INTRODUCTION AU DEVELOPPEMENT SOUS QGIS 3

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... Néanmoins, même avec ce type d'outils, et même avec des applications métiers, très vite il conviendra de développer ses propres traitements pour faire des choses intéressantes. 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 lorsque l'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 « +, -, x, / » sont disponibles et pour afficher le résultat d'une opération il faut avoir recours à la commande « print » :

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

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

```
>>> texte = 'texte'
>>> liste = [2,1,5]
>>> listeplusgrande = liste + [2,4]
>>> print liste[2]
5
```

```
>>> liste[2] = 7
>>> print liste
[2,1,7]
>>> tab = [[2,1,10],[6,2,8]]
>>> print tab[1][2]
8
```

Enfin, les boucles s'exécutent de cette manière :

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

2) PREMIERES INTERACTIONS AVEC PYQGIS

La première chose à connaître sur PyQGIS concerne l'interaction avec les couches chargées dans QGIS.

```
>>> canvas = qgis.utils.iface.mapCanvas() # Interaction avec la carte
>>> allLayers = canvas.layers()# Récupère les couches chargées dans QGIS
>>> couche_noeud = allLayers[0] # Récupère la couche supérieure
>>> print(couche_noeud.name())# Affiche le nom de la couche
```

Néanmoins, cette méthode est liée aux couches affichées dans le « canvas », ainsi il n'est par exemple plus possible d'accéder (depuis QGIS 3) à des couches textuelles non géocodées par ce biais. Il est alors possible de passer par les instances de « QgsProject ». Attention, l'ordre des couches correspond alors à leur ordre d'importation.

```
>>> allLayers = list(QgsProject.instance().mapLayers().values())
```

Pour être sûr de manipuler la bonne couche, il convient parfois de l'appeler plutôt par son nom.

```
>>> couche_noeud = QgsProject.instance().mapLayersByName('Villes')[0]
```

Une fois la couche récupérée, pour consulter les champs et les valeurs correspondantes, il faudra utiliser le « dataprovider ».

```
>>> provider = couche_noeud.dataProvider() # Interaction avec la couche arc
>>> print(provider.fields()[0].name()) # Affiche le nom du premier champ
>>> nbcol = provider.fields().count() # Récupère le nombre de champs
>>> nameid = provider.fieldNameIndex('Nom') # Index du champ Nom
```

Dans ce cadre, pour parcourir les valeurs d'un champ il faudra créer une boucle parcourant les entités.

```
>>> feat = QgsFeature() # Permet de manipuler les entités des couches
>>> fit = provider.getFeatures() # Récupère les entités
>>> table = [] # Variable pour stocker les noms
>>> while fit.nextFeature(feat):
            print(feat.attributes()[nameid]) # Affiche les noms
            table = table + [str(feat.attributes()[nameid])] # Stocke les noms
```

Il est aussi parfois pertinent d'interagir avec les objets sélectionnés par l'utilisateur pour récupérer les valeurs.

```
>>> layer = iface.activeLayer() # Récupère la couche sélectionnable
>>> print(layer.selectedFeatureCount())# Affiche le nombre d'entités sélectionnées
>>> objet = layer.selectedFeatures()[0] # Récupère l'entité sélectionnée
>>> print(objet.attributes()[nameid]) # Affiche le nom de cet entité
```

Il est souvent nécessaire de modifier une couche en créant des champs et en complétant les données issues des traitements effectués. Pour cela, il est possible d'utiliser les fonctions « addAttributes » et « changeAttributeValue ».

```
>>> provider.addAttributes([QgsField('NOM_MAJ', QVariant.String)]) # Création champ
>>> couche_noeud.startEditing() # Mode Edition
>>> for i in range(len(table)):
        maj = table[i].upper() # Mise en capitale
        couche_noeud.changeAttributeValue(i, nbcol, maj) # Modification
>>> couche_noeud.commitChanges() # Sauvegarde des modifications
```

