CETE Nord Picardie

Risques et Développement des Territoires

pour comprendre le présent et construire un avenir durable

DDTM du Nord

Service Urbanisme et Connaissance Territoriale

Gestion et Valorisation des Données

Novembre 2011

Conception d'un plugin Python pour QGIS

Mise en œuvre pratique et valorisation

Liberté • Égalité • Fraternité
RÉPUBLIQUE FRANÇAISE

Ministère de l'Écologie, du Développement durable, des Transports et du Logement

Maîtrise d'ouvrage de l'étude : DDTM du Nord

Pilote de l'étude : Rémi BOREL

Références administratives

N° d'affaire : **10034763001**

Historique des versions du document

Version	Date	Auteur	Commentaires
1	*27/10/11*	*Rémi BOREL*	
2			
3			
4			

Chargé d'affaire pilote - Affaire suivie par

Tél : Rémi BOREL
Mél. Remi.Borel@developpement-durable.gouv.fr

Rédacteur

Rémi BOREL

Contributeurs

Jérôme DOUCHÉ
Aurélien AGUIRRE

Relecteurs

Frédéric LASSERON
Aurélien AGUIRRE

Visas techniques

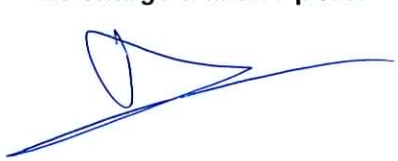
Le chargé d'affaire pilote	Le relecteur
	

Table des matières

1 – Introduction

Ce rapport a pour objectif de permettre à un utilisateur de QGIS de facilement mettre en œuvre un plugin Python dans QGIS. Certains pré-requis sont néanmoins nécessaires, mais pas développés dans ce rapport : le langage Python, le détail de l'API QGIS et la librairie graphique Qt pour Python.

Comme son nom l'indique, un plugin Python dans QGIS doit être écrit et développé en Python. Un plugin QGIS est l'équivalent d'un « MBX » pour Mapinfo ou d'une « ToolBox » pour ArcGIS. C'est une extension de l'application qui permet de créer des nouvelles fonctionnalités, ou d'assembler des fonctionnalités existantes. Dans ce document, le terme «extension» pourra également être utilisé afin de désigner le mot « plugin ».

Ce document s'articule en 4 parties principales :

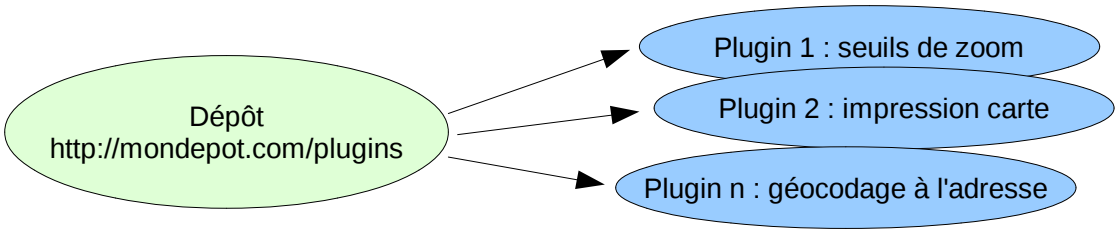
- Où se situent les plugins Python dans QGIS ?
- Quels fichiers dois-je créer ? Existe-t-il des modèles ?
- Exemple de plugin simple pas à pas,
- Comment diffuser et valoriser mon plugin.

Les pré-requis sont développés dans certaines références figurant en annexe. Quelques recherches sur Internet peuvent également offrir de nombreux compléments. Moyennant une bonne compréhension des principes et en adaptant certains codes existants, il est possible d'arriver à produire des plugins sans pour autant être un expert de Qt et Python. Il ne faut pas hésiter à installer des plugins de contributeurs et examiner leur structure. Ce document ne présente qu'une méthode de conception parmi d'autres.

2 – L'emplacement des plugins

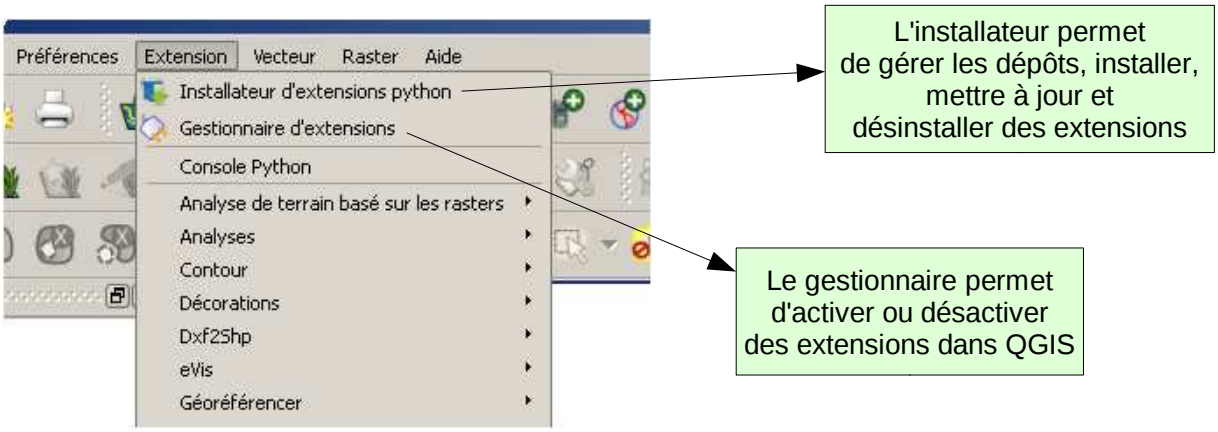
2.1 – Le système de dépôt d'extensions de QGIS

Les plugins QGIS sont gérés par des systèmes de « dépôts » d'extensions. A un dépôt correspond une URL. A une URL est associé une série de plugin.



Remarque : la création de dépôts est détaillée en fin de document.

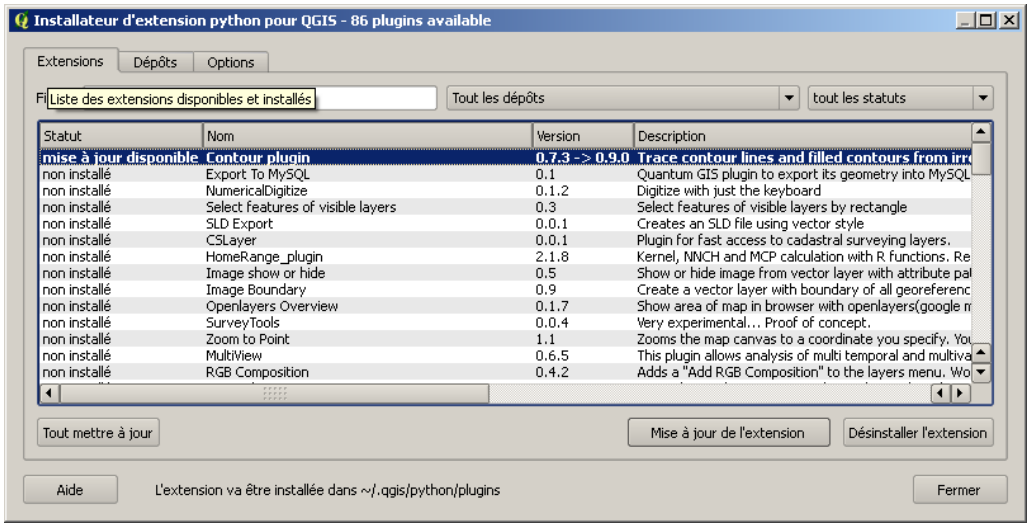
Dans QGIS, le menu extension permet de gérer ces dépôts :



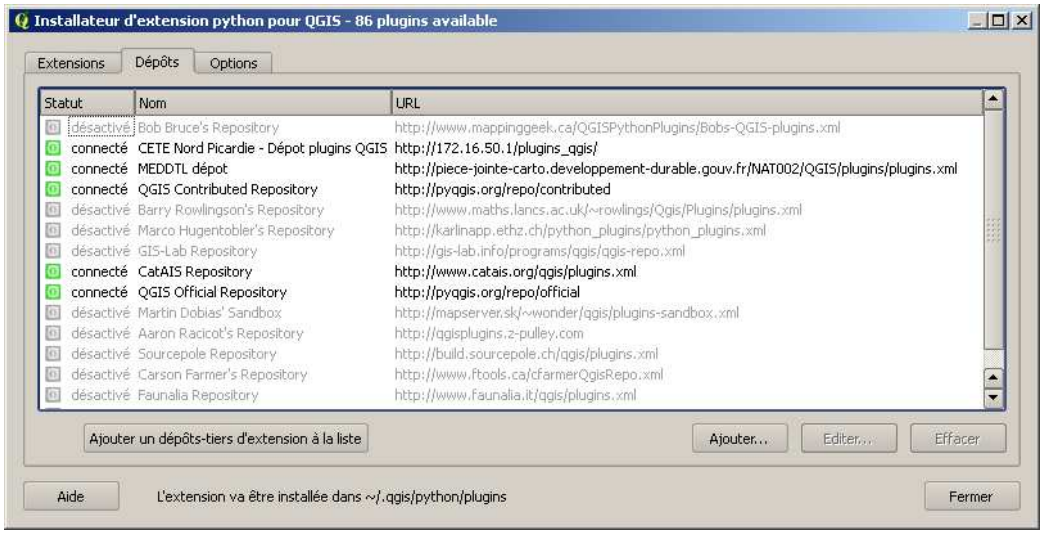
2.1.1. L'installateur d'extensions

L'installateur d'extensions contient deux parties principales:

- la consultation des plugins disponibles sur les dépôts,
- la gestion des dépôts.



L'onglet de consultation permet de visualiser rapidement quels plugins sont installés, ceux qui bénéficient d'une mise à jour. L'installation d'un nouveau plugin se fait simplement en cliquant sur « Installer l'extension ».



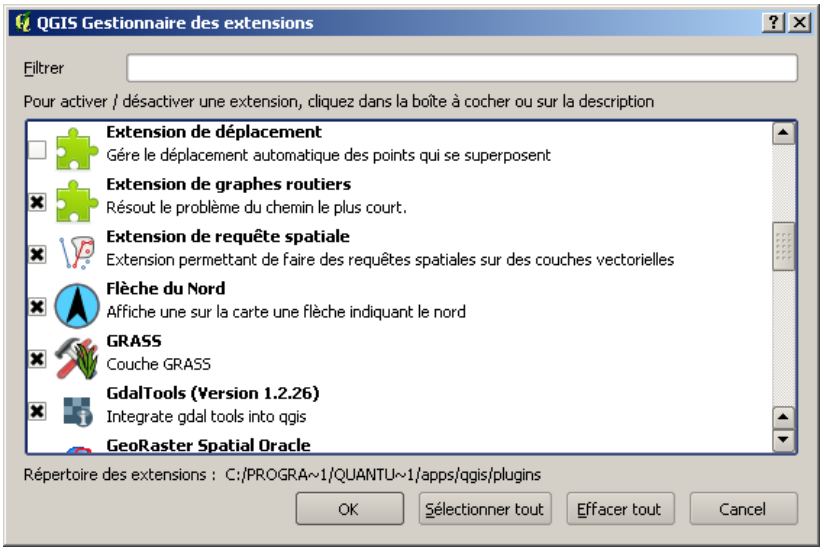
L'onglet de gestion des dépôts permet de renseigner / supprimer / désactiver des dépôts. Il s'agit de déclarer une URL (le dépôt) et lui donner un nom. Sur l'image ci-dessus, on voit les dépôts actifs et connectés en vert, et les dépôts désactivés en gris.

