PyQGIS developer cookbook

Version 2.6

QGIS Project

22 May 2015

Table des matières

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 une bonne idée générale des principales fonctionnalités.

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

****A FAIRE : **** Faire fonctionner PyQGIS (compilation manuelle, débogage)

Il y a de nombreuses façons d'utiliser les dépendances python de QGIS ; elles sont mentionnées en détail dans les sections suivantes :
– lancer des commandes dans la console Python de QGIS
– créer et utiliser des plugins en Python
– créer des applications personnalisées basées sur l'API QGIS
Il y a une documentation complète de l'API QGIS qui détaille les classes des librairies QGIS. l'API Python de QGIS est presque identique à l'API en C++.

Il existe quelques ressources sur la programmation sous PyQGIS sur blog QGIS (Note du traducteur : ce site est obsolète). Vous pouvez consulter QGIS tutorial ported to Python pour visualiser quelques exemple d'applications externes. Une bonne source d'information en ce qui concerne les extensions est de simplement télécharger des extensions depuis le dépôt des extensions et d'examiner leur code. De plus, le répertoire `python/plugins/` de votre installation QGIS contient quelques extensions qui vous permettront d'apprendre comment développer et comment réaliser la majorité des tâches courantes.

1.1 La Console Python

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

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

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

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

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

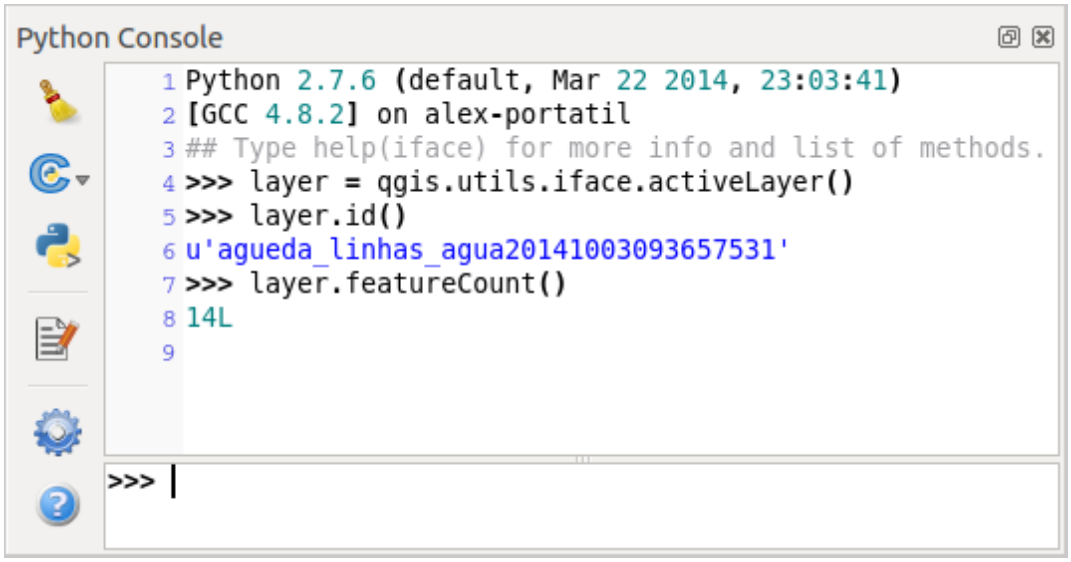

FIGURE 1.1 – La Console Python de QGIS

1.2 Extensions Python

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

De nombreuses extensions couvrant diverses fonctionnalités ont été écrites depuis l'introduction du support de Python. L'installateur d'extensions permet aux utilisateurs de facilement chercher, mettre à niveau et supprimer les extensions Python. Voir la page du ' Dépôt des Extensions Python <http ://www.qgis.org/wiki/Python_Plugin_Repositories>'_ pour diverses sources d'extensions.

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

1.3 Applications Python

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

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

1.3.1 Utiliser PyQGIS dans une application personnalisée

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.

D'abord, vous devez importer le module qgis, indiquer les chemins QGIS pour trouver les ressources (base de données des projections, fournisseurs de données, etc.). Lorsque vous définissez une base de chemin et que le deuxième argument vaut `True`, QGIS initialisera tous les chemins avec un répertoire qui sera le répertoire de base indiqué par défaut. Appeler la fonction `initQgis()` est important pour laisser QGIS rechercher les fournisseurs de données disponibles.

```
from qgis.core import *

# supply path to where is your qgis installed
QgsApplication.setPrefixPath("/path/to/qgis/installation", True)

# load providers
QgsApplication.initQgis()
```

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 :-)

Quand vous avez fini d'utiliser la librairie QGIS, appelez la fonction `exitQgis()` pour vous assurer que tout est nettoyé (par ex, effacer le registre de couche cartographique et supprimer les couches) :

```
QgsApplication.exitQgis()
```

1.3.2 Exécuter des applications personnalisées

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

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

Ceci peut être corrigé en définissant la variable d'environnement `PYTHONPATH`. Dans les commandes suivantes, `qgispath` doit être remplacé par le réel chemin d'accès au dossier d'installation de QGIS :
– sur Linux : **export PYTHONPATH=/qgispath/share/qgis/python**
– sur Windows : **set PYTHONPATH=c :\qgispath\python**
Le chemin vers les modules PyQGIS est maintenant connu. Néanmoins, ils dépendent des bibliothèques `qgis_core` et `qgis_gui` (les modules Python qui servent d'encapsulage). Le chemin vers ces bibliothèques est inconnu du système d'exploitation et vous allez encore récupérer une erreur d'import (le message peut varier selon le système) :

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

Corrigez ce problème en ajoutant les répertoires d'emplacement des bibliothèques QGIS au chemin de recherche de l'éditeur dynamique de liens :
– sur Linux : **export LD_LIBRARY_PATH=/qgispath/lib**
– sur Windows : **set PATH=C :\qgispath ;%PATH%**
Ces commandes peuvent être écrites dans un script de lancement qui gérera le démarrage. Lorsque vous déployez des applications personnalisées qui utilisent PyQGIS, il existe généralement deux possibilités :
– Imposer à l'utilisateur d'installer QGIS sur la plate-forme avant d'installer l'application. L'installateur de l'application devrait s'occuper des emplacements par défaut des bibliothèques QGIS et permettre à l'utilisateur de préciser un chemin si ce dernier n'est pas trouvé. Cette approche a l'avantage d'être plus simple mais elle impose plus d'actions à l'utilisateur.
– Créer un paquet QGIS qui contiendra votre application. Publier l'application sera plus complexe et le paquet d'installation sera plus volumineux mais l'utilisateur n'aura pas à télécharger et à installer d'autres logiciels.
Les deux modèles de déploiement peuvent être mélangés : déployer une application autonome sous Windows et Mac OS X et laisser l'installation de QGIS par l'utilisateur (via son gestionnaire de paquets) pour Linux.

CHAPITRE 2

Chargement de couches

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

2.1 Couches vectorielles

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

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

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

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

– La bibliothèque OGR (shapefiles et beaucoup d'autres formats) — la source de données est le chemin du fichier :

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

– Base de données PostGIS — la source de données est une chaîne de texte contenant toutes les informations nécessaires à la création d'une connexion à la base de données PostgreSQL. La classe `QgsDataSourceURI` peut générer ce texte pour vous. Notez que QGIS doit être compilé avec le support Postgres pour que ce fournisseur soit disponible. :

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

  vlayer = QgsVectorLayer(uri.uri(), "layer_name_you_like", "postgres")
  ```

– CSV ou autres fichiers de texte délimité — pour ouvrir un fichier délimité par le point-virgule, ayant des champs "x" et "y" respectivement pour coordonnées x et y, vous devriez utiliser quelque chose comme ceci :

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

Note : à partir de la version 1.7 de QGIS, la chaîne de caractère du fournisseur de données peut être structurée en tant qu'URL. Ainsi, le chemin doit être préfixé avec *file :///*. Il est également possible d'utiliser des géométries formatées en WKT (Well-Known-Text) à la place des champs "x" et "y" et de spécifier le système de référence de coordonnées. Par exemple

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

– Fichiers GPX — le fournisseur de données "gpx" lit les pistes, les routes et les points de navigation des fichiers gpx. Pour ouvrir un fichier, il faut spécifier le type utilisé (track/route/waypoint) dans l'url

