Warning)
77             else:
78                 QgsMessageLog.logMessage(
79                     'RandomTask "{}" Exception: {exception}'.format(
80                         name=self.description(),
81                         exception=self.exception),
82                     MESSAGE_CATEGORY, Qgis.Critical)

```

(suite sur la page suivante)

```

83         raise self.exception
84
85     def cancel(self):
86         QgsMessageLog.logMessage(
87             'RandomTask "{name}" was canceled'.format(
88                 name=self.description()),
89             MESSAGE_CATEGORY, QgsInfo)
90         super().cancel()
91
92
93 longtask = RandomIntegerSumTask('waste cpu long', 20)
94 shorttask = RandomIntegerSumTask('waste cpu short', 10)
95 minitask = RandomIntegerSumTask('waste cpu mini', 5)
96 shortsubtask = RandomIntegerSumTask('waste cpu subtask short', 5)
97 longsubtask = RandomIntegerSumTask('waste cpu subtask long', 10)
98 shortestsubtask = RandomIntegerSumTask('waste cpu subtask shortest', 4)
99
100 # Add a subtask (shortsubtask) to shorttask that must run after
101 # minitask and longtask has finished
102 shorttask.addSubTask(shortsubtask, [minitask, longtask])
103 # Add a subtask (longsubtask) to longtask that must be run
104 # before the parent task
105 longtask.addSubTask(longsubtask, [], QgsTask.ParentDependsOnSubTask)
106 # Add a subtask (shortestsubtask) to longtask
107 longtask.addSubTask(shortestsubtask)
108
109 QgsApplication.taskManager().addTask(longtask)
110 QgsApplication.taskManager().addTask(shorttask)
111 QgsApplication.taskManager().addTask(minitask)

```

```

1 RandomIntegerSumTask(0): Started task "waste cpu subtask shortest"
2 RandomIntegerSumTask(0): Started task "waste cpu short"
3 RandomIntegerSumTask(0): Started task "waste cpu mini"
4 RandomIntegerSumTask(0): Started task "waste cpu subtask long"
5 RandomIntegerSumTask(3): Task "waste cpu subtask shortest" completed
6 RandomTotal: 25452 (with 100 iterations)
7 RandomIntegerSumTask(3): Task "waste cpu mini" completed
8 RandomTotal: 23810 (with 100 iterations)
9 RandomIntegerSumTask(3): Task "waste cpu subtask long" completed
10 RandomTotal: 26308 (with 100 iterations)
11 RandomIntegerSumTask(0): Started task "waste cpu long"
12 RandomIntegerSumTask(3): Task "waste cpu long" completed
13 RandomTotal: 22534 (with 100 iterations)

```

## 15.2.2 Tâche de la fonction

Créer une tâche à partir d'une fonction (doSomething dans cet exemple). Le premier paramètre de la fonction contiendra la `QgsTask` pour la fonction. Un paramètre important (nommé) est `on_finished`, qui spécifie une fonction qui sera appelée lorsque la tâche sera terminée. La fonction `doSomething` dans cet exemple a un paramètre supplémentaire nommé `wait_time`.

```

1 import random
2 from time import sleep
3
4 MESSAGE_CATEGORY = 'TaskFromFunction'
5
6 def doSomething(task, wait_time):
7     """
8     Raises an exception to abort the task.

```

(suite sur la page suivante)

(suite de la page précédente)

```

9     Returns a result if success.
10    The result will be passed, together with the exception (None in
11    the case of success), to the on_finished method.
12    If there is an exception, there will be no result.
13    """
14    QgsMessageLog.logMessage('Started task {}'.format(task.description()),
15                             MESSAGE_CATEGORY, Qgis.Info)
16    wait_time = wait_time / 100
17    total = 0
18    iterations = 0
19    for i in range(100):
20        sleep(wait_time)
21        # use task.setProgress to report progress
22        task.setProgress(i)
23        arandominteger = random.randint(0, 500)
24        total += arandominteger
25        iterations += 1
26        # check task.isCanceled() to handle cancellation
27        if task.isCanceled():
28            stopped(task)
29            return None
30        # raise an exception to abort the task
31        if arandominteger == 42:
32            raise Exception('bad value!')
33    return {'total': total, 'iterations': iterations,
34           'task': task.description()}
35
36 def stopped(task):
37     QgsMessageLog.logMessage(
38         'Task "{name}" was canceled'.format(
39             name=task.description()),
40         MESSAGE_CATEGORY, Qgis.Info)
41
42 def completed(exception, result=None):
43     """This is called when doSomething is finished.
44     Exception is not None if doSomething raises an exception.
45     result is the return value of doSomething."""
46     if exception is None:
47         if result is None:
48             QgsMessageLog.logMessage(
49                 'Completed with no exception and no result '\
50                 '(probably manually canceled by the user)',
51                 MESSAGE_CATEGORY, Qgis.Warning)
52         else:
53             QgsMessageLog.logMessage(
54                 'Task {name} completed\n'
55                 'Total: {total} ( with {iterations} '
56                 'iterations)'.format(
57                     name=result['task'],
58                     total=result['total'],
59                     iterations=result['iterations']),
60                 MESSAGE_CATEGORY, Qgis.Info)
61     else:
62         QgsMessageLog.logMessage("Exception: {}".format(exception),
63                                   MESSAGE_CATEGORY, Qgis.Critical)
64     raise exception
65
66 # Create a few tasks
67 task1 = QgsTask.fromFunction('Waste cpu 1', doSomething,
68                              on_finished=completed, wait_time=4)
69 task2 = QgsTask.fromFunction('Waste cpu 2', doSomething,

```

(suite sur la page suivante)

```

70         on_finished=completed, wait_time=3)
71 QgsApplication.taskManager().addTask(task1)
72 QgsApplication.taskManager().addTask(task2)

```

```

1 RandomIntegerSumTask(0): Started task "waste cpu subtask short"
2 RandomTaskFromFunction(0): Started task Waste cpu 1
3 RandomTaskFromFunction(0): Started task Waste cpu 2
4 RandomTaskFromFunction(0): Task Waste cpu 2 completed
5 RandomTotal: 23263 ( with 100 iterations)
6 RandomTaskFromFunction(0): Task Waste cpu 1 completed
7 RandomTotal: 25044 ( with 100 iterations)

```

### 15.2.3 Tâche à partir d'un algorithme de traitement

Créer une tâche qui utilise l'algorithme `qgis:randompointsinextent` pour générer 50000 points aléatoires à l'intérieur d'une étendue spécifiée. Le résultat est ajouté au projet de manière sûre.

```

1 from functools import partial
2 from qgis.core import (QgsTaskManager, QgsMessageLog,
3                       QgsProcessingAlgRunnerTask, QgsApplication,
4                       QgsProcessingContext, QgsProcessingFeedback,
5                       QgsProject)
6
7 MESSAGE_CATEGORY = 'AlgRunnerTask'
8
9 def task_finished(context, successful, results):
10     if not successful:
11         QgsMessageLog.logMessage('Task finished unsuccessfully',
12                                 MESSAGE_CATEGORY, Qgs.Warning)
13     output_layer = context.getMapLayer(results['OUTPUT'])
14     # because getMapLayer doesn't transfer ownership, the layer will
15     # be deleted when context goes out of scope and you'll get a
16     # crash.
17     # takeMapLayer transfers ownership so it's then safe to add it
18     # to the project and give the project ownership.
19     if output_layer and output_layer.isValid():
20         QgsProject.instance().addMapLayer(
21             context.takeResultLayer(output_layer.id()))
22
23 alg = QgsApplication.processingRegistry().algorithmById(
24     'qgis:randompointsinextent')
25 context = QgsProcessingContext()
26 feedback = QgsProcessingFeedback()
27 params = {
28     'EXTENT': '0.0,10.0,40,50 [EPSG:4326]',
29     'MIN_DISTANCE': 0.0,
30     'POINTS_NUMBER': 50000,
31     'TARGET_CRS': 'EPSG:4326',
32     'OUTPUT': 'memory:My random points'
33 }
34 task = QgsProcessingAlgRunnerTask(alg, params, context, feedback)
35 task.executed.connect(partial(task_finished, context))
36 QgsApplication.taskManager().addTask(task)

```

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



### 16.1 Structurer les plugins Python

- *Ecriture d'un plugin*
  - *Fichiers de l'extension*
- *Contenu de l'extension*
  - *Métadonnées de l'extension*
  - *\_\_init\_\_.py*
  - *mainPlugin.py*
  - *Fichier de ressources*
- *Documentation*
- *Traduction*
  - *Exigences en matière de logiciels*
  - *Fichiers et répertoire*
    - *Fichier .pro*
    - *Fichier .ts*
    - *Fichier .qm*
  - *Traduction en utilisant un fichier Makefile*
  - *Chargement de l'extension*
- *Conseils et Astuces*
  - *Rechargeur de plugins*
  - *Accès aux plugins*
  - *Messages log*
  - *Partage de votre extension*

Voici quelques étapes à suivre afin de créer des extensions :

1. *Idée* : Ayez une idée de ce que vous voulez faire avec votre nouveau plugin QGIS. Pourquoi le faites-vous ? Quel problème voulez-vous résoudre ? Existe-t-il déjà un autre plugin pour ce problème ?
2. *Créer des fichiers* : certains sont essentiels (voir *Fichiers de l'extension*)
3. *Écrire le code* : écrire le code dans les fichiers appropriés
4. *Tester* : *Rechargez votre extension* pour vérifier si tout est OK
5. *Publier* : Publiez votre plugin dans le dépôt QGIS ou créez votre propre dépôt comme un « arsenal » d'« armes SIG » personnelles.

## 16.1.1 Ecriture d'un plugin

Depuis l'introduction des plugins Python dans QGIS, un certain nombre de plugins sont apparus. L'équipe QGIS maintient un *Dépôt officiel des extensions QGIS*. Vous pouvez utiliser leur source pour en savoir plus sur la programmation avec PyQGIS ou découvrir si vous dupliquez l'effort de développement.

### Fichiers de l'extension

Voici la structure du dossier pour notre plugin servant d'exemple

```
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*
```

Quelle est la signification de ces fichiers :

- `__init__.py` = Le point de départ du plugin. Il doit avoir la méthode `classFactory()` et peut avoir n'importe quel autre code d'initialisation.
- `mainPlugin.py` = Le code de travail principal du plugin. Contient toutes les informations sur les actions du plugin et le code principal.
- `.file :resources.qrc` = Le document `.xml` créé par Qt Designer. Contient les chemins relatifs aux ressources des formulaires.
- `.file :resources.py` = La traduction en Python du fichier `.qrc` décrit ci-dessus.
- `form.ui` = l'interface conçue sur Qt Designer
- `form.py` = La traduction en Python du fichier `form.ui` décrit ci-dessus.
- `metadata.txt` = Contient des informations générales, la version, le nom et quelques autres métadonnées utilisées par le site web des plugins et l'infrastructure des plugins.

Ici est un moyen de créer les fichiers de base (squelette) d'un plugin Python QGIS typique.

Il existe un plugin QGIS appelé [Plugin Builder 3](#) qui crée un modèle de plugin pour QGIS. C'est l'option recommandée, car il produit des sources compatibles 3.x.

**Avertissement :** Si vous prévoyez de télécharger le plugin dans le *Dépôt officiel des extensions QGIS* vous devez vérifier que votre plugin suit certaines règles supplémentaires, requises pour le plugin *Validation*

## 16.1.2 Contenu de l'extension

Vous trouverez ici des informations et des exemples sur ce qu'il faut ajouter dans chacun des fichiers de la structure décrite ci-dessus.

### Métadonnées de l'extension

Tout d'abord, le gestionnaire de plugin doit récupérer quelques informations de base sur le plugin, comme son nom, sa description, etc. Le fichier `metadata.txt` est le bon endroit pour mettre ces informations.

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

Nom des métadonnées	Requis	Notes
Nom	Vrai	une courte chaîne de caractères contenant le nom du plugin
Version minimale de QGIS	Vrai	notation en pointillés de la version minimale de QGIS
Version maximale de QGIS	Faux	notation en pointillés de la version maximale de QGIS
description	Vrai	texte court qui décrit le plugin, pas d'HTML autorisé
au sujet	Vrai	texte plus long qui décrit le plugin en détail, pas de HTML autorisé
version	Vrai	chaîne courte avec la notation en pointillés de la version
auteur	Vrai	nom de l'auteur
e-mail	Vrai	e-mail de l'auteur, uniquement affiché sur le site web pour les utilisateurs connectés, mais visible dans le gestionnaire de plugin après l'installation du plugin
Suivi des modifications	Faux	de type texte, indique les modifications de version. Peut être multiligne, le HTML n'est pas autorisé
experimental	Faux	booléen, <code>True</code> ou <code>False</code> , indiquant si cette version est expérimentale
deprecated	Faux	booléen, <code>True</code> ou <code>False</code> permettant de savoir si l'extension est obsolète ou pas. S'applique à toute l'extension.
les mots-clé	Faux	liste séparée par des virgules, les espaces sont autorisés à l'intérieur des balises individuelles
page d'accueil	Faux	une URL valide pointant vers la page d'accueil de votre plugin
repository	Vrai	une URL valide du dépôt du code
tracker	Faux	une URL valide pour le signalement des bugs et demandes
icon	Faux	un nom de fichier ou un chemin d'accès relatif (par rapport au dossier de base du paquet compressé du plugin) d'une image accessible sur le web (PNG, JPEG)
category	Faux	soit <code>Raster</code> , <code>Vector</code> , <code>Database</code> ou <code>Web</code>
plugin_dependencies	Faux	Liste des autres plugins à installer, séparés par des virgules de type PIP
serveur	Faux	le drapeau booléen, « <code>true</code> » ou « <code>false</code> », détermine si le plugin a une interface serveur
hasProcessing-Provider	Faux	le drapeau booléen, « <code>true</code> » ou « <code>false</code> », détermine si le plugin fournit des algorithmes de traitement

Par défaut, les plugins sont placés dans le menu *Plugins* (nous verrons dans la section suivante comment ajouter une entrée de menu pour votre plugin) mais ils peuvent aussi être placés dans les menus *Raster*, *Vecteur*, *base de données* et *Web*.

Une entrée de métadonnées « catégorie » correspondante existe pour le spécifier, de sorte que le plugin peut être classé en conséquence. Cette entrée de métadonnées sert de conseil aux utilisateurs et leur indique où (dans quel menu) se trouve le plugin. Les valeurs autorisées pour « category » sont : `Vecteur`, `Raster`, `Base de données` ou `Web`. Par exemple, si votre plugin sera disponible à partir du menu « Raster », ajoutez ceci à `metadata.txt`.

```
category=Raster
```

**Note :** Si `qgisMaximumVersion` est vide, il sera automatiquement mis à la version majeure plus `.99` lorsqu'il sera téléchargé dans le *Dépôt officiel des extensions QGIS*.

---

Un exemple pour ce fichier 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

; Since QGIS 3.8, a comma separated list of plugins to be installed
; (or upgraded) can be specified.
; The example below will try to install (or upgrade) "MyOtherPlugin" version 1.12
; and any version of "YetAnotherPlugin"
plugin_dependencies=MyOtherPlugin==1.12,YetAnotherPlugin
```

## \_\_init\_\_.py

Ce fichier est requis par le système d'importation de Python. De plus, QGIS exige que ce fichier contienne une fonction `classFactory()`, qui est appelée lorsque le plugin est chargé dans QGIS. Elle reçoit une référence à l'instance de `QgisInterface` et doit retourner un objet de la classe de votre plugin à partir du `mainplugin.py` — dans notre cas, il s'appelle `TestPlugin` (voir ci-dessous). Voici à quoi doit ressembler le fichier `__init__.py`.

