INTRODUCTION AU DEVELOPPEMENT SOUS QGIS

QGIS, comme tous les logiciels SIG, ne constitue pas une fin en soi. En effet, la version de base de ce logiciel libre offre relativement peu d'outils de traitement de l'information géographique, à l'instar d'ArcGIS sans ses fameuses *toolbox*… Initialement, toute la puissance de QGIS résidait dans sa complémentarité avec Grass. Désormais, QGIS dispose d'une communauté de développeurs importante et dynamique qui développe des extensions qui lui sont propres. L'extension FTools est une bonne illustration de cette dynamique. Cette extension permettant d'effectuer de nombreux traitements sur des objets vectoriels figure dorénavant parmi les extensions de base installées avec QGIS. Pour développer de telles extensions et ainsi améliorer les capacités de QGIS, il faut avoir recours à un langage de programmation comme Python. Pour commencer à programmer en Python sous QGIS, il faut aller dans « Extension -> Console Python ». Des lignes de commande apparaissent, il convient dès lors de bien les utiliser. Ce complément de cours propose une introduction à Python et donne des exemples concrets permettant d'aborder le développement sous QGIS.

1) INTRODUCTION CURSIVE A PYTHON

Python est un langage qui peut s'utiliser dans de nombreux contextes et s'adapter à tout type d'utilisation grâce à des bibliothèques spécialisées à chaque traitement. Ce langage est néanmoins particulièrement utilisé comme langage de script pour automatiser des tâches simples mais fastidieuses ou contraignantes. Il est aussi particulièrement répandu dans le monde scientifique et possède de nombreuses extensions destinées aux applications numériques. On l'utilise également comme langage de développement pour des prototypes lorsqu'on a besoin d'une application fonctionnelle avant de l'optimiser avec un langage de plus bas niveau. Plus précisément, Python est un langage de programmation objet, interprété, multi-paradigme, et multi-plateforme.

Python a été conçu pour être un langage lisible. Il vise ainsi à être visuellement épuré. De plus, ce langage utilise des mots anglais là où d'autres langages utilisent de la ponctuation (peu compréhensible pour un non-initié). Il possède également moins de constructions syntaxiques que de nombreux langages structurés tels que C, Perl, ou Pascal. Les commentaires sont indiqués par le caractère croisillon (#).

De plus, les blocs sont identifiés par l'indentation (*ie.* l'ajout de tabulations ou d'espaces dans un fichier texte) au lieu d'accolades comme en C ou C++ ou de « begin ... end » comme en Pascal. Une augmentation de l'indentation marque le début d'un bloc et une réduction de

l'indentation marque la fin du bloc courant. C'est pourquoi, Python requiert une rigueur dans l'écriture des scripts et en fait un langage que je trouve pédagogique, car il oblige de ne pas coder « vulgairement ». Ainsi, à une ligne correspond une commande et à une colonne correspond un niveau.

Commençons par l'affectation qui se fait à l'aide du signe égalité :

```
>>> a=2
>>> b=3
```

L'ensemble des opérations « +, -, ×, / » sont disponibles et pour afficher le résultat d'une opération il faut avoir recours à la commande « print » :

```
>>> e=a+b
>>> print e
5
```

Attention ! La division réserve des surprises car c'est une division entière :

```
>>> e=a/b
>>> print e
0
```

Pour obtenir, la valeur généralement attendue, il faut utiliser la commande « float() » :

```
>>> e=a/float(b)
>>> print e
0.666666666667
```

Les objets couramment utilisés en Python sont les chaînes de caractère et les listes :

```
>>> a="tr"
>>> print a
tr
>>> b=2
>>> e=a+b
Traceback (most recent call last):
File "<input>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
>>> b=str(b)
>>> print(b)
2
>>> e=a+b
>>> print e
tr2
>>> liste=[1,2]
>>> print liste
[1, 2]
>>> liste=liste+[4,5,6]
>>> print liste
[1, 2, 4, 5, 6]
```

```
>>> print liste[4]
6
>>> print liste[0]
1
```

Enfin, les boucles s'exécutent de cette manière :

```
>>> for i in range(3) :
... print i
...
0
1
2
```

Pour plus d'informations, il existe d'excellentes introductions :

http://www.larsen-b.com/static/intro_python/

http://tdc-www.harvard.edu/Python.pdf

2) Application : Analyser des graphes spatiaux dans QGIS a l'aide de NetworkX

Le SIG libre QGIS propose une console Python permettant d'interagir avec l'ensemble des outils de QGIS et par conséquent de développer de petits programmes automatisant plusieurs tâches. Dans les faits, il est plus souvent intéressant d'utiliser les nombreuses bibliothèques Python existantes afin d'étendre les capacités de QGIS. Par exemple, il existe de nombreuses bibliothèques permettant de réaliser de l'analyse de graphe (igraph, NetworkX…). Ici, nous allons utiliser NetworkX, bibliothèque développée originalement par Aric Hagberg, Dan Schult, et Pieter Swart (http://networkx.github.com/).