```
uri = "path/to/gpx/file.gpx?type=track"
vlayer = QgsVectorLayer(uri, "layer_name_you_like", "gpx")
```

– Base de données Spatialite — supporté depuis la version 1.1 de QGIS. Tout comme avec les bases de données PostGIS, la classe `QgsDataSourceURI` peut être utilisée pour générer l'identifiant de la source de données :

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

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

– Géométries MySQL de type WKB, au travers d'OGR — la source de données est la chaîne de connexion à la table

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

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

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

L'URI peut être créée en utilisant la bilbiothèque standard `urllib`.

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

2.2 Couches raster

Pour accéder aux fichiers raster, QGIS utilise la bibliothèque GDAL. Elle gère un large nombre de formats de fichiers. Si vous avez du mal à ouvrir certains fichiers, vérifiez si votre installation de GDAL gère ce format particulier (tous les formats ne sont pas gérés par défaut). Pour ouvrir un raster depuis un fichier, indiquez son nom de fichier et son répertoire d'emplacement

```
fileName = "/path/to/raster/file.tif"
fileInfo = QFileInfo(fileName)
baseName = fileInfo.baseName()
rlayer = QgsRasterLayer(fileName, baseName)
```

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

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

```
layer_name = 'elevation'
uri = QgsDataSourceURI()
uri.setParam ('url', 'http://localhost:8080/geoserver/wcs')
uri.setParam ( "identifier", layer_name)
rlayer = QgsRasterLayer(uri, 'my_wcs_layer', 'wcs')
```

Vous pouvez également charger une couche raster depuis un serveur WMS. Néanmoins il ne sera pas possible d'utiliser les réponses du type GetCapabilities depuis l'API de QGIS : vous devez connaître les couches que vous voulez charger

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

2.3 Registre de couches cartographiques

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

Ajouter une couche au registre

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

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

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

A FAIRE : Plus d'informations sur le registre de cartes ?

CHAPITRE 3

Utiliser des couches raster

Cette section liste différentes opérations réalisables avec des couches raster.

3.1 Détails d'une couche

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

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

3.2 Style de représentation

Lorsqu'un raster est chargé, il récupère un style de représentation par défaut basé sur son type. Ce style peut être modifié dans les propriétés de la couche ou en programmant. On peut retrouver les styles d'affichage suivants :

Index	Constante : QgsRasterLater.X	Commentaire
1	SingleBandGray	Image mono-bande représentée avec une plage de niveaux de gris.
2	SingleBandPseudo-Color	Image mono-bande représentée avec un algorithme de pseudo-couleurs.
3	PalettedColor	Image “Palette” représentée avec une table de couleurs.
4	PalettedSingle-BandGray	Image “Palette” représentée en niveaux de gris.
5	PalettedSingle-BandPseudoColor	Couche “Palette” dessinée avec un algorithme de pseudo-couleur
7	MultiBandSingle-BandGray	Couche contenant au moins 2 bandes mais une seule bande est représentée selon une plage de niveaux de gris.
8	MultiBandSingle-BandPseudoColor	Couche contenant au moins 2 bandes mais une seule bande est représentée à l’aide d’un algorithme de pseudo-couleur.
9	MultiBandColor	Couche contenant au moins 2 bandes, calquée sur un espace de couleur RGB.

Pour interroger le style de représentation :

```
>>> rlayer.drawingStyle()
9
```

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

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

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

3.2.1 Rasters mono-bande

Ils sont rendus en niveaux de gris par défaut. Pour modifier le style de représentation à la pseudo-couleur :

```
>>> rlayer.setDrawingStyle(QgsRasterLayer.SingleBandPseudoColor)
>>> rlayer.setColorShadingAlgorithm(QgsRasterLayer.PseudoColorShader)
```

Le `PseudoColorShader` est un moteur de rendu qui met en valeur les faibles valeurs en bleu et les valeurs fortes en rouge. L’autre moteur, `FreakOutShader` utilise des couleurs plus fantaisistes et selon la documentation, il effrayera votre grand-mère et fera aboyer vos chiens !

Il existe également le moteur `ColorRampShader` qui calque les couleurs selon la carte des couleurs. Il dispose de trois modes d’interpolation des valeurs :
– linéaire (`INTERPOLATED`) : les couleurs résultent d’une interpolation linéaire des entrées de couleur de la carte qui sont en dessous et au dessus de la valeur du pixel actuel.
– discret (`DISCRETE`) : la couleur est utilisée depuis l’entrée de la carte de couleur avec une valeur supérieure ou égale.
– exact (`EXACT`) : la couleur n’est pas interpolée. Seuls les pixels dont la valeur équivaut aux entrées de la carte de couleur sont représentés.
Pour appliquer une rampe de couleur interpolée qui part de vert vers le jaune (pour les pixels dont la valeur s’échelonne de 0 à 255) :

```
>>> rlayer.setColorShadingAlgorithm(QgsRasterLayer.ColorRampShader)
>>> lst = [ QgsColorRampShader.ColorRampItem(0, QColor(0,255,0)), \
    QgsColorRampShader.ColorRampItem(255, QColor(255,255,0)) ]
>>> fcn = rlayer.rasterShader().rasterShaderFunction()
```

```
>>> fcn.setColorRampType(QgsColorRampShader.INTERPOLATED)
>>> fcn.setColorRampItemList(lst)
```

Pour retourner aux niveaux de gris par défaut, utilisez :

```
>>> rlayer.setDrawingStyle(QgsRasterLayer.SingleBandGray)
```

3.2.2 Rasters multi-bandes

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

```
>>> rlayer.setGreenBandName(rlayer.bandName(1))
>>> rlayer.setRedBandName(rlayer.bandName(2))
```

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

```
>>> rlayer.setDrawingStyle(QgsRasterLayer.MultiBandSingleBandPseudoColor)
>>> rlayer.setGrayBandName(rlayer.bandName(1))
>>> rlayer.setColorShadingAlgorithm(QgsRasterLayer.PseudoColorShader)
>>> # now set the shader
```

3.3 Rafraîchir les couches

Si vous avez effectué des modifications sur le style d'une couche et tenez à ce qu'elles soient automatiquement visibles pour l'utilisateur, appelez ces méthodes :

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

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

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

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

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

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

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

3.4 Interrogation des données

Voici comment faire pour réaliser une interrogation de donnée sur les bandes d'une couche raster sur un point donné :

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

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

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

CHAPITRE 4

Utilisation de couches vectorielles

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

4.1 Itérer sur une couche vecteur

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

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

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

    # fetch attributes
    attrs = feature.attributes()

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

Les attributs peuvent être retrouvés selon leur index :

```
idx = layer.fieldNameIndex('name')
print feature.attributes()[idx]
```

4.1.1 Itérer sur une sélection d'entités

Méthodes pratiques.

Pour les cas présentés ci-dessus, au cas où vous présumez qu'il y a une sélection en cours dans la couche vecteur, vous pouvez utiliser la méthode `features()` de l'extension interne Traitements de la manière suivant :

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

Ce code va itérer sur toutes les entités d'une couche, dans le cas où il n'y a pas de sélection, ou sur les entités sélectionnés dans le cas contraire.

Si vous avez uniquement besoin des entités sélectionées, vous pouvez utiliser la méthode :func : *selectedFeatures* de la couche vecteur :

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

4.1.2 Itérer sur un sous-ensemble d'entités

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

```
request=QgsFeatureRequest()
request.setFilterRect(areaOfInterest)
for f in layer.getFeatures(request):
    ...
```

La requête peut être utilisée pour définir les données rappatriées pour chaque entité de manière à ce que l'itérateur retourne toutes les entités mais uniquement des données partielles pour chacune d'entre elles.

```
# Only return selected fields
request.setSubsetOfAttributes([0,2])
# More user friendly version
request.setSubsetOfAttributes(['name','id'],layer.pendingFields())
# Don't return geometry objects
request.setFlags(QgsFeatureRequest.NoGeometry)
```

4.2 Modifier des couches vecteur

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

```
caps = layer.dataProvider().capabilities()
```

En utilisant n'importe laquelle des méthodes qui suivent pour l'édition de couches vecteur, les changement sont directement validés dans le dispositif de stockage d'information 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*.

4.2.1 Ajout d'Entités

Créez quelques instance de `QgsFeature` et passez les sous forme de liste à la méthode `addFeatures()` du fournisseur. Elle vous renverra deux valeurs : le résultat (vrai/faux) et la liste des entités ajoutées (leur identifiant est paramétré pas le stockage de données).