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

# any other initialisation needed
```

## mainPlugin.py

C'est là que la magie se produit et c'est à cela que ressemble la magie : (par exemple `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 from 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):
```

(suite sur la page suivante)

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

Les seules fonctions de plugin qui doivent exister dans le fichier source du plugin principal (par exemple `mainPlugin.py`) sont :

- « `__init__` » qui permet l'accès à l'interface de QGIS
- `initGui()` appelé lorsque le plugin est chargé
- `unload()` appelé lorsque le plugin est déchargé

Dans l'exemple ci-dessus, `addPluginToMenu` est utilisé. Cela ajoutera l'action de menu correspondante au menu *Plugins*. D'autres méthodes existent pour ajouter l'action à un autre menu. Voici une liste de ces méthodes :

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

Elles ont toutes la même syntaxe que la méthode `addPluginToMenu`.

Il est recommandé d'ajouter votre menu de plugin à l'une de ces méthodes prédéfinies afin de conserver une certaine cohérence dans l'organisation des entrées de plugin. Toutefois, vous pouvez ajouter votre groupe de menus personnalisés directement à la barre de menu, comme le montre l'exemple suivant :

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

    self.action = QAction(QIcon(":/plugins/testplug/icon.png"),
                          "Test plugin",
                          self.iface.mainWindow())
    self.action.setObjectName("testAction")
    self.action.setWhatsThis("Configuration for test plugin")
    self.action.setStatusTip("This is status tip")
    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()
```

N'oubliez pas de donner à `QAction` et `QMenu` `objectName` un nom spécifique à votre plugin pour qu'il puisse être personnalisé.

## Fichier de ressources

Vous pouvez voir que dans `initGui()` nous avons utilisé une icône du fichier de ressources (appelé `resources.qrc` dans notre cas)

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

Il est bon d'utiliser un préfixe qui n'entrera pas en collision avec d'autres plugins ou d'autres parties de QGIS, sinon vous risquez d'obtenir des ressources dont vous ne voulez pas. Il vous suffit maintenant de générer un fichier Python qui contiendra les ressources. C'est fait avec la commande `pyrcc5` :

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

**Note :** Dans les environnements Windows, tenter d'exécuter la commande **pyrcc5** à partir de l'invite de commande ou de Powershell entraînera probablement l'erreur « Windows ne peut pas accéder au périphérique, au chemin ou au fichier spécifié [...] ». La solution la plus simple est probablement d'utiliser le shell OSGeo4W, mais si vous êtes à l'aise pour modifier la variable d'environnement PATH ou pour spécifier explicitement le chemin de l'exécutable, vous devriez pouvoir le trouver à l'adresse <Your QGIS Install Directory>\bin\pyrcc5.exe.

Et c'est tout... rien de compliqué :)

Si vous avez tout fait correctement, vous devriez pouvoir trouver et charger votre plugin dans le gestionnaire de plugins et voir un message dans la console lorsque l'icône de la barre d'outils ou l'élément de menu approprié est sélectionné.

Lorsque vous travaillez sur un véritable plugin, il est judicieux d'écrire le plugin dans un autre répertoire (de travail) et de créer un makefile qui générera des fichiers d'interface utilisateur + de ressources et d'installer le plugin dans votre installation QGIS.

### 16.1.3 Documentation

La documentation du plugin peut être écrite sous forme de fichiers d'aide HTML. Le module `qgis.utils` fournit une fonction, `showPluginHelp()` qui ouvrira le navigateur de fichiers d'aide, de la même manière que les autres aides QGIS.

La fonction `showPluginHelp()` recherche les fichiers d'aide dans le même répertoire que le module appelant. Elle cherchera, à son tour, `index-ll_cc.html`, `index-ll.html`, `index-en.html`, `index-en_us.html` et `index.html`, en affichant celui qu'elle trouve en premier. Ici, `ll_cc` est la locale QGIS. Cela permet d'inclure plusieurs traductions de la documentation avec le plugin.

La fonction `showPluginHelp()` peut également prendre les paramètres `packageName`, qui identifie un plugin spécifique pour lequel l'aide sera affichée, `filename`, qui peut remplacer « index » dans les noms des fichiers recherchés, et `section`, qui est le nom d'une balise d'ancrage html dans le document sur lequel le navigateur sera positionné.

### 16.1.4 Traduction

En quelques étapes, vous pouvez configurer l'environnement pour la localisation du plugin de sorte que, selon les paramètres locaux de votre ordinateur, le plugin sera chargé dans différentes langues.

#### Exigences en matière de logiciels

La façon la plus simple de créer et de gérer tous les fichiers de traduction est d'installer [Qt Linguist](#). Dans un environnement GNU/Linux basé sur Debian, vous pouvez l'installer en tapant :

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

#### Fichiers et répertoire

Lorsque vous créez l'extension, vous devriez trouver le répertoire « `i18n` » dans le répertoire principal de l'extension. Tous les fichiers de traduction doivent être dans ce répertoire.

### Fichier .pro

Premièrement, vous devez créer un fichier « .pro ». Il s'agit d'un fichier projet qui peut être géré par Qt Linguist.

Dans ce fichier « .pro », vous devez spécifier tous les fichiers et formulaires que vous voulez traduire. Ce fichier est utilisé pour mettre en place les fichiers et les variables de localisation. Un fichier projet possible, correspondant à la structure de notre plugin *example plugin* :

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

Votre plugin peut suivre une structure plus complexe, et il peut être réparti sur plusieurs fichiers. Si c'est le cas, gardez à l'esprit que `pylupdate5`, le programme que nous utilisons pour lire le fichier `pro` et mettre à jour la chaîne traduisible, ne développe pas les caractères génériques, vous devez donc placer chaque fichier explicitement dans le fichier `pro`. Votre fichier de projet pourrait alors ressembler à quelque chose comme ceci :

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

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

Enfin, avec la variable « TRANSLATIONS », vous pouvez spécifier les langages de traduction que vous souhaitez.

**Avertissement :** Assurez-vous de nommer le fichier `ts` comme `votre_plugin_ + langue + .ts` sinon le chargement de la langue échouera ! Utilisez le raccourci de 2 lettres pour la langue (**it** pour l'italien, **de** pour l'allemand, etc...)

### Fichier .ts

Une fois que vous avez créé le `.pro`, vous êtes prêt à générer le(s) fichier(s) `.ts` pour la (les) langue(s) de votre plugin.

Lancez un terminal, allez au dossier `your_plugin/i18n` et saisissez :

```
pylupdate5 your_plugin.pro
```

Vous devriez voir le(s) fichier(s) `your_plugin_language.ts`.

Ouvrez le fichier `.ts` avec **Qt Linguist** et commencez à traduire.

### Fichier .qm

Lorsque vous avez fini de traduire votre plugin (si certaines chaînes ne sont pas terminées, la langue source de ces chaînes sera utilisée), vous devez créer le fichier `.qm` (le fichier `.ts` compilé qui sera utilisé par QGIS).

Il suffit d'ouvrir un cd de terminal dans le répertoire `your_plugin/i18n` et de taper ::

```
lrelease your_plugin.ts
```

Vous devriez maintenant voir le(s) fichier(s) `your_plugin.qm` dans le dossier `i18n`.



## Traduction en utilisant un fichier Makefile

Vous pouvez également utiliser le makefile pour extraire des messages du code python et des dialogues Qt, si vous avez créé votre plugin avec Plugin Builder. Au début du Makefile, il y a une variable LOCALES :

```
LOCALES = en
```

Ajoutez l'abréviation de la langue à cette variable, par exemple pour la langue hongroise :

```
LOCALES = en hu
```

Vous pouvez maintenant générer ou mettre à jour le fichier `hu.ts` (et le fichier `en.ts` aussi) à partir des sources par :

```
make transup
```

Après cela, vous avez mis à jour le fichier `.ts` pour toutes les langues définies dans la variable LOCALES. Utilisez **Qt Linguist** pour traduire les messages du programme. Pour terminer la traduction, les fichiers « `.qm` » peuvent être créés par le transcompilateur :

```
make transcompile
```

Vous devez distribuer les fichiers « `.ts` » avec votre plugin.

## Chargement de l'extension

Pour voir la traduction de votre plugin, ouvrez QGIS, changez la langue (*parametre* [?](#) *Options* [?](#) *General*) et redémarrez QGIS.

Vous devriez voir votre plugin dans la bonne langue.

**Avertissement :** Si vous changez quelque chose dans votre plugin (nouvelle interface utilisateur, nouveau menu, etc.), vous devez **générer à nouveau** la version mise à jour des fichiers `.ts` et `.qm`, donc exécuter à nouveau la commande ci-dessus.

## 16.1.5 Conseils et Astuces

### Rechargeur de plugins

Pendant le développement de votre plugin, vous devrez fréquemment le recharger dans QGIS pour le tester. C'est très facile en utilisant le plugin **Plugin Reloader**. Vous pouvez le trouver avec le Plugin Manager.

### Accès aux plugins

Vous pouvez accéder à toutes les classes de plugins installés depuis QGIS en utilisant python, ce qui peut être pratique pour le débogage.

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

### Messages log

Les plugins ont leur propre onglet dans le panneau `log_message_panel`.

### Partage de votre extension

QGIS héberge des centaines de plugins dans le dépôt de plugins. Pensez à partager le vôtre ! Cela permettra d'étendre les possibilités de QGIS et les gens pourront apprendre de votre code. Tous les plugins hébergés peuvent être trouvés et installés à partir de QGIS grâce au gestionnaire de plugins.

Informations et prérequis consultables ici : [plugins.qgis.org](http://plugins.qgis.org).

## 16.2 Extraits de code

- *Comment appeler une méthode par un raccourci clavier*
- *Comment basculer les couches*
- *Comment accéder à la table d'attributs des entités sélectionnées*
- *Interface pour le plugin dans le dialogue des options*

Cette section contient des extraits de code pour faciliter le développement de plugins.

### 16.2.1 Comment appeler une méthode par un raccourci clavier

Dans le plug-in, ajoutez à la fonction `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)
```

À `unload()` ajouter

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

La méthode qui est appelée lorsque l'on appuie sur CTRL+I

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

### 16.2.2 Comment basculer les couches

Il existe une API pour accéder aux couches de la légende. Voici un exemple qui permet de basculer la visibilité de la couche active

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

## 16.2.3 Comment accéder à la table d'attributs des entites sélectionnées

```

1 def change_value(value):
2     """Change the value in the second column for all selected features.
3
4     :param value: The new value.
5     """
6     layer = iface.activeLayer()
7     if layer:
8         count_selected = layer.selectedFeatureCount()
9         if count_selected > 0:
10            layer.startEditing()
11            id_features = layer.selectedFeatureIds()
12            for i in id_features:
13                layer.changeAttributeValue(i, 1, value) # 1 being the second column
14            layer.commitChanges()
15        else:
16            iface.messageBar().pushCritical("Error",
17                "Please select at least one feature from current layer")
18    else:
19        iface.messageBar().pushCritical("Error", "Please select a layer")
20
21 # The method requires one parameter (the new value for the second
22 # field of the selected feature(s)) and can be called by
23 change_value(50)

```

## 16.2.4 Interface pour le plugin dans le dialogue des options

Vous pouvez ajouter un onglet d'options de plugin personnalisé à *Settings* [\[?\] Options](#). Ceci est préférable à l'ajout d'une entrée spécifique dans le menu principal pour les options de votre plugin, car cela permet de garder tous les paramètres de l'application QGIS et les paramètres du plugin dans un seul endroit, ce qui est facile à découvrir et à naviguer pour les utilisateurs.

L'extrait suivant ajoute un nouvel onglet vierge pour les paramètres du plugin, prêt à être rempli avec toutes les options et paramètres spécifiques à votre plugin. Vous pouvez diviser les classes suivantes en différents fichiers. Dans cet exemple, nous ajoutons deux classes dans le fichier `mainPlugin.py`.

```

1 class MyPluginOptionsFactory(QgsOptionsWidgetFactory):
2
3     def __init__(self):
4         super().__init__()
5
6     def icon(self):
7         return QIcon('icons/my_plugin_icon.svg')
8
9     def createWidget(self, parent):
10        return ConfigOptionsPage(parent)
11
12
13 class ConfigOptionsPage(QgsOptionsPageWidget):
14
15     def __init__(self, parent):
16         super().__init__(parent)
17         layout = QHBoxLayout()
18         layout.setContentsMargins(0, 0, 0, 0)
19         self.setLayout(layout)

```

Enfin, nous ajoutons les importations et modifions la fonction `__init__`:

```

1 from qgis.PyQt.QtWidgets import QHBoxLayout
2 from qgis.gui import QgsOptionsWidgetFactory, QgsOptionsPageWidget
3
4
5 class MyPlugin:
6     """QGIS Plugin Implementation."""
7
8     def __init__(self, iface):
9         """Constructor.
10
11         :param iface: An interface instance that will be passed to this class
12             which provides the hook by which you can manipulate the QGIS
13             application at run time.
14         :type iface: QgsInterface
15         """
16         # Save reference to the QGIS interface
17         self.iface = iface
18
19
20     def initGui(self):
21         self.options_factory = MyPluginOptionsFactory()
22         self.options_factory.setTitle(self.tr('My Plugin'))
23         iface.registerOptionsWidgetFactory(self.options_factory)
24
25     def unload(self):
26         iface.unregisterOptionsWidgetFactory(self.options_factory)

```

**Astuce :** Vous pouvez appliquer une logique similaire pour ajouter l'option personnalisée du plugin au dialogue des propriétés de la couche en utilisant les classes `QgsMapLayerConfigWidgetFactory` et `QgsMapLayerConfigWidget`.

## 16.3 Utilisation de la classe `QgsPluginLayer`

Si votre plugin utilise ses propres méthodes pour le rendu des couches, la meilleure façon de l'implémenter passe par l'écriture de vos propres types en vous basant sur la classe `QgsPluginLayer`

### 16.3.1 Héritage de la classe `QgsPluginLayer`

Vous trouverez ci dessous un exemple simple d'implémentation de la classe `QgsPluginLayer`, basé sur le code de l'extension `Watermark`

Le moteur de rendu personnalisé est la partie de l'implémentation qui définit le rendu réel dans le canevas de carte.

```

1 class WatermarkLayerRenderer(QgsMapLayerRenderer):
2
3     def __init__(self, layerId, rendererContext):
4         super().__init__(layerId, rendererContext)
5
6     def render(self):
7         image = QImage("/usr/share/icons/hicolor/128x128/apps/qgis.png")
8         painter = self.rendererContext().painter()
9         painter.save()
10        painter.drawImage(10, 10, image)
11        painter.restore()
12        return True
13

```

(suite sur la page suivante)

(suite de la page précédente)

```

14 class WatermarkPluginLayer(QgsPluginLayer):
15
16     LAYER_TYPE="watermark"
17
18     def __init__(self):
19         super().__init__(WatermarkPluginLayer.LAYER_TYPE, "Watermark plugin layer")
20         self.setValid(True)
21
22     def createMapRenderer(self, rendererContext):
23         return WatermarkLayerRenderer(self.id(), rendererContext)
24
25     def setTransformContext(self, ct):
26         pass
27
28     # Methods for reading and writing specific information to the project file can
29     # also be added:
30
31     def readXml(self, node, context):
32         pass
33
34     def writeXml(self, node, doc, context):
35         pass

```

Le plugin layer peut être ajouté a un projet et à la carte comme n'importe quelle autre couche :

```

plugin_layer = WatermarkPluginLayer()
QgsProject.instance().addMapLayer(plugin_layer)

```

Quand vous chargez un projet contenant ce type de couches, une classe constructeur est necessaire

```

1 class WatermarkPluginLayerType(QgsPluginLayerType):
2
3     def __init__(self):
4         super().__init__(WatermarkPluginLayer.LAYER_TYPE)
5
6     def createLayer(self):
7         return WatermarkPluginLayer()
8
9     # You can also add GUI code for displaying custom information
10    # in the layer properties
11    def showLayerProperties(self, layer):
12        pass
13
14
15    # Keep a reference to the instance in Python so it won't
16    # be garbage collected
17    plt = WatermarkPluginLayerType()
18
19    assert QgsApplication.pluginLayerRegistry().addPluginLayerType(plt)

```

## 16.4 Paramètres de l'IDE pour l'écriture et le débogage de plugins

- *Plugins utiles pour écrire des plugins Python*
- *Une note sur la configuration de votre IDE sous Linux et Windows*
- *Débogage à l'aide de l'IDE Pyscripter (Windows)*
- *Débogage à l'aide d'Eclipse et PyDev*
  - *Installation*
  - *Mise en place d'Eclipse*
  - *Configuration du débogueur*
  - *Faire comprendre l'API à eclipse*
- *Débogage avec PyCharm sur Ubuntu avec QGIS compilé*
- *Débogage à l'aide de PDB*

Bien que chaque programmeur ait son éditeur IDE/Text préféré, voici quelques recommandations pour mettre en place des IDE populaires pour l'écriture et le débogage de plugins Python QGIS.

