

Utiliser Python avec des données géographiques: une introduction

Christian Kaiser

1. Aperçu

Python est un langage de programmation puissant et facile à utiliser et apprendre. Python est très utilisé comme **langage de scriptage** dans beaucoup de logiciels, notamment dans le domaine SIG (p.ex. QuantumGIS ou ArcGIS). Il permet de développer rapidement des petits scripts comme des applications complètes. Python possède également une riche bibliothèque d'extensions permettant d'accéder à des nombreuses librairies spécialisées. En plus, il est facile d'ajouter des nouvelles fonctionnalités grâce à des modules.

Python est un **langage interprété**, ce qui rend possible le développement rapide sans compilation. Des langages comme C ou C++ sont des langages compilés, c'est-à-dire le code doit d'abord être traduit en code machine avant d'être exécuté. Dans le cas d'un langage interprété, cette étape de compilation est faite pendant l'exécution du logiciel. Ceci rend le code un peu plus lent, mais accélère et facilite considérablement le développement.

Python est un langage open-source; l'interpréteur Python ainsi que les librairies standard peuvent être téléchargés librement depuis le site Web Python: www.python.org.

Python est un langage multi-plateforme. L'interpréteur et le code écrit tournent en principe sur toutes les plateformes majeures.

1.1. Documentation

Python possède une riche documentation accessible sur Internet, sous www.python.org (en anglais). La page <http://wiki.python.org/moin/FrenchLanguage> donne un aperçu de la documentation Python en français (l'aperçu lui-même est en anglais...).

Nous allons utiliser beaucoup la documentation, car il est impossible de mémoriser toutes les fonctions et astuces.

Il est possible de télécharger l'archive complet, si vous ne possédez pas tout le temps une connexion Internet.

Il y a aussi de nombreux tutoriels pour apprendre Python, par exemple:
<http://dept-info.labri.fr/ENSEIGNEMENT/projetprog1/TutorielPython.pdf>

Ce tutoriel est plus complet que ce cours, mais présente évidemment uniquement une introduction à Python, sans focaliser sur les données spatiales.

2. Un bref exemple pour démarrer

Python peut être utilisé de deux manières différentes: en **mode interactif** et en **mode script** où un fichier enregistré sur le disque est exécuté. Nous commençons d'abord par le mode interactif. Sur un système Unix, l'interpréteur Python peut généralement être lancé depuis le Terminal en tapant

```
python
```

Parfois, un interpréteur un peu plus sophistiqué est installé; il s'appelle ipython (pour interactive python). Pour le lancer, simplement taper

```
ipython
```

Python démarre alors avec quelques informations sur la version, et puis attend notre code:

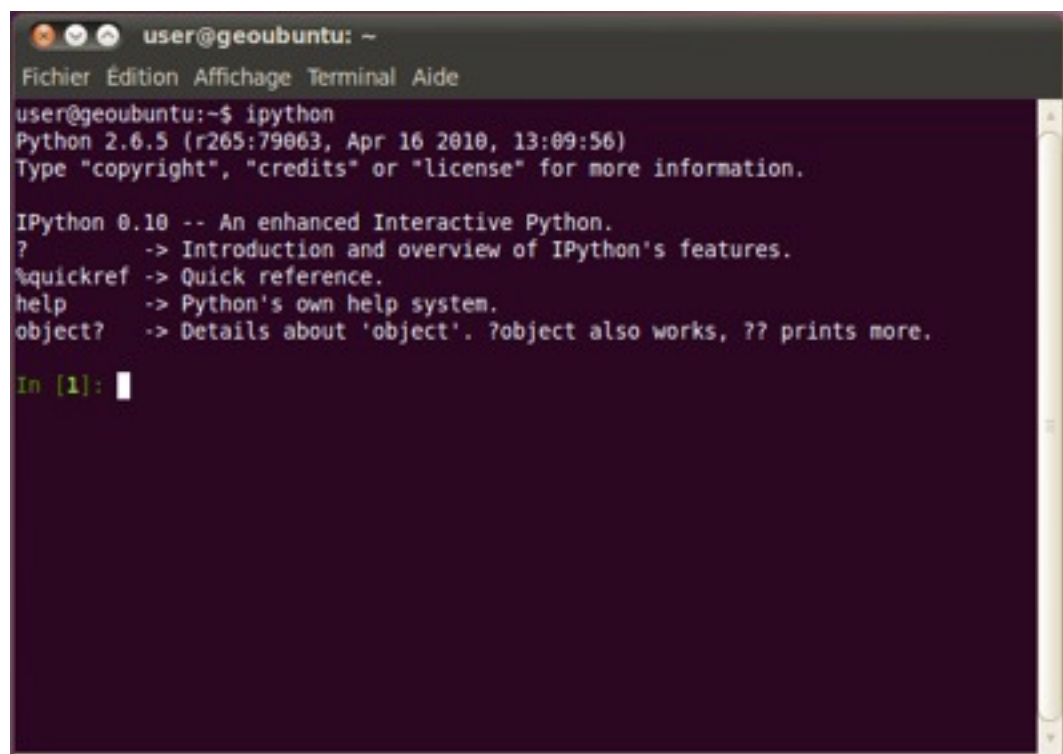


Figure 1. La console Python.

Essayons le code suivant:

```
a = "Paul"
if a == "Paul":
    print "Hello Paul"
else:
    print "Hello inconnu"
```

Ceci montre déjà quelques caractéristiques typiques de Python:

- a = "Paul" définit une nouvelle **variable** contenant un «string»
- Il n'y a **pas de caractère de fin de ligne** comme dans C ou Java (mais ça marche tout de même avec un point-virgule...)
- Dans une condition if ... else ..., il n'y a **pas d'accolades** {...}, mais un double-point
- **L'indentation est importante:** elle permet de regrouper les instructions

Pour quitter la console Python, vous pouvez taper Ctrl-D.

2.1. Commentez votre code!

Il est très important d'écrire des commentaires qui font du sens. Ils sont là pour que vous compreniez votre code plus tard, et pour que votre collègue le comprenne aussi.

En Python, tout ce qui suit un dièse (#) est ignoré jusqu'à la fin de la ligne. Essayez:

```
# Moi, je vous dis que ce programme ne fait presque rien
print "Bonjour!"    # Cette ligne affiche "Bonjour!"
# Vous pouvez utiliser le dièse aussi pour "désactiver" une ligne:
# print "Moi je ne fais rien"
```

En plus, Python connaît un commentaire spécial, le **docstring**, qui sert à la documentation. Il est placé tout au début du fichier, d'une fonction, ou d'une classe, entre des triple-apostrophes. Il est typiquement utilisé pour informer ce le code est censé faire, et comment il faut l'utiliser. Un exemple (on verra les fonctions plus loin):

```
def au_carre(x):
    """Calcul le carré de la valeur d'entrée.
    Prend qu'un seul argument numérique."""
    return x**2
```

2.2. Les variables

Une variable est un nom pour un morceau de données stocké dans la mémoire vive de l'ordinateur. Un nom de variable peut être composé de caractères (sans accents), de chiffres et «underscores» (soulignés). Le nom doit commencer par un caractère. Python distingue entre minuscules et majuscules. Essayez:

```
adresse = "Avenue de la Gare, Lausanne"
code_postale = 1003
distance_a_la_gare_km = 0.15
print adresse, " se trouve à ", distance_a_la_gare_km, "km"
```

Par convention, on remplace les espaces par des underscores, et tous les noms de variables sont en minuscules. On utilisera les majuscules plus tard pour les noms de classes.