```
if caps & QgsVectorDataProvider.AddFeatures:
  feat = QgsFeature()
  feat.addAttribute(0, 'hello')
  feat.setGeometry(QgsGeometry.fromPoint(QgsPoint(123, 456)))
  (res, outFeats) = layer.dataProvider().addFeatures([feat])
```

4.2.2 Suppression d'Entités

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

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

4.2.3 Modifier des Entités

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

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

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

if caps & QgsVectorDataProvider.ChangeGeometries:
  geom = QgsGeometry.fromPoint(QgsPoint(111,222))
  layer.dataProvider().changeGeometryValues({ fid : geom })
```

4.2.4 Ajout et Suppression de Champs

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

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

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

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

```
layer.updateFields()
```

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

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

Pour savoir si une couche est en mode édition, utilisez la fonction `isEditing()` — les fonctions d'éditions fonctionnent seulement lorsque le mode d'édition est activé. Utilisation des fonctions d'éditions :

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

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

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

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

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

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

if problem_occurred:
  layer.destroyEditCommand()
  return

# ... more editing ...

layer.endEditCommand()
```

La fonction `beginEditCommand()` créera une commande interne "activée" et enregistrera les changements à suivre de la couche vecteur. Lors de l'appel à la fonction `endEditCommand()`, la commande est poussée sur la pile d'annulation et l'utilisateur peut alors cliquer sur les boutons Annuler/Refaire. Au cas où quelquechose tournerait mal lors des changements, la méthode `destroyEditCommand()` method supprimera la commande de la pile et annulera tous les changements réalisés depuis que la commande est active.

Pour lancer le mode édition, il existe la méthode `startEditing()`. Pour arrêter l'édition, vous pouvez utiliser `commitChanges()` et `rollback()`. Néanmoins, vous n'avez pas besoin de les utiliser et vous pouvez laisser cette fonctionnalité activable par l'utilisateur.

4.4 Utilisation des index spatiaux

Les index spatiaux peuvent améliorer fortement les performances de votre code si vous réalisez de fréquentes requêtes sur une couche vecteur. Imaginez par exemple que vous écrivez un algorithme d'interpolation et que

pour une position donnée, vous devez déterminer les 10 points les plus proches dans une couche de points, dans l'objectif d'utiliser ces points pour calculer une valeur interpolée. Sans index spatial, la seule méthode pour QGIS de trouver ces 10 points est de calculer la distance entre tous les points de la couche et l'endroit indiqué et de comparer ces distances entre-elles. Cela peut prendre beaucoup de temps spécialement si vous devez répeter l'opération sur plusieurs emplacements. Si index spatial existe pour la couche, l'opération est bien plus efficace.

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

Les index spatiaux ne sont pas créés par défaut pour une couche vecteur mais vous pouvez le faire facilement de cette manière :

1. créez l'index spatial — le code qui suit créé un index vide

   ```
   index = QgsSpatialIndex()
   ```

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

   ```
   index.insertFeature(feat)
   ```

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

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

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

4.5 Ecrire dans des couches vecteur

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

Il y a deux façons d'exporter une couche vectorielle :
– A partir d'une instance de la classe `QgsVectorLayer`

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

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

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

Le troisième paramètre indique l'encodage du texte en sortie. Seuls certains pilotes ont besoin de ce paramètre pour fonctionner correctement, les fichiers Shape sont dans ce cas. Néanmoins, vous ne devriez pas rencontrer de problèmes tant que vous n'utilisez pas un jeu de caractères international. Le quatrième paramètre que nous avons laissé à `None` peut indiquer un SCR de destination, si une instance valide de `QgsCoordinateReferenceSystem` est utilisée, la couche est transformée dans ce SCR.
Consultez les formats gérés par OGR pour trouver les noms de pilote valides. Vous ne devez indiquer cette valeur dans la colonne "Code". Vous pouvez indiquer optionnellement d'exporter uniquement les entités sélectionnées ou utiliser des options de création spécifiques à chaque pilote ou encore indiquer au pilote de ne pas créer d'attributs. Consultez la documentation pour la syntaxe complète.
– Directement depuis les entités

```
# define fields for feature attributes. A list of QgsField objects is needed
fields = [QgsField("first", QVariant.Int),
          QgsField("second", QVariant.String)]

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

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

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

# delete the writer to flush features to disk (optional)
del writer
```

4.6 Fournisseur de données en mémoire

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

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

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

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

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

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

crs=définition Spécifie le système de coordonnée de référence, où définition peut être sous n'importe laquelle des formes acceptées par `QgsCoordinateReferenceSystem.createFromString()`

index=yes Spécifie que le fournisseur utilisera un index spatial

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

L'exemple suivant montre une URI intégrant toutes ces options

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

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

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

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

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

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

Finalement, vérifions que tout s'est bien déroulé

```
# show some stats
print "fields:", len(pr.fields())
print "features:", pr.featureCount()
e = layer.extent()
print "extent:", e.xMin(),e.yMin(),e.xMax(),e.yMax()

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

4.7 Apparence (Symbologie) des couches vecteur

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

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

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

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

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

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

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

Il peut également exister des moteurs de rendu personnalisés et vous ne pouvez donc pas considérer qu'il existe juste ces trois types. Vous pouvez interroger le singleton `QgsRendererV2Registry` pour savoir quels sont les moteurs de rendu disponibles.

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

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

4.7.1 Moteur de rendu à symbole unique

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

4.7.2 Moteur de rendu à symboles catégorisés

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

Pour obtenir la liste des catégories

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

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

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

4.7.3 Moteur de rendu à symboles gradués

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

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

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

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

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

```
from qgis.core import *

myVectorLayer = QgsVectorLayer(myVectorPath, myName, 'ogr')
myTargetField = 'target_field'
myRangeList = []
myOpacity = 1
# Make our first symbol and range...
myMin = 0.0
myMax = 50.0
myLabel = 'Group 1'
```

```
myColour = QtGui.QColor('#ffee00')
mySymbol1 = QgsSymbolV2.defaultSymbol(myVectorLayer.geometryType())
mySymbol1.setColor(myColour)
mySymbol1.setAlpha(myOpacity)
myRange1 = QgsRendererRangeV2(myMin, myMax, mySymbol1, myLabel)
myRangeList.append(myRange1)
#now make another symbol and range...
myMin = 50.1
myMax = 100
myLabel = 'Group 2'
myColour = QtGui.QColor('#00eeff')
mySymbol2 = QgsSymbolV2.defaultSymbol(
     myVectorLayer.geometryType())
mySymbol2.setColor(myColour)
mySymbol2.setAlpha(myOpacity)
myRange2 = QgsRendererRangeV2(myMin, myMax, mySymbol2 myLabel)
myRangeList.append(myRange2)
myRenderer = QgsGraduatedSymbolRendererV2('', myRangeList)
myRenderer.setMode(QgsGraduatedSymbolRendererV2.EqualInterval)
myRenderer.setClassAttribute(myTargetField)

myVectorLayer.setRendererV2(myRenderer)
QgsMapLayerRegistry.instance().addMapLayer(myVectorLayer)
```

4.7.4 Travailler avec les symboles

Pour la représentation des symboles, il existe la classe de base `QgsSymbolV2` qui est dérivée en trois sous-classes :
– `QgsMarkerSymbolV2` — pour les entités ponctuelles.
– `QgsLineSymbolV2` — pour les entités linéaires.
– `QgsFillSymbolV2` — pour les entités polygonales.
Chaque symbole est constitué d'une ou plusieurs couche de symboles (classes dérivées de `QgsSymbolLayerV2`). Les couches de symboles font le rendu, la classe du symbole sert seulement de conteneur pour les couches de symbole.