Les principaux dépôts sont :

- Le dépôt officiel de QGIS : http://pyqgis.org/repo/official
- Le dépôt « contributeur » de QGIS : http://pyqgis.org/repo/contributed

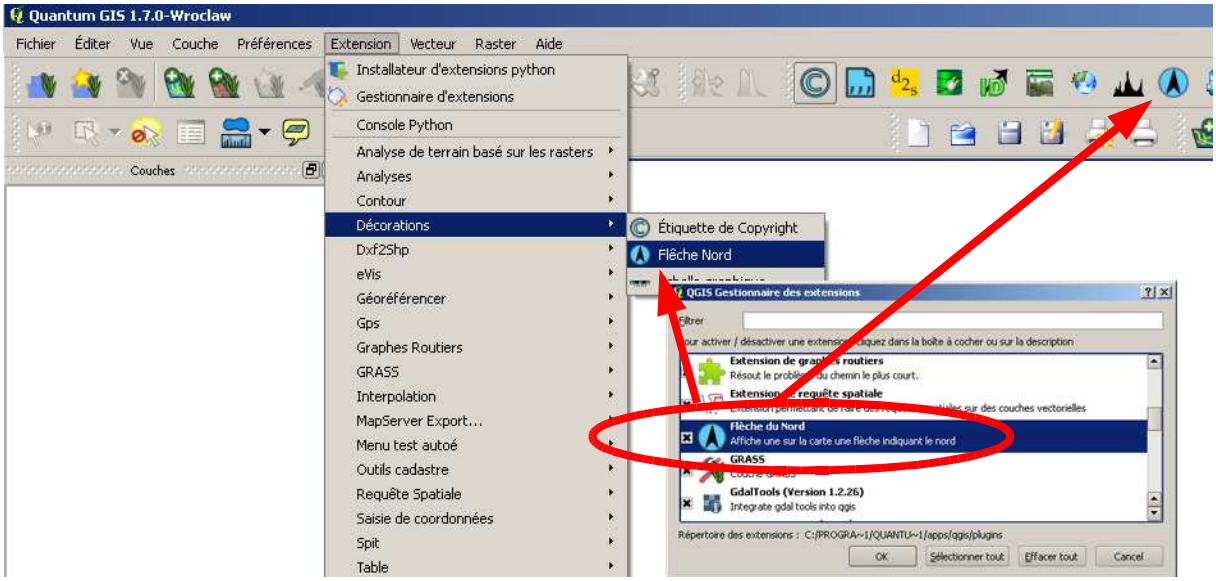
Remarque : penser à vérifier la configuration du proxy dans les préférences générales de QGIS. Cela est important pour pouvoir se connecter à des dépôts qui peuvent être sur Intranet et Internet.

2.1.2. Le gestionnaire d'extensions



Une fois un plugin installé via l'installateur d'extensions, le gestionnaire d'extensions permet de l'activer ou non dans QGIS. L'interface graphique proposée permet de visualiser les plugins disponibles sous forme de liste avec une information synthétique : logo, titre et sous titre.

Quand un plugin est activé (case cochée), les menus et barres d'outils correspondant sont créés dans l'interface graphique de QGIS. Cf image ci-dessous :

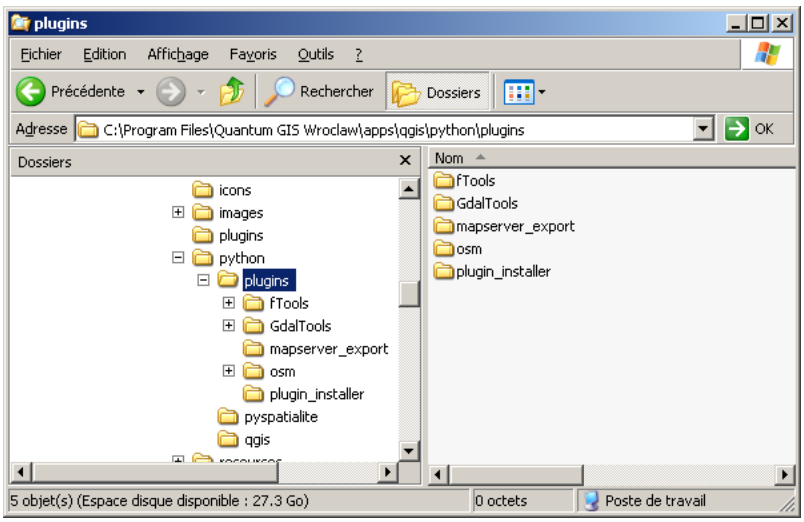


2.2 – Emplacement dans le dossier « programmes » de QGIS

Les plugins QGIS sont constitués d'un ensemble de fichiers – dont leur contenu sera détaillé par la suite. A chaque démarrage de QGIS, ces fichiers sont chargés afin d'être disponibles dans l'application. C'est ce que l'on constate lors des étapes « Démarrage de python » et « Reconstitution des extensions chargées » du lancement de QGIS.

Les extensions « obligatoires » et théoriquement non modifiables de QGIS sont localisées dans le dossier d'installation du logiciel (ex : Program Files pour Windows).

La copie d'écran ci-dessous montre l'emplacement de ces plugins :

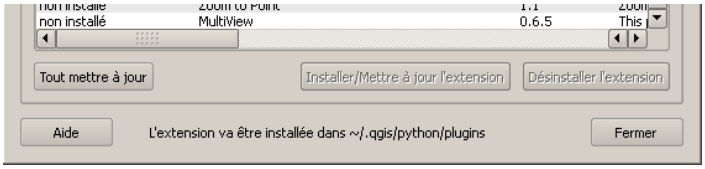


On remarque le dossier « plugin_installer » qui n'est autre que le plugin permettant de gérer les plugins dans QGIS ! Certains autres plugins indispensables comme GdalTools ou fTools sont également dans ce dossier.

Remarque : il est déconseillé de modifier ce répertoire, car cela peut affecter la stabilité de QGIS.

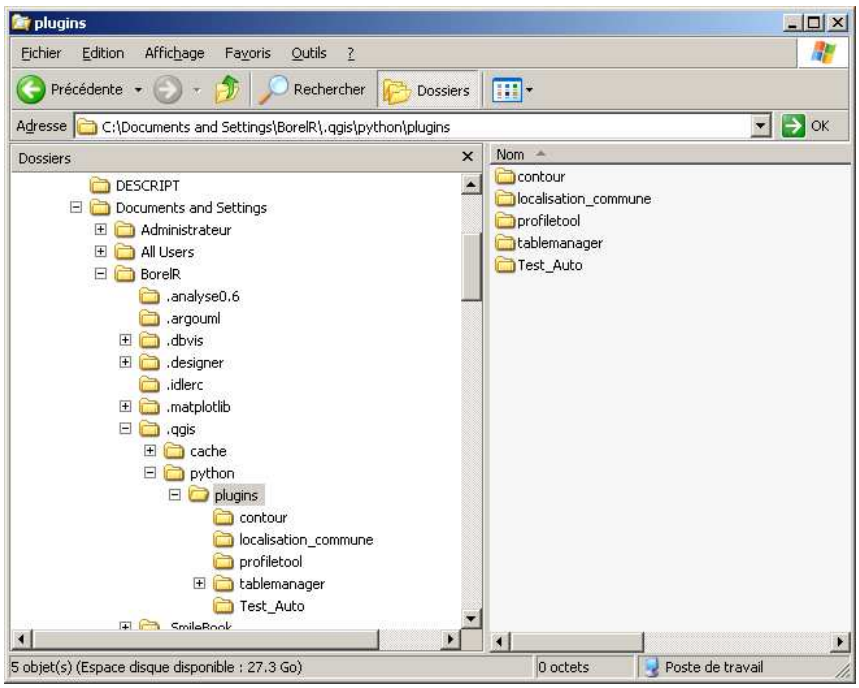
2.3 – Emplacement dans le dossier utilisateur

Les plugins téléchargés via le gestionnaire d'extensions ne sont heureusement pas installés dans le dossier « programmes » de QGIS, mais dans un répertoire utilisateur. L'installateur d'extensions rappelle d'ailleurs cet emplacement (cf ci-dessous) :



Le caractère « ~ » signifie « dossier utilisateur ». Sous Windows, il se situe dans « Documents and Settings ». Sous Linux dans « /home ».

La copie d'écran ci-dessous donne un aperçu de ce dossier sous Windows.

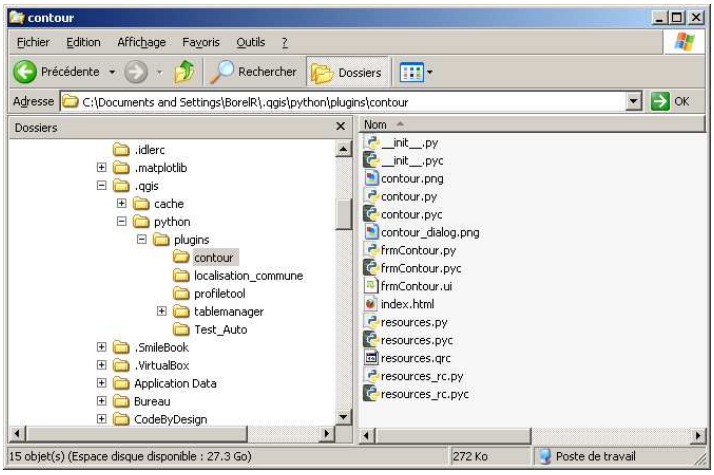


Chaque sous-dossier représente une extension.