Enfin, créer une couche et gérer les géométries fait partie des éléments fondamentaux de PyQGIS. La création de couche passe principalement par quatre étapes. La première concerne la création de la couche à l'aide de la fonction « QgsVectorLayer ». La deuxième consiste à créer sa structure avec la création des champs comme précédemment. La troisième étape consiste à créer les entités à l'aide des fonctions « setGeometry » et « setAttributes » et de les ajouter à la couche créée avec la fonction « addFeatures ». Enfin, après avoir tout sauvegardé, il faudra penser à rajouter cette couche à l'interface.

```
>>> vln = QgsVectorLayer("Point?crs=epsg:27572", "Nodes", "memory")# Création couche
>>> prn = vln.dataProvider()
>>> prn.addAttributes( [QgsField("Nom", QVariant.String)]) # Création des champs
>>> fet = QgsFeature()
>>> fet.setGeometry( objet.geometry() ) # Création géométrie d'une entité
>>> fet.setAttributes( [ objet.attributes()[nameid] ]) # Propriété de l'entité
>>> vln.startEditing()
>>> prn.addFeatures( [ fet ] ) # Ajout de l'entité à la couche
>>> vln.commitChanges()
>>> QgsProject.instance().addMapLayer(vln) # Ajout de la couche à l'interface
```

Les fonctions « fromPointXY », « fromPolylineXY » ou « fromPolygonXY » permettent de facilement créer des objets géométriques « à la main ».

```
>>> point = QgsPointXY(600000, 2000000) # Création d'un point avec ses coordonnées
>>> fet = QgsFeature() # Création d'une entité
>>> fet.setGeometry( QgsGeometry.fromPointXY(point)) # Géométrie de l'entité
>>> fet.setAttributes( [ " ? " ]) # Attribut de l'entité
>>> vln.startEditing() # Modification de la couche concernée
>>> prn.addFeatures( [ fet ] ) # Ajout de l'entité à la couche
>>> vln.commitChanges() # Sauvegarde de la modification
```

3) AUGMENTER LES POTENTIALITES DE QGIS AVEC D'AUTRES BIBLIOTHEQUES PYTHON

A partir du moment où l'on possède quelques bases en Python et en PyQgis, on peut alors entrevoir d'augmenter les potentialités de Qgis. En effet, Python dispose de nombreuses bibliothèques qui mises en commun avec QGis viennent offrir des potentialités supplémentaires. L'import des bibliothèques est potentiellement très simple.

```
>>> import networkx # Importation de la bibliothèque networkx
```

Après il est possible d'appeler les fonctions de cette bibliothèque très simplement, en reprenant le nom de cette bibliothèque.

```
>>> G = networkx.Graph() # Création d'un graphe (réseau)
```

Utiliser le nom entier de la bibliothèque pour chaque fonction peut vite se révéler fastidieux, on peut alors donner à la bibliothèque un nom plus court.

```
>>> import networkx as nx
>>> Gdir = nx.DiGraph()
```

Ensuite, il faut savoir utiliser la bibliothèque en question. Ici Networkx permet de faire de l'analyse de réseaux. On peut ainsi à l'aide de cette bibliothèque calculer assez simplement l'importance des éléments constituant ce réseau. Pour cela, il faut un tableau (une liste) contenant l'ensemble des arcs de ce graphe. Un arc possède un point de départ, un point d'arrivée et éventuellement des poids, comme par exemple une distance. Récupérons ces données à l'aide de nos connaissances PyQgis et aux données chargées.

```
>>> couche_arc = QgsProject.instance().mapLayersByName('Autoroutes')[0]
>>> provider2 = couche_arc.dataProvider() # Interaction avec la couche arc
>>> feat = QgsFeature() # Permet de manipuler les entités des couches
>>> fit = provider2.getFeatures() # Récupère les entités
>>> table_arc = [] # Variable pour stocker les noms
>>> while fit.nextFeature(feat):
            dist = float(feat.attributes()[0])
            dep = int(feat.attributes()[2])
            arr = int(feat.attributes()[3])
            table_arc = table_arc + [[dep, arr, dist]]
```