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

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

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

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

Travailler avec des couches de symboles

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

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

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

Sortie

```
EllipseMarker
FontMarker
SimpleMarker
SvgMarker
VectorField
```

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

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

Créer des types personnalisés de couches de symbole

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

```
class FooSymbolLayer(QgsMarkerSymbolLayerV2):

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

  def layerType(self):
    return "FooMarker"

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

  def startRender(self, context):
    pass

  def stopRender(self, context):
    pass

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

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

La méthode `layerType()` détermine le nom de la couche de symbole. Elle doit être unique parmi toutes les couches de symbole déjà existantes. Des propriétés sont utilisées pour la persistance des attributs. La méthode `clone()` renvoie une copie de la couche de symbole avec des attributs complètement identiques. Enfin, il reste

les méthodes de rendu : `startRender()` est appelée avant le rendu de la première entité, `stopRender()` lorsque le rendu est terminé. La méthode `renderPoint()` s'occupe du rendu. Les coordonnées du ou des point(s) sont déjà transformées dans le SCR de sortie.

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

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

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

    self.layer = None

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

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

  def symbolLayer(self):
    return self.layer

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

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

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

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

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

```
class FooSymbolLayerMetadata(QgsSymbolLayerV2AbstractMetadata):

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

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

  def createSymbolLayerWidget(self):
    return FooSymbolLayerWidget()

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

Vous devez renseigner le type de couche (la même renvoyée par la couche) et le type de symbole (marker/line/fill) au constructeur de la classe parent. La méthode `createSymbolLayer()` s'occupe de créer l'instance d'une couche de symbole avec les attributs indiqués dans le dictionnaire *props*. (Attention, les clefs sont des instances QString et non des objets Python "str"). Et il existe également la méthode `createSymbolLayerWidget()` qui renvoie l'interface de paramétrage pour ce type de couche de symbole.

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

4.7.5 Créer ses propres moteurs de rendu

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

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

```
import random

class RandomRenderer(QgsFeatureRendererV2):
  def __init__(self, syms=None):
    QgsFeatureRendererV2.__init__(self, "RandomRenderer")
    self.syms = syms if syms else [QgsSymbolV2.defaultSymbol(QGis.Point), QgsSymbolV2.defaultSymbo

  def symbolForFeature(self, feature):
    return random.choice(self.syms)

  def startRender(self, context, vlayer):
    for s in self.syms:
      s.startRender(context)

  def stopRender(self, context):
    for s in self.syms:
      s.stopRender(context)

  def usedAttributes(self):
    return []

  def clone(self):
    return RandomRenderer(self.syms)
```

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

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

```
class RandomRendererWidget(QgsRendererV2Widget):
  def __init__(self, layer, style, renderer):
    QgsRendererV2Widget.__init__(self, layer, style)
    if renderer is None or renderer.type() != "RandomRenderer":
      self.r = RandomRenderer()
    else:
      self.r = renderer
    # setup UI
    self.btn1 = QgsColorButtonV2("Color 1")
    self.btn1.setColor(self.r.syms[0].color())
    self.vbox = QVBoxLayout()
    self.vbox.addWidget(self.btn1)
    self.setLayout(self.vbox)
    self.connect(self.btn1, SIGNAL("clicked()"), self.setColor1)

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

  def renderer(self):
    return self.r
```

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

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

```
class RandomRendererMetadata(QgsRendererV2AbstractMetadata):
  def __init__(self):
    QgsRendererV2AbstractMetadata.__init__(self, "RandomRenderer", "Random renderer")

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

QgsRendererV2Registry.instance().addRenderer(RandomRendererMetadata())
```

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

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

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

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

4.8 Sujets complémentaires

****A FAIRE : ** ** Créer/modifier des symboles qui fonctionnent avec un style (`QgsStyleV2`) basé sur les rampes de couleur (`QgsVectorColorRampV2`). Moteur de rendu basé sur les ensembles de règles (voir'cet article <http ://snorf.net/blog/2014/03/04/symbology-of-vector-layers-in-qgis-python-plugins>'_), explorer les registres des couches de symbole et des moteurs de rendu.

CHAPITRE 5

Manipulation de la géométrie

Les points, lignes et polygones représentant un objet spatial sont couramment appelées des géométries. Dans QGIS, ils sont représentés par la classe `QgsGeometry`. Tous les types de géométrie sont admirablement présentés dans la page de discussion JTS.

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.

5.1 Construction de géométrie

Il existe plusieurs options pour créer une géométrie :
– depuis des coordonnées

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

Les coordonnées sont indiquées à l'aide de la classe `QgsPoint`.
La polyligne (Linestring) est représentée par une succession de points. Le polygone est représenté par une succession de polylignes en anneaux (c'est-à-dire des polylignes fermées). Le premier anneau représente l'anneau externe (la limite), les potentiels anneaux ultérieures sont des trous dans le polygone.
Les géométries multi-parties sont d'un niveau plus complexe : les multipoints sont une succession de points, les multilignes une succession de lignes et les multipolygones une succession de polygones.

– depuis un Well-Known-Text (WKT)

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

– depuis un Well-Known-Binary (WKB)

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

5.2 Accéder à la Géométrie

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

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

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

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

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

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

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

5.3 Prédicats et opérations géométriques

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

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

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

Les calculs de surfaces et de périmètres ne prennent pas en compte le SCR lorsque qu'on utilise les méthodes de la classe `QgsGeometry` class. Pour réaliser des calculs plus précis sur les surfaces ou les distances, la classe `QgsDistanceArea` peut être utilisée. Si les projections sont désactivées, les calculs seront faits en mode planaire sinon, ils tiendront compte de l'ellipsoïde. Lorsqu'aucune ellipsoïde n'est définie explicitement, les paramètres WGS84 sont utilisés pour les calculs.

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

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

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

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

– Modification de la géométrie : Algorithme de Reprojection
– Distance et surface avec la classe `QgsDistanceArea` : Algorithme Matrice des Distances
– Algorithme de conversion de multi-parties en partie unique

CHAPITRE 6

Support de projections

6.1 Système de coordonnées de référence

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

– Spécifier le SCR par son identifiant

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

QGIS utilise trois identifiants différents pour chaque système de référence :
– `PostgisCrsId` — Identifiants utilisés dans les bases de données PostGIS.
– `InternalCrsId` — Identifiants utilisés dans la base de données QGIS.
– `EpsgCrsId` — Identifiants définis par l'organisation EPSG.
Sauf indication contraire dans le deuxième paramètre, le SRID de PostGIS est utilisé par défaut.

– Spécifier le SCR par un Well-Known-Text (WKT)

```
wkt = 'GEOGCS["WGS84", DATUM["WGS84", SPHEROID["WGS84", 6378137.0, 298.257223563]],'
       PRIMEM["Greenwich", 0.0], UNIT["degree",0.017453292519943295],'
       AXIS["Longitude",EAST], AXIS["Latitude",NORTH]]'
crs = QgsCoordinateReferenceSystem(wkt)
```

– Créer un SCR non valide et utiliser ensuite une des fonctions `create*()` pour l'initaliser. Dans l'exemple qui suit, nous utilisons une chaîne de caractères Proj4 pour initialiser une projection

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

Il faut vérifier si la création (c'est à dire la recherche dans la base de données) du SCR a réussi. La fonction : : func : *isValid* doit retourner : const : *true*.

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

Accéder à l'information sur le système de référence spatiale

```
print "QGIS CRS ID:", crs.srsid()
print "PostGIS SRID:", crs.srid()
print "EPSG ID:", crs.epsg()
print "Description:", crs.description()
print "Projection Acronym:", crs.projectionAcronym()
print "Ellipsoid Acronym:", crs.ellipsoidAcronym()
print "Proj4 String:", crs.proj4String()
# check whether it's geographic or projected coordinate system
print "Is geographic:", crs.geographicFlag()
```

```
# check type of map units in this CRS (values defined in QGis::units enum)
print "Map units:", crs.mapUnits()
```

6.2 Projections

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

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

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

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

CHAPITRE 7

Utiliser le Canevas de carte

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

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

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

Pour résumer, l'architecture du canevas de carte repose sur trois concepts :
– le canevas de carte — pour visualiser la carte
– des objets de canevas — objets additionnels qui peuvent être affichés dans le canevas de carte
– les outils cartographiques — pour interagir avec le canevas de carte

7.1 Intégrer un canevas de carte

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

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

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

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

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

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

Maintenant nous pouvons ajouter des couches cartographiques. Nous allons ouvrir d'abord une couche et l'ajouter au registre des couches de la carte. Ensuite, nous paramétrerons l'emprise de la carte et nous affecterons la liste des couches à destination du canevas :

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

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

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

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

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

7.2 Utiliser les outils cartographiques avec le canevas

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

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

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

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

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

    self.setCentralWidget(self.canvas)

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

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

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

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

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

    self.pan()

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

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

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

Vous pouvez copier le code ci-dessus dans un fichier (par exemple : `mywnd.py`) et essayer de l'ouvrir depuis la console Python de QGIS. Le code qui suit affichera la couche actuellement sélectionnée dans le nouveau canevas créé :

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

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

7.3 Contour d'édition et symboles de sommets

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

Pour afficher une polyligne :

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

Pour afficher un polygone :

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

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

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

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

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

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

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

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

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

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

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

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

7.4 Ecrire des outils cartographiques personnalisés

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

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

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

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

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

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

def canvasReleaseEvent(self, e):
    self.isEmittingPoint = False
    r = self.rectangle()
    if r is not None:
      print "Rectangle:", r.xMin(), r.yMin(), r.xMax(), r.yMax()

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

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

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

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

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

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

    return QgsRectangle(self.startPoint, self.endPoint)

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

7.5 Ecrire des éléments de canevas de carte personnalisés

TODO Comment créer un objet de canevas de carte ?