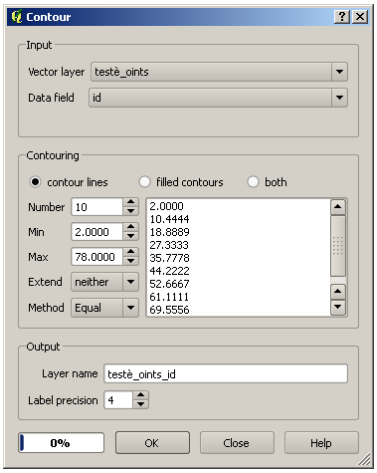
3 – Les fichiers de base d'un plugin

3.1 – Vue d'ensemble des fichiers

Comme il l'a été décrit précédemment, un plugin n'est en fait qu'un dossier dans une arborescence, que ce soit dans le dossier programmes ou le dossier utilisateur. Ce dossier contient plusieurs fichiers permettant d'afficher et d'utiliser le plugin dans QGIS. Ci-dessous un exemple avec le plugin « Contour » disponible sur le dépôt « contributeurs QGIS » :



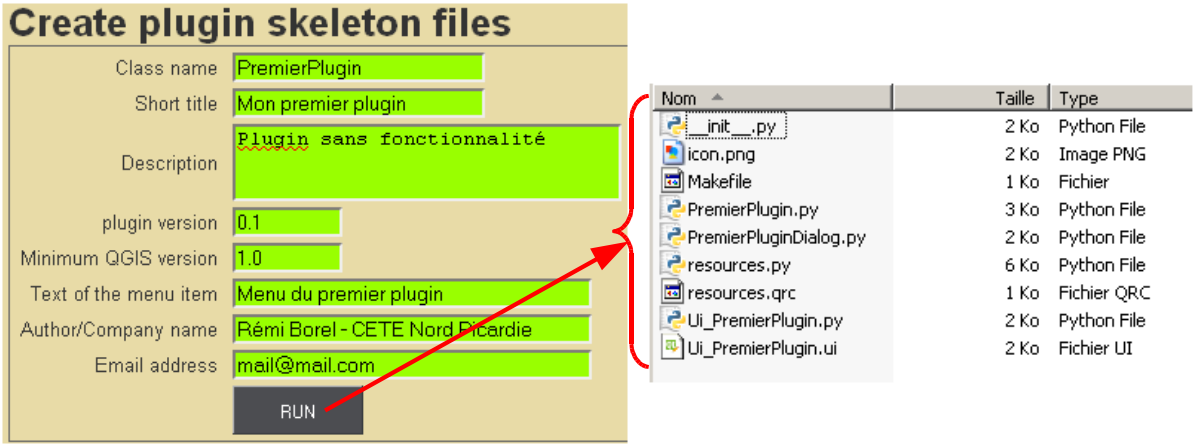
Vision du plugin dans l'arborescence



Vision du plugin dans QGIS

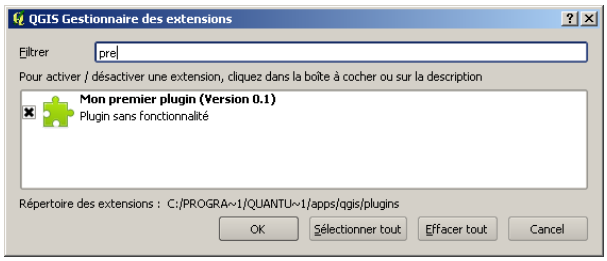
Afin de fonctionner, le plugin doit être constitué de quelques fichiers obligatoires, répondant à des contraintes de contenu et de nommage. Il existe à l'adresse suivante: http://www.dimitrisk.gr/qgis/creator/ un outil gratuit permettant de générer le squelette d'un plugin sans fonctionnalité.

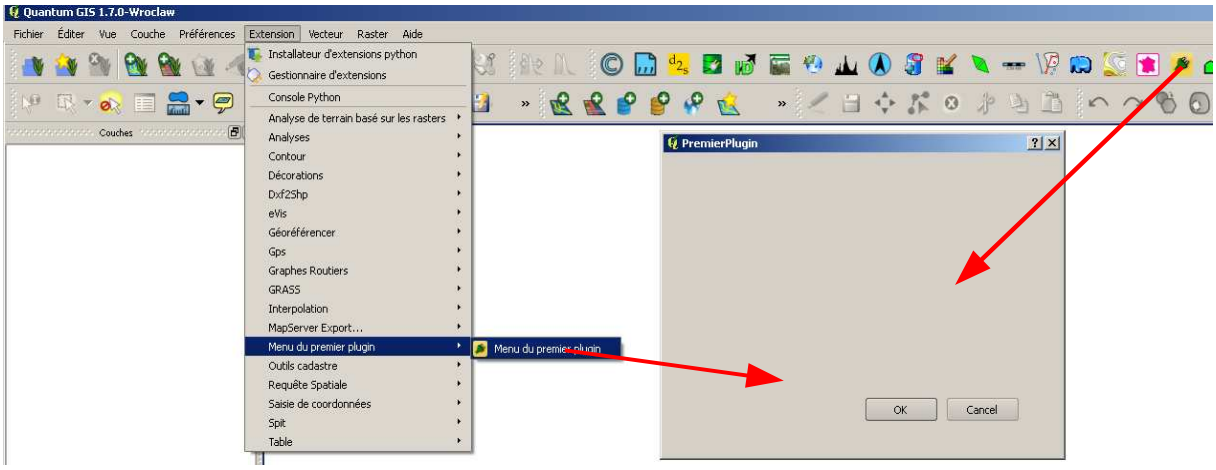
Voici les fichiers générés par cet outil avec les paramètres suivants :



Pour faire fonctionner ce plugin dans QGIS, on peut copier le dossier «PremierPlugin » dans le répertoire utilisateur des plugins.

Au redémarrage de QGIS, ce nouveau plugin sera disponible dans le gestionnaire d'extensions. En le sélectionnant, il devient actif dans QGIS :





Remarque : il est possible que ce plugin factice ne fonctionne pas aussi bien que ci-dessus : erreurs, pas d'icône, lancement impossible. Tous ces paramétrages seront détaillés par la suite.

Les fichiers indispensables au fonctionnement d'un plugin sont les suivants :

- __init__.py : c'est le fichier d'initialisation du plugin,
- PremierPlugin.py : la classe principale du plugin. Elle donne les emplacements où doivent être implantés les boutons et barres d'outils dans QGIS,
- Ui_PremierPlugin.py/ui : les fichiers permettant de « dessiner » l'interface graphique,
- PremierPluginDialog.py : le fichier gérant les interactions entre les outils graphiques (boutons, menus déroulants, etc.) et les actions associées à ces outils.

On peut trouver d'autres fichiers facultatifs, mais donnant un meilleur rendu au plugin :

- icon.png : le logo du plugin,
- ressources.py/qrc : la compilation du logo en langage Python,
- un fichier Python de fonctions spécifiques (conversion, formatage, .etc.),
- éventuellement des fichiers html pour gérer l'aide.

Important : La séparation des différents fichiers du Plugin permet de gérer les aspects fonctionnels et graphiques de manière séparée. Concrètement, il est possible de travailler sur de nouvelles fonctionnalités sans modifier l'interface graphique. A l'inverse, il est possible de modifier l'interface graphique sans que cela n'aie de répercutions sur les fonctionnalités.

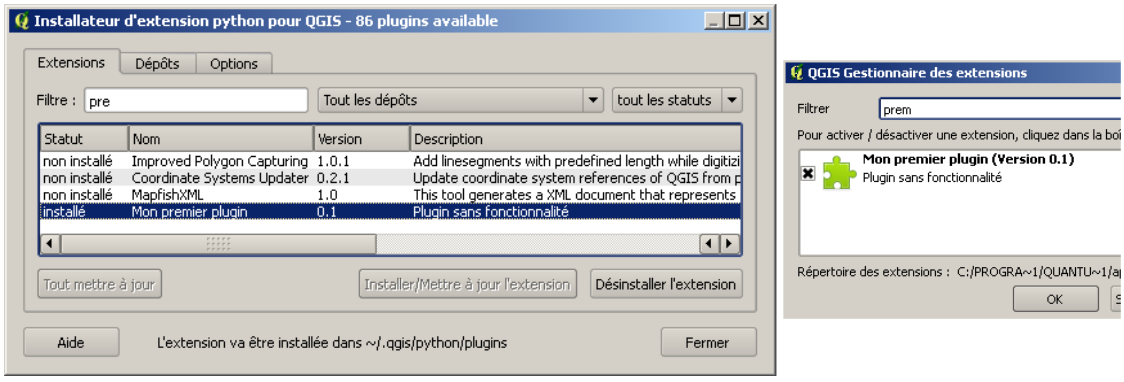
Le contenu et les liens entre ces fichiers est décliné dans les paragraphes suivants.

3.2 – Le fichier d'initialisation

Le fichier d'initialisation doit contenir obligatoirement les fonctions écrites ci-dessous, en respectant la casse :

```
def name():
  return "Mon premier plugin"
def description():
  return "Plugin sans fonctionnalité"
def version():
  return "Version 0.1"
def authorName():
  return "Nom de l'auteur"
def qgisMinimumVersion():
  return "1.0"
def classFactory(iface):
  from PremierPlugin import PremierPlugin
  return PremierPlugin(iface)
```

Les chaînes de caractères retournées par ces fonctions sont destinées à être affichées dans le gestionnaire d'extensions. Il faut absolument bien les renseigner car la lisibilité et la compréhension du plugin en dépendent.



Vu du plugin « Mon premier plugin » dans l'installateur et le gestionnaire d'extensions

De manière assez logique :

- La fonction « name » donne le nom du plugin,
- La fonction « description » en donne la description,
- La fonction « version » en donne la version,
- La fonction « qgisMinimumVersion » donne la version minimale de QGIS à utiliser,
- La fonction « classFactory » appelle la classe principale du plugin. Le nom de la classe principale du plugin doit y apparaître. Si possible, ce nom doit être repris dans le nom du fichier Python (ici PremierPlugin.py).

3.3 – La classe principale

Dans notre exemple, la classe principale « PremierPlugin » se situe dans le ficher PremierPlugin.py. Les lignes de code ci-dessous décrivent pas à pas le contenu de ce fichier et de cette classe :