### 16.4.1 Plugins utiles pour écrire des plugins Python

Certains plugins sont pratiques pour écrire des plugins Python. De *Plugins* [☞](#) *Manage and Install plugins...*, install :

- *Rechargeur de plugin* : Cela vous permet de recharger un plugin et d'effectuer de nouvelles modifications sans avoir à redémarrer QGIS.
- *Premiers secours* : Ceci ajoutera une console Python et un débogueur local pour inspecter les variables lorsqu'une exception est levée d'un 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.*

### 16.4.2 Une note sur la configuration de votre IDE sous Linux et Windows

**Sur Linux**, il suffit généralement d'ajouter les emplacements de la bibliothèque QGIS à la variable d'environnement PYTHONPATH de l'utilisateur. Sous la plupart des distributions, cela peut être fait en éditant `~/ .bashrc` ou `~/ .bash-profile` avec la ligne suivante (testé sur OpenSUSE Tumbleweed) :

```
export PYTHONPATH="$PYTHONPATH:/usr/share/qgis/python/plugins:/usr/share/qgis/
↳python"
```

Enregistrez le fichier et implémentez les paramètres d'environnement en utilisant la commande shell suivante :

```
source ~/.bashrc
```

**Sur Windows**, vous devez vous assurer que vous avez les mêmes paramètres d'environnement et que vous utilisez les mêmes bibliothèques et interpréteur que QGIS. La façon la plus rapide de le faire est de modifier le fichier de démarrage de QGIS.

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

### 16.4.3 Débogage à l'aide de l'IDE Pyscripter (Windows)

Pour l'utilisation de Pyscripter IDE, voici ce que vous devez faire :

1. Faites une copie de `qgis-unstable.bat` et renommez le `pyscripter.bat`.
2. Ouvrez-le dans un éditeur. Et supprimez la dernière ligne, celle qui lance QGIS.
3. Ajoutez une ligne qui pointe vers votre exécutable Pyscripter et ajoutez l'argument de la ligne de commande qui définit la version de Python à utiliser
4. Ajoutez également l'argument qui pointe vers le dossier où Pyscripter peut trouver la dll Python utilisée par QGIS, vous pouvez le trouver sous le dossier bin de votre installation OSGeoW

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

5. Maintenant, lorsque vous double-cliquez sur ce fichier, Pyscripter se lance, avec le chemin d'accès correct.

Plus populaire que Pyscripter, Eclipse est un choix commun parmi les développeurs. Dans la section suivante, nous expliquerons comment le configurer pour développer et tester des plugins.

### 16.4.4 Débogage à l'aide d'Eclipse et PyDev

#### Installation

Pour utiliser Eclipse, assurez-vous que vous avez installé les éléments suivants

- Eclipse
- Aptana Studio 3 Plugin or PyDev
- QGIS 2.x
- Vous pouvez également installer **Remote Debug**, un plugin QGIS. Pour le moment, il est encore expérimental, donc activez  *Plugins expérimentaux* sous *Plugins*  *Gérer et installer des plugins...*  *Options* au préalable.

Pour préparer votre environnement à l'utilisation d'Eclipse dans Windows, vous devez également créer un fichier batch et l'utiliser pour démarrer Eclipse :

1. Localisez le dossier dans lequel se trouve `qgis_core.dll`. Normalement, il s'agit de `C:\OSGeo4W\apps\qgis\bin`, mais si vous avez compilé votre propre application QGIS, il se trouve dans votre dossier de compilation dans `output/bin/RelWithDebInfo`.
2. Localisez votre exécutable `eclipse.exe`.
3. Créez le script suivant et utilisez-le pour démarrer eclipse lors du développement des plugins QGIS.

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

#### Mise en place d'Eclipse

1. Dans Eclipse, créez un nouveau projet. Vous pouvez sélectionner *Projet général* et lier vos sources réelles par la suite, de sorte que l'endroit où vous placez ce projet n'a pas vraiment d'importance.
2. Faites un clic droit sur votre nouveau projet et choisissez *New*  *Folder*.
3. Cliquez sur *avance* et choisissez *lien vers un autre lieu (dossier lié)*. Si vous avez déjà des sources que vous voulez déboguer, choisissez celles-ci. Si vous n'en avez pas, créez un dossier comme cela a déjà été expliqué.

Maintenant, dans la vue *Project Explorer*, votre arbre des sources apparaît et vous pouvez commencer à travailler avec le code. Vous disposez déjà de la coloration syntaxique et de tous les autres puissants outils de l'IDE.

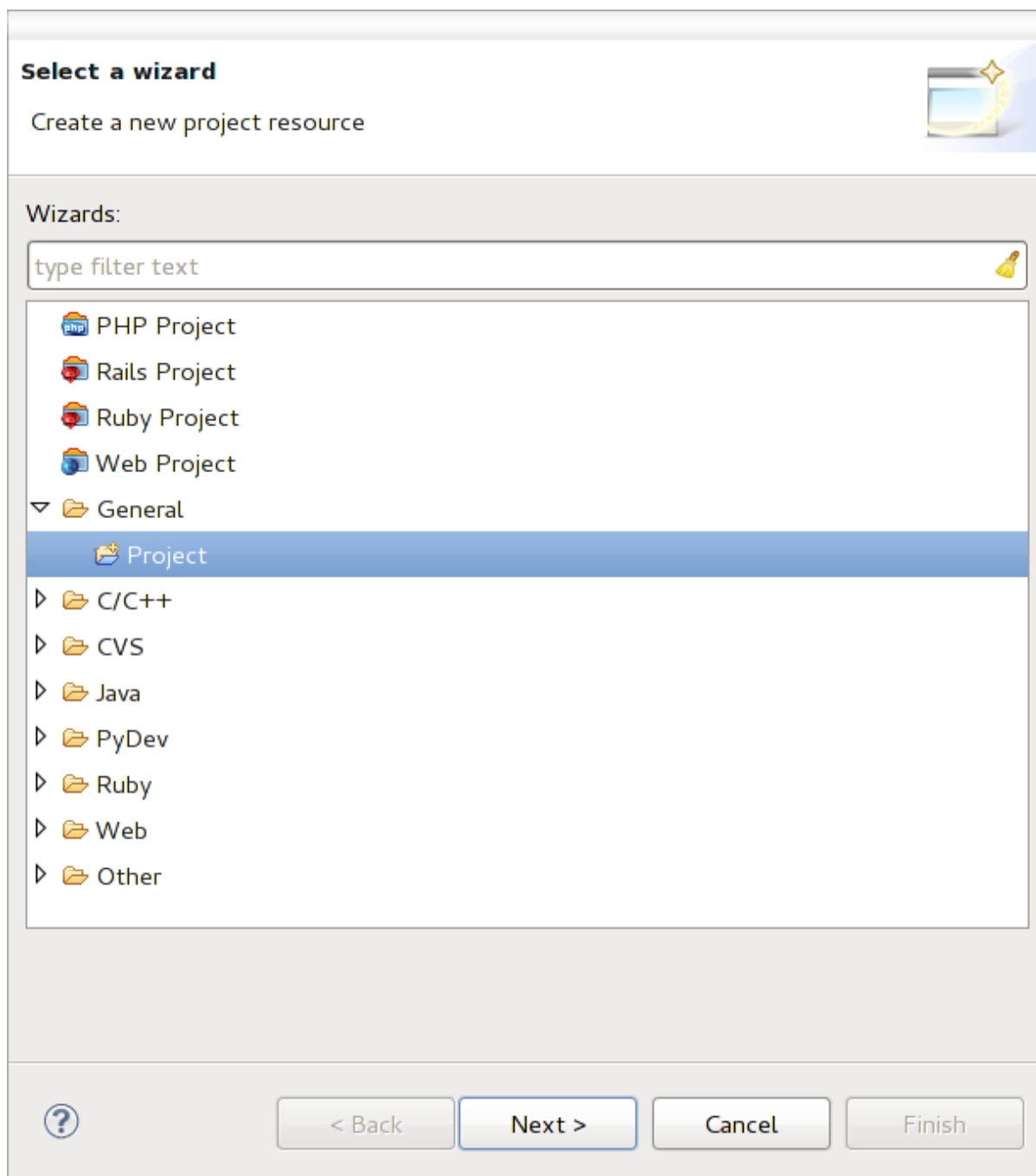


Fig. 16.1 – Projet Eclipse



## Configuration du débogueur

Pour faire fonctionner le débogueur :

1. Passez à la perspective Debug dans Eclipse (*Window* → *Open Perspective* → *Other* → *Debug*).
2. démarrer le serveur de débogage PyDev en choisissant *PyDev* → *Start Debug Server*.
3. Eclipse attend maintenant une connexion de QGIS à son serveur de débogage et lorsque QGIS se connectera au serveur de débogage, il lui permettra de contrôler les scripts python. C'est exactement pour cela que nous avons installé le plugin *Remote Debug*. Démarrez donc QGIS au cas où vous ne l'auriez pas déjà fait et cliquez sur le symbole de bogue.

Vous pouvez maintenant définir un point d'arrêt et dès que le code le touche, l'exécution s'arrête et vous pouvez inspecter l'état actuel de votre plugin. (Le point d'arrêt est le point vert dans l'image ci-dessous, que vous pouvez définir en double-cliquant dans l'espace blanc à gauche de la ligne où vous voulez que le point d'arrêt soit défini).

```

87         self.webView.debuggerActionManager.setEnabled( true )
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
104

```

Fig. 16.2 – Point d'arrêt

Une chose très intéressante que vous pouvez utiliser maintenant est la console de débogage. Assurez-vous que l'exécution est actuellement arrêtée à un point d'arrêt, avant de poursuivre.

1. Ouvrez la vue Console (*Fenêtre* → *Afficher la vue*). Elle affichera la console *Debug Server* qui n'est pas très intéressante. Mais il y a un bouton *Open Console* qui vous permet de passer à une console de débogage PyDev plus intéressante.
2. Cliquez sur la flèche à côté du bouton *Open Console* et choisissez *PyDev Console*. Une fenêtre s'ouvre pour vous demander quelle console vous voulez démarrer.
3. Choisissez *PyDev Debug Console*. Au cas où elle serait grisée et vous demanderait de démarrer le débogueur et de sélectionner la trame valide, assurez-vous que le débogueur distant est connecté et que vous êtes actuellement sur un point d'arrêt.

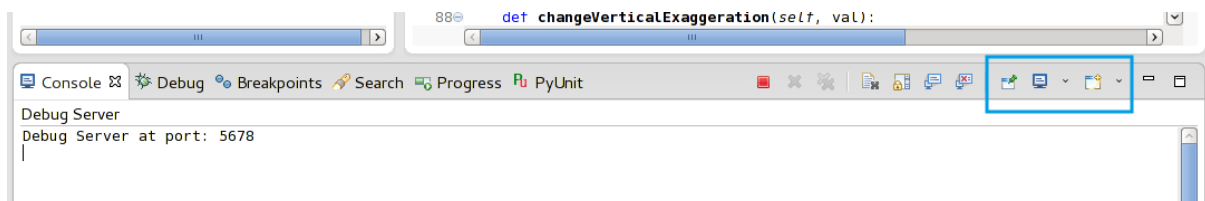


Fig. 16.3 – PyDev Debug Console

Vous disposez maintenant d'une console interactive qui vous permet de tester les commandes dans le contexte actuel. Vous pouvez manipuler des variables ou faire des appels d'API ou tout ce que vous voulez.

---

**Astuce :** Un peu ennuyeux, c'est que chaque fois que vous entrez une commande, la console repasse sur le serveur de débogage. Pour arrêter ce comportement, vous pouvez cliquer sur le bouton *Console Pin* lorsque vous êtes sur la page du serveur de débogage et il devrait se souvenir de cette décision au moins pour la session de débogage en cours.

---

## Faire comprendre l'API à Eclipse

Une fonction très pratique est de faire en sorte qu'Eclipse connaisse réellement l'API QGIS. Cela lui permet de vérifier si votre code contient des fautes de frappe. Mais ce n'est pas tout : Eclipse peut également vous aider à effectuer l'autocomplétion des importations et des appels d'API.

Pour ce faire, Eclipse analyse les fichiers de la bibliothèque QGIS et diffuse toutes les informations. La seule chose que vous devez faire est de dire à Eclipse où se trouvent les bibliothèques.

1. Cliquez sur *Window* *Preferences* *PyDev* *Interpreter* *Python*.

Vous verrez votre interpréteur python configuré dans la partie supérieure de la fenêtre (actuellement python2.7 pour QGIS) et quelques onglets dans la partie inférieure. Les onglets intéressants pour nous sont *Bibliothèques* et *Constructions forcées*.

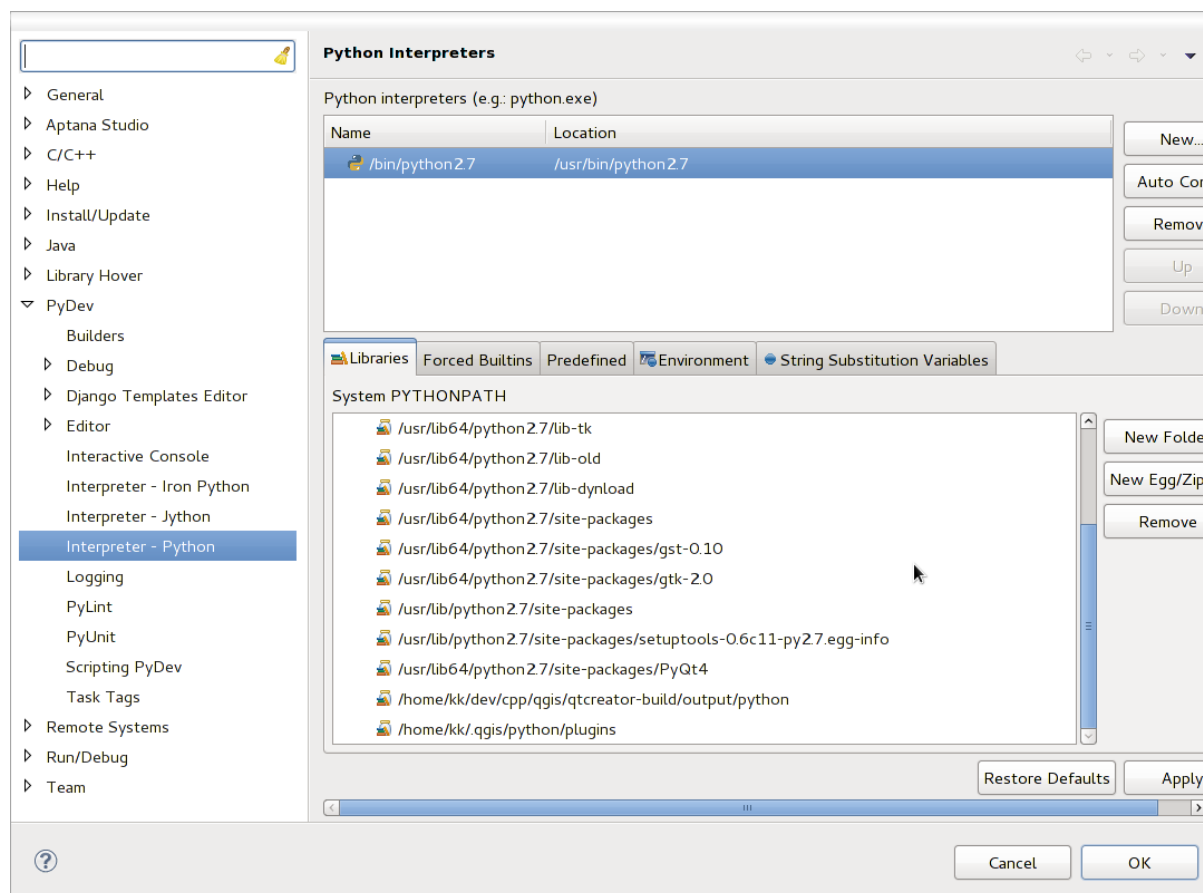


Fig. 16.4 – PyDev Debug Console

2. Ouvrez d'abord l'onglet Bibliothèques.
3. Ajoutez un nouveau dossier et choisissez le dossier python de votre installation QGIS. Si vous ne savez pas où se trouve ce dossier (ce n'est pas le dossier des plugins) :
  1. Ouvrir QGIS
  2. Démarrer une console python
  3. Entrez `qgis`.
  4. et appuyez sur la touche Entrée. Il vous indiquera le module QGIS qu'il utilise et son chemin.
  5. Enlevez le « `/qgis/__init__.pyc` » de ce chemin et vous avez le chemin que vous cherchez.
4. Vous devez également ajouter votre dossier de plugins ici (il se trouve dans le dossier `python/plugins` sous le dossier `user_profile`).