Pour utiliser une bibliothèque Python dans QGIS, il faut la télécharger (suivre le lien suivant pour la télécharger [1]), puis dézipper le fichier source et placer ce fichier dans le répertoire des bibliothèques Python de QGIS. Pour la version 1.8 de QGIS, le répertoire se situe dans : « C:\Program Files\Quantum GIS Lisboa\apps\Python27\Lib\site-packages » (attention, cette version de QGIS n'a pas été choisie par hasard, en effet elle implémente Python 2.7 qui est nécessaire pour la dernière version de NetworkX). Toutes les bibliothèques Python déjà présentes dans QGIS se trouvent dans ce répertoire (MapPlotLib, PyQt4…). La bonne complémentarité entre QGIS et Python est ici illustrée, car il n'y a plus rien à faire. Tout est installé et il est alors possible de réaliser de l'analyse de graphe avec QGIS, alors même qu'initialement ce n'était pas le cas.

Pour vérifier, il suffit d'ouvrir QGIS, d'aller dans « Extension -> Console Python », puis de taper les lignes de code suivantes :

```
>>> import networkx as nx  # Import de la bibliothèque NetworkX
>>> G=nx.Graph()  # Création d'un graphe G
>>> G.add_edges_from([(1,2),(1,3)])  # Création des arcs du graph G
>>> nx.clustering(G)  # Calcul les clustering coefficient des noeuds
{1: 0.0, 2: 0.0, 3: 0.0}
>>> nx.degree(G)  # Calcul les degrés des noeuds
{1: 2, 2: 1, 3: 1}
```

L'étape suivante est un peu plus complexe que la précédente. En effet, sans même avoir une grande connaissance de Python et de QGIS, il est possible d'utiliser la bibliothèque NetworkX dans QGIS en étudiant les tutoriels disponibles (le lien vers les tutoriels [2]). Faire interagir Python et QGIS ajoute un niveau de difficulté supplémentaire. Ainsi, partons du fait que deux couches ont été chargées dans QGIS. La première couche est composée de nœuds caractérisés par leur identifiant. La deuxième couche est composée d'arcs caractérisés par leur identifiant, les deux identifiants des nœuds situés aux extrémités de l'arc et leur longueur. L'enjeu est alors de créer le graphe « NetworkX » à partir de ces deux couches. Pour cela, il faut récupérer les attributs des couches à l'aide de Python :

```
>>> canvas=qgis.utils.iface.mapCanvas() # Interaction avec la carte et les couches
>>> allLayers = canvas.layers() # Récupère les deux couches chargées dans QGIS
>>> couche_noeud = allLayers[0] # Récupère la couche supérieure
>>> couche_arc = allLayers[1]  # Récupère la couche inférieure
>>> provider = couche_arc.dataProvider() # Interaction avec la couche arc
>>> field=provider.fields() # Récupère les champs de la couche arc
>>> startind=provider.fieldNameIndex('depart') # Index du champ "départ"
>>> endind=provider.fieldNameIndex('arrivee')   # Index du champ "arrivée"
>>> distind=provider.fieldNameIndex('distance_K') # Index du champ "distance_K"
>>> feat = QgsFeature()   # Permet de manipuler les entités des couches
>>> allAttrs = provider.attributeIndexes()   # Récupère les index des entités
>>> provider.select(allAttrs)   # Sélectionne toutes les entités
>>> table=[] # Création d'une table qui va récupérer les informations
>>> while provider.nextFeature(feat): # Boucle parcourant l'ensemble des entités
...     attrs = feat.attributeMap() # Récupère tous les attributs des entités
...     attrsstart=attrs.values()[startind].toInt()[0]  # Attribut nœud de départ
...     attrsend=attrs.values()[endind].toInt()[0]  # Attribut nœud d'arrivée
...     attrsdist=attrs.values()[distind].toInt()[0] # Attribut distance
...     table=table+[(attrsstart,attrsend,attrsdist)] # Ajoute les attributs
récupérés à la table
```

Une fois les attributs des couches permettant de créer le graphe récupérés et après les avoir collectés dans un tableau, il est possible d'analyser ce graphe à l'aide de la bibliothèque NetworkX :

```
>>> import networkx as nx  # Import de la bibliothèque NetworkX
>>> G=nx.Graph() # Création d'un graphe G
>>> G.add_weighted_edges_from(table) # Ajout des entités de la table au graphe G
```

```
>>> b=nx.betweenness_centrality(G,weight='weight') # Calcul de la centralité des noeuds du graphe G
```