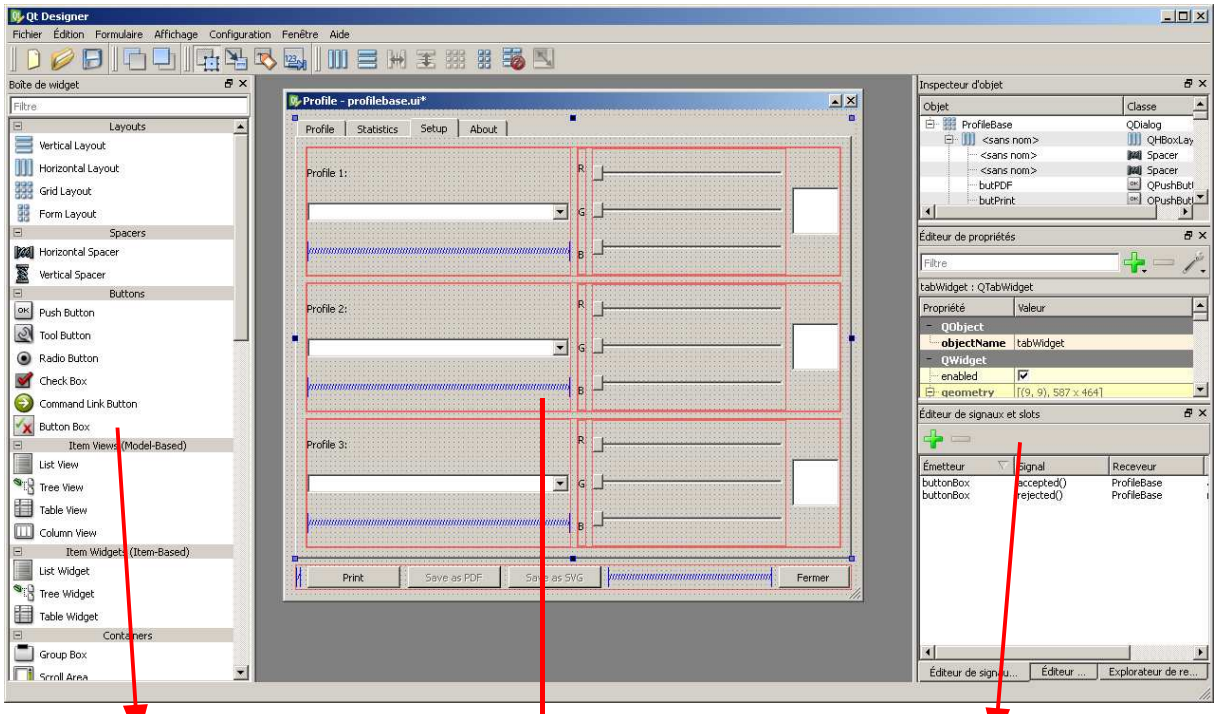
```
# Import des librairies Qt (pour l'interface graphique)
from PyQt4.QtCore import *
from PyQt4.QtGui import *
from qgis.core import *
# Import des ressources nécessaires (images)
import resources
#. Import de la classe gérant les interactions interface graphique / fonctionnalités (voir plus loin)
from PremierPluginDialog import PremierPluginDialog

# Début de la classe principale du plugin
class PremierPlugin:

  def __init__(self, iface):  # Méthode obligatoire d'initialisation de la classe
    # L'objet iface est une instance de la classe QgisInterface de l'API QGIS
    self.iface = iface  # Code à laisser tel quel

  def initGui(self):  # Méthode intégrant le plugin à QGIS depuis le gestionnaire d'extensions
    # Création de l'icône qui, une fois cliquée, ouvrira l'interface graphique du plugin
    self.action = QAction(QIcon(":/plugins/PremierPlugin/icon.png"), \
       "Titre de l'icône", self.iface.mainWindow())
    # Ligne qui lance le plugin (méthode «  run ») au moment du clic (cf remarque ci-après)
    QObject.connect(self.action, SIGNAL("triggered()"), self.run)

    # Ajout de l'icône dans le menu «  Nom du menu » et dans la barre d'outils
    self.iface.addToolBarIcon(self.action)
    self.iface.addPluginToMenu("&Nom du menu", self.action)

  def unload(self):  # Méthode enlevant les références au plugin dans QGIS
    # Suppression de l'icône dans le menu «  Nom du menu » et dans la barre d'outils
    self.iface.removePluginMenu("&Nom du menu",self.action)
    self.iface.removeToolBarIcon(self.action)

  def run(self): # Méthode ouvrant l'interface graphique du plugin
    # Ouverture de la classe gérant les interactions interface graphique / fonctionnalités (voir plus loin)
    dlg = PremierPluginDialog()
    # Afficher l'interface graphique
    dlg.show()
    result = dlg.exec_()
    # Regarde si le bouton OK a été pressé
    if result == 1:
      pass # Si c'est le cas, mettre ici le code donnant des fonctionnalités au plugin …
```

De par la structure du fichier ci-dessus, on remarque qu'il est très aisé de modifier le nom des menus, de créer plusieurs icônes dans un même menu, chaque icône ayant une fonctionnalité spécifique.

La ligne *« QObject.connect(self.action, SIGNAL("triggered()"), self.run) »* ci-dessus signifie que lorsque l'objet action (l'icône) est déclenché, la méthode run est lancée. Ceci est une déclinaison particulière des *signaux et slots* propres à Qt. Ces signaux et slots permettent d'associer des actions à des événements. Ils seront largement utilisés et détaillés dans les parties suivantes.

3.4 – L'interface graphique

Dans un plugin Python pour QGIS, la partie la plus simple à concevoir est l'interface graphique en tant que telle : les boutons, les éléments de formulaire, les listes déroulantes, etc. Pour cela, il est possible d'utiliser l'outil Qt Designer ou Qt Creator[1].



Zone de sélection des objets de l'interface graphique

Zone de création graphique de l'interface

Zone de gestion des propriétés des objets

Aperçu de la fenêtre principale de Qt Designer

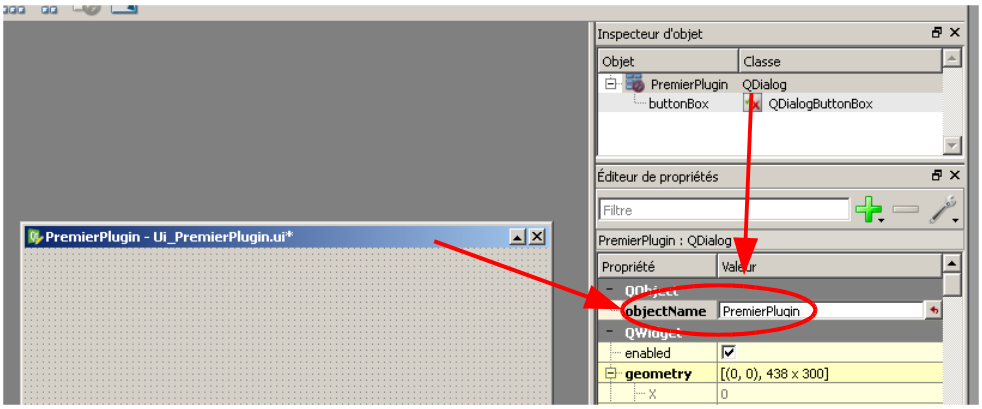
1 Cf en annexe pour l'installation de Qt Designer et la génération automatique de script Python

La création de l'interface graphique ne demande aucune compétence en programmation. Qt Designer enregistre le formulaire dans un fichier ayant une extension «ui », un simple fichier XML. Ce fichier « ui » peut ensuite être converti automatiquement en code Python via une application en ligne de commande [1].

Comme cela est écrit plus haut, il est possible de modifier son interface graphique sans pour autant que cela altère négativement les fonctionnalités. Il suffit pour cela de re-générer le fichier « py » à chaque fois que le fichier « ui » est modifié.

Remarque 1 : traditionnellement, le nom du fichier de l'interface graphique est «Ui_ » suivi du nom de la classe principale du Plugin. Pour l'exemple courant: `Ui_PremierPlugin.py/ui`.

Remarque 2 : le nom de la classe « mère » QDialog dans Qt Designer doit être celui de la classe principale de notre Plugin.



Nom de la classe du QDialog

Ici : « PremierPlugin »

Remarque 3 : veiller à nommer clairement tous les objets de l'interface graphique. Ne pas utiliser le nommage par défaut qui donne un nom identique à tous les objets à l'exception d'un numéro. Donner des noms « français », même s'ils sont longs, car ils vont devoir être appelés dans le fichier d'interactions.
Il est plus facile de faire la différence entre `case_a_cocher_shp`, `case_a_cocher_tab` et `case_a_cocher_kml`qu'entre `checkBox_1`, `checkBox_2` et `checkBox_3`.

3.5 – L'interaction interface graphique / fonctionnalités

Le fichier d'interactions interface graphique / fonctionnalités permet d'associer des actions à des événements se produisant dans l'interface graphique. C'est le principe des formulaires.

Le fichier minimal, généré automatiquement pour l'outil en ligne[2] ressemble à ceci :

```
# chargement des modules Qt génériques
from PyQt4 import QtCore, QtGui
# chargement du fichier d'interface graphique créé à l'étape précédente.
from Ui_PremierPlugin import Ui_PremierPlugin

# creation de la boite de dialogue, appelée dans la méthode Run de la Classe principale (PremierPlugin.py)
class PremierPluginDialog(QtGui.QDialog):
  def __init__(self):
    QtGui.QDialog.__init__(self)
    # Appel de la classe créée dans Qt Designer, importée en début de fichie
    self.ui = Ui_PremierPlugin ()
    self.ui.setupUi(self)
```

La classe PremierPluginDialog ci-dessus est réduite au strict minimum car l'interface graphique par défaut ne contient que les boutons « OK » et « annuler ». Dans la méthode d'initialisation `__init__`, il est possible d'ajouter les éléments suivants :

- l'appel à des fonctions extérieures contenues dans un fichier Python annexe,
- l'appel de l'API QGIS. C'est important pour interagir avec la fenêtre carte par exemple,
- les signaux et slots Qt, qui permettent d'appeler des fonctions lors d'événements.

La section suivante détaille concrètement la manière de mettre en œuvre ces principes.

2 Cf bibliographie

4 – Exemple : création d'un plugin de localisation à la commune

4.1 – Objectifs du plugin

Pour cet exemple, nous allons construire un plugin de localisation à la commune ayant deux fonctionnalités :

- identification d'une commune par son code INSEE,
- identification d'une commune par son nom.

Lorsqu'une commune est identifiée, la fenêtre carte de QGIS doit se centrer sur la commune choisie.
Le plugin doit être accessible dans la barre de menus et la barre d'outils.

Pour cela, il est nécessaire de disposer d'une source de données. Plusieurs méthodes sont possibles :

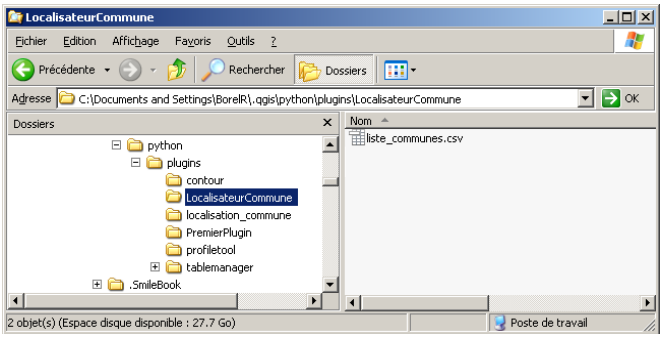
- disposer d'un fichier texte lu au lancement du Plugin puis mis en mémoire,
- lire dans un fichier Shape communal,
- interroger un serveur de base de données[3].

Pour cet exemple, nous allons travailler avec un fichier texte structuré au format tableur (CSV) et contenant les colonnes suivantes : code INSEE, nom de commune sans casse, xmin, ymin, xmax et ymax. Les deux premières colonnes permettent d'interroger les communes. Les 4 dernières permettent d'accéder à leur emprise.

```
"59001";"ABANCOURT";713154;7014138;716629;7016757
"59002";"ABSCON";719393;7024035;723273;7027559
"59003";"AIBES";775732;7013387;780222;7017637
"59004";"AIX";719584;7043298;723908;7045694
"59005";"ALLENNES LES MARAIS";694836;7047264;697503;7051256
"59006";"AMFROIPRET";751615;7019181;753712;7021399
"59007";"ANHIERS";710106;7033001;711794;7035066
"59008";"ANICHE";716844;7024069;720021;7027671
"59009";"VILLENEUVE D ASCQ";708198;7055791;714749;7064040
"59010";"ANNEUX";707222;7005011;710399;7007746
```

Exemple de contenu du fichier CSV communal (10 premières lignes)

Le nom de la classe principale de notre plugin sera **`LocalisateurCommune`** Il faut donc créer un dossier du même nom dans le répertoire des plugins QGIS (cf § 2.3).