5. Passez ensuite à l'onglet *Forced Builtins*, cliquez sur *Nouveau...* et entrez `qgis`. Ceci fera en sorte qu'Eclipse analyse l'API QGIS. Vous voulez probablement aussi qu'Eclipse connaisse l'API PyQt. Par conséquent, ajoutez également PyQt en tant qu'intégration forcée. Cela devrait probablement déjà être présent dans votre onglet bibliothèques.
6. Cliquez sur *OK* et vous avez terminé.

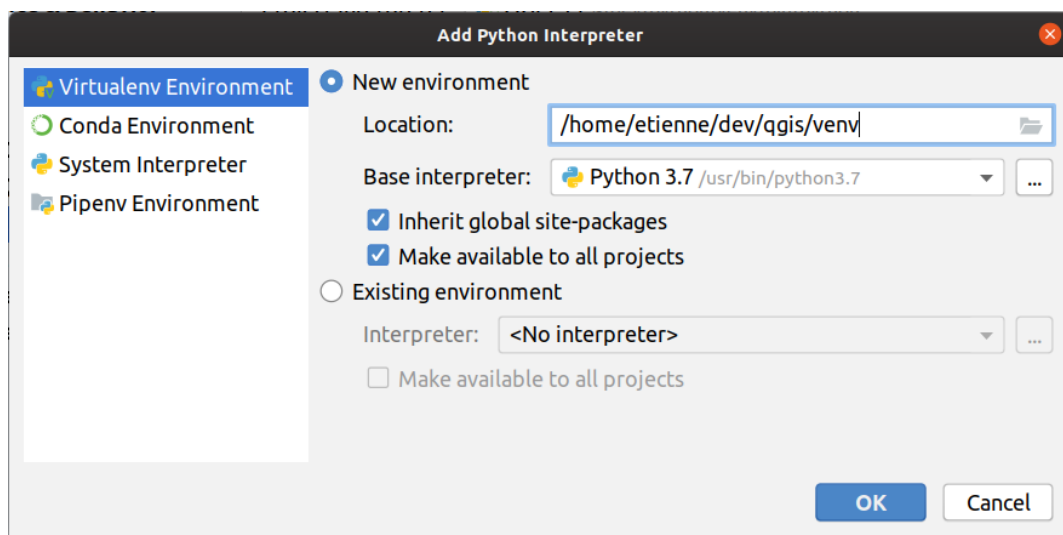
**Note :** Chaque fois que l'API QGIS change (par exemple, si vous compilez QGIS Master et que le fichier SIP a changé), vous devez retourner à cette page et cliquer simplement sur *appliquer*. Cela permettra à Eclipse d'analyser à nouveau toutes les bibliothèques.

## 16.4.5 Débogage avec PyCharm sur Ubuntu avec QGIS compilé

PyCharm est un IDE pour Python développé par JetBrains. Il existe une version gratuite appelée Community Edition et une version payante appelée Professional. Vous pouvez télécharger PyCharm sur le site web : <https://www.jetbrains.com/pycharm/download>

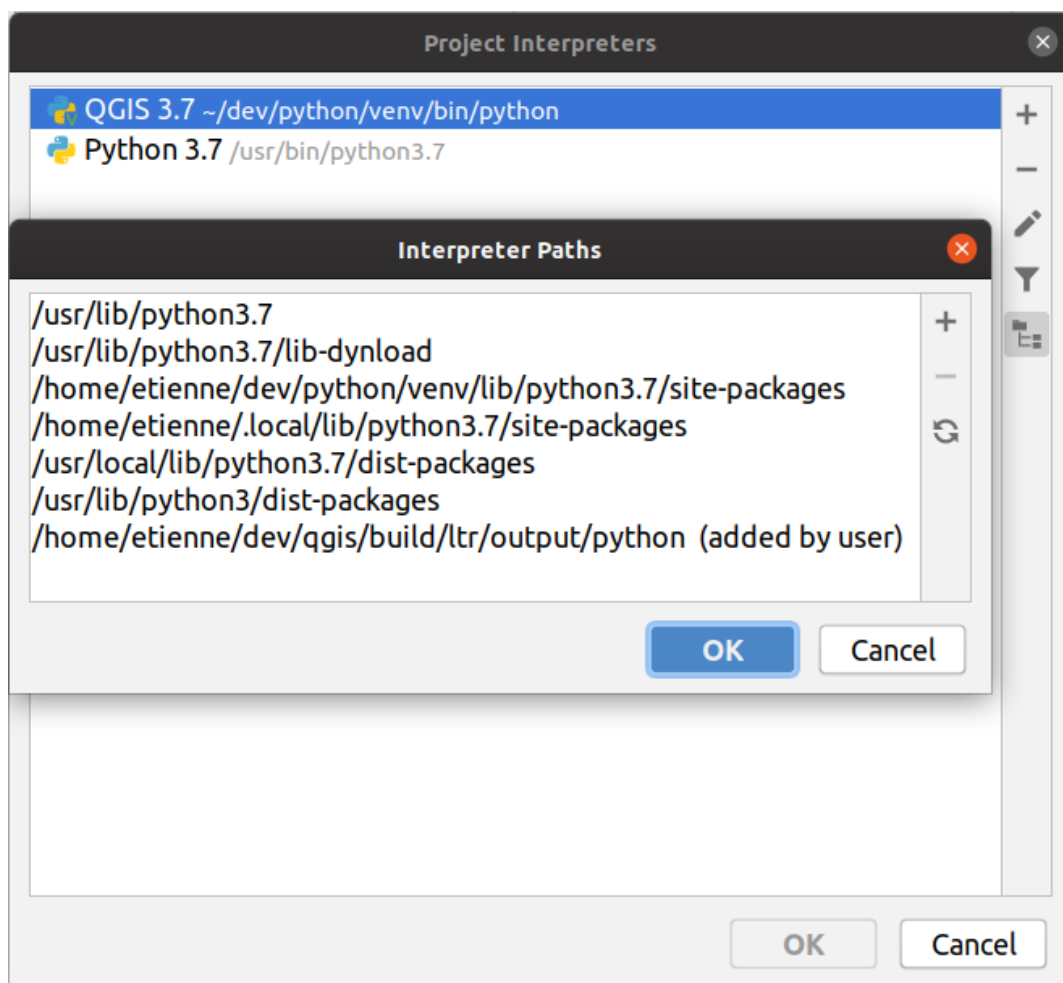
Nous supposons que vous avez compilé QGIS sur Ubuntu avec le répertoire de compilation donné `~/dev/qgis/build/master`. Il n'est pas obligatoire d'avoir un QGIS auto-compilé, mais seul celui-ci a été testé. Les chemins doivent être adaptés.

1. Dans PyCharm, dans *propriétés projet*, *Project Interpreter*, nous allons créer un environnement virtuel Python appelé QGIS.
2. Cliquez sur le petit engrenage et ensuite sur *Ajouter*.
3. Sélectionnez *Virtualenv environment*.
4. Sélectionnez un emplacement générique pour tous vos projets Python tel que `~/dev/qgis/venv` car nous utiliserons cet interpréteur Python pour tous nos plugins.
5. Choisissez un interpréteur de base Python 3 disponible sur votre système et cochez les deux options suivantes *Hériter des ensembles site-packages* et *Rendre disponible à tous les projets*.



1. Cliquez sur *OK*, revenez sur la petite gear et cliquez sur *Show all* (Afficher tout).
2. Dans la nouvelle fenêtre, sélectionnez votre nouvel interprète QGIS et cliquez sur la dernière icône du menu vertical *Montre les chemins pour l'interprète sélectionné*.
3. Enfin, ajoutez le chemin absolu suivant à la liste : `file :~/dev/qgis/build/master/output/python`.

1. Redémarrez PyCharm et vous pourrez commencer à utiliser ce nouvel environnement virtuel Python pour tous vos plugins.



PyCharm connaîtra l'API QGIS et aussi l'API PyQt si vous utilisez Qt fourni par QGIS comme « `from qgis.PyQt.QtCore import QDir` ». L'auto-complétion devrait fonctionner et PyCharm peut inspecter votre code.

Dans la version professionnelle de PyCharm, le débogage à distance fonctionne bien. Pour la version communautaire, le débogage à distance n'est pas disponible. Vous ne pouvez avoir accès qu'à un débogueur local, ce qui signifie que le code doit être exécuté à l'intérieur de PyCharm (sous forme de script ou de test unitaire), et non dans le QGIS lui-même. Pour le code Python qui tourne *dans* QGIS, vous pouvez utiliser le plugin *First Aid* mentionné ci-dessus.

### 16.4.6 Débogage à l'aide de PDB

Si vous n'utilisez pas un IDE tel qu'Eclipse ou PyCharm, vous pouvez déboguer en utilisant PDB, en suivant ces étapes.

1. Ajoutez d'abord ce code à l'endroit où vous souhaitez déboguer

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

2. Ensuite, lancez QGIS depuis la ligne de commande.

Sur Linux, faites :

```
$ ./Qgis
```

Sur macOS faire :

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

3. Et lorsque l'application atteint votre point d'arrêt, vous pouvez taper dans la console !

**A FAIRE :** Ajouter des informations sur les tests

## 16.5 Déblocage de votre plugin

- *Métadonnées et noms*
- *Code et aide*
- *Dépôt officiel des extensions QGIS*
  - *Autorisations*
  - *Gestion de la confiance*
  - *Validation*
  - *Structure de l'extension*

Une fois que votre plugin est prêt et que vous pensez qu'il pourrait être utile à certaines personnes, n'hésitez pas à le télécharger sur *Dépôt officiel des extensions QGIS*. Sur cette page, vous trouverez également des instructions sur la façon de préparer le plugin pour qu'il fonctionne bien avec l'installateur de plugin. Si vous souhaitez créer votre propre dépôt de plugins, vous pouvez également créer un simple fichier XML qui répertoriera les plugins et leurs métadonnées.

Veuillez porter une attention particulière aux suggestions suivantes :

### 16.5.1 Métadonnées et noms

- éviter d'utiliser un nom trop similaire aux plugins existants
- si votre plugin a une fonctionnalité similaire à un plugin existant, veuillez expliquer les différences dans le champ À propos, afin que l'utilisateur sache lequel utiliser sans avoir besoin de l'installer et de le tester
- éviter de répéter « plugin » dans le nom du plugin lui-même
- utilisez le champ description dans les métadonnées pour une description d'une ligne, le champ À propos pour des instructions plus détaillées
- inclure un dépôt de code, un système de suivi des bogues et une page d'accueil ; cela augmentera considérablement les possibilités de collaboration et peut être fait très facilement avec l'une des infrastructures web disponibles (GitHub, GitLab, Bitbucket, etc.)
- choisissez les balises avec soin : évitez celles qui ne sont pas informatives (par exemple le vecteur) et préférez celles qui sont déjà utilisées par d'autres (voir le site du plugin)
- ajouter une icône appropriée, ne pas laisser celle par défaut ; voir l'interface QGIS pour une suggestion du style à utiliser

### 16.5.2 Code et aide

- ne pas inclure les fichiers générés (ui\_\*.py, resources\_rc.py, fichiers d'aide générés...) et les trucs inutiles (par exemple .gitignore) dans le dépôt
- ajouter le plugin au menu approprié (Vecteur, Raster, Web, Base de données)
- le cas échéant (plugins effectuant des analyses), envisagez d'ajouter le plugin en tant que sous-plugin du cadre Processing : cela permettra aux utilisateurs de l'exécuter en lot, de l'intégrer dans des flux de travail plus complexes et vous libérera de la charge de concevoir une interface
- inclure au moins une documentation minimale et, si cela est utile pour tester et comprendre, des échantillons de données.

### 16.5.3 Dépôt officiel des extensions QGIS

Vous trouverez le dépôt *officiel* des extensions QGIS ici : <https://plugins.qgis.org/>.

Pour pouvoir utiliser le dépôt officiel, vous devez obtenir un identifiant OSGEO sur le portail web de l'OSGEO <[https://www.osgeo.org/community/getting-started-osgeo/osgeo\\_userid/](https://www.osgeo.org/community/getting-started-osgeo/osgeo_userid/)>`\_.

Une fois que vous aurez téléchargé votre plugin, il sera approuvé par un membre du personnel et vous en serez informé.

**A FAIRE :** Insérer un lien vers le document de gouvernance

#### Autorisations

Ces règles ont été mises en œuvre dans le dépôt officiel des plugins :

- chaque utilisateur enregistré peut ajouter un nouveau plugin
- Les utilisateurs du *staff* peuvent approuver ou désapprouver toutes les versions de plugin
- les utilisateurs qui ont la permission spéciale *plugins.can\_approve* voient les versions qu'ils téléchargent automatiquement approuvées
- les utilisateurs qui ont la permission spéciale *plugins.can\_approve* peuvent approuver les versions téléchargées par d'autres tant qu'ils sont dans la liste des *propriétaires* du plugin
- un plugin particulier ne peut être supprimé et édité que par les utilisateurs du *staff* et les propriétaires du plugin
- si un utilisateur sans permission *plugins.can\_approve* télécharge une nouvelle version, la version du plugin n'est pas automatiquement approuvée.

## Gestion de la confiance

Les membres du staff peuvent accorder la *confiance* à des créateurs de plugins sélectionnés en définissant la permission `plugins.can_approve` via l'application frontale.

La vue détaillée du plugin offre des liens directs pour accorder la confiance au créateur du plugin ou aux *propriétaires* du plugin.

## Validation

Les métadonnées du plugin sont automatiquement importées et validées à partir du paquet compressé lorsque le plugin est téléchargé.