Utilisons ensuite des fonctions networkx intéressantes.

```
>>> G.add_weighted_edges_from(table_arc)
>>> deg = G.degree()
>>> bet = nx.betweenness_centrality(G,weight='weight')
```

Enfin, sauvegardons intelligemment les résultats.

```
>>> nbcol = provider.fields().count()
>>> provider.addAttributes( [QgsField("Between", QVariant.Double)])
>>> feat = QgsFeature()
>>> fit = provider.getFeatures()
>>> i = 0
>>> couche_noeud.startEditing() # Modification de la couche concernée
>>> while fit.nextFeature(feat):
            id_noeud = int(feat.attributes()[0])
            couche_noeud.changeAttributeValue(i, nbcol, bet[id_noeud])
            i= i +1
>>> couche_noeud.commitChanges() # Sauvegarde des modifications
```

Le tour est joué. Il est maintenant possible de facilement cartographier les résultats grâce à QGIS.

4) CREER DES PETITS PROGRAMMES POUR DES UTILISATEURS EXTERIEURS

Pour commercer à créer des outils, il convient de pouvoir dialoguer avec l'utilisateur. En effet, pour que les codes développés jusqu'à présent soient utilisables par d'autres personnes en différentes circonstances, il faut que l'utilisateur puisse indiquer quelles couches et quels champs utiliser, quels indicateurs calculer… Pour cela, Python propose par exemple la fonction « input ». Néanmoins, il conviendra d'utiliser des bibliothèques qui permettent de développer des interfaces personnalisées et d'avoir recours à des boites de dialogue prédéfinies. Une des plus courantes en Python est « PyQt ».

```
>>> nom = QInputDialog.getText(None,"N°","Entrez votre nom :")
>>> QMessageBox.information(None, "Information", "Hello " + nom[0])
>>> val = QInputDialog.getInt(None,"N°","Entrez un entier compris entre 0 et 57 ")
>>> print(table[val[0]])
>>> nom_champ = provider.fields().names()
>>> n = QInputDialog.getItem(None, "Champ", "Choisissez le champ Nom", nom_champ)
>>> nameid = provider.fieldNameIndex(n[0])
>>> fit = provider.getFeatures()
>>> while fit.nextFeature(feat):
            print(feat.attributes()[nameid])
```

En utilisant PyQT, on peut aussi créer ses propres interfaces, mais on ne va pas détailler cela ici. En revanche, maintenant que l'on est en mesure d'interagir avec un utilisateur, il est possible de créer des petits programmes pertinents. Ainsi, en reprenant des éléments codés plus hauts, on peut créer un programme qui récupère le nom du champ contenant les identifiants des nœuds. Pour les arcs, ce programme récupère aussi les noms des champs correspondant aux nœuds de départ, aux nœuds d'arrivée et aux distances. On part alors du principe que l'utilisateur a chargé une couche nœud et une couche arc dans QGIS, la couche nœud étant placée au-dessus de la couche arc. Ce code permettra de calculer la centralité des nœuds et de stocker le résultat dans un nouveau champ. Pour écrire et sauvegarder ce code, QGIS propose un petit éditeur. Enregistrez le code sous le nom voulu et exécutez-le.

```
import networkx as nx
canvas = qgis.utils.iface.mapCanvas()
allLayers = canvas.layers()
couche_noeud = allLayers[0]
couche_arc = allLayers[1]
provider = couche_noeud.dataProvider()
provider2 = couche_arc.dataProvider()
nom_champ_n = provider.fields().names()
nom_champ_a = provider2.fields().names()
n = QInputDialog.getItem(None, "Couche Noeud", "Champ Identifant", nom_champ_n)
a = QInputDialog.getItem(None, "Couche Arc", "Champ Noeud Départ", nom_champ_a)
a2 = QInputDialog.getItem(None, "Couche Arc", "Champ Noeud Arrivée", nom_champ_a)
a3 = QInputDialog.getItem(None, "Couche Arc", "Champ Distance", nom_champ_a)
nid = provider.fieldNameIndex(n[0])
aid = provider2.fieldNameIndex(a[0])
aid2 = provider2.fieldNameIndex(a2[0])
aid3= provider2.fieldNameIndex(a3[0])
feat = QgsFeature()
fit = provider2.getFeatures()
table_arc = []
while fit.nextFeature(feat):
    dep = int(feat.attributes()[aid])
    arr = int(feat.attributes()[aid2])
    dist = float(feat.attributes()[aid3])
    table_arc = table_arc + [[dep, arr, dist]]
G = nx.Graph()
G.add_weighted_edges_from(table_arc)
bet = nx.betweenness_centrality(G,weight='weight')
nbcol = provider.fields().count()
provider.addAttributes( [QgsField("Between", QVariant.Double)])
fit = provider.getFeatures()
i = 0
couche_noeud.startEditing() # Modification de la couche concernée
while fit.nextFeature(feat):
    id_noeud = int(feat.attributes()[nid])
    couche_noeud.changeAttributeValue(i, nbcol, bet[id_noeud])
    i= i +1
couche_noeud.commitChanges()
QMessageBox.information(None, "Information", "Calcul fini")
```