On dispose dans ce répertoire le fichiers de communes.

3 Le module psycopg2 de Python permet de communiquer avec un serveur PostgreSQL

4.2 – Préalable : l'encodage des caractères

Afin d'éviter des problèmes d'affichage des caractères accentués, il est nécessaire de définir l'encodage de chaque fichier Python. Pour cela, il faut:

- éditer le fichier « py » dans l'encodage voulu,
- mettre dans la toute première ligne du fichier cet encodage.

Pour des raisons d'universalité, il est conseillé d'utilisé l'encodage UTF-8. Celui-ci est géré par des éditeurs de texte comme SciTe ou Notepad++.

La première ligne de chaque fichier est alors :

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

S'il reste quelques erreurs à l'affichage, comme par exemple un « Ã© » à la place d'un « é », il reste la solution de rajouter un « u » devant la chaîne de caractères. Par exemple :

```
machaine = "Donner à voir"
```

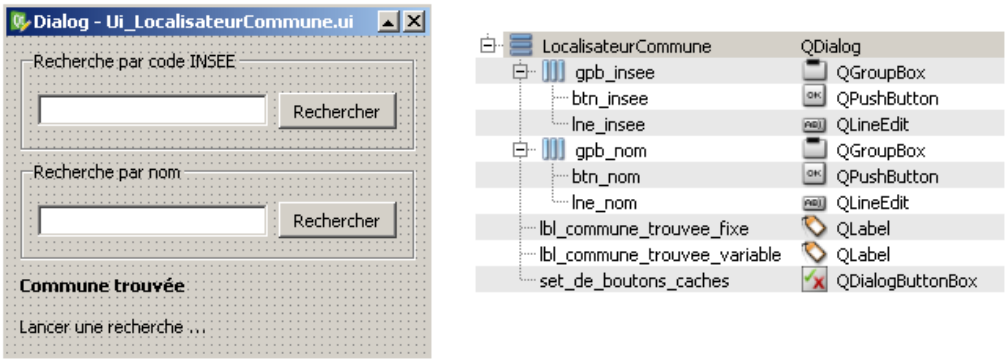
devient

```
machaine = u"Donner à voir"
```

Cette remarque est valable pour les versions 2.x de Python. En version 3, les fichiers « py » doivent être obligatoirement codés en UTF-8. Actuellement, QGIS 1.7..0 fonctionne avec la version 2.5.

4.3 – L'interface graphique

Lancer Qt Designer et essayer[4] de dessiner l'interface graphique, avec les noms d'objets et la forme tels que ci-dessous. L'enregistrer sous le nom **`Ui_LocalisateurCommune.ui`** dans le dossier précédemment créé.



Vue de l'interface graphique et de ses objets

L'interface graphique se compose de trois parties: d'une part les zones de recherche par code INSEE et par nom, comportant toutes deux un édition de ligne et un bouton rechercher. D'autre part, la zone inférieure, montrant à l'utilisateur le résultat de la recherche.

Comme indiqué dans la section 3, les noms ont été choisis de manière à être facilement repérables dans l'interface. De plus, le nom de la classe principale QDialog est bien **`LocalisateurCommune`**

Remarque: quand le fichier « ui » est prêt, penser à le convertir en « py » via l'utilitaire pyuic4 (cf annexe 1).

4 Si besoin, le code source de cette interface est disponible en annexe 2 ...

4.4 – Le fichier d'initialisation, la classe principale et l'icône

4.4.1. Le fichier d'initialisation

Le fichier d'initialisation doit s'appeler **`__init__.py`**. Il est important de rappeler que les fonctions présentes dans ce fichier sont obligatoires car elles permettent de référencer le plugin au démarrage de QGIS.

Penser à bien écrire le nom de la classe principale dans la fonction **`classFactory`**.

4.4.2. La classe principale

Pour des raisons de facilité de lecture, la classe principale doit porter le nom du dossier dans lequel est rangé le plugin. Le fichier Python contenant cette classe principale doit lui aussi porter ce nom. Ici, c'est **`LocalisateurCommune.py`**.

Le contenu de ce fichier est calqué sur le modèle présenté dans la section 3.

4.4.3. L'icône du plugin

Pour l'instant, la seule référence à l'icône qui doit être affichée dans QGIS figure dans la ligne:

```
Qaction(QIcon(":/plugins/LocalisateurCommune/icon.png"), ....
```

du fichier de la classe principale.

Ceci n'est pas suffisant pour charger cette image, car QGIS a besoin de les « compiler » pour les utiliser. Pour cela, il faut créer un fichier « qrc ». Ce dernier fait référence à toutes les ressources externes à compiler. Il est structuré comme un fichier XML. La compilation de ces ressources est possible grâce à l'utilitaire **`pyrcc4`**[5]. Celui-ci est disponible au même niveau que pyuic4.

Le fichier « qrc » faisant référence à cette ressource icon.png est le suivant :

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

A condition que l'icône existe (format PNG de 24px de côté) et qu'elle soit localisée dans le dossier du plugin, la commande :

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

générera le fichier resources.py dont QGIS a besoin pour afficher l'icône.

5 Cf annexe 1 pour voir comment avoir accès à pyrcc4 et pyuic4

4.5 – Le fichier de fonctions annexes

A ce stade, le plugin peut s'afficher dans QGIS, mais ne propose toujours pas de fonctionnalités. Il s'agit juste d'une interface graphique dont les deux boutons «Rechercher» sont inactifs. Certaines fonctionnalités doivent être développées, dont certaines complètement indépendantes de QGIS ou de notre interface graphique. Nous allons créer ici un fichier spécifique de fonctions. Ceci présente les avantages de pouvoir :

- facilement réutiliser ces fonctions dans d'autres projets,
- faire évoluer les fonctionnalités sans toucher aux autres fichiers,
- faire évoluer l'interface sans toucher au fichier de fonctions.

Dans le cas précis de notre plugin, les fonctions indépendantes pressenties sont:

Fonctionnalité	***Paramètres en entrée***	***Valeurs en sortie***
Ouvrir le fichier CSV des communes pour le mettre en mémoire	Le nom du fichier CSV	Un tableau associatif (dictionnaire) avec comme clé le code INSEE
Recherche une commune via un code INSEE	Code INSEE	Code INSEE
Recherche une commune via un nom	Chaîne de caractères	Code INSEE
Donner les informations constituant une commune	Code INSEE	Tableau d'informations sur la commune

Ces fonctions peuvent être testées en dehors de QGIS puisqu'elles sont en écrites Python « pur ». Une fois qu'elles ont été éprouvées, il est possible de les lier à l'interface graphique.

Toutes ces fonctions sont écrites dans un fichier **`fonctions.py`**. Le fichier final est disponible en annexe. On constatera que certaines sous-fonctions peuvent être pratiques comme par exemple la suppression des accents.

4.6 – Le fichier d'interactions

Dans le modèle proposé dans la section 3, le fichier d'interactions ne contient aucune référence aux objets de l'interface graphique: il se contente de l'initialiser. En ce qui concerne notre plugin exemple, il faut créer 2 événements :

- le clic sur le bouton « Rechercher » par code INSEE,
- le clic sur le bouton « Rechercher » par nom de commune.

Il faut également créer les action associées à chacun des événements.

4.6.1. La gestion des événements

La déclaration de ces événements doit être faite dans la méthode d'initialisation **__init__** de la classe. Elle suit le principe des signaux et slots de Qt[6]. La syntaxe générale est la suivante, pour le clic sur un des boutons rechercher :

```
self.connect(self.ui.btn_insee,SIGNAL("clicked()"),self.cherche_insee)
```

Dès que l'objet **btn_insee** reçoit le signal **clicked()** (cliqué), il faut exéctuer la méthode **cherche_insee**.

On peut également associer une action au fait d'appuyer sur « Entrée » au clavier quand le champ de saisie est activé :

```
self.connect(self.ui.lne_insee, SIGNAL("returnPressed()"), self.ui.btn_insee, SLOT("click()"))
```

Cette instruction signifie qu'un appui sur la touche entrée dans la ligne d'édition du code INSEE est équivalent à un clic sur le bouton de recherche. Cela présente l'avantage d'éviter de passer par une fonction spécifique et réduit les risques de bugs.

Remarque importante : pour les objets de type QDialog , l'appui sur la touche « Entrée » implique par défaut la validation du formulaire. Cela peut poser des problèmes quand on associe une action supplémentaire à l'appui sur Entrée (comme c'est le cas ci-dessus) et que notre plugin n'a pas de notion de validation. Deux solutions sont envisageables pour palier à ce problème:

- l'utilisation d'un QWidget, pour lequel le bouton Entrée n'a par défaut aucune action,
- l'insertion dans le QDialog d'un QdialogButtonBox contenant un bouton Ok et étant réduit à une taille nulle.

Pour cet exemple, c'est la deuxième solution qui a été choisie. Le code source en annexe 2 prend en compte cette particularité.

4.6.2. La création des actions associées aux événements

Comme indiqué ci-dessus, un événement est associé à une action. Les actions sont simplement des méthodes (fonctions) à écrire dans la classe **LocalisateurCommuneDialog**

Par exemple, la méthode ci-dessous récupère le contenu du champ **lne_nom** , lui applique une fonction de nettoyage de chaîne et l'affiche dans le label en bas de l'interface graphique:

```
def cherche_nom(self):
    insee = self.ui.lne_nom.text().toUtf8()
    insee = u"%s" % fonctions.nettoie_chaine_majuscule(insee)
    self.ui.lbl_commune_trouvee_variable.setText(insee)
```

Par extension, il est possible de développer n'importe quelle fonction (ou méthode) au sein de la classe LocalisateurCommuneDialog. Dès qu'une fonction n'a aucun rapport avec des actions QGIS ou la gestion des objets de l'interface graphique, le principe est de la mettre dans le module fonctions.py précédemment décrit.

6 Plus d'infos dans la section bibliographie

5 – Diffusion d'un plugin, création d'un dépôt

5.1 – Introduction

Afin de communiquer un plugin à un autre utilisateur de QGIS, la méthode la plus directe est d'effectuer des copies du répertoire dudit plugin dans le dossier des plugin de chaque QGIS concerné. Ainsi, au prochain démarrage, QGIS chargera ce nouveau plugin. Cette méthode a priori efficace n'est pourtant pas la plus pertinente car :

- comme évoqué en section 2, QGIS possède un installateur d'extensions,
- cet installateur d'extension détecte les plugins mis à jour,
- en procédant par copie de dossiers, la mise à jour de l'extension peut être fastidieuse, si de nombreux postes sont concernés.