CHAPITRE 8

Rendu cartographique et Impression

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

8.1 Rendu simple

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

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

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

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

render = QgsMapRenderer()

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

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

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

# do the rendering
render.render(p)

p.end()

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

8.2 Sortie utilisant un composeur de carte

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

Le composeur est composé d'un ensemble de classes. Elles appartiennent toute à la bibliothèque core. L'application QGIS dispose d'une interface graphique dédiée pour le placement des éléments mais celle-ci n'est pas disponible dans la bibliothèque graphique. Si vous n'êtes pas familier de la structure de Vue Graphique de Qt, nous vous encourageons à lire cette documentation car le composeur est basé dessus.

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

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

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

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

– carte — cet élément indique aux bibliothèques l'emplacement de la carte. Dans l'exemple ci-dessous, nous créons une carte et l'étirons sur toute la taille de la page

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

– étiquette — permet l'affichage d'étiquettes. Il est possible de modifier la police, la couleur, l'alignement et les marges

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

– légende

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

– Échelle graphique

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

– flèche
– image
– couche
– table

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

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

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

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

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

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

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

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

8.2.1 Sortie vers une image raster

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

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

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

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

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

8.2.2 Export en PDF

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

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

pdfPainter = QPainter(printer)
paperRectMM = printer.pageRect(QPrinter.Millimeter)
```

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

CHAPITRE 9

Expressions, Filtrage et Calcul de valeurs

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

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

Les opérations suivantes sont disponibles :
– opérateurs arithmétiques : `+`, `-`, `*`, `/`, `^`
– parenthèses : pour faire respecter la précédence des opérateurs : `(1 + 1) * 3`
– les unaires plus et moins : `-12`, `+5`
– fonctions mathématiques : `sqrt`, `sin`, `cos`, `tan`, `asin`, `acos`, `atan`
– fonctions géométriques : `$area`, `$length`
– fonctions de conversion : `to int`, `to real`, `to string`

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`

9.1 Analyse syntaxique d'expressions

```
>>> exp = QgsExpression('1 + 1 = 2')
>>> exp.hasParserError()
False
>>> exp = QgsExpression('1 + 1 = ')
>>> exp.hasParserError()
True
>>> exp.parserErrorString()
PyQt4.QtCore.QString(u'syntax error, unexpected $end')
```

9.2 Évaluation des expressions

9.2.1 Expressions basiques

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

9.2.2 Expressions avec entités

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

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

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

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

9.2.3 Gestion des erreurs

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

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

print value
```

9.3 Exemples

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

```
def where(layer, exp):
  print "Where"
  exp = QgsExpression(exp)
  if exp.hasParserError():
    raise Expection(exp.parserErrorString())
  exp.prepare(layer.pendingFields())
  for feature in layer.getFeatures():
    value = exp.evaluate(feature)
    if exp.hasEvalError():
      raise ValueError(exp.evalErrorString())
    if bool(value):
      yield feature
```

```
layer = qgis.utils.iface.activeLayer()
for f in where(layer, 'Test > 1.0'):
  print f + " Matches expression"
```

CHAPITRE 10

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 indiquer une clef qui sera utilisée pour accéder à la variable : pour la couleur préférée de l'utilisateur, vous pouvez utiliser la clef "couleur_favorite" ou tout autre chaîne de caractères explicite. Nous vous recommandons de donner de la structure dans le nommage des clefs.

Nous pouvons faire des différences entre différents types de paramètres :

– **Paramètres globaux** — ils sont liés à l'utilisateur d'une machine en particulier. QGIS enregistre lui-même un certain nombre de variables globales, par exemple, la taille de la fenêtre principale ou la tolérance d'accrochage par défaut. Cette fonctionnalité est fournie directement par la bibliothèque Qt grâce à la classe QSettings. Par défaut, cette classe enregistrer les paramètres dans le moyen "natif" du système d'exploitation, c'est à dire : la base de registre sous Windows, un fichier .plist sous Mac OS C ou un fichier .ini sous GNU/Linux. La documentation QSettings est facile à comprendre et c'est pourquoi nous allons présenter un simple exemple :

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

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

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

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

```
proj = QgsProject.instance()

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

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

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

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

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

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

CHAPITRE 11

Communiquer avec l'utilisateur

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

11.1 Afficher des messages : La classe `QgsMessageBar`

Utiliser des boîte 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

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

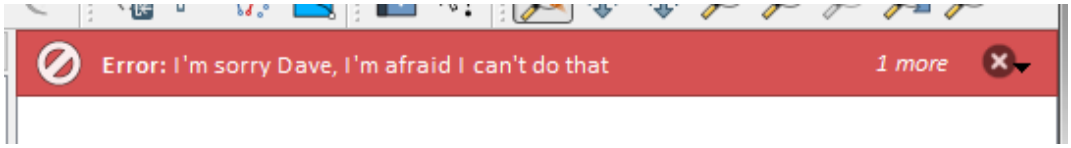

FIGURE 11.1 – Barre de message de QGis

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

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

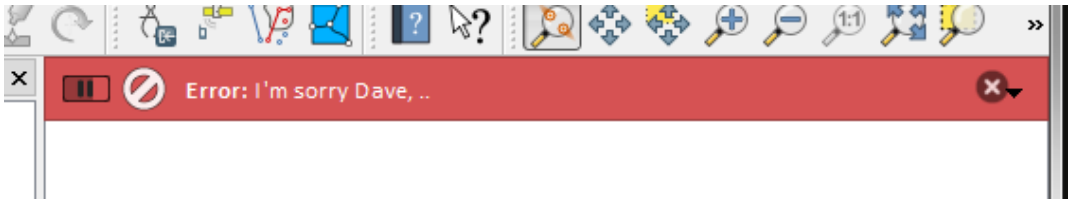

FIGURE 11.2 – Barre de message de Qgis avec décompte

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

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

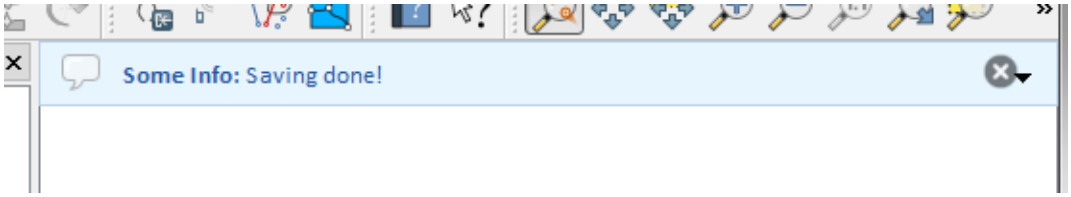

FIGURE 11.3 – Barre de message QGis (info)