Il est évidemment possible d'utiliser les variables pour des opérations et calculs:

```
distance_a_la_gare_metres = distance_a_la_gare_km * 1000
```

```
print distance_a_la_gare_metres
```

Il est possible d'attribuer la même valeur à plusieurs variables:

```
a = b = c = 1
print a, b, c
```

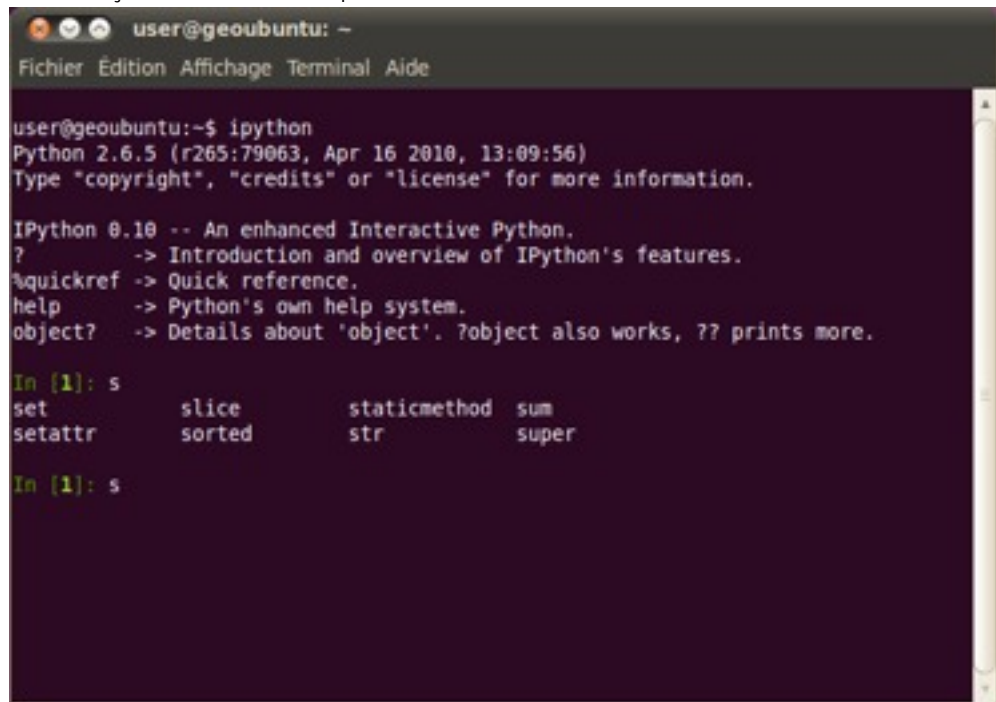
Pour supprimer une variable, on peut taper:

```
del a          # ça efface la variable a
```

2.3. Quelques astuces pour la console interactive

La console interactive iPython offre deux mécanismes qui peuvent considérablement simplifier la vie du programmeur:

- Si commencez à taper un nom d'une variable ou d'une fonction, et vous appuyez sur la touche **Tab**, iPython va automatiquement lister toutes les variables et fonctions commençant avec le texte tapé:



```
user@geoubuntu: ~
Fichier Edition Affichage Terminal Aide

user@geoubuntu:~$ ipython
Python 2.6.5 (r265:79063, Apr 16 2010, 13:09:56)
Type "copyright", "credits" or "license" for more information.

IPython 0.10 -- An enhanced Interactive Python.
?          -> Introduction and overview of IPython's features.
%quickref  -> Quick reference.
help       -> Python's own help system.
object?    -> Details about 'object'. ?object also works, ?? prints more.

In [1]: s
set      slice      staticmethod  sum
setattr  sorted      str           super

In [1]: s
```

- Si vous tapez `?nom_fonction`, iPython affichera la documentation pour `nom_fonction`. Essayez p.ex. `?str`.