Voici quelques règles de validation que vous devez connaître lorsque vous souhaitez télécharger un plugin sur le dépôt officiel :

1. le nom du dossier principal contenant votre plugin doit contenir uniquement des caractères ASCII (A-Z et a-z), des chiffres et les caractères de soulignement (`_`) et moins (`-`), il ne peut pas non plus commencer par un chiffre
2. `metadata.txt` est requis
3. toutes les métadonnées requises énumérées dans *metadata table* doivent être présentes
4. le champ de métadonnées de la *version* doit être unique

## Structure de l'extension

Selon les règles de validation, le paquet compressé (`.zip`) de votre plugin doit avoir une structure spécifique pour être validé comme plugin fonctionnel. Comme le plugin sera décompressé dans le dossier `plugins` des utilisateurs, il doit avoir son propre répertoire dans le fichier `.zip` pour ne pas interférer avec les autres plugins. Les fichiers obligatoires sont : `metadata.txt` et `__init__.py`. Mais il serait bon d'avoir un `README` et bien sûr une icône pour représenter le plugin (`resources.qrc`). Voici un exemple de ce à quoi devrait ressembler un `plugin.zip`.

```
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

---

- *Créer à partir de zéro*
- *Mise à jour d'un plugin*

Selon le type de plugin que vous allez développer, il peut être préférable d'ajouter ses fonctionnalités en tant qu'algorithme de traitement (ou un ensemble d'entre eux). Cela permettrait une meilleure intégration dans QGIS, des fonctionnalités supplémentaires (puisqu'il peut être exécuté dans les composants de Processing, tels que le modèleur ou l'interface de traitement par lots), et un temps de développement plus rapide (puisque Processing prendra 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.

### 17.1 Créer à partir de zéro

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

1. Installez le plugin **Plugin Builder**
2. Créez une nouvelle extension à l'aide de Plugin Builder. Lorsque l'application vous demande le modèle à utiliser, sélectionnez « Processing Provider ».
3. 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.

## 17.2 Mise à jour d'un plugin

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

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

```
hasProcessingProvider=yes
```

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

```
1 from qgis.core import QgsApplication
2 from processing_provider.provider import Provider
3
4 class YourPluginName():
5
6     def __init__(self):
7         self.provider = None
8
9     def initProcessing(self):
10        self.provider = Provider()
11        QgsApplication.processingRegistry().addProvider(self.provider)
12
13    def initGui(self):
14        self.initProcessing()
15
16    def unload(self):
17        QgsApplication.processingRegistry().removeProvider(self.provider)
```

3. Vous pouvez créer un dossier `processing_provider` avec trois fichiers dedans :
  - `__init__.py` avec rien dedans. Ceci est nécessaire pour faire un paquet Python valide.
  - `provider.py` qui va créer le provider processing et exposer vos algorithmes.

```
1 from qgis.core import QgsProcessingProvider
2
3 from processing_provider.example_processing_algorithm import
4     ↪ExampleProcessingAlgorithm
5
6 class Provider(QgsProcessingProvider):
7
8     def loadAlgorithms(self, *args, **kwargs):
9         self.addAlgorithm(ExampleProcessingAlgorithm())
10        # add additional algorithms here
11        # self.addAlgorithm(MyOtherAlgorithm())
12
13    def id(self, *args, **kwargs):
14        """The ID of your plugin, used for identifying the provider.
15
16        This string should be a unique, short, character only string,
17        eg "qgis" or "gdal". This string should not be localised.
18        """
19        return 'yourplugin'
20
21    def name(self, *args, **kwargs):
22        """The human friendly name of your plugin in Processing.
23
24        This string should be as short as possible (e.g. "Lastools", not
25        "Lastools version 1.0.1 64-bit") and localised.
26        """
27        return self.tr('Your plugin')
28
29    def icon(self):
```

(suite sur la page suivante)

(suite de la page précédente)

```
30     """Should return a QIcon which is used for your provider inside
31     the Processing toolbox.
32     """
33     return QgsProcessingProvider.icon(self)
```

— `example_processing_algorithm.py` qui contient le fichier de l'algorithme d'exemple. Copiez/collez le contenu du fichier `modèle de script` et mettez-le à jour selon vos besoins.

4. Vous pouvez maintenant recharger votre plugin dans QGIS et vous devriez voir votre script d'exemple dans la boîte à outils de traitement and modeleur.

Les extraits de code sur cette page nécessitent les importations suivantes si vous êtes en dehors de la console pyqgis :

```
from qgis.core import (
    QgsVectorLayer,
    QgsPointXY,
)
```



---

## Bibliothèque d'analyse de réseau

---

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

La bibliothèque d'analyse de réseau peut être utilisée pour :

- créer un graphe mathématique à partir des données géographiques (couches vecteurs de polylignes)
- mettre en œuvre les méthodes de base de la théorie des graphes (actuellement, seul l'algorithme de Dijkstra est utilisé)

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.

### 18.1 Information générale

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

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

### 18.2 Construire un graphe

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

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

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

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

La conversion d'une couche vecteur vers le graphe se fait à l'aide du modèle de programmation `Builder`. Un graphe est construit à l'aide d'un « Director ». Il n'y a qu'un seul Director pour l'instant : `QgsVectorLayerDirector`. Le director définit les paramètres de base qui seront utilisés pour construire un graphe à partir d'une couche vecteur linéaire, utilisée par le constructeur pour créer le graphe. Actuellement, comme dans le cas du director, un seul constructeur existe : `QgsGraphBuilder`, qui crée des objets `QgsGraph`. Vous pouvez vouloir implémenter vos propres constructeurs qui construiront un graphe compatible avec des bibliothèques telles que `BGL` ou `NetworkX`.

Pour calculer les propriétés des bords, on utilise le modèle de programmation `strategy`. Pour l'instant, seules les stratégies `QgsNetworkDistanceStrategy` (qui prend en compte la longueur du trajet) et `QgsNetworkSpeedStrategy` (qui prend également en compte la vitesse) sont disponibles. Vous pouvez mettre en œuvre votre propre stratégie qui utilisera tous les paramètres nécessaires. Par exemple, le plugin `RoadGraph` utilise une stratégie qui calcule le temps de parcours en utilisant la longueur des bords et la valeur de la vitesse à partir des attributs.

Il est temps de plonger dans le processus.

Tout d'abord, pour utiliser cette bibliothèque, nous devons importer le module d'analyse

```
from qgis.analysis import *
```

Ensuite, quelques exemples pour créer un directeur

```
1 # don't use information about road direction from layer attributes,
2 # all roads are treated as two-way
3 director = QgsVectorLayerDirector(vectorLayer, -1, '', '', '',
4     ↳QgsVectorLayerDirector.DirectionBoth)
5
6 # use field with index 5 as source of information about road direction.
7 # one-way roads with direct direction have attribute value "yes",
8 # one-way roads with reverse direction have the value "1", and accordingly
9 # bidirectional roads have "no". By default roads are treated as two-way.
10 director = QgsVectorLayerDirector(vectorLayer, 5, 'yes', '1', 'no',
    ↳QgsVectorLayerDirector.DirectionBoth)
```

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

```
1 director = QgsVectorLayerDirector(vectorLayer,
2     directionFieldId,
3     directDirectionValue,
4     reverseDirectionValue,
5     bothDirectionValue,
6     defaultDirection)
```

Voici la liste complète de la signification de ces paramètres :

- `vectorLayer` — 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 de la route par défaut. Cette valeur sera utilisée pour les routes où le champ `directionFieldId` n'est pas défini ou a une valeur différente de l'une des trois valeurs spécifiées ci-dessus. Les valeurs possibles sont :
  - « `QgsVectorLayerDirector.DirectionForward` » — Aller simple direct
  - `QgsVectorLayerDirector.DirectionBackward` — Inversion à sens unique
  - `QgsVectorLayerDirector.DirectionBoth` — Dans les deux sens

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

```

1 # The index of the field that contains information about the edge speed
2 attributeId = 1
3 # Default speed value
4 defaultValue = 50
5 # Conversion from speed to metric units ('1' means no conversion)
6 toMetricFactor = 1
7 strategy = QgsNetworkSpeedStrategy(attributeId, defaultValue, toMetricFactor)

```

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

```

director = QgsVectorLayerDirector(vectorLayer, -1, '', '', '', 3)
director.addStrategy(strategy)

```

Nous pouvons maintenant utiliser le constructeur, qui va créer le graphique. Le constructeur de la classe `QgsGraphBuilder` prend plusieurs arguments :

- `crs` — système de référence des coordonnées à utiliser. Argument obligatoire.
- `otfEnabled` — utiliser la reprojection « à la volée » ou non. Par défaut `const :True` (utilisez OTF).
- `TopologyTolerance` — 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(vectorLayer.crs())

```

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

```

startPoint = QgsPointXY(1179720.1871, 5419067.3507)
endPoint = QgsPointXY(1180616.0205, 5419745.7839)

```

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])

```

## 18.3 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 chemin le plus court est un graphique pondéré dirigé (ou plus précisément un arbre) ayant les propriétés suivantes :

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

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

La méthode `shortestTree` est utile lorsque vous voulez vous promener dans l'arbre qui a le chemin le plus court. Elle crée toujours un nouvel objet graphique (`QgsGraph`) et accepte trois variables :

- `source` — Graphique d'entrée
- `startVertexIdx` — index du point sur l'arbre (la racine de l'arbre)
- `criterionNum` — nombre de propriétés de bord à utiliser (à partir de 0).

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

La méthode `dijkstra` a les mêmes arguments, mais renvoie deux tableaux. Dans le premier tableau, l'élément  $n$  contient l'index du front entrant ou -1 s'il n'y a pas de front entrant. Dans le second tableau, l'élément  $n$  contient la distance entre la racine de l'arbre et le sommet  $n$  ou `DOUBLE_MAX` si le sommet  $n$  est inaccessible depuis la racine.

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

Voici un code très simple pour afficher l'arbre du chemin le plus court en utilisant le graphique créé avec la méthode `shortestTree` (sélectionner la couche de chaîne de caractères dans le panneau *couches* et remplacer les coordonnées par les vôtres).

**Avertissement :** Utilisez ce code uniquement à titre d'exemple, il crée beaucoup d'objets `QgsRubberBand` et peut être lent sur de grands ensembles de données.

```
1 from qgis.core import *
2 from qgis.gui import *
3 from qgis.analysis import *
4 from qgis.PyQt.QtCore import *
5 from qgis.PyQt.QtGui import *
6
7 vectorLayer = QgsVectorLayer('testdata/network.gpkg|layername=network_lines',
8   ↳'lines')
9 director = QgsVectorLayerDirector(vectorLayer, -1, '', '', '',
10   ↳QgsVectorLayerDirector.DirectionBoth)
11 strategy = QgsNetworkDistanceStrategy()
12 director.addStrategy(strategy)
13 builder = QgsGraphBuilder(vectorLayer.crs())
14
15 pStart = QgsPointXY(1179661.925139, 5419188.074362)
16 tiedPoint = director.makeGraph(builder, [pStart])
17 pStart = tiedPoint[0]
```

(suite sur la page suivante)



(suite de la page précédente)

```

16 graph = builder.graph()
17
18
19 idStart = graph.findVertex(pStart)
20
21 tree = QgsGraphAnalyzer.shortestTree(graph, idStart, 0)
22
23 i = 0
24 while (i < tree.edgeCount()):
25     rb = QgsRubberBand(iface.mapCanvas())
26     rb.setColor (Qt.red)
27     rb.addPoint (tree.vertex(tree.edge(i).fromVertex()).point())
28     rb.addPoint (tree.vertex(tree.edge(i).toVertex()).point())
29     i = i + 1

```

Même chose mais en utilisant la méthode `dijkstra`.

```

1 from qgis.core import *
2 from qgis.gui import *
3 from qgis.analysis import *
4 from qgis.PyQt.QtCore import *
5 from qgis.PyQt.QtGui import *
6
7 vectorLayer = QgsVectorLayer('testdata/network.gpkg|layername=network_lines',
8 ↪ 'lines')
9
10 director = QgsVectorLayerDirector(vectorLayer, -1, '', '', '',
11 ↪ QgsVectorLayerDirector.DirectionBoth)
12 strategy = QgsNetworkDistanceStrategy()
13 director.addStrategy(strategy)
14 builder = QgsGraphBuilder(vectorLayer.crs())
15
16 pStart = QgsPointXY(1179661.925139, 5419188.074362)
17 tiedPoint = director.makeGraph(builder, [pStart])
18 pStart = tiedPoint[0]
19
20 graph = builder.graph()
21
22 idStart = graph.findVertex(pStart)
23
24 (tree, costs) = QgsGraphAnalyzer.dijkstra(graph, idStart, 0)
25
26 for edgeId in tree:
27     if edgeId == -1:
28         continue
29     rb = QgsRubberBand(iface.mapCanvas())
30     rb.setColor (Qt.red)
31     rb.addPoint (graph.vertex(graph.edge(edgeId).fromVertex()).point())
32     rb.addPoint (graph.vertex(graph.edge(edgeId).toVertex()).point())

```

### 18.3.1 Trouver les chemins les plus courts

Pour trouver le chemin optimal entre deux points, l'approche suivante est utilisée. Les deux points (début A et fin B) sont « liés » au graphique lors de sa construction. Ensuite, en utilisant la méthode `shortestTree` ou `dijkstra`, nous construisons l'arbre du chemin le plus court avec une racine au point de départ A. Dans le même arbre, nous trouvons également le point final B et commençons à parcourir l'arbre du point B au point A. L'algorithme complet peut être écrit sous la forme :

```

1 assign T = B
2 while T != B
3     add point T to path
4     get incoming edge for point T
5     look for point TT, that is start point of this edge
6     assign T = TT
7 add point A to path

```

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

Voici un exemple de code pour la console Python QGIS (vous devrez peut-être charger et sélectionner une couche de chaîne de caractères dans la table des matières et remplacer les coordonnées dans le code par les vôtres) qui utilise la méthode `shortestTree`.

```

1 from qgis.core import *
2 from qgis.gui import *
3 from qgis.analysis import *
4
5 from qgis.PyQt.QtCore import *
6 from qgis.PyQt.QtGui import *
7
8 vectorLayer = QgsVectorLayer('testdata/network.gpkg|layername=network_lines',
9 ↪ 'lines')
10 builder = QgsGraphBuilder(vectorLayer.sourceCrs())
11 director = QgsVectorLayerDirector(vectorLayer, -1, '|', '|', '|', ↪
12 ↪ QgsVectorLayerDirector.DirectionBoth)
13
14 startPoint = QgsPointXY(1179661.925139,5419188.074362)
15 endPoint = QgsPointXY(1180942.970617,5420040.097560)
16
17 tiedPoints = director.makeGraph(builder, [startPoint, endPoint])
18 tStart, tStop = tiedPoints
19
20 graph = builder.graph()
21 idxStart = graph.findVertex(tStart)
22
23 tree = QgsGraphAnalyzer.shortestTree(graph, idxStart, 0)
24
25 idxStart = tree.findVertex(tStart)
26 idxEnd = tree.findVertex(tStop)
27
28 if idxEnd == -1:
29     raise Exception('No route!')
30
31 # Add last point
32 route = [tree.vertex(idxEnd).point()]
33
34 # Iterate the graph
35 while idxEnd != idxStart:
36     edgeIds = tree.vertex(idxEnd).incomingEdges()
37     if len(edgeIds) == 0:
38         break
39     edge = tree.edge(edgeIds[0])

```

(suite sur la page suivante)

(suite de la page précédente)

```

38     route.insert(0, tree.vertex(edge.fromVertex()).point())
39     idxEnd = edge.fromVertex()
40
41     # Display
42     rb = QgsRubberBand(iface.mapCanvas())
43     rb.setColor(Qt.green)
44
45     # This may require coordinate transformation if project's CRS
46     # is different than layer's CRS
47     for p in route:
48         rb.addPoint(p)

```

Et voici le même échantillon mais en utilisant la méthode `dijkstra`.