Ce petit éditeur est bien pratique. Néanmoins, lorsque l'on commence à écrire des codes plus complexes, il convient d'utiliser de meilleurs éditeurs : comme Notepad++ et surtout Atom. Depuis la ligne de commande vous pourrez utiliser les fonctions « exec » et « open » pour exécuter votre code.

5) FAIRE SA PREMIERE EXTENSION

La dernière étape consiste éventuellement à transformer ses petits programmes en de belles extensions. En effet, pour utiliser le programme développé il faut ouvrir la console python, puis ouvrir l'éditeur. Dans cet

éditeur il faut alors ouvrir le programme, puis l'exécuter. Ces étapes sont peu digestes pour un non initié. Par conséquent, donner accès à votre programme directement depuis l'interface QGis semble plus pertinent. Pour cela, il faut un fichier __init__.py, le fichier du code et un fichier avec les metadonnées (metadata.txt).

Le fichier __init__.py peut simplement être composé d'une fonction qui appelle le code développé (test.py) et plus précisément une classe de ce fichier qui contiendra le code.

```
def classFactory(iface):
  from .test import Test
  return Test(iface)
```

Le fichier de metadata.txt va décrire les principales caractéristiques de l'extension. Il peut se composer comme suit :

```
[general]
name=TEST
description=Extension
category=Vector
version=1.0
qgisMinimumVersion=3.0

author=Serge Lhomme
email=serge.lhomme@u-pec.fr

class_name=Test
```

Enfin, il va falloir légèrement modifier le code créé en plaçant votre code dans une fonction run, elle-même placée dans la classe référencée dans le fichier __init__.py. Seule la première ligne devra être modifiée de « canvas = qgis.utils.iface.mapCanvas() » en « canvas = self.iface.mapCanvas() »

```
class Test:

      def run(self):
            Le code
```

Dans cette classe, il convient d'avoir une fonction d'initialisation qui définit l'interface et une fonction initGui

```
      def __init__(self, iface):
            self.iface = iface

      def initGui(self):
            self.action = QAction("Test", self.iface.mainWindow())
            self.action.triggered.connect(self.run)
            self.iface.addPluginToMenu("Test", self.action)
```

Enfin, il faudra en amont de ce code placer les bibliothèques à appeler, notamment celle de PyQGIS et bien entendu NetworkX.

```
from qgis.core import *
from PyQt5.QtCore import *
from PyQt5.QtGui import *
from PyQt5.QtWidgets import *
from qgis.PyQt.QtWidgets import QAction
import network as nx
```

Placez vos trois fichiers dans un dossier zippé reprenant le nom de votre extension (ici « Test »), puis dans QGIS allez dans « Extension -> Installer/Gérer les extensions », vous pourrez alors installer votre extension depuis un fichier zippé.