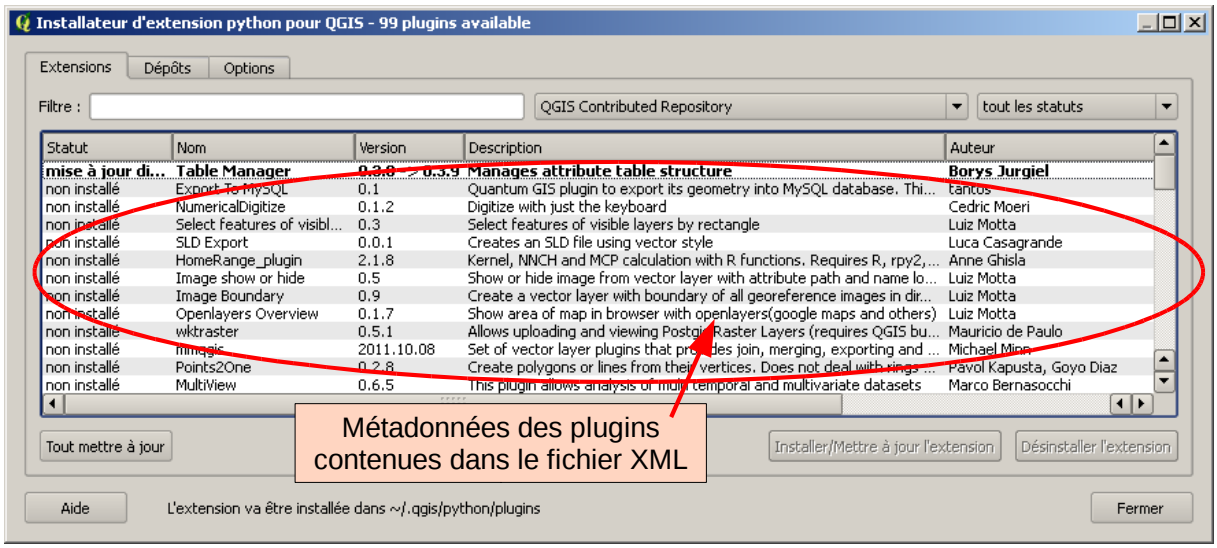
La diffusion de plugin par le système de dépôts permet de gérer tous ces aspects. Elle nécessite de disposer d'un serveur WEB, ou de passer par le dépôt des contributeurs QGIS[7]. La première méthode, plus générale, est développée dans ce document.

5.2 – Les fichiers obligatoires

Un dépôt ne peut être interrogeable à condition qu'il respecte certaines conditions obligatoires:

- un fichier XML doit référencer tous les plugins du dépôt,
- à chaque plugin doit être associé un fichier zip téléchargeable.

Quand l'installateur d'extensions se connecte à un dépôt, il doit pouvoir accéder au fichier XML, sorte de fichier de « métadonnées » des plugins. C'est le contenu de ce fichier qui permet d'afficher les informations relatives à chaque plugin dans l'installateur (cf image ci-dessous) :



Quand l'utilisateur clique sur le bouton « Installer », QGIS télécharge le fichier zip contenant le plugin, le décompresse et le copie automatiquement dans le dossier des plugins. On remarque dans l'image ci-dessus que la gestion des mises à jour est facilitée, via les lignes en gras.

7 Disponible à l'adresse http://pyqgis.org/manager/python_plugin/list

5.2.1. Le fichier XML des métadonnées

Le fichier XML doit respecter a minima la structure ci-dessous, dont les informations sont calquées sur le plugin développé en section 4. On suppose pour cet exemple que le fichier XML créé se nomme liste_plugins.xml, et qu'il est accessible à l'adresse suivante : http://www.monserveur.com/plugins_qgis/liste_plugins.xml. Cette adresse est de fait l'adresse du dépôt que l'on peut ajouter à l'installateur d'extensions QGIS.

```
<?xml version = '1.0' encoding = 'UTF-8'?>
<plugins>
    <pyqgis_plugin version="0.1" name="Localisateur à la commune" >
        <description>Localisation par code INSEE ou par nom</description>
        <homepage>http://www.monserveur.com/ </homepage>
        <file_name>LocalisateurCommune.zip </file_name>
        <author_name>Rémi BOREL - CETE Nord Picardie </author_name>
        <download_url>http://www.monserveur.com/plugins_qgis/LocalisateurCommune.zip </download_url>
        <qgis_minimum_version>1.6.0</qgis_minimum_version>
    </pyqgis_plugin>
</plugins>
```

Il est important de bien éditer le fichier dans l'encodage spécifié dans la première ligne. Tout comme pour les scripts python, l'UTF-8 est conseillé. Le fichier XML contient autant de balises **`pyqgis_plugin`** que de plugins à référencer.

Il est surtout *fondamental* de recopier de manière strictement identique les informations contenues dans le fichier **`__init__.py`**, de manière à garantir une cohérence.

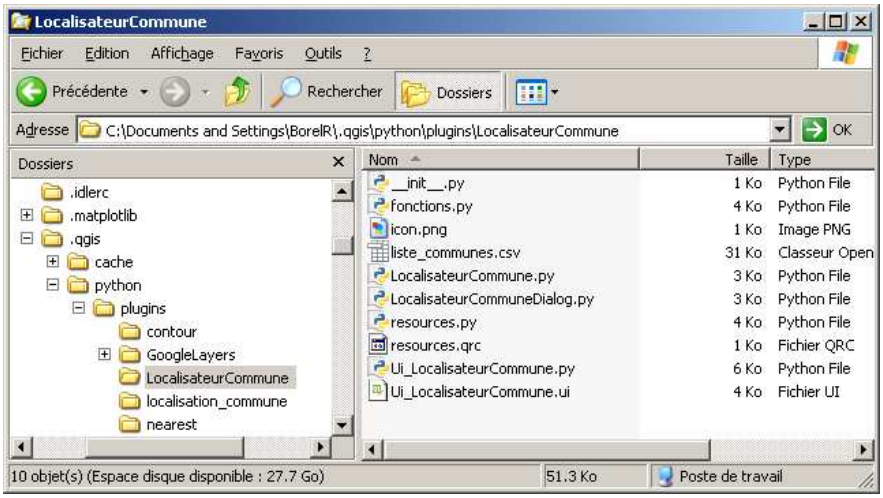
La balise **`homepage`** n'a pas de lien avec le plugin. C'est en général le site internet personnel ou professionnel de l'auteur. Les balises **`file_name`** et **`download_url`** doivent faire exactement référence au fichier à télécharger (cf paragraphe suivant). Sinon, le plugin ne pourra pas être installé.

L'affichage du fichier XML dans un navigateur donne le résultat suivant :

Il est possible d'assigner au fichier XML une feuille de styles de manière à la rendre plus lisible. Ceci est détaillé §5.3.

5.2.2. L'archive ZIP du contenu du plugin

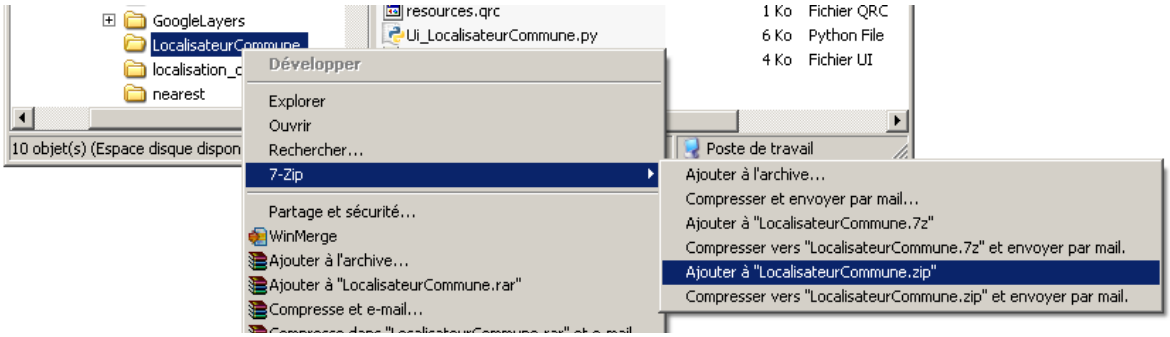
Il suffit de regrouper dans une archive zip l'ensemble des fichiers nécessaires au fonctionnement du plugin, plus certains fichiers utiles à des développeurs qui souhaiteraient modifier le plugin. Par exemple, les fichiers « ui » sont intéressants à diffuser car ils sont facilement éditables dans Qt Designer. En revanche, les fichiers pyc ne sont pas utiles, car ils sont recréés à chaque chargement de QGIS. L'image ci-dessous montre les fichiers nécessaires au fonctionnement du plugin développé en section 4 :



Pour que le plugin fonctionne, il faut que l'archive zip contienne à sa racine le dossier du plugin. Ici, il s'agit du dossier LocalisateurCommune. Un logiciel comme 7zip[8], généralement installé par défaut sur les machines du ministère est très fiable et très facile d'utilisation.

Afin de garantir les meilleurs résultats, la méthode la plus simple est :

1. bouton droit sur le dossier du plugin,
2. menu contextuel « 7zip... »,
3. enfin, « Ajouter à nomdudossier.zip ».



C'est cette archive qu'il faut ensuite déposer à la même adresse que cette indiquée dans la balise **`download_url`** du fichier XML des plugins.

8 Logiciel libre, téléchargeable sur http://www.7-zip.org/

5.3 – Les fichiers facultatifs

5.3.1. Pour mettre en forme le XML

L'objectif est de mettre en forme la page WEB affichée lors de l'appel de l'URL du plugin. Cela permet de communiquer l'URL et de faire connaître ses plugins. Deux fichiers sont nécessaires :

- un fichier « XSL » qui est en fait structuré comme un XML. C'est une sorte de template pour l'affichage du fichier XML. Il permet de décrire les balises du fichier XML de base,
- un fichier CSS, comme en HTML. Ce fichier va permettre de créer des styles pour chaque type de balise du fichier XML.

De plus, il faut ajouter la ligne suivante :

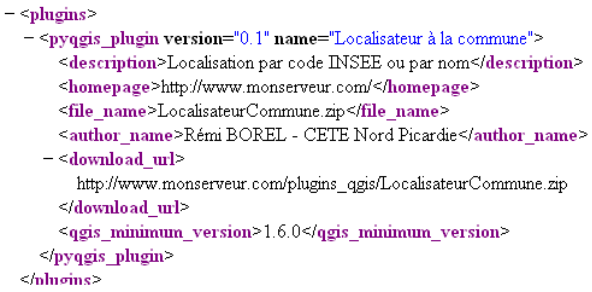
```
<?xml-stylesheet type="text/xsl" href="liste_plugins.xsl" ?>
```

en deuxième ligne du fichier XML de base. C'est cette ligne qui appelle le fichier XSL.