```

1  from qgis.core import *
2  from qgis.gui import *
3  from qgis.analysis import *
4
5  from qgis.PyQt.QtCore import *
6  from qgis.PyQt.QtGui import *
7
8  vectorLayer = QgsVectorLayer('testdata/network.gpkg|layername=network_lines',
9  ↪ 'lines')
10 director = QgsVectorLayerDirector(vectorLayer, -1, '|', '|', '|', ↪
11 ↪ QgsVectorLayerDirector.DirectionBoth)
12 strategy = QgsNetworkDistanceStrategy()
13 director.addStrategy(strategy)
14
15 builder = QgsGraphBuilder(vectorLayer.sourceCrs())
16
17 startPoint = QgsPointXY(1179661.925139, 5419188.074362)
18 endPoint = QgsPointXY(1180942.970617, 5420040.097560)
19
20 tiedPoints = director.makeGraph(builder, [startPoint, endPoint])
21 tStart, tStop = tiedPoints
22
23 graph = builder.graph()
24 idxStart = graph.findVertex(tStart)
25 idxEnd = graph.findVertex(tStop)
26
27 (tree, costs) = QgsGraphAnalyzer.dijkstra(graph, idxStart, 0)
28
29 if tree[idxEnd] == -1:
30     raise Exception('No route!')
31
32 # Total cost
33 cost = costs[idxEnd]
34
35 # Add last point
36 route = [graph.vertex(idxEnd).point()]
37
38 # Iterate the graph
39 while idxEnd != idxStart:
40     idxEnd = graph.edge(tree[idxEnd]).fromVertex()
41     route.insert(0, graph.vertex(idxEnd).point())
42
43 # Display
44 rb = QgsRubberBand(iface.mapCanvas())
45 rb.setColor(Qt.red)
46
47 # This may require coordinate transformation if project's CRS

```

(suite sur la page suivante)

```

46 # is different than layer's CRS
47 for p in route:
48     rb.addPoint(p)

```

### 18.3.2 Surfaces de disponibilité

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

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

Pour trouver les zones de disponibilité, nous pouvons utiliser la méthode `dijkstra` de la classe `QgsGraphAnalyzer`. Il suffit de comparer les éléments du tableau des coûts avec une valeur prédéfinie. Si le coût [i] est inférieur ou égal à une valeur prédéfinie, alors le sommet i est à l'intérieur de la zone de disponibilité, sinon il est à l'extérieur.

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 :

```

1  director = QgsVectorLayerDirector(vectorLayer, -1, '', '', '', ↵
   ↪QgsVectorLayerDirector.DirectionBoth)
2  strategy = QgsNetworkDistanceStrategy()
3  director.addStrategy(strategy)
4  builder = QgsGraphBuilder(vectorLayer.crs())
5
6
7  pStart = QgsPointXY(1179661.925139, 5419188.074362)
8  delta = iface.mapCanvas().getCoordinateTransform().mapUnitsPerPixel() * 1
9
10 rb = QgsRubberBand(iface.mapCanvas(), True)
11 rb.setColor(Qt.green)
12 rb.addPoint(QgsPointXY(pStart.x() - delta, pStart.y() - delta))
13 rb.addPoint(QgsPointXY(pStart.x() + delta, pStart.y() - delta))
14 rb.addPoint(QgsPointXY(pStart.x() + delta, pStart.y() + delta))
15 rb.addPoint(QgsPointXY(pStart.x() - delta, pStart.y() + delta))
16
17 tiedPoints = director.makeGraph(builder, [pStart])
18 graph = builder.graph()
19 tStart = tiedPoints[0]
20
21 idStart = graph.findVertex(tStart)
22
23 (tree, cost) = QgsGraphAnalyzer.dijkstra(graph, idStart, 0)
24
25 upperBound = []
26 r = 1500.0
27 i = 0
28 tree.reverse()
29
30 while i < len(cost):
31     if cost[i] > r and tree[i] != -1:
32         outVertexId = graph.edge(tree[i]).toVertex()
33         if cost[outVertexId] < r:
34             upperBound.append(i)
35     i = i + 1

```

(suite sur la page suivante)

(suite de la page précédente)

```
36
37 for i in upperBound:
38     centerPoint = graph.vertex(i).point()
39     rb = QgsRubberBand(iface.mapCanvas(), True)
40     rb.setColor(Qt.red)
41     rb.addPoint(QgsPointXY(centerPoint.x() - delta, centerPoint.y() - delta))
42     rb.addPoint(QgsPointXY(centerPoint.x() + delta, centerPoint.y() - delta))
43     rb.addPoint(QgsPointXY(centerPoint.x() + delta, centerPoint.y() + delta))
44     rb.addPoint(QgsPointXY(centerPoint.x() - delta, centerPoint.y() + delta))
```



### 19.1 Introduction

QGIS Server, c'est trois choses différentes :

1. Bibliothèque QGIS Server : une bibliothèque qui fournit une API pour la création de services web OGC
2. QGIS Server FCGI : une application binaire FCGI `qgis_maserv.fcgi` qui, avec un serveur web, implémente un ensemble de services OGC (WMS, WFS, WCS etc.) et d'API OGC (WFS3/OAPIF)
3. Développement QGIS Server : une application binaire de serveur de développement `qgis_mapserver` qui implémente un ensemble de services OGC (WMS, WFS, WCS etc.) et des API OGC (WFS3/OAPIF)

Ce chapitre du livre de cuisine se concentre sur le premier sujet et, en expliquant l'utilisation de l'API QGIS Server, il montre comment il est possible d'utiliser Python pour étendre, améliorer ou personnaliser le comportement du serveur ou comment utiliser l'API QGIS Server pour intégrer QGIS Server dans une autre application.

Il existe plusieurs façons de modifier le comportement de QGIS Server ou d'étendre ses capacités pour offrir de nouveaux services ou API personnalisés, voici les principaux scénarios auxquels vous pouvez être confrontés :

- EMBEDDING → Utiliser l'API QGIS Server depuis une autre application Python
- STANDALONE → Run QGIS Server as a standalone WSGI/HTTP service
- FILTERS → Améliorer/personnaliser QGIS Server avec des plugins de filtrage
- SERVICES → Ajouter un nouveau *SERVICE*
- OGC APIs → Ajouter une nouvelle API *OGC*

L'intégration et les applications autonomes nécessitent l'utilisation de l'API Python de QGIS Server directement à partir d'un autre script ou application Python. Les autres options sont mieux adaptées lorsque vous souhaitez ajouter des fonctionnalités personnalisées à une application binaire standard de QGIS Server (FCGI ou serveur de développement) : dans ce cas, vous devrez écrire un plugin Python pour l'application serveur et enregistrer vos filtres, services ou API personnalisés.

## 19.2 Principes de base de l'API du serveur

Les classes fondamentales impliquées dans une application typique de QGIS Server sont les suivantes :

- `QgsServer` l'instance du serveur (typiquement une seule instance pour toute la durée de vie de l'application)
- `QgsServerRequest` l'objet de la requête (généralement recréé sur chaque requête)
- `QgsServerResponse` l'objet de réponse (généralement recréé à chaque requête)
- `QgsServer.handleRequest(request, response)` traite la requête et remplit la réponse

Le flux de travail QGIS Server CGI ou serveur de développement peut être résumé comme suit :

```

1 initialize the QgsApplication
2 create the QgsServer
3 the main server loop waits forever for client requests:
4     for each incoming request:
5         create a QgsServerRequest request
6         create a QgsServerResponse response
7         call QgsServer.handleRequest(request, response)
8             filter plugins may be executed
9         send the output to the client

```

Dans la méthode `QgsServer.handleRequest(request, response)` les callbacks des plugins de filtre sont appelés et `QgsServerRequest` et `QgsServerResponse` sont mis à la disposition des plugins par le biais de la `QgsServerInterface`.

**Avertissement :** Les classes QGIS Server ne sont pas sûres pour les threads, vous devez toujours utiliser un modèle de multitraitement ou des conteneurs lorsque vous construisez des applications évolutives basées sur l'API du serveur QGIS.

## 19.3 Autonome ou intégré

Pour les applications serveur autonomes ou intégré, vous devrez utiliser directement les classes de serveur mentionnées ci-dessus, en les intégrant dans une implémentation de serveur web qui gère toutes les interactions du protocole HTTP avec le client.

Voici un exemple minimal d'utilisation de l'API QGIS Server (sans la partie HTTP) :

```

1 from qgis.core import QgsApplication
2 from qgis.server import *
3 app = QgsApplication([], False)
4
5 # Create the server instance, it may be a single one that
6 # is reused on multiple requests
7 server = QgsServer()
8
9 # Create the request by specifying the full URL and an optional body
10 # (for example for POST requests)
11 request = QgsBufferServerRequest(
12     'http://localhost:8081/?MAP=/qgis-server/projects/helloworld.qgs' +
13     '&SERVICE=WMS&REQUEST=GetCapabilities')
14
15 # Create a response objects
16 response = QgsBufferServerResponse()
17
18 # Handle the request
19 server.handleRequest(request, response)
20
21 print(response.headers())
22 print(response.body().data().decode('utf8'))

```

(suite sur la page suivante)



(suite de la page précédente)

```

23
24 app.exitQgis()

```

Voici un exemple complet d'application autonome développée pour le test continu des intégrations sur le dépôt de code source de QGIS, il présente un large ensemble de filtres de plugins et de schémas d'authentification différents (non destinés à la production car ils ont été développés à des fins de test uniquement mais toujours intéressants pour l'apprentissage) :

[https://github.com/qgis/QGIS/blob/master/tests/src/python/qgis\\_wrapped\\_server.py](https://github.com/qgis/QGIS/blob/master/tests/src/python/qgis_wrapped_server.py)

## 19.4 Plugins de serveur

Les plugins python du serveur sont chargés une fois lorsque l'application QGIS Server démarre et peuvent être utilisés pour enregistrer des filtres, des services ou des API.

La structure d'un plugin serveur est très similaire à son homologue de bureau, un objet `QgsServerInterface` est mis à la disposition des plugins et ceux-ci peuvent enregistrer un ou plusieurs filtres, services ou API personnalisés dans le registre correspondant en utilisant une des méthodes exposées par l'interface serveur.

### 19.4.1 Plugins pour filtres de serveur

Les filtres existent en trois possibilités différentes et peuvent être instanciés en sous-classant l'une des classes ci-dessous et en appelant la méthode correspondante de `QgsServerInterface` :

Type de filtre	Classe de base	Enregistrement de <code>QgsServerInterface</code>
I/O	<code>QgsServerFilter</code>	<code>registerFilter</code>
Contrôle d'accès	<code>QgsAccessControlFilter</code>	<code>registerAccessControl</code>
Cache	<code>QgsServerCacheFilter</code>	<code>registerServerCache</code>

#### Filtres I/O

Les filtres I/O peuvent modifier l'entrée et la sortie du serveur (la demande et la réponse) des services de base (WMS, WFS, etc.), ce qui permet d'effectuer tout type de manipulation du flux de travail des services. Il est possible, par exemple, de restreindre l'accès à des couches sélectionnées, d'injecter une feuille de style XSL dans la réponse XML, d'ajouter un filigrane à une image WMS générée, etc.

A partir de là, il vous sera peut-être utile de jeter un coup d'oeil rapide à l'API [server plugins docs](#).

Chaque filtre doit mettre en œuvre au moins un des trois rappels :

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

Tous les filtres ont accès à l'objet requête/réponse (`QgsRequestHandler`) et peuvent manipuler toutes ses propriétés (entrée/sortie) et lever des exceptions (mais d'une manière assez particulière comme nous le verrons plus loin).

Voici le pseudo-code montrant comment le serveur traite une requête typique et quand les rappels du filtre sont appelés :

```

1 for each incoming request:
2     create GET/POST request handler
3     pass request to an instance of QgsServerInterface
4     call requestReady filters
5     if there is not a response:
6         if SERVICE is WMS/WFS/WCS:

```

(suite sur la page suivante)

```
7         create WMS/WFS/WCS service
8         call service's executeRequest
9             possibly call sendResponse for each chunk of bytes
10            sent to the client by a streaming services (WFS)
11         call responseComplete
12         call sendResponse
13     request handler sends the response to the client
```

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

### 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).

### sendResponse

Il est appelé chaque fois qu'une sortie est envoyée à **FCGI** `stdout` (et de là, au client), cela est normalement fait après que les services centraux aient terminé leur processus et après que le hook `responseComplete` ait été appelé, mais dans quelques cas, XML peut devenir si énorme qu'une implémentation XML de streaming était nécessaire (WFS `GetFeature` est l'un d'entre eux), dans ce cas, `sendResponse` est appelé plusieurs fois avant que la réponse ne soit complète (et avant que `responseComplete` ne soit appelé). La conséquence évidente est que `sendResponse` est normalement appelé une fois mais peut exceptionnellement être appelé plusieurs fois et dans ce cas (et seulement dans ce cas) il est également appelé avant `responseComplete`.

`sendResponse` est le meilleur endroit pour manipuler directement la sortie du service de base et tandis que `responseComplete` est généralement aussi une option, `sendResponse` est la seule option viable en cas de services de streaming.

### responseComplete

Il est appelé une fois lorsque les services centraux (s'ils sont touchés) ont terminé leur processus et que la demande est prête à être envoyée au client. Comme indiqué ci-dessus, il est normalement appelé avant `sendResponse` sauf pour les services de streaming (ou autres filtres de plugin) qui auraient pu appeler `sendResponse` plus tôt.

`responseComplete` est l'endroit idéal pour fournir l'implémentation de nouveaux services (WPS ou services personnalisés) et pour effectuer une manipulation directe de la sortie provenant des services de base (par exemple pour ajouter un filigrane sur une image WMS).

## Lever les exceptions d'un plugin

Un certain travail reste à faire sur ce sujet : l'implémentation actuelle peut distinguer les exceptions gérées et non gérées en définissant une propriété `QgsRequestHandler` à une instance de `QgsMapServiceException`, de cette façon le code C++ principal peut attraper les exceptions python gérées et ignorer les exceptions non gérées (ou mieux : les enregistrer).

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.

## Écriture d'une extension serveur

Un plugin serveur est un plugin Python QGIS standard tel que décrit dans *Développer des extensions Python*, qui fournit juste une interface supplémentaire (ou alternative) : un plugin de bureau QGIS typique a accès à l'application QGIS par le biais de la `QgisInterface`, un plugin serveur a seulement accès à une `QgsServerInterface` lorsqu'il est exécuté dans le contexte de l'application QGIS Server.

Pour que QGIS serveur sache qu'un plugin a une interface serveur, une entrée spéciale de métadonnées est nécessaire (dans `metadata.txt`) :

```
server=True
```

**Important** : Seuls les plugins qui ont le jeu de métadonnées `server=True` seront chargés et exécutés par QGIS Server.

L'exemple de plugin présenté ici (avec beaucoup d'autres) est disponible sur github à l'adresse <https://github.com/elpaso/qgis3-server-vagrant/tree/master/resources/web/plugins>, quelques plugins de serveur sont également publiés dans le dépôt officiel de plugins QGIS.

## Fichiers de l'extension

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

```
1 PYTHON_PLUGINS_PATH/
2   HelloServer/
3     __init__.py    --> *required*
4     HelloServer.py --> *required*
5     metadata.txt  --> *required*
```

### `__init__.py`

Ce fichier est requis par le système d'importation de Python. De plus, QGIS Server exige que ce fichier contienne une fonction `serverClassFactory()`, qui est appelée lorsque le plugin est chargé dans QGIS Server au démarrage du serveur. Elle reçoit une référence à l'instance de `QgsServerInterface` et doit retourner une instance de la classe de votre plugin. Voici à quoi ressemble le plugin d'exemple `__init__.py`.

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

## HelloServer.py

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

Un plugin serveur consiste généralement en un ou plusieurs callbacks regroupés dans les instances d'un `QgsServerFilter`.

Chaque `QgsServerFilter` implémente un ou plusieurs des callbacks suivants :

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

L'exemple suivant met en œuvre un filtre minimal qui imprime *HelloServer!* dans le cas où le paramètre **SERVICE** est égal à « HELLO ».