```
def showError():
    pass

widget = iface.messageBar().createMessage("Missing Layers", "Show Me")
button = QPushButton(widget)
button.setText("Show Me")
button.pressed.connect(showError)
widget.layout().addWidget(button)
iface.messageBar().pushWidget(widget, QgsMessageBar.WARNING)
```

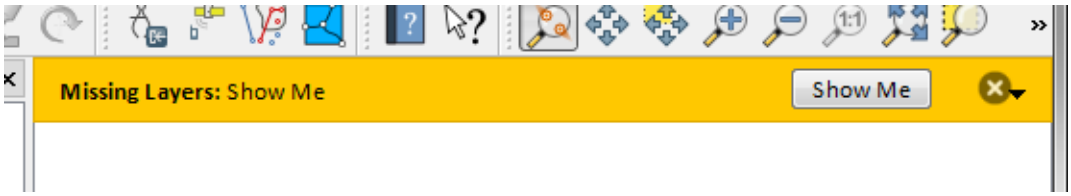

FIGURE 11.4 – Barre de message QGis avec un bouton

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

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

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

11.2 Afficher la progression

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

```
import time
from PyQt4.QtGui import QProgressBar
from PyQt4.QtCore import *
progressMessageBar = iface.messageBar().createMessage("Doing something boring...")
progress = QProgressBar()
progress.setMaximum(10)
progress.setAlignment(Qt.AlignLeft|Qt.AlignVCenter)
```

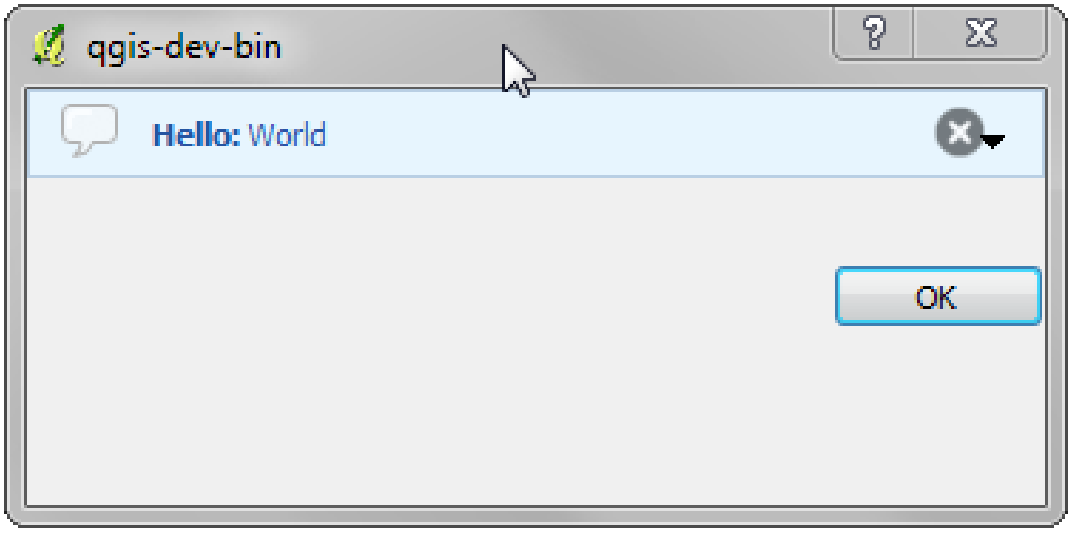

FIGURE 11.5 – Barre de message QGis avec une boîte de dialogue personnalisée

```
progressMessageBar.layout().addWidget(progress)
iface.messageBar().pushWidget(progressMessageBar, iface.messageBar().INFO)
for i in range(10):
    time.sleep(1)
    progress.setValue(i + 1)
iface.messageBar().clearWidgets()
```

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

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

11.3 Journal

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

```
QgsMessageLog.logMessage("Your plugin code has been executed correctly", QgsMessageLog.INFO)
QgsMessageLog.logMessage("Your plugin code might have some problems", QgsMessageLog.WARNING)
QgsMessageLog.logMessage("Your plugin code has crashed!", QgsMessageLog.CRITICAL)
```

CHAPITRE 12

Développer des extensions Python

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

Les extensions Python sont listées avec les extensions C++ dans le gestionnaire d'extension. Voici les chemins où elles peuvent être situées :
– UNIX/Mac : `~/.qgis/python/plugins` et `(qgis_prefix)/share/qgis/python/plugins`
– Windows : `~/.qgis/python/plugins` et `(qgis_prefix)/python/plugins`
Sous Windows, le répertoire Home (noté ci-dessus par ~) est généralement situé dans un emplacement du type `C:\Documents and Settings\(user)` (sous Windows XP et inférieur) ou dans `C:\Users\(user)`. Etant donné que QGIS utilise Python 2.7, les sous-répertoires de ces chemins doivent contenir un fichier __init__.py pour pouvoir les considérer comme des paquets Python qui peuvent être importés en extensions.

Étapes :

1. *Idée* : Avoir une idée de ce que vous souhaitez faire avec votre nouvelle extension. Pourquoi le faites-vous ? Quel problème souhaitez-vous résoudre ? N'y a-t-il pas déjà une autre extension pour ce problème ?
2. *Créer des fichiers* : Créer les fichiers décrits plus loin. Un point de départ (`__init.py__`). Remplissez les fichiers *Métadonnées de l'extension* (`metadata.txt`). Un corps principal de l'extension (`mainplugin.py`). Un formulaire créé avec QT-Designer (`form.ui`), et son fichier de ressources `resources.qrc`.
3. *Écrire le code* : Écrire le code à l'intérieur du fichier `mainplugin.py`
4. *Test* : Fermez et ré-ouvrez QGIS et importez à nouveau votre extension. Vérifiez si tout est OK.
5. *Publier* : Publiez votre extension dans le dépôt QGIS ou créez votre propre dépôt tel un "arsenal" pour vos "armes SIG" personnelles.

12.1 Écriture d'une extension

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

12.1.1 Fichiers de l'extension

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

```
PYTHON_PLUGINS_PATH/
  MyPlugin/
    __init__.py     --> *required*
    mainPlugin.py   --> *required*
```

```
metadata.txt   --> *required*
resources.qrc  --> *likely useful*
resources.py   --> *compiled version, likely useful*
form.ui        --> *likely useful*
form.py        --> *compiled version, likely useful*
```

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

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

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

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

12.2 Contenu de l'extension

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

12.2.1 Métadonnées de l'extension

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

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

Nom de la métadonnée	Requis	Notes
name	Vrai	texte court contenant le nom de l'extension
qgisMinimumVersion	Vrai	version minimum de QGIS en notation par points
qgisMaximumVersion	Faux	version maximum de QGIS en notation par points
description	Vrai	un texte court qui décrit l'extension. Le HTML n'est pas autorisé
about	Faux	un texte long qui décrit l'extension en détail, pas de HTML autorisé
version	Vrai	texte court avec le numéro de version par points
author	Vrai	nom de l'auteur
email	Vrai	e-mail de l'auteur, *n'apparaîtra pas* sur le site web
changelog	Faux	texte, peut être multi-lignes, pas de HTML autorisé
experimental	Faux	indicateur booléen, *Vrai* ou *Faux*
deprecated	Faux	indicateur booléen, *Vrai* ou *Faux*, s'applique à l'extension entière et pas simplement à la version chargée
tags	Faux	liste séparée par une virgule, les espaces sont autorisés à l'intérieur des balises individuelles
homepage	Faux	une URL valide pointant vers la page d'accueil de l'extension
repository	Faux	une URL valide pour le dépôt du code source
tracker	Faux	une URL valide pour les billets et rapports de bugs
icon	Faux	un nom de fichier ou un chemin relatif (par rapport au dossier racine de l'extension, fichiers compressés)
category	Faux	soit *Raster*, *Vector*, *Database* ou *Web*

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

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

```
category=Raster
```

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

Un exemple pour ce fichier metadata.txt

```
; the next section is mandatory

[general]
name=HelloWorld
email=me@example.com
author=Just Me
qgisMinimumVersion=2.0
description=This is an example plugin for greeting the world.
    Multiline is allowed:
    lines starting with spaces belong to the same
    field, in this case to the "description" field.
    HTML formatting is not allowed.
about=This paragraph can contain a detailed description
    of the plugin. Multiline is allowed, HTML is not.
version=version 1.2
; end of mandatory metadata

; start of optional metadata
category=Raster
changelog=The changelog lists the plugin versions
```

```
    and their changes as in the example below:
    1.0 - First stable release
    0.9 - All features implemented
    0.8 - First testing release

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

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

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

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

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

12.2.2 __init__.py

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

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

## any other initialisation needed
```

12.2.3 mainPlugin.py

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

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

# initialize Qt resources from file resources.py
import resources

class TestPlugin:

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

  def initGui(self):
    # create action that will start plugin configuration
```

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

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

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

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

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

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

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

Les seules fonctions de l'extension qui doivent exister dans le fichier source principal de l'extension (ex : `mainPlugin.py`) sont :
– `__init__` –> qui donne accès à l'interface de QGIS.
– `initGui()` –> appelée lorsque l'extension est chargée.
– `unload()` –> chargée lorsque l'extension est déchargée.
Vous pouvez voir que dans l'exemple ci-dessus, la fonction `addPluginToMenu()` est utilisée. Elle ajoute l'entrée de menu correspondante au menu *Extension*. Il existe d'autres méthodes pour ajouter l'action dans un menu différent. Voici une liste de ces méthodes :
– `addPluginToRasterMenu()`
– `addPluginToVectorMenu()`
– `addPluginToDatabaseMenu()`
– `addPluginToWebMenu()`
Toutes ont la même syntaxe que la méthode `addPluginToMenu()`.