3. Types de données Python

Python connaît un certain nombre de types de données, p.ex. pour les chaînes de caractères (type: **string**), les nombres entiers (type: **integer**), les nombre à virgule flottante (type: **float**), ou les arrays (type: **list**). Pour la plupart du temps, ces types fonctionnent de **manière implicite**, c'est-à-dire il ne faut pas les déclarer, comme p.ex. dans C. Par contre, on ne peut pas utiliser un integer à la place d'un string, ou un string à la place d'un float, etc. Il y a des fonctions de **conversion** pour faire cela.

3.1. Les chaînes de caractères

Les chaînes de caractères (string) peuvent être déclarées très simplement avec des guillemets, apostrophes, ou triple guillemets:

```
str1 = "Une chaîne de caractères"
str2 = 'C\'est aussi une chaîne de caractères'
str3 = """Et ça aussi, c'est une chaîne de caractères.
C'est une chaîne de caractères sur plusieurs lignes!"""
```

Remarquez que dans la deuxième ligne, nous avons «protégé» l'apostrophe à l'intérieur du texte par un «backslash». C'est pour éviter que le string se termine à cet endroit déjà. On peut faire de même pour un string avec guillemets:

```
str4 = "Dis \"bonjour\""
```

ou plus simplement:

```
str5 = 'Dis "bonjour"'
```

L'utilisation de guillemets ou apostrophes est strictement équivalent (contrairement à C ou PHP p.ex.).

Nous pouvons demander la longueur d'un string avec `len`:

```
print len(str5)
```

Pour extraire uniquement une sélection d'une chaîne, on peut utiliser des indices. P.ex. les 4 premiers caractères de `str1` peuvent être obtenu comme suit:

```
str1[0:4]          # Les indices en Python commencent à 0
```

ce qui est équivalent à

```
str1[:4]          # Le 0 initial peut être omis
```

Si on veut omettre les 3 derniers caractères, il suffit d'utiliser des indices négatifs:

```
str1[:-3]
```

Et si on veut juste les 3 derniers caractères:

```
str1[-3:]
```

Évidemment, on peut aussi juste extraire un seul caractère:

```
str1[4]           # Le 5ème caractère
```

Quelques autres fonctions utiles

Beaucoup de fonctions permettent de manipuler les chaînes de caractères. Voici un bref aperçu sur les fonctions les plus utiles. Les fonctions suivent le string (ou le nom de la variable), séparées par un point.

Pour **enlever des espaces blancs** (espace, tabulateur, retour de lignes etc.) au début et à la fin d'une chaîne:

```
" C'est très pratique pour nettoyer \n".strip()
```

(le «\n» est un signe de «nouvelle ligne»).

Pour enlever un ou plusieurs caractères spécifiques:

```
"--- Enlever les traits-d'union et espaces ---".strip('- ')
```

Pour **chercher un texte** à l'intérieur d'une chaîne:

```
"auto" in "C'est une belle automobile"    # Retourne True or False
"C'est une belle automobile".find("auto")
    # Retourne l'endroit dans la chaîne ou "auto" a été trouvé,
    # ou -1 si la chaîne n'a pas été trouvée
```

Et pour **remplacer un texte** par un autre:

```
"C'est une belle voiture".replace('voiture', 'moto')
```

Pour convertir une chaîne en liste en utilisant un caractère de séparation:

```
"Séparer les mots".split()
"Séparer les phrases. Par des points.".split('.')
```

Nous allons discuter les listes plus en détail un peu plus loin.

Une documentation plus exhaustive des méthodes s'appliquant à des string peut être trouvée à l'URL: <http://docs.python.org/library/stdtypes.html#string-methods> ou dans la machine virtuelle: <http://localhost/pydoc/library/stdtypes.html#string-methods>

Les chaînes unicode

Dans Python, il y a également des chaînes de caractères un peu différentes: les chaînes unicode. Unicode permet d'utiliser les caractères de tous les alphabets. Les chaînes unicodes sont pour la plupart du temps transparentes dans Python. Pour créer un string unicode, il suffit de mettre un «u» devant la chaîne:

```
u"C'est une chaîne de caractères Unicode"
```