```

1 class HelloFilter(QgsServerFilter):
2
3     def __init__(self, serverIface):
4         super().__init__(serverIface)
5
6     def requestReady(self):
7         QgsMessageLog.logMessage("HelloFilter.requestReady")
8
9     def sendResponse(self):
10        QgsMessageLog.logMessage("HelloFilter.sendResponse")
11
12    def responseComplete(self):
13        QgsMessageLog.logMessage("HelloFilter.responseComplete")
14        request = self.serverInterface().requestHandler()
15        params = request.parameterMap()
16        if params.get('SERVICE', '').upper() == 'HELLO':
17            request.clear()
18            request.setResponseHeader('Content-type', 'text/plain')
19            # Note that the content is of type "bytes"
20            request.appendBody(b'HelloServer!')
```

Les filtres doivent être enregistrés dans la **serverIface** comme dans l'exemple suivant :

```

class HelloServerServer:
    def __init__(self, serverIface):
        serverIface.registerFilter(HelloFilter(), 100)
```

Le second paramètre de `registerFilter` fixe une priorité qui définit l'ordre des rappels ayant le même nom (la priorité la plus basse est invoquée en premier).

En utilisant les trois rappels, les plugins peuvent manipuler l'entrée et/ou la sortie du serveur de nombreuses manières différentes. À chaque instant, l'instance du plugin a accès à la `QgsRequestHandler` par le biais de la `QgsServerInterface`. La classe `QgsRequestHandler` a de nombreuses méthodes qui peuvent être utilisées pour modifier les paramètres d'entrée avant d'entrer dans le traitement de base du serveur (en utilisant `requestReady()`) ou après que la requête ait été traitée par les services de base (en utilisant `sendResponse()`).

Les exemples suivants montrent quelques cas d'utilisation courants :

## Modifier la couche en entrée

L'exemple de plugin contient un exemple de test qui modifie les paramètres d'entrée provenant de la chaîne de requête, dans cet exemple un nouveau paramètre est injecté dans le « parameterMap » (déjà analysé), ce paramètre est ensuite visible par les services centraux (WMS etc.), à la fin du traitement des services centraux nous vérifions que le paramètre est toujours là :

```

1 class ParamsFilter(QgsServerFilter):
2
3     def __init__(self, serverIface):
4         super(ParamsFilter, self).__init__(serverIface)
5
6     def requestReady(self):
7         request = self.serverInterface().requestHandler()
8         params = request.parameterMap()
9         request.setParameter('TEST_NEW_PARAM', 'ParamsFilter')
10
11    def responseComplete(self):
12        request = self.serverInterface().requestHandler()
13        params = request.parameterMap()
14        if params.get('TEST_NEW_PARAM') == 'ParamsFilter':
15            QgsMessageLog.logMessage("SUCCESS - ParamsFilter.responseComplete")
16        else:
17            QgsMessageLog.logMessage("FAIL - ParamsFilter.responseComplete")

```

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

```

1  src/core/qgsmessagelog.cpp: 45: (logMessage) [0ms] 2014-12-12T12:39:29 plugin[0] ↵
↵HelloServerServer - loading filter ParamsFilter
2  src/core/qgsmessagelog.cpp: 45: (logMessage) [1ms] 2014-12-12T12:39:29 Server[0] ↵
↵Server plugin HelloServer loaded!
3  src/core/qgsmessagelog.cpp: 45: (logMessage) [0ms] 2014-12-12T12:39:29 Server[0] ↵
↵Server python plugins loaded
4  src/mapserver/qgshhttprequesthandler.cpp: 547: (requestStringToParameterMap) [1ms] ↵
↵inserting pair SERVICE // HELLO into the parameter map
5  src/mapserver/qgsserverfilter.cpp: 42: (requestReady) [0ms] QgsServerFilter ↵
↵plugin default requestReady called
6  src/core/qgsmessagelog.cpp: 45: (logMessage) [0ms] 2014-12-12T12:39:29 plugin[0] ↵
↵SUCCESS - ParamsFilter.responseComplete

```

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).

---

**Astuce :** Si vous voulez vraiment implémenter un service personnalisé, il est recommandé de sous-classer `QgsService` et d'enregistrer votre service sur `registerFilter` en appelant son `registerService(service)`

---

## 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 :

```

1 from qgis.server import *
2 from qgis.PyQt.QtCore import *
3 from qgis.PyQt.QtGui import *
4
5 class WatermarkFilter(QgsServerFilter):
6
7     def __init__(self, serverIface):
8         super().__init__(serverIface)
9
10    def responseComplete(self):
11        request = self.serverInterface().requestHandler()
12        params = request.parameterMap()
13        # Do some checks
14        if (params.get('SERVICE').upper() == 'WMS' \
15            and params.get('REQUEST').upper() == 'GETMAP' \
16            and not request.exceptionRaised()):
17            QgsMessageLog.logMessage("WatermarkFilter.responseComplete: image_
↳ready %s" % request.parameter("FORMAT"))
18            # Get the image
19            img = QImage()
20            img.loadFromData(request.body())
21            # Adds the watermark
22            watermark = QImage(os.path.join(os.path.dirname(__file__), 'media/
↳watermark.png'))
23            p = QPainter(img)
24            p.drawImage(QRect( 20, 20, 40, 40), watermark)
25            p.end()
26            ba = QByteArray()
27            buffer = QBuffer(ba)
28            buffer.open(QIODevice.WriteOnly)
29            img.save(buffer, "PNG" if "png" in request.parameter("FORMAT") else
↳"JPG")
30            # Set the body
31            request.clearBody()
32            request.appendBody(ba)

```

Dans cet exemple, la valeur du paramètre **SERVICE** est vérifiée et si la demande entrante est un **WMS GETMAP** et qu'aucune exception n'a été définie par un plugin exécuté précédemment ou par le service central (WMS dans ce cas), l'image générée par le WMS est récupérée dans le tampon de sortie et l'image en filigrane est ajoutée. L'étape finale consiste à effacer le tampon de sortie et à le remplacer par l'image nouvellement générée. Veuillez noter que dans une situation réelle, nous devons également vérifier le type d'image demandé au lieu de supporter uniquement les PNG ou JPG.

## Filtres de contrôle d'accès

Les filtres de contrôle d'accès donnent au développeur un contrôle fin sur les couches, les entites et les attributs auxquels il peut accéder, les rappels suivants peuvent être mis en œuvre dans un filtre de contrôle d'accès :

- layerFilterExpression(layer)
- layerFilterSubsetString(layer)
- layerPermissions(layer)
- authorizedLayerAttributes(layer, attributes)
- allowToEdit(layer, feature)
- cacheKey()

## Fichiers de l'extension

Voici la structure des répertoires de notre exemple de plugin :

```

1 PYTHON_PLUGINS_PATH/
2   MyAccessControl/
3     __init__.py    --> *required*
4     AccessControl.py --> *required*
5     metadata.txt   --> *required*
```

### `__init__.py`

Ce fichier est requis par le système d'importation de Python. Comme pour tous les plugins QGIS Server , ce fichier contient une fonction `serverClassFactory()`, qui est appelée lorsque le plugin est chargé dans QGIS Server au démarrage. Elle reçoit une référence à une instance de `QgsServerInterface` et doit retourner une instance de la classe de votre plugin. Voici à quoi ressemble le plugin d'exemple `__init__.py` :

```

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

### `AccessControl.py`

```

1 class AccessControlFilter(QgsAccessControlFilter):
2
3     def __init__(self, server_iface):
4         super().__init__(server_iface)
5
6     def layerFilterExpression(self, layer):
7         """ Return an additional expression filter """
8         return super().layerFilterExpression(layer)
9
10    def layerFilterSubsetString(self, layer):
11        """ Return an additional subset string (typically SQL) filter """
12        return super().layerFilterSubsetString(layer)
13
14    def layerPermissions(self, layer):
15        """ Return the layer rights """
16        return super().layerPermissions(layer)
17
18    def authorizedLayerAttributes(self, layer, attributes):
19        """ Return the authorised layer attributes """
20        return super().authorizedLayerAttributes(layer, attributes)
21
22    def allowToEdit(self, layer, feature):
23        """ Are we authorise to modify the following geometry """
24        return super().allowToEdit(layer, feature)
25
26    def cacheKey(self):
27        return super().cacheKey()
28
29 class AccessControlServer:
30
31    def __init__(self, serverIface):
32        """ Register AccessControlFilter """
33        serverIface.registerAccessControl(AccessControlFilter(self.serverIface), 100)
```

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.

### 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 ».

### 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 ».

### layerPermissions

Limiter l'accès à la couche.

Retourne un objet de type `LayerPermissions`, qui a les propriétés :

- `canRead` voir dans le `GetCapabilities` et acces lecture seule.
- `canInsert` pour pouvoir insérer une nouvelle entité .
- `canUpdate` pour pouvoir mettre à jour une entité.
- `canDelete` pour pouvoir supprimer une entité.

Exemple :

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

Pour tout limiter à un accès en lecture seule.

### 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".



## 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.

## cacheKey

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

## 19.4.2 Services personnalisés

Dans QGIS Server, les services de base tels que WMS, WFS et WCS sont implémentés en tant que sous-classes de `QgsService`.

Pour implémenter un nouveau service qui sera exécuté lorsque le paramètre de la chaîne de requête SERVICE correspondra au nom du service, vous pouvez implémenter votre propre `QgsService` et enregistrer votre service sur le `serviceRegistry` en appelant son `registerService(service)`.

Voici un exemple d'un service de personnalisation appelé CUSTOM :

```
1 from qgis.server import QgsService
2 from qgis.core import QgsMessageLog
3
4 class CustomServiceService(QgsService):
5
6     def __init__(self):
7         QgsService.__init__(self)
8
9     def name(self):
10        return "CUSTOM"
11
12    def version(self):
13        return "1.0.0"
14
15    def allowMethod(method):
16        return True
17
18    def executeRequest(self, request, response, project):
19        response.setStatuscode(200)
20        QgsMessageLog.logMessage('Custom service executeRequest')
21        response.write("Custom service executeRequest")
22
23
24 class CustomService():
25
26    def __init__(self, serverIface):
27        serverIface.serviceRegistry().registerService(CustomServiceService())
```

### 19.4.3 API personnalisées

Dans QGIS Server, les API OGC de base telles que OAPIF (alias WFS3) sont implémentées sous forme de collections de `QgsServerOgcApiHandler` sous-classes qui sont enregistrées dans une instance de `QgsServerOgcApi` (ou sa classe parente `QgsServerApi`).

Pour implémenter une nouvelle API qui sera exécutée lorsque le chemin de l'URL correspondra à une certaine URL, vous pouvez implémenter vos propres instances `QgsServerOgcApiHandler`, les ajouter à une instance `QgsServerOgcApi` et enregistrez l'API sur le `serviceRegistry` en appelant son `registerApi(api)`.

Voici un exemple d'API personnalisée qui sera exécutée lorsque l'URL contient « /customapi » :

```

1 import json
2 import os
3
4 from qgis.PyQt.QtCore import QBuffer, QIODevice, QTextStream, QRegularExpression
5 from qgis.server import (
6     QgsServiceRegistry,
7     QgsService,
8     QgsServerFilter,
9     QgsServerOgcApi,
10    QgsServerQueryStringParameter,
11    QgsServerOgcApiHandler,
12 )
13
14 from qgis.core import (
15     QgsMessageLog,
16     QgsJsonExporter,
17     QgsCircle,
18     QgsFeature,
19     QgsPoint,
20     QgsGeometry,
21 )
22
23
24 class CustomApiHandler(QgsServerOgcApiHandler):
25
26     def __init__(self):
27         super(CustomApiHandler, self).__init__()
28         self.setContentTypes([QgsServerOgcApi.HTML, QgsServerOgcApi.JSON])
29
30     def path(self):
31         return QRegularExpression("/customapi")
32
33     def operationId(self):
34         return "CustomApiXYCircle"
35
36     def summary(self):
37         return "Creates a circle around a point"
38
39     def description(self):
40         return "Creates a circle around a point"
41
42     def linkTitle(self):
43         return "Custom Api XY Circle"
44
45     def linkType(self):
46         return QgsServerOgcApi.data
47
48     def handleRequest(self, context):
49         """Simple Circle"""
50
51         values = self.values(context)

```

(suite sur la page suivante)

(suite de la page précédente)

```

52     x = values['x']
53     y = values['y']
54     r = values['r']
55     f = QgsFeature()
56     f.setAttributes([x, y, r])
57     f.setGeometry(QgsCircle(QgsPoint(x, y), r).toCircularString())
58     exporter = QgsJsonExporter()
59     self.write(json.loads(exporter.exportFeature(f)), context)
60
61     def templatePath(self, context):
62         # The template path is used to serve HTML content
63         return os.path.join(os.path.dirname(__file__), 'circle.html')
64
65     def parameters(self, context):
66         return [QgsServerQueryStringParameter('x', True, ↵
↵QgsServerQueryStringParameter.Type.Double, 'X coordinate'),
67                 QgsServerQueryStringParameter(
68                     'y', True, QgsServerQueryStringParameter.Type.Double, 'Y↵
↵coordinate'),
69                 QgsServerQueryStringParameter('r', True, ↵
↵QgsServerQueryStringParameter.Type.Double, 'radius')]
70
71
72 class CustomApi():
73
74     def __init__(self, serverIface):
75         api = QgsServerOgcApi(serverIface, '/customapi',
76                               'custom api', 'a custom api', '1.1')
77         handler = CustomApiHandler()
78         api.registerHandler(handler)
79         serverIface.serviceRegistry().registerApi(api)

```

Les extraits de code sur cette page nécessitent les importations suivantes si vous êtes en dehors de la console pyqgis :

```

1  from qgis.PyQt.QtCore import (
2      QRectF,
3  )
4
5  from qgis.core import (
6      QgsProject,
7      QgsLayerTreeModel,
8  )
9
10 from qgis.gui import (
11     QgsLayerTreeView,
12 )

```



### 20.1 Interface utilisateur

#### Changer l'apparence

```
1 from qgis.PyQt.QtWidgets import QApplication
2
3 app = QApplication.instance()
4 app.setStyleSheet(".QWidget {color: blue; background-color: yellow;}")
5 # You can even read the stylesheet from a file
6 with open("testdata/file.qss") as qss_file_content:
7     app.setStyleSheet(qss_file_content.read())
```

#### Changer l'icône et le titre

```
1 from qgis.PyQt.QtGui import QIcon
2
3 icon = QIcon("/path/to/logo/file.png")
4 iface.mainWindow().setWindowIcon(icon)
5 iface.mainWindow().setWindowTitle("My QGIS")
```

### 20.2 Réglages

#### Get QgsSettings list

```
1 from qgis.core import QgsSettings
2
3 qs = QgsSettings()
4
5 for k in sorted(qs.allKeys()):
6     print(k)
```

## 20.3 Barres d'outils

### Supprimer une barre d'outils

```
1 toolbar = iface.helpToolBar()
2 parent = toolbar.parentWidget()
3 parent.removeToolBar(toolbar)
4
5 # and add again
6 parent.addToolBar(toolbar)
```

### Supprimer la barre d'outils action

```
actions = iface.attributesToolBar().actions()
iface.attributesToolBar().clear()
iface.attributesToolBar().addAction(actions[4])
iface.attributesToolBar().addAction(actions[3])
```

## 20.4 Menus

### Supprimer menu

```
1 # for example Help Menu
2 menu = iface.helpMenu()
3 menubar = menu.parentWidget()
4 menubar.removeAction(menu.menuAction())
5
6 # and add again
7 menubar.addAction(menu.menuAction())
```

## 20.5 Canevas

### Accéder au canevas

```
canvas = iface.mapCanvas()
```

### Changer la couleur du canevas

```
from qgis.PyQt.QtCore import Qt

iface.mapCanvas().setCanvasColor(Qt.black)
iface.mapCanvas().refresh()
```

### Intervalle de mise à jour de la carte

```
from qgis.core import QgsSettings
# Set milliseconds (150 milliseconds)
QgsSettings().setValue("/qgis/map_update_interval", 150)
```

## 20.6 Couches

### Ajouter une couche vecteur

```
layer = iface.addVectorLayer("testdata/airports.shp", "layer name you like", "ogr")
if not layer or not layer.isValid():
    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

```
1 from qgis.core import QgsVectorLayer
2 layer = QgsVectorLayer("Point?crs=EPSG:4326", "layer name you like", "memory")
3 QgsProject.instance().addMapLayer(layer)
4
5 layers_names = []
6 for layer in QgsProject.instance().mapLayers().values():
7     layers_names.append(layer.name())
8
9 print("layers TOC = {}".format(layers_names))
```

```
layers TOC = ['layer name you like']
```

### Sinon

```
layers_names = [layer.name() for layer in QgsProject.instance().mapLayers().
↳values()]
print("layers TOC = {}".format(layers_names))
```

```
layers TOC = ['layer name you like']
```

### Rechercher une couche par son nom

```
from qgis.core import QgsProject

layer = QgsProject.instance().mapLayersByName("layer name you like")[0]
print(layer.name())
```

```
layer name you like
```

### Définir la couche active

```
from qgis.core import QgsProject

layer = QgsProject.instance().mapLayersByName("layer name you like")[0]
iface.setActiveLayer(layer)
```

### Rafraîchissement de la couche à l'intervalle