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

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

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

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

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

N'oubliez pas de paramétrer la propriété objectName des QAction et QMenu avec un nom spécifique à votre extension.

12.2.4 Fichier de ressources

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

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

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

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

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

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

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

12.3 Documentation

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

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

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

CHAPITRE 13

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

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

13.1 Note sur la configuration de l'EDI sous Windows

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

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

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

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

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

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

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

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

13.2 Débogage à l'aide d'Eclipse et PyDev

13.2.1 Installation

Afin d'utiliser Eclipse, assurez-vous d'avoir installé
– Eclipse
– l'extension Aptana d'Eclipse ou PyDev
– QGIS 2.0

13.2.2 Préparation de QGIS

Il faut faire un peu de préparation dans QGIS lui-même. Deux extensions sont intéressantes : *Remote Debug* et *Plugin reloader*.
– Allez dans *Extension → Installer/Gérer les extensions*.
– Cherchez l'extension *Remote Debug* (pour l'instant, elle est en version expérimentale et vous devrez donc activer les extensions expérimentales dans l'onglet Options pour la trouver). Installez-la
– Cherchez l'extension *Plugin reloader* et installez-la de la même manière. Elle vous permettra de recharger une extension sans avoir à redémarrer QGIS.

13.2.3 Configuration d'Eclipse

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

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

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

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

13.2.4 Configurer le débogueur

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

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

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

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

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

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

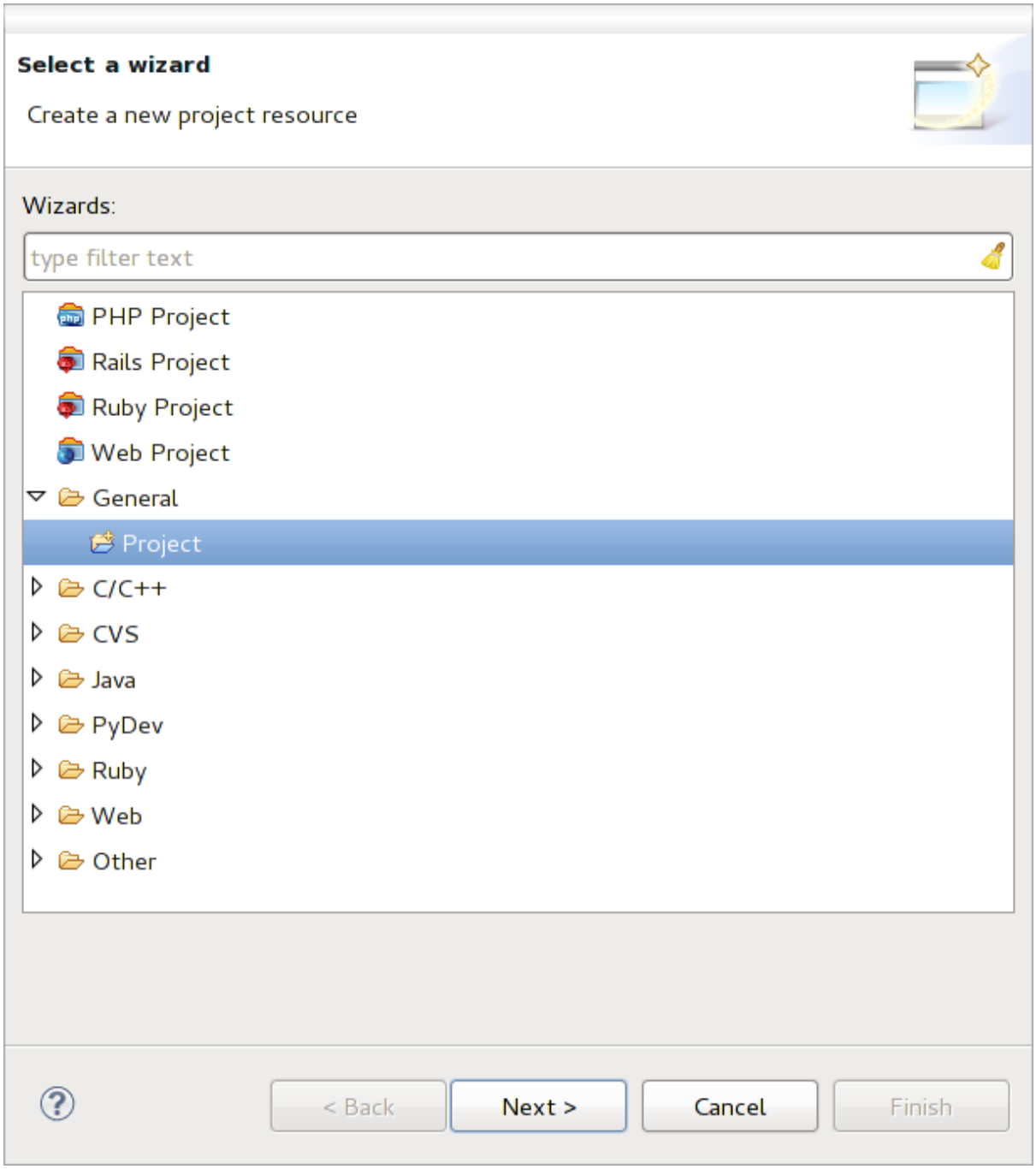

FIGURE 13.1 – Projet Eclipse

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

FIGURE 13.2 – Point d'arrêt

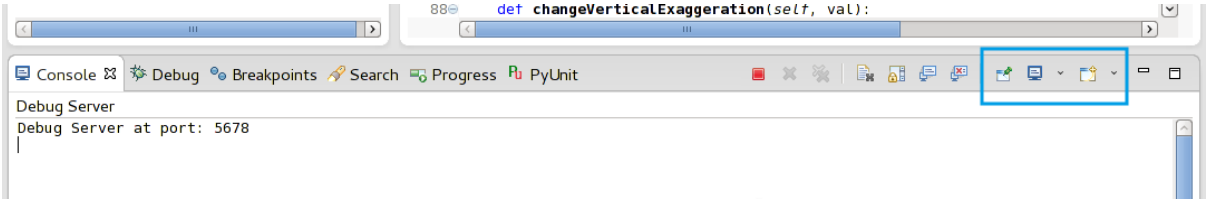

FIGURE 13.3 – Console de débogage de PyDev

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

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

13.2.5 Permettre à Eclipse de comprendre l'API

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

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

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

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

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

Vous devriez également ajouter le répertoire de vos extensions (sous Linux : `~/.qgis/python/plugins`).

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

Cliquer sur *OK* et c'est fini.