Il est possible de convertir une chaîne unicode vers une chaîne 8-bits d'un encodage donné, et vice-versa:

```
u"àèè".encode('utf-8')
unicode('àèè', 'iso-8859-1')
```

3.2. Un peu de maths...

Vous pouvez utiliser Python comme calculatrice, avec tous les opérateurs classiques:
+ - * / ** (puissance), % (reste d'une division):

```
1 + 2
3 * (2 + 4)
2**16
8 / 2
8 % 3
```

Par contre, si vous tapez:

```
5 / 2
```

vous aurez une petite surprise: le résultat est 2 et non 2.5...

Python donne automatiquement à chaque variable un **type**: un nombre entier (**integer**), un nombre à virgule flottante (**float**), une chaîne de caractère (**string**), etc.

La division d'un nombre entier par un nombre entier est en conséquence toujours un nombre entier. On peut créer un float à la place en tapant:

```
5.0 / 2
```

ou forcer une conversion:

```
float(5) / 2
```

Notez que la conversion d'un seul des deux nombre est suffisant.

Les fonctions float(), int() et str() permettent de changer de type de variable. Attention, la conversion en integer supprime simplement les chiffres décimaux, penser à arrondir avant:

```
int( round( 155.0 / 53 ) ) # arrondir à l'entier le plus proche
int( floor( 155.0 / 53 ) )
```

3.3. Faisons une liste...

Python connaît un grand nombre de types pour des collections de données. Le type «list» en est le plus flexible. Une liste est similaire à un array, mais peut contenir des données de types différents. Une liste s'écrit entre crochets, les différents éléments sont séparés par des virgules:

```
lst = [ "pomme", "banane", 1, 5.4 ]
```

L'accès aux éléments de la liste se fait de manière similaire aux éléments d'une chaîne de caractère:

```
lst[2]          # Le 3ème élément de la liste
lst[:2]         # Les 2 premiers éléments
```

Une liste peut également contenir une autre liste:

```
nested_list = [ [1,2,3], [4,5,6], [7,8,9] ]
```


Pour accéder au 2ème élément de la première «sous-liste», on peut simplement écrire:

```
nested_list[0][1]
```

Les listes imbriquées peuvent être très pratique par exemple pour faire une liste de coordonnées...

Il est aussi possible de construire une liste contenant des numéros continus:

```
range(4)          # [0,1,2,3]
range(1,4)        # [1,2,3]; la liste s'arrête à 4-1
range(1,10,2)     # [1,3,5,7,9]; la liste s'arrête à 10-1,
                  # et fait des pas de 2
```

Les éléments d'une liste peuvent être modifiés (contrairement aux chaînes de caractères):

```
a = [1, 4, 3]
a[1] = 2
```

Il est aussi possible de modifier plusieurs éléments à la fois:

```
a[0:2] = [5,4]
```

Il est possible de supprimer un élément de la liste par:

```
del a[1]
```

Pour supprimer plusieurs éléments:

```
a[0:2] = []
```

Pour insérer un élément (ou plusieurs) au début de la liste:

```
a[:0] = [5]
```

Et pour ajouter un élément à la fin, ou plusieurs...

```
a.append(6)
a += [7,8,9]    # équivalent à: a = a + [7,8,9]
```

Notez que `a.append([7,8,9])` ajoute la liste comme dernier élément de la liste...