Ci dessous les différentes vues de la page WEB de notre dépôt de plugin : http://www.monserveur.com/plugins_qgis/liste_plugins.xml en fonction de la présence ou non des 2 fichiers annexes.

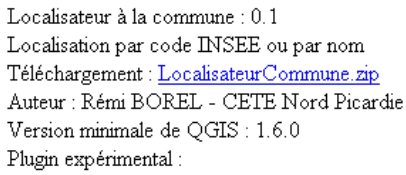
Fichier XML seul

Apparaît comme dans un éditeur de texte



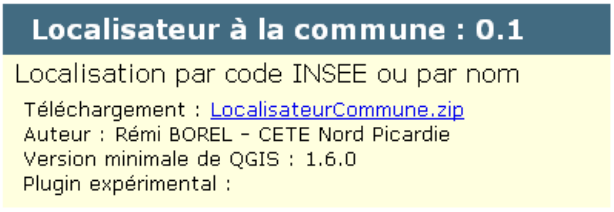
Fichier XML
+ fichier XSL

Pas de sytle mais une description des champs



Fichier XML
+ fichier XSL
+ fichier CSS

Description des champs et style



Il n'est pas nécessaire de connaître strictement le fonctionnement et la syntaxe des fichiers XSL et CSS. Les exemples de fichiers XSL et CSS donnés en annexe sont adaptables et suffisants pour cet usage.

5.3.2. Pour l'accès via l'URL

Dans l'état actuel de la configuration, l'accès au dépôt se fait indifféremment dans un navigateur comme dans QGIS via l'URL : http://www.monserveur.com/plugins_qgis/liste_plugins.xml

Il est possible de ne pas faire apparaître la référence au nom du fichier XML dans l'URL. Pour les utilisateurs d'Apache, il faut créer un fichier nommé « **.htaccess** » (avec le point) au même endroit que le fichier XML. Ce fichier doit simplement contenir la ligne :

```
DirectoryIndex liste_plugins.xml
```

6 – Annexe 1 : Qt Designer et fichiers Python

6.1 – Installation de Qt Designer

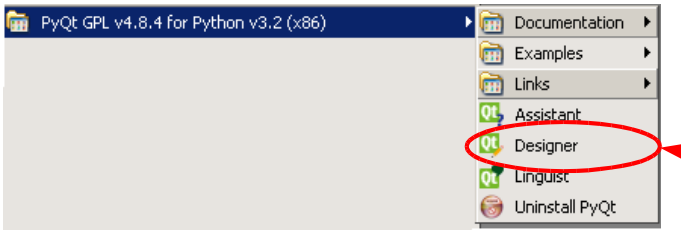
Qt Designer peut être installé selon au moins deux méthodes :

- Via le framework complet Qt téléchargeable gratuitement sur Internet,
- Via l'installation de Python et du module PyQt.

La première méthode est très complète, mais ne permet pas d'avoir accès aux scripts permettant de convertir les fichiers « ui » en fichiers « py ». Ce document présentera donc la seconde méthode.

Le langage Python est librement téléchargeable sur le site python.org. L'installation d'une version au choix de Python sous Windows se fait comme pour n'importe quel logiciel. Python se trouve alors dans le répertoire C:\PythonXX, où XX représente le numéro de version.
Remarque : installer une version de Python correspondant à celle de QGIS.

Le module Python PyQt est lui aussi librement téléchargeable sur internet[9]. Son installation est également très simple et bien guidée. A son issue, un répertoire C:\PythonXX\Lib\site-packages\PyQt4 est créé, et le menu démarrer comporte le sous menu correspondant.



Qt Designer permet d'éditer des interfaces graphiques facilement

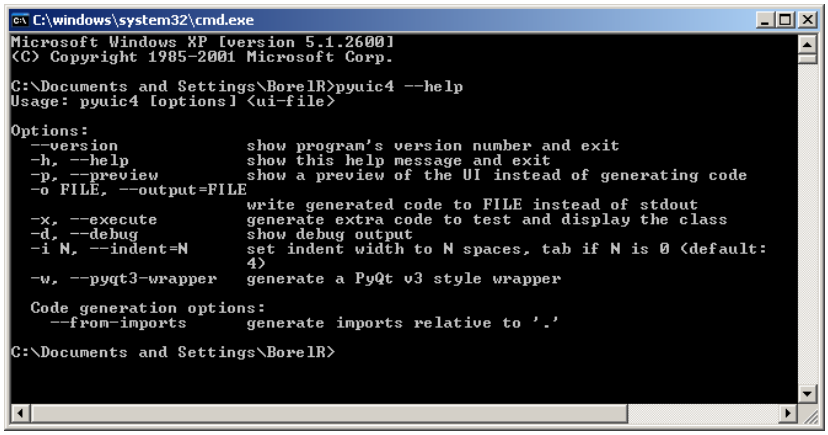
6.2 – Génération automatique de fichiers Python

Une fois l'interface graphique créée sous Qt Designer, on obtient un fichier «ui ». Afin de le convertir en fichier Python « py », on peut utiliser un utilitaire de conversion. Celui-ci se trouve dans le répertoire C:\PythonXX\Lib\site-packages\PyQt4 et porte le nom de `pyuic4.bat`. C'est un utilitaire en ligne de commandes.

Afin de l'utiliser depuis n'importe quel répertoire, il faut rajouter C:\PythonXX\Lib\site-packages\PyQt4 au PATH de Windows.

9 http://www.riverbankcomputing.co.uk/software/pyqt/download

La commande `pyuic4 --help` donne alors l'aide de l'utilitaire :

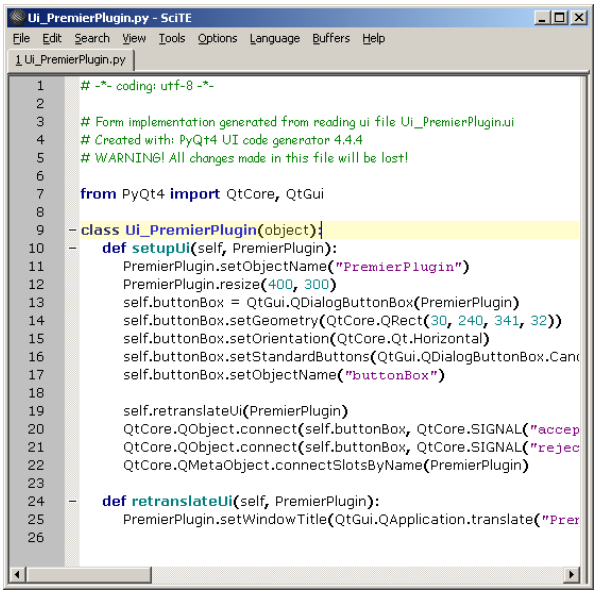


Si l'on a par exemple conçu le fichier interface.ui, et que l'on veut obtenir interface.py, il faut exécuter la commande suivante :

```
pyuic4 -o interface.py interface.ui
```

Le résultat est le suivant :

Contenu du fichier « ui »



Contenu du fichier « py »

On trouve également l'utilitaire pyrcc4, qui fonctionne sous le même principe.

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

7 – Annexe 2 : codes sources de l'exemple développé

7.1 – L'interface graphique

Contenu à mettre dans le fichier **`Ui_LocalisateurCommune.ui`**:

```
<?xml version="1.0" encoding="UTF-8"?>
<ui version="4.0">
 <class>LocalisateurCommune</class>
 <widget class="QDialog" name="LocalisateurCommune">
  <property name="geometry">
   <rect>
    <x>0</x>
    <y>0</y>
    <width>263</width>
    <height>202</height>
   </rect>
  </property>
  <property name="windowTitle">
   <string>Localisateur à la commune</string>
  </property>
  <layout class="QVBoxLayout" name="verticalLayout">
   <item>
    <widget class="QGroupBox" name="gpb_insee">
     <property name="title">
      <string>Recherche par code INSEE</string>
     </property>
     <layout class="QHBoxLayout" name="horizontalLayout">
      <item>
       <widget class="QLineEdit" name="lne_insee"/>
      </item>
      <item>
       <widget class="QPushButton" name="btn_insee">
        <property name="text">
         <string>Rechercher</string>
        </property>
       </widget>
      </item>
     </layout>
    </widget>
   </item>
   <item>
    <widget class="QGroupBox" name="gpb_nom">
     <property name="title">
      <string>Recherche par nom</string>
     </property>
     <layout class="QHBoxLayout" name="horizontalLayout_2">
      <item>
       <widget class="QLineEdit" name="lne_nom"/>
      </item>
```

```
   <item>
    <widget class="QPushButton" name="btn_nom">
     <property name="text">
      <string>Rechercher</string>
     </property>
    </widget>
   </item>
  </layout>
 </widget>
</item>
<item>
 <widget class="QLabel" name="lbl_commune_trouvee_fixe">
  <property name="sizePolicy">
   <sizepolicy hsizetype="Preferred" vsizetype="Minimum">
    <horstretch>0</horstretch>
    <verstretch>0</verstretch>
   </sizepolicy>
  </property>
  <property name="minimumSize">
   <size>
    <width>0</width>
    <height>20</height>
   </size>
  </property>
  <property name="maximumSize">
   <size>
    <width>16777215</width>
    <height>20</height>
   </size>
  </property>
  <property name="font">
   <font>
    <weight>75</weight>
    <bold>true</bold>
   </font>
  </property>
  <property name="text">
   <string>Commune trouvée</string>
  </property>
 </widget>
</item>
<item>
 <widget class="QLabel" name="lbl_commune_trouvee_variable">
  <property name="sizePolicy">
   <sizepolicy hsizetype="Preferred" vsizetype="Minimum">
    <horstretch>0</horstretch>
    <verstretch>0</verstretch>
   </sizepolicy>
  </property>
  <property name="minimumSize">
   <size>
    <width>0</width>
    <height>20</height>
   </size>
  </property>
  <property name="maximumSize">
   <size>
    <width>16777215</width>
    <height>20</height>
```

```
    </size>
   </property>
   <property name="text">
    <string>Lancer une recherche ... </string>
   </property>
  </widget>
 </item>
 <item>
  <widget class="QDialogButtonBox" name="set_de_boutons_caches" >
   <property name="maximumSize">
    <size>
     <width>0</width>
     <height>0</height>
    </size>
   </property>
   <property name="standardButtons" >
    <set>QDialogButtonBox::Ok</set>
   </property>
  </widget>
 </item>
</layout>
</widget>
<resources/>
<connections/>
</ui>
```

7.2 – Le fichier d'initialisation

Contenu à mettre dans le fichier **__init__.py** :

```
# coding: utf-8
def name():
  return u"Localisateur à la commune"
def description():
  return "Localisation par code INSEE ou par nom"
def version():
  return "Version 0.1"
def authorName():
  return "Rémi BOREL - CETE Nord Picardie"
def qgisMinimumVersion():
  return "1.6.0"
def classFactory(iface):
  from LocalisateurCommune import LocalisateurCommune
  return LocalisateurCommune(iface)
```