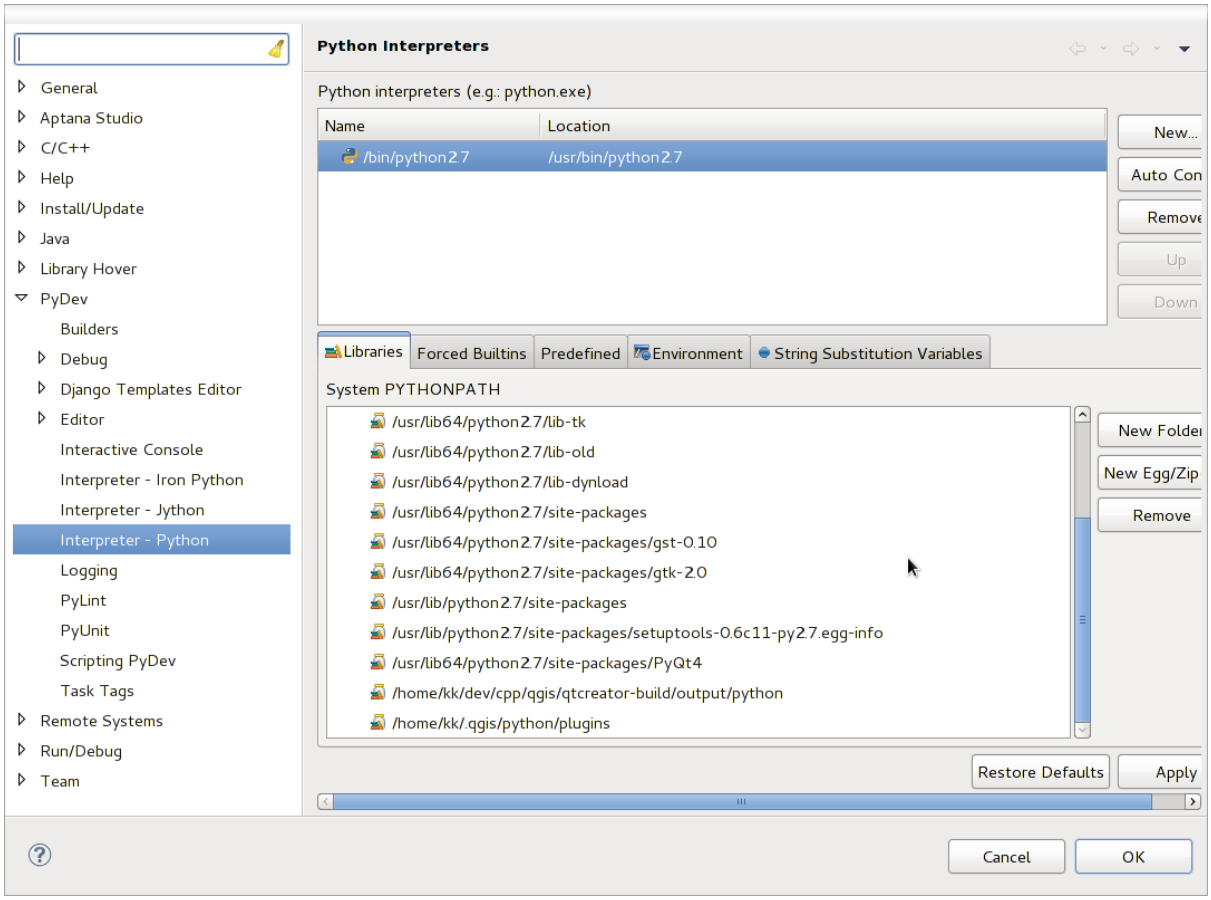

FIGURE 13.4 – Console de débogage de PyDev

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

Pour une autre configuration d'Eclipse pour travailler avec des extensions Python de QGIS, consultez ce lien

13.3 Débogage à l'aide de PDB

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

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

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

Ensuite exécutez QGIS depuis la ligne de commande.

Sur Linux, faites :

$ ./Qgis

Sur Mac OS X, faites :

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

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

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

CHAPITRE 14

Utiliser une extension de couches

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

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

14.1 Héritage de QgsPluginLayer

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

```
class WatermarkPluginLayer(QgsPluginLayer):

  LAYER_TYPE="watermark"

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

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

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

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

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

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

```
class WatermarkPluginLayerType(QgsPluginLayerType):

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

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

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

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

CHAPITRE 15

Compatibilité avec les versions précédentes de QGIS

15.1 Menu Extension

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

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

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

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

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

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

CHAPITRE 16

Publier votre extension

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

16.1 Dépôt officiel des extensions Python

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

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

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

A FAIRE : Insérer un lien vers le document de gouvernance

16.1.1 Permissions

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

16.1.2 Gestion de la confiance

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

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

16.1.3 Validation

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

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

1. le nom du dossier principal contenant votre extension ne doit contenir que des caractères ASCII (A-Z et a-z), des chiffres et les caractères underscore(_) et moins (-), sans cependant commencer par un chiffre
2. `metadata.txt` est requis
3. Toutes les métadonnées requises listées dans *metadata table* doivent être présentes.
4. Le champ de métadonnée *version* doit être unique

16.1.4 Structure d'une extension

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

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

CHAPITRE 17

Extraits de code

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

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

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

```
self.keyAction = QAction("Test Plugin", self.iface.mainWindow())
self.iface.registerMainWindowAction(self.keyAction, "F7") # action1 triggered by F7 key
self.iface.addPluginToMenu("&Test plugins", self.keyAction)
QObject.connect(self.keyAction, SIGNAL("triggered()"),self.keyActionF7)
```

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

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

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

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

17.2 Comment activer des couches :

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

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

17.3 Comment accéder à la table attributaire des entités sélectionnées

```
def changeValue(self, value):
  layer = self.iface.activeLayer()
  if(layer):
    nF = layer.selectedFeatureCount()
    if (nF > 0):
    layer.startEditing()
    ob = layer.selectedFeaturesIds()
```

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

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

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

CHAPITRE 18

Bibliothèque d'analyse de réseau

Depuis la révision ee19294562 (QGIS >= 1.8), la nouvelle bibliothèque d'analyse de réseau a été ajoutée à la bibliothèque principale d'analyse de QGis. La bibliothèque :
– créé un graphe mathématique à partir de données géographiques (couches vecteurs de polylignes)
– implémente des méthodes simples de la théorie des graphes (pour l'instant, uniquement avec l'algorithme Dijkstra).
La bibliothèque d'analyse de réseau a été créée en exportant les fonctions de l'extension principale RoadGraph. Vous pouvez en utiliser les méthodes dans des extensions ou directement dans la console Python.

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

Pour calculer les propriétés des arcs, une stratégie basée sur les motifs de programmation est employée. Pour l'instant, seule la stratégie QgsDistanceArcProperter est disponible ; elle prend en compte la longueur de la route.

Vous pouvez implémenter votre propre stratégie qui utilisera tous les paramètres nécessaires. Par exemple, l'extension RoadGraph utilise une stratégie qui détermine le temps de trajet en utilisant les longueurs d'arc et la vitesse à partir d'attributs.

Il est temps de plonger dans le processus.

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

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

Ensuite, quelques exemples pour créer un directeur

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

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

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

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

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

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

```
properter = QgsDistanceArcProperter()
```

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

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

Nous pouvons maintenant utiliser le constructeur qui créera le graphe. Le constructeur de la classe QgsGraphBuilder utilise plusieurs arguments :
– crs — système de coordonnées de référence à utiliser. Argument obligatoire.
– otfEnabled — utiliser ou non la projection "à la volée". La valeur par défaut est const :*True* (oui, utiliser OTF).
– topologyTolerance — la tolérance topologique. La valeur par défaut est 0.
– ellipsoidID — ellipsoïde à utiliser. Par défaut "WGS84".

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

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

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

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

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

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

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

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

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

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

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

La méthode `shortestTree()` est utile lorsque vous voulez approcher l'arbre du chemin le plus court. Elle créé toujours un nouvel objet de graphe (QgsGraph) et elle accepte trois variables :
– source — graphe en entrée
– startVertexIdx — index du point sur l'arbre (la racine de l'arbre)
– criterionNum — nombre de propriétés d'arc à utiliser (en partant de 0).

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

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

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

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

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

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

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

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

graph = builder.graph()

idStart = graph.findVertex(pStart)

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

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

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

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

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

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

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

graph = builder.graph()

idStart = graph.findVertex(pStart)

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

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

18.3.1 Trouver les chemins les plus courts

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

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

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

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

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

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

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

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

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

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

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

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

if idStop == -1:
  print "Path not found"
```

```
else:
  p = []
  while (idStart != idStop):
    l = tree.vertex(idStop).inArc()
    if len(l) == 0:
      break
    e = tree.arc(l[0])
    p.insert(0, tree.vertex(e.inVertex()).point())
    idStop = e.outVertex()

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

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

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

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

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

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

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

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

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

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

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

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

  p.append(tStart)

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

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

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

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

Voici un exemple :

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

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

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

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

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

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

idStart = graph.findVertex(tStart)

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

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

for i in upperBound:
```

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