Pour tester si une valeur donnée se trouve dans une liste, ou non:

```
8 in a
9 not in a
```

Il est aussi possible de concaténer deux listes:

```
a = [1,2,3]
b = [4,5,6]
ab = a + b
```

Il est possible de trier une liste (sort) ou l'inverser (reverse):

```
a.sort()      # ne retourne rien, mais modifie la liste directement
a.reverse()   # modifie aussi directement la liste
```

3.4. Ou un tuple...?

Un tuple est similaire à une liste, sauf que les éléments ne peuvent pas être modifiés. Un tuple peut être créé simplement en séparant plusieurs valeurs par des virgules, et éventuellement en entourant avec des parenthèses:

```
b = 123, 456, 'abc'
c = (543, 454.4, 'def')
```

Un tuple peut contenir un autre tuple, ou une liste:

```
b = 123, ('a', 'b'), [4,5,6]
```

Pour créer un tuple vide, ou avec un élément, c'est un peu particulier:

```
b = ()          # tuple vide
b = 1,          # notez la virgule, elle est nécessaire
b = (1,)        # c'est la même chose
```

L'accès aux éléments des tuples se fait de la même manière que pour les listes:

```
c[0:2]
```

Il est possible de convertir un tuple en liste et vice-versa:

```
t = 1,2,3
list(t)
l = [4,5,6]
tuple(l)
```

3.5. Les ensembles...

Un ensemble (set) est une collection non-ordonnée d'éléments similaire à un tuple, sauf qu'un ensemble contient uniquement des éléments uniques:

```
fruits = ['pomme', 'poire', 'orange', 'pomme', 'banane']
set(fruits)
```

Il est aussi possible d'appliquer quelques opérateurs sur les ensembles.

```
a = set('python')          # contient les caractères uniques
b = set('apprentissage')

a - b          # éléments dans a mais pas dans b
a | b          # éléments dans a ou b
a & b          # éléments communs de a et b
a ^ b          # éléments dans a ou b, mais pas dans les deux
```

3.6. Et un dictionnaire...

Le dictionnaire, parfois aussi connu sous le nom de tableau associatif ou «hash table», est une collection d'éléments où chaque élément est identifié par une clé unique. Cette clé peut être tout élément non-modifiable, c'est-à-dire un string, un chiffre, ou un tuple (s'il ne

contient pas d'élément modifiable, tel qu'une liste). **Un dictionnaire est un ensemble non-ordonné de couples clé:valeur.** Pour créer un dictionnaire, on utilise des accolades:

```
d = { 'a': 1, 'b': 'asdf', 3.4: [1,2,3] }
```

L'identification d'un élément se fait à travers la clé:

```
d[3.4]  
d['a']
```

Il est possible de modifier les valeur d'un dictionnaire ou d'ajouter un nouvel élément:

```
d['nouveau'] = 'arbre'  
d['a'] = 2
```

Pour supprimer un couple clé:valeur, on peut utiliser:

```
del d['a']
```

Pour créer un dictionnaire vide, il suffit d'écrire:

```
d2 = {}
```

Extraire une valeur pour une clé non existante est une erreur. Par contre, la méthode `get` permet d'extraire une valeur pour une clé, et dans le cas où la clé n'existe pas, de choisir une valeur par défaut:

```
d.get('n_existe_pas', 0) # retourne 0
```

La méthode `keys` retourne une liste contenant toutes les clés, et `values` une liste de toutes les valeurs:

```
d.keys()  
d.values()
```

Les dictionnaires n'ont pas d'ordre défini. Donc pour extraire une liste de valeurs par ordre alphabétique, il faut demander la liste des clés, et puis appliquer la méthode `sort` sur cette liste. Le code suivant permet de le faire (nous allons revenir sur les boucles plus tard, donc si vous ne comprenez pas encore ce code, ce n'est pas grave):

```
d = {'Ford': 25000, 'Mercedes': 52000, 'Audi': 38000}  
d_cles = d.keys()  
d_cles.sort()  
valeurs_triees_par_cle = [ d[k] for k in d_cles ]
```

ou sous forme de tuples clé:valeur:

```
[ (k, d[k]) for k in d_cles ]
```

Et finalement, pour savoir si une clé existe:

```
d.has_key('cle')
```

Évidemment, il y a une documentation exhaustive pour les dictionnaires aussi:

<http://docs.python.org/library/stdtypes.html#mapping-types-dict> ou
<http://localhost/pydoc/library/stdtypes.html#mapping-types-dict>

3.7. Revenons sur les chaînes de caractères

Dans Python, il est possible de construire des strings en utilisant plusieurs types de données, potentiellement en formatant les nombres. Ce type de formatage est connu de plusieurs langages, p.ex. `printf` en C. Le code suivant

```
print "J'ai %i pommes" % 3
```

est converti en «J'ai 3 pommes».