7.3 – Le fichier de la classe principale

Contenu à mettre dans le fichier **`LocalisateurCommune.py`**:

```
# coding: utf-8

# Import des librairies Qt (pour l'interface graphique)
from PyQt4.QtCore import *
from PyQt4.QtGui import *
from qgis.core import *
# Import des ressources nécessaires (images)
import resources, fonctions, os
#. Import de la classe gérant les interactions interface graphique / fonctionnalités (voir plus loin)
from LocalisateurCommuneDialog import LocalisateurCommuneDialog

# Début de la classe principale du plugin
class LocalisateurCommune:

    def __init__(self, iface):  # Méthode obligatoire d'initialisation de la classe
        # L'objet iface est une instance de la classe QgisInterface de l'API QGIS
        self.iface = iface  # Code à laisser tel quel

    def initGui(self):  # Méthode intégrant le plugin à QGIS depuis le gestionnaire d'extensions
        # Création de l'icône qui, une fois cliquée, ouvrira l'interface graphique du plugin
        self.action = QAction(QIcon(":/plugins/LocalisateurCommune/icon.png" ),
        u"Localisateur à la commune" , self.iface.mainWindow())
        # Ligne qui lance le plugin (méthode «  run ») au moment du clic (cf remarque ci-après)
        QObject.connect(self.action, SIGNAL("triggered()" ), self.run)

        # Ajout de l'icône dans le menu «  Nom du menu » et dans la barre d'outils
        self.iface.addToolBarIcon(self.action)
        self.iface.addPluginToMenu(u"&Localisateur à la commune" , self.action)

    def unload(self):  # Méthode enlevant les références au plugin dans QGIS
        # Suppression de l'icône dans le menu «  Nom du menu » et dans la barre d'outils
        self.iface.removePluginMenu(u"&Localisateur à la commune" ,self.action)
        self.iface.removeToolBarIcon(self.action)

    def run(self): # Méthode ouvrant l'interface graphique du plugin

        #Vérification de l'existence du fichier de communes
        rep = QgsApplication.qgisSettingsDirPath()
        fichier = os.path.join(u"%s" % rep, 'python', 'plugins', 'LocalisateurCommune' ,
'liste_communes.csv' )
        erreur, dict_communes = fonctions.ouvre_csv_communes(fichier)

        if erreur=="OK":  #Si pas d'erreur, on ouvre l'interface
            dlg = LocalisateurCommuneDialog(self, dict_communes)  #le self est nécessaire
pour accéder à QGIS dans le fichier LocalisateurCommuneDialog
            # Afficher l'interface graphique
            dlg.show()
            dlg.exec_()
        else:
            QMessageBox.warning(self.iface.mainWindow(), "ERREUR", erreur)
```

7.4 – Le fichier des fonctions

Contenu à mettre dans le fichier **`fonctions.py`**:

```
# coding: utf-8
import csv
import re
import unicodedata

#-------------------------CHAINES DE CARATERES------------------------------
def ouvre_csv_communes(fichier_csv):
    """Ouvre le fichier des communes et met le résultat dans un dictionnaire
    ayant comme clé le code INSEE
    """
    try:
        csvfile = open(fichier_csv)
    except:
        return "Erreur : fichier de communes introuvable" ,{}
    try:

        dialect = csv.Sniffer().sniff(csvfile.read(20000))
        csvfile.seek(0)
        reader = csv.reader(csvfile, dialect)
        dict_communes = {}
        for ligne in reader:
            dict_communes[ligne[0]] = [ligne[1], int(ligne[2]), int(ligne[3]), int(ligne[4]),
int(ligne[5]) ]
        csvfile.close()
        return "OK", dict_communes
    except: #Retourne False (erreur) s'il y a un problème
        return "Erreur : lecture du fichier impossible" ,{}

def cherche_insee(insee, dict_communes):
    """Cherche dans le tableau de données l'existence du code INSEE saisi
    Retourne une chaine vide si le code n'existe pas
    """
    liste_codes = dict_communes.keys()  #Les codes INSEE sont les clés du dictionnaire
    if insee in liste_codes:
        return insee
    else:
        return ""

def cherche_nom(nom, dict_communes):
    """Cherche dans le tableau de données la ville
    se rapprochant le plus du nom saisi
    """
    #Nettoyage de la chaine en entrée
    nom = nettoie_chaine_majuscule(nom)
    if nom == '':
        return ""
    #Comparaison du nom saisi à tous les noms de communes
    distance_mini = 100000000000
    insee_mieux = ""
    for insee in dict_communes.keys():
        d = levenshtein(nom, dict_communes[insee][0])
        if d<distance_mini:
            distance_mini = d
```

```
                insee_mieux = insee
            if d==0:
                break
    return insee_mieux

def infos_commune(insee, dict_communes):
    """Renvoie les infos d'une commune en fonction de son code INSEE.
    C'est un tuple avec le libellé sans casse, xmin, ymin, xmax et ymax
    """
    try:
        return dict_communes[insee]
    except:
        return ["Aucune commune trouvée", 0, 0, 0, 0]

#-------------------------SOUS FONCTIONS-----------------------------

def nettoie_chaine_majuscule(chaineutf8):
    """Met une chaine en majuscule, supprime les caracteres accentues
    et supprime les blancs inutiles (inclus les underscore)
    """
    #Enlève les accents
    chaineutf8 = unicode(chaineutf8, "utf-8" )
    chaineutf8 =unicodedata.normalize('NFD',chaineutf8)
    chaineutf8 = chaineutf8.encode('ascii','ignore')
    #Met  en majuscules
    chaineutf8 = chaineutf8.upper()
    #enlève caractères spéciux :
    chaineutf8 = re.sub('\W', ' ', chaineutf8)
    chaineutf8 = re.sub('_', ' ', chaineutf8)
    #Enlève les séries de blancs
    chaineutf8 = chaineutf8.strip()
    chaineutf8 = re.sub('\s+', ' ', chaineutf8)
    return chaineutf8

def levenshtein(a,b):
    """Calcule la distance de Levenshtein
    entre les chaines a and b.
    """
    n, m = len(a), len(b)
    if n > m:
        # Make sure n <= m, to use O(min(n,m)) space
        a,b = b,a
        n,m = m,n

    current = list(range(n+1))
    for i in range(1,m+1):
        previous, current = current, [i]+[0]*n
        for j in range(1,n+1):
            add, delete = previous[j]+1, current[j-1]+1
            change = previous[j-1]
            if a[j-1] != b[i-1]:
                change = change + 1
            current[j] = min(add, delete, change)

    return current[n]
```

7.5 – Le fichier d'interactions

Contenu à mettre dans le fichier **`LocalisateurCommuneDialog.py`**:

```
# coding: utf-8

# chargement des modules Qt génériques
from PyQt4.QtCore import *
from PyQt4.QtGui import *

#Import de QGIS
from qgis import *
from qgis.core import *
from qgis.gui import *

# chargement du fichier d'interface graphique
from Ui_LocalisateurCommune import Ui_LocalisateurCommune

import fonctions

# creation de la boite de dialogue, appelée dans la méthode Run de la Classe principale (PremierPlugin.py)
class LocalisateurCommuneDialog(QDialog):
    def __init__(self, classe_iface_parent, dict_communes):  #On récupère le dictionnaire des communes
        QDialog.__init__(self)
        # Appel de la classe créée dans Qt Designer, importée en début de fichier
        self.ui = Ui_LocalisateurCommune()
        self.ui.setupUi(self)

        #IMPORTANT :
        #Recupetation dans self.iface des proprietes d'acces à l'interface ! (self.iface)
        self.iface = classe_iface_parent.iface
        #et aussi le QgsMapCanvas
        self.canvas = self.iface.mapCanvas()

        #GESTION DE l'APPUI SUR ENTREE PAR ZONE DE BOUTON
        self.connect(self.ui.lne_nom, SIGNAL("returnPressed()"), self.ui.btn_nom, SLOT("click()"))
        self.connect(self.ui.lne_insee, SIGNAL("returnPressed()"), self.ui.btn_insee, SLOT("click()"))

        #Clic sur la recherche par code insee et par nom
        self.connect(self.ui.btn_insee,SIGNAL("clicked()"),self.cherche_insee)
        self.connect(self.ui.btn_nom,SIGNAL("clicked()"),self.cherche_nom)

        #Chargement des communes
        self.dict_communes = dict_communes


    def cherche_nom(self):
        #~ QMessageBox.warning(self, "ERREUR", "PASSAGE DANS NOM")
        nom = self.ui.lne_nom.text().toUtf8()
        #Recherche de la commune
        insee_trouve = fonctions.cherche_nom(nom, self.dict_communes)
        self.affiche_resultat_et_zoom(insee_trouve)
```

```
def cherche_insee(self):
    #~ QMessageBox.warning(self, "ERREUR", "PASSAGE DANS INSEE")
    insee = self.ui.lne_insee.text().toUtf8()
    insee = u"%s" % fonctions.nettoie_chaine_majuscule(insee)
    #Recherche de la commune
    insee_trouve = fonctions.cherche_insee(insee, self.dict_communes)
    self.affiche_resultat_et_zoom(insee_trouve)


def affiche_resultat_et_zoom(self, insee):
    #Recupération des récultats
    resultat = fonctions.infos_commune(insee, self.dict_communes)
    self.ui.lbl_commune_trouvee_variable.setText(unicode(resultat[0], 'utf-8'))
    #Zoom sur l'emprise
    if resultat[0]>100000:
        self.zoom_extent(resultat[1], resultat[2], resultat[3], resultat[4])

def zoom_extent(self, xmin, ymin, xmax, ymax):
    #On agrandit l'emprise au cas ou (min 500m)
    if xmax-xmin<500:
        xmin = xmin - (500-(xmax-xmin))/2
        xmax = xmax + (500-(xmax-xmin))/2
    if ymax-ymin<500:
        ymin = ymin - (500-(ymax-ymin))/2
        ymax = ymax + (500-(ymax-ymin))/2
    #Creation du rectangle d'emprise :
    rec = QgsRectangle(xmin, ymin, xmax, ymax)
    self.canvas.setExtent(rec)
    self.canvas.refresh()
```

8 – Bibliographie – Sources

QGIS Téthys : Notions élémentaires du langage Python
CP2I / DO su Sud-Est
Février 2011

Création de « squelette » pour plugin QGIS
http://www.dimitrisk.gr/qgis/creator/

Comment écrire des plugins Python pour QGIS
http://www.qgis.org/wiki/Writing_Python_Plugins

Déposer des plugins sur le dépôt des contributeurs QGIS
http://pyqgis.org/manager/python_plugin/list

Documentation de référence pour l'API QGIS
http://doc.qgis.org/head/index.html

Documentation de référence pour la librairie Qt
http://doc.qt.nokia.com/latest/

Manuel utilisateur pour Qt Designer
http://doc.qt.nokia.com/stable/designer-manual.html

Les ressources dans Qt
http://doc.qt.nokia.com/latest/resources.html