```
1 from qgis.core import QgsProject
2
```

(suite sur la page suivante)

(suite de la page précédente)

```
3 layer = QgsProject.instance().mapLayersByName("layer name you like")[0]
4 # Set seconds (5 seconds)
5 layer.setAutoRefreshInterval(5000)
6 # Enable auto refresh
7 layer.setAutoRefreshEnabled(True)
```

### Afficher des méthodes

```
dir(layer)
```

### Ajouter une nouvelle entité à l'aide d'un formulaire d'attributs

```
1 from qgis.core import QgsFeature, QgsGeometry
2
3 feat = QgsFeature()
4 geom = QgsGeometry()
5 feat.setGeometry(geom)
6 feat.setFields(layer.fields())
7
8 iface.openFeatureForm(layer, feat, False)
```

### Ajouter une nouvelle entité sans un formulaire d'attributs

```
1 from qgis.core import QgsGeometry, QgsPointXY, QgsFeature
2
3 pr = layer.dataProvider()
4 feat = QgsFeature()
5 feat.setGeometry(QgsGeometry.fromPointXY(QgsPointXY(10,10)))
6 pr.addFeatures([feat])
```

### Retourner entités

```
for f in layer.getFeatures():
    print(f)
```

```
<qgis._core.QgsFeature object at 0x7f45cc64b678>
```

### Obtenir les entités sélectionnées

```
for f in layer.selectedFeatures():
    print(f)
```

### Obtenir l'id des entités sélectionnées

```
selected_ids = layer.selectedFeatureIds()
print(selected_ids)
```

### Créer une couche mémoire à partir de certaines entites Ids

```
from qgis.core import QgsFeatureRequest

memory_layer = layer.materialize(QgsFeatureRequest().setFilterFids(layer.
    ↳selectedFeatureIds()))
QgsProject.instance().addMapLayer(memory_layer)
```

### Renvoyer la géométrie

```
# Point layer
for f in layer.getFeatures():
    geom = f.geometry()
    print('%f, %f' % (geom.asPoint().y(), geom.asPoint().x()))
```



```
10.000000, 10.000000
```

### Déplacer une géométrie

```
1 from qgis.core import QgsFeature, QgsGeometry
2 poly = QgsFeature()
3 geom = QgsGeometry.fromWkt("POINT(7 45)")
4 geom.translate(1, 1)
5 poly.setGeometry(geom)
6 print(poly.geometry())
```

```
<QgsGeometry: Point (8 46)>
```

### Définir le SCR

```
from qgis.core import QgsProject, QgsCoordinateReferenceSystem

for layer in QgsProject.instance().mapLayers().values():
    layer.setCrs(QgsCoordinateReferenceSystem('EPSG:4326'))
```

### Afficher le SCR

```
1 from qgis.core import QgsProject
2
3 for layer in QgsProject.instance().mapLayers().values():
4     crs = layer.crs().authid()
5     layer.setName('{} ({}).format(layer.name(), crs))
```

### Cacher une colonne du champ

```
1 from qgis.core import QgsEditorWidgetSetup
2
3 def fieldVisibility (layer, fname):
4     setup = QgsEditorWidgetSetup('Hidden', {})
5     for i, column in enumerate(layer.fields()):
6         if column.name() == fname:
7             layer.setEditorWidgetSetup(idx, setup)
8             break
9     else:
10        continue
```

### Couche depuis le WKT

```
1 from qgis.core import QgsVectorLayer, QgsFeature, QgsGeometry, QgsProject
2
3 layer = QgsVectorLayer('Polygon?crs=epsg:4326', 'Mississippi', 'memory')
4 pr = layer.dataProvider()
5 poly = QgsFeature()
6 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))")
7 poly.setGeometry(geom)
8 pr.addFeatures([poly])
9 layer.updateExtents()
10 QgsProject.instance().addMapLayers([layer])
```

### Charger toutes les couches vecteur à partir de GeoPackage

```
1 fileName = "testdata/sublayers.gpkg"
2 layer = QgsVectorLayer(fileName, "test", "ogr")
3 subLayers = layer.dataProvider().subLayers()
```

(suite sur la page suivante)

```

4
5 for subLayer in subLayers:
6     name = subLayer.split('!!!:!!!')[1]
7     uri = "%s|layername=%s" % (fileName, name,)
8     # Create layer
9     sub_vlayer = QgsVectorLayer(uri, name, 'ogr')
10    # Add layer to map
11    QgsProject.instance().addMapLayer(sub_vlayer)

```

### Charger couche tuile (couche XYZ)

```

1 from qgis.core import QgsRasterLayer, QgsProject
2
3 def loadXYZ(url, name):
4     rasterLyr = QgsRasterLayer("type=xyz&url=" + url, name, "wms")
5     QgsProject.instance().addMapLayer(rasterLyr)
6
7 urlWithParams = 'type=xyz&url=https://a.tile.openstreetmap.org/%7Bz%7D/%7Bx%7D/%7By
↵%7D.png&zmax=19&zmin=0&crs=EPSG3857'
8 loadXYZ(urlWithParams, 'OpenStreetMap')

```

### Enlever toutes les couches

```
QgsProject.instance().removeAllMapLayers()
```

### Enlever tout

```
QgsProject.instance().clear()
```

## 20.7 Table des matières

### Accès aux couches cochées

```
iface.mapCanvas().layers()
```

### Supprimer le menu contextuel

```

1 ltv = iface.layerTreeView()
2 mp = ltv.menuProvider()
3 ltv.setMenuProvider(None)
4 # Restore
5 ltv.setMenuProvider(mp)

```

## 20.8 Table des matières (avancé)

### Noeud racine

```

1 from qgis.core import QgsVectorLayer, QgsProject, QgsLayerTreeLayer
2
3 root = QgsProject.instance().layerTreeRoot()
4 node_group = root.addGroup("My Group")
5
6 layer = QgsVectorLayer("Point?crs=EPSG:4326", "layer name you like", "memory")
7 QgsProject.instance().addMapLayer(layer, False)
8

```

(suite de la page précédente)

```

9 node_group.addLayer(layer)
10
11 print(root)
12 print(root.children())

```

### Accéder au premier nœud enfant

```

1 from qgis.core import QgsLayerTreeGroup, QgsLayerTreeLayer, QgsLayerTree
2
3 child0 = root.children()[0]
4 print(child0.name())
5 print(type(child0))
6 print(isinstance(child0, QgsLayerTreeLayer))
7 print(isinstance(child0.parent(), QgsLayerTree))

```

```

My Group
<class 'qgis._core.QgsLayerTreeGroup'>
False
True

```

### Trouver des groupes et des nœuds

```

1 from qgis.core import QgsLayerTreeGroup, QgsLayerTreeLayer
2
3 def get_group_layers(group):
4     print('- group: ' + group.name())
5     for child in group.children():
6         if isinstance(child, QgsLayerTreeGroup):
7             # Recursive call to get nested groups
8             get_group_layers(child)
9         else:
10            print(' - layer: ' + child.name())
11
12
13 root = QgsProject.instance().layerTreeRoot()
14 for child in root.children():
15     if isinstance(child, QgsLayerTreeGroup):
16         get_group_layers(child)
17     elif isinstance(child, QgsLayerTreeLayer):
18         print('- layer: ' + child.name())

```

```

- group: My Group
- layer: layer name you like

```

### Trouver un groupe à partir du nom

```
print(root.findGroup("My Group"))
```

```
<qgis._core.QgsLayerTreeGroup object at 0x7fd75560cee8>
```

### Trouver la couche par id

```
print(root.findLayer(layer.id()))
```

```
<qgis._core.QgsLayerTreeLayer object at 0x7f56087af288>
```

### Ajouter couche

```
1 from qgis.core import QgsVectorLayer, QgsProject
2
3 layer1 = QgsVectorLayer("Point?crs=EPSG:4326", "layer name you like 2", "memory")
4 QgsProject.instance().addMapLayer(layer1, False)
5 node_layer1 = root.addLayer(layer1)
6 # Remove it
7 QgsProject.instance().removeMapLayer(layer1)
```

### Ajouter groupe

```
1 from qgis.core import QgsLayerTreeGroup
2
3 node_group2 = QgsLayerTreeGroup("Group 2")
4 root.addChildNode(node_group2)
5 QgsProject.instance().mapLayersByName("layer name you like")[0]
```

### Déplacer la couche

```
1 layer = QgsProject.instance().mapLayersByName("layer name you like")[0]
2 root = QgsProject.instance().layerTreeRoot()
3
4 myLayer = root.findLayer(layer.id())
5 myClone = myLayer.clone()
6 parent = myLayer.parent()
7
8 myGroup = root.findGroup("My Group")
9 # Insert in first position
10 myGroup.insertChildNode(0, myClone)
11
12 parent.removeChildNode(myLayer)
```

### Déplacer la couche chargée vers un groupe spécifique

```
1 QgsProject.instance().addMapLayer(layer, False)
2
3 root = QgsProject.instance().layerTreeRoot()
4 myGroup = root.findGroup("My Group")
5 myOriginalLayer = root.findLayer(layer.id())
6 myLayer = myOriginalLayer.clone()
7 myGroup.insertChildNode(0, myLayer)
8 parent.removeChildNode(myOriginalLayer)
```

### Changer la visibilité

```
myGroup.setItemVisibilityChecked(False)
myLayer.setItemVisibilityChecked(False)
```

### Est le groupe sélectionné

```
1 def isMyGroupSelected( groupName ):
2     myGroup = QgsProject.instance().layerTreeRoot().findGroup( groupName )
3     return myGroup in iface.layerTreeView().selectedNodes()
4
5 print(isMyGroupSelected( 'my group name' ))
```

```
False
```

### Étendre le nœud

```
print(myGroup.isExpanded())
myGroup.setExpanded(False)
```

**Truc de nœud caché**

```

1 from qgis.core import QgsProject
2
3 model = iface.layerTreeView().layerTreeModel()
4 ltv = iface.layerTreeView()
5 root = QgsProject.instance().layerTreeRoot()
6
7 layer = QgsProject.instance().mapLayersByName('layer name you like')[0]
8 node = root.findLayer(layer.id())
9
10 index = model.node2index(node)
11 ltv.setRowHidden(index.row(), index.parent(), True)
12 node.setCustomProperty('nodeHidden', 'true')
13 ltv.setCurrentIndex(model.node2index(root))

```

**signale du noeud**

```

1 def onWillAddChildren(node, indexFrom, indexTo):
2     print("WILL ADD", node, indexFrom, indexTo)
3
4 def onAddedChildren(node, indexFrom, indexTo):
5     print("ADDED", node, indexFrom, indexTo)
6
7 root.willAddChildren.connect(onWillAddChildren)
8 root.addedChildren.connect(onAddedChildren)

```

**Supprimer couche**

```
root.removeLayer(layer)
```

**Supprimer groupe**

```
root.removeChildNode(node_group2)
```

**Créer nouvelle table des matières (TDM)**

```

1 root = QgsProject.instance().layerTreeRoot()
2 model = QgsLayerTreeModel(root)
3 view = QgsLayerTreeView()
4 view.setModel(model)
5 view.show()

```

**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")

```

## 20.9 Traitement algorithmes

### Obtenir la liste d'algorithmes

```

1 from qgis.core import QgsApplication
2
3 for alg in QgsApplication.processingRegistry().algorithms():
4     if 'buffer' == alg.name():
5         print("{}: {} --> {}".format(alg.provider().name(), alg.name(), alg.
↳ displayName()))

```

```
QGIS (native c++):buffer --> Buffer
```

### Obtenir l'aide des algorithmes

Sélection aléatoire

```

from qgis import processing
processing.algorithmHelp("native:buffer")

```

```
...
```

### Exécuter l'algorithme

Dans cet exemple, le résultat est stocké dans une couche en mémoire temporaire, qui est ajoutée au projet.

```

from qgis import processing
result = processing.run("native:buffer", {'INPUT': layer, 'OUTPUT': 'memory:'})
QgsProject.instance().addMapLayer(result['OUTPUT'])

```

```
Processing(0): Results: {'OUTPUT': 'output_d27a2008_970c_4687_b025_f057abbd7319'}
```

### Combien d'algorithmes ?

```
len(QgsApplication.processingRegistry().algorithms())
```

### Combien de fournisseurs y a-t-il ?

```

from qgis.core import QgsApplication

len(QgsApplication.processingRegistry().providers())

```

### Combien d'expressions y a-t-il ?

```

from qgis.core import QgsExpression

len(QgsExpression.Functions())

```

## 20.10 Décorateurs

### Droits d'auteur

```

1 from qgis.PyQt.Qt import QTextDocument
2 from qgis.PyQt.QtGui import QFont
3
4 mQFont = "Sans Serif"
5 mQFontSize = 9
6 mLabelQString = "© QGIS 2019"

```

(suite sur la page suivante)

(suite de la page précédente)

```

7 mMarginHorizontal = 0
8 mMarginVertical = 0
9 mLabelQColor = "#FF0000"
10
11 INCHES_TO_MM = 0.0393700787402 # 1 millimeter = 0.0393700787402 inches
12 case = 2
13
14 def add_copyright(p, text, xOffset, yOffset):
15     p.translate( xOffset , yOffset )
16     text.drawContents(p)
17     p.setWorldTransform( p.worldTransform() )
18
19 def _on_render_complete(p):
20     deviceHeight = p.device().height() # Get paint device height on which this_
↳painter is currently painting
21     deviceWidth = p.device().width() # Get paint device width on which this_
↳painter is currently painting
22     # Create new container for structured rich text
23     text = QTextDocument()
24     font = QFont()
25     font.setFamily(mQFont)
26     font.setPointSize(int(mQFontSize))
27     text.setDefaultFont(font)
28     style = "<style type=\"text/css\"> p {color: " + mLabelQColor + "}</style>"
29     text.setHtml( style + "<p>" + mLabelQString + "</p>" )
30     # Text Size
31     size = text.size()
32
33     # RenderMillimeters
34     pixelsInchX = p.device().logicalDpiX()
35     pixelsInchY = p.device().logicalDpiY()
36     xOffset = pixelsInchX * INCHES_TO_MM * int(mMarginHorizontal)
37     yOffset = pixelsInchY * INCHES_TO_MM * int(mMarginVertical)
38
39     # Calculate positions
40     if case == 0:
41         # Top Left
42         add_copyright(p, text, xOffset, yOffset)
43
44     elif case == 1:
45         # Bottom Left
46         yOffset = deviceHeight - yOffset - size.height()
47         add_copyright(p, text, xOffset, yOffset)
48
49     elif case == 2:
50         # Top Right
51         xOffset = deviceWidth - xOffset - size.width()
52         add_copyright(p, text, xOffset, yOffset)
53
54     elif case == 3:
55         # Bottom Right
56         yOffset = deviceHeight - yOffset - size.height()
57         xOffset = deviceWidth - xOffset - size.width()
58         add_copyright(p, text, xOffset, yOffset)
59
60     elif case == 4:
61         # Top Center
62         xOffset = deviceWidth / 2
63         add_copyright(p, text, xOffset, yOffset)
64
65     else:

```

(suite sur la page suivante)

(suite de la page précédente)

```
66     # Bottom Center
67     yOffset = deviceHeight - yOffset - size.height()
68     xOffset = deviceWidth / 2
69     add_copyright(p, text, xOffset, yOffset)
70
71     # Emitted when the canvas has rendered
72     iface.mapCanvas().renderComplete.connect(_on_render_complete)
73     # Repaint the canvas map
74     iface.mapCanvas().refresh()
```

## 20.11 Compositeur

### Obtenir la mise en page d'impression par nom

```
1 composerTitle = 'MyComposer' # Name of the composer
2
3 project = QgsProject.instance()
4 projectLayoutManager = project.layoutManager()
5 layout = projectLayoutManager.layoutByName(composerTitle)
```

## 20.12 Sources

- [API Python de QGIS \(PyQGIS\)](#)
- [API C++ de QGIS](#)
- [Questions QGIS sur StackOverFlow](#)
- [Script de Klas Karlsson](#)