Le signe % signifie qu'il s'agit d'un formatage, le caractère suivant définit quel type de données doit être formaté. Les caractères de formatage les plus utilisés sont:

i ou d	nombre entier positif ou négatif
f	nombre à virgule flottante
s	string

Pour formater plus d'une valeur, il suffit d'écrire plusieurs signe de formatage, et de fournir un tuple ou une liste avec le bon nombre d'éléments.

Il est possible de formater les nombres en indiquant par exemple une longueur minimale:

```
"Il est précisément %i:%02i:%02i" % (9, 3, 1)
```

Dans cet exemple, les minutes et secondes sont formatées avec deux chiffres au minimum; le 0 indique de remplacer des chiffres manquants avec des 0.

Le même s'applique à des nombre à virgule flottante:

```
print "Cet article coûte CHF %07.2f" % 45.1
```

ce qui est converti en «Cet article coûte CHF 0045.10» (7.2 signifie une longueur totale de 7 caractères (y inclut le point décimal), et 2 chiffres après le point décimal).

Il est aussi possible d'utiliser des dictionnaires pour formater une chaîne:

```
d = {'article': 'sac', 'prix': 58.9}
print "Article: %(article)s au prix de CHF %(prix).2f" % d
```

4. Python en mode batch

Le mode interactif de Python est très pratique pour rapidement tester une ligne de code, mais il ne se prête pas pour des scripts plus longs. Surtout, généralement, on aimerait probablement réutiliser le code déjà écrit. En plus, certaines consoles Python interactives ne sont pas très sophistiquées. Dans ce cas, nous pouvons enregistrer l'ensemble des commandes Python dans un fichier avec extension **.py**, et l'exécuter avec la commande

```
python fichier.py
```

directement depuis le Terminal.

Si on veut exécuter un fichier Python, et entrer dans la console interactive après la fin du script, on peut taper

```
python -i fichier.py
```

Sous Windows, un fichier Python peut généralement être lancé avec un simple double-clic, ou de manière similaire depuis la Console (l'emplacement de Python peut être différent suivant l'installation):

```
C:\python26\python.exe fichier.py
```

Sous Unix, il est possible de rendre exécutable un fichier Python comme un outil de ligne de commande. Ceci est possible en plaçant la ligne suivante (appelée «shebang») au début du fichier (vraiment tout au début):

```
#!/usr/bin/env python
```

Il faut encore rendre exécutable le fichier avec la commande suivante:

```
chmod +x fichier.py
```

Par la suite, le code peut être lancé comme un programme normal:

```
./fichier.py      # si fichier.py est dans le dossier courant  
fichier.py       # si le dossier de fichier.py est dans le PATH
```

Dans ce cas, il n'est plus nécessaire d'utiliser l'extension **.py**. Si vous renommez votre script **fichier.py** en **mon_programme** et vous le placez dans un dossier contenu dans la variable **PATH**, vous pouvez exécuter le programme simplement avec

```
mon_programme
```

(le contenu de la variable d'environnement **PATH** peut être visualisé avec la commande «echo \$PATH» directement dans le Terminal (pas dans la console Python).

Encodage d'un fichier Python

Par défaut, un fichier Python contient uniquement des caractères ASCII, sans accents ou autres caractères spéciaux. Si nous voulons écrire des caractères non-ASCII, il faut déclarer l'encodage du fichier sur la deuxième ligne du fichier (après le shebang, mais avant le docstring):

```
# -*- coding: iso-8859-1 -*-
```

Alternativement, l'encodage **utf-8** est également très utilisé.