Enfin, il est possible de récupérer les résultats obtenus pour les implémenter dans la couche « nœuds » :

```
>>> provider2 = couche_noeud.dataProvider() # interaction avec les données
>>> nbcol=int(provider2.fieldCount()) # Récupère le nombre de champs de la couche
>>> from PyQt4.QtCore import *  # Import de la bibliothèque PyQt4
>>> from PyQt4.QtGui import *  # Import de la bibliothèque PyQt4
>>> provider2.addAttributes([QgsField("centralite", QVariant.Double)])
>>> taille=len(b)  # Récupère le nombre de nœuds
>>> couche_noeud.startEditing() # Permet l'édition de la couche nœuds
>>> for k in range(taille):  # Boucle qui parcoure l'ensemble des nœuds
…       couche_noeud.changeAttributeValue(k,nbcol,b[k+1]) # Attribution des valeurs
>>> couche_noeud.commitChanges() # Enregistrement des changements
```

Toutes ces lignes de commande peuvent être récupérées et réorganisées pour créer un petit programme permettant de calculer les centralités des nœuds d'un graphe spatial. Pour l'exécuter dans GIS, il suffit de taper la ligne de commande suivante :

```
>>> execfile("C:/Users/XXXXX/Desktop/centralite.py")  # Appelle le programme à exécuter
```

Voilà cette application a permis de présenter de manière concrète l'utilisation de python dans QGIS. Il est désormais possible d'effectuer de l'analyse de réseaux sous QGIS à l'aide de la bibliothèque NetworkX en interagissant avec l'interface QGIS (récupération des données affichées, modification des couches).

3) Application pour QGIS 2.0

QGIS 2.0 introduit quelques différences de programmation (assez mineures dans le cas présenté) qui empêchent d'utiliser le code ci-dessus en l'état. Il faut donc effectuer quelques modifications pour le rendre compatible avec QGIS 2.0. Ainsi, vous trouverez en rouge les éléments à modifier pour que le code présenté ci-dessus soit valide.

```
>>> canvas=qgis.utils.iface.mapCanvas() # Interaction avec la carte et les couches
>>> allLayers = canvas.layers() # Récupère les deux couches chargées dans QGIS
>>> couche_noeud = allLayers[0] # Récupère la couche supérieure
>>> couche_arc = allLayers[1]  # Récupère la couche inférieure
>>> provider = couche_arc.dataProvider() # Interaction avec la couche arc
>>> field=provider.fields() # Récupère les champs de la couche arc
>>> startind=provider.fieldNameIndex('depart') # Index du champ "départ"
>>> endind=provider.fieldNameIndex('arrivee')   # Index du champ "arrivée"
>>> distind=provider.fieldNameIndex('distance_K') # Index du champ "distance_K"
>>> feat = QgsFeature()   # Permet de manipuler les entités des couches
>>> allAttrs = provider.attributeIndexes()   # Récupère les index des entités
```

```
>>> provider.select(allAttrs)   # Sélectionne toutes les entités
>>> fit1 = provider.getFeatures(
QgsFeatureRequest().setSubsetOfAttributes([startind,endind,distind]) )
>>> table=[] # Création d'une table qui va récupérer les informations
>>> while fit1.nextFeature(feat): # Boucle parcourant l'ensemble des entités
…     attrs = feat.attributeMap() # Récupère tous les attributs des entités
…     attrsstart=attrs.values()[startind].toInt()[0]  # Attribut nœud de départ
…     attrsend=attrs.values()[endind].toInt()[0]  # Attribut nœud d'arrivée
…     attrsdist=attrs.values()[distind].toInt()[0] # Attribut distance
…     attrsstart=int(feat.attributes()[startind]) # Attribut nœud de départ
…     attrsend= int(feat.attributes()[endind])  # Attribut nœud d'arrivée
…     attrsdist= int(feat.attributes()[distind]) # Attribut distance
…     table=table+[(attrsstart,attrsend,attrsdist)] # Ajoute les attributs
récupérés à la table
```

Le code exploitant NetworkX n'est pas à modifier. En revanche, le code permettant de stocker les résultats doit légèrement être modifié :

```
>>> provider2 = couche_noeud.dataProvider() # interaction avec les données
>>> nbcol=int(provider2.fieldCount()) # Récupère le nombre de champs de la couche
>>> nbcol=int(provider2.fields().count())
>>> from PyQt4.QtCore import *  # Import de la bibliothèque PyQt4
>>> from PyQt4.QtGui import *  # Import de la bibliothèque PyQt4
>>> provider2.addAttributes([QgsField("centralite", QVariant.Double)])
>>> taille=len(b)  # Récupère le nombre de nœuds
>>> couche_noeud.startEditing() # Permet l'édition de la couche nœuds
>>> for k in range(taille):  # Boucle qui parcoure l'ensemble des nœuds
…     couche_noeud.changeAttributeValue(k,nbcol,b[k+1]) # Attribution des valeurs
>>> couche_noeud.commitChanges() # Enregistrement des changements
```