Passer des arguments à un script Python

Il est possible d'appeler un script Python avec des arguments, par exemple:

```
python fichier.py Paul
```

Vous pouvez récupérer les arguments dans votre script Python à l'aide du module **sys** (plus sur les modules plus tard):

```
import sys          # Charger le module sys
# sys.argv est une liste contenant tous les arguments
# Le premier argument (sys.argv[0]) correspond au nom du programme
nombre_arguments = len(sys.argv)
print "Bonjour %s" % sys.argv[1]    # Affiche 'Bonjour Paul'
```

5. Les exceptions: gestion des erreurs

Si une erreur survient dans une ligne de code, Python imprimera un message d'erreur avec quelques informations sur la nature de l'erreur. Une erreur est aussi appelée «exception» (même si les erreurs sont parfois plutôt la règle...). Lors d'une exception, l'exécution du programme est stoppée.

Il est possible de stopper nous-même l'exécution d'un code, en appuyant sur Ctrl-C, ou parfois la touche DEL (dépendant de l'interpréteur).

Parfois, il est aussi utile «d'intercepter» une exception, typiquement si on veut tester si une opération est possible ou non. Ça peut se faire comme suit:

```
d = {'a': 1}
try:
    print d['b']      # génère une exception
except:
    print "la clé b n'existe pas"
```

6. Structures de contrôle: boucles et conditions

6.1. Les conditions

Les conditions marchent de la manière suivante dans Python:

```
# raw_input demande une entrée par l'utilisateur
a = int(raw_input('Entrer un nombre entier: '))
if a == 0:
    print "Vous avez saisi zéro"
elif a > 10:
    print "Vous avez saisi une grande valeur"
elif a <= -1:
    print "Le chiffre saisi est négatif"
else:
    print "Vous avez saisi une petite valeur"
```

Seulement la partie `if` est obligatoire, `elif` et `else` sont facultatifs. Il n'y a pas besoin de mettre des parenthèses ou des accolades dans Python, par contre il y a un double-point à la fin de la condition. Le code à exécuter de manière conditionnelle est indenté au bon niveau (typiquement 4 espaces, ceci est généralement géré automatiquement par votre éditeur de texte).

Vous pouvez évidemment aussi faire des conditions plus sophistiquées:

```
a = int(raw_input('Entrer un nombre entier: '))
b = int(raw_input('Entrer un autre nombre entier: '))
if a > 0 and b > 0:
    print "Les deux nombres sont positifs"
elif a > 0 or b > 0:
    print "Au moins un des deux nombre est positif"
```

6.2. Les boucles

En Python, il y a deux types de boucles, les boucles `for` et `while`. La boucle `while` est exécuté tant qu'une condition est remplie:

```
# une boucle infinie
while True:          # "True" et "False" sont des mots du langage
    pass             # pass est une commande qui ne fait rien
```

Alternativement, la boucle `for` permet de faire une itération sur tous les éléments d'une collection (ce qui est différent de la plupart des langages, comme C ou Java p.ex.):

```
for i in [1,2,3,4]:
    print i
```

La fonction `range` déjà vu dans la section sur les listes peut être pratique dans ce cas:


```
for i in range(0, 10):  
    print i
```

Il est aussi possible de faire une itération sur un dictionnaire:

```
d = {'a': 1, 'b': 2, 'c': 3}  
for k in d:  
    print 'cle: ', k, ' / valeur: ', d[k]
```

Une chaîne de caractère est aussi traitée comme une collection:

```
s = 'un texte'  
for c in s:  
    print c
```

Par contre, il n'est pas prudent de modifier une collection sur laquelle on est en train de parcourir. Dans ce cas, il faut d'abord faire une copie de la collection, ce qui est simple dans le cas d'une liste:

```
a = [1,2,3]  
for i in a[:]:  
    a[:0] = [i]
```

L'instruction `a[:]` renvoie en fait une copie de la liste complète...

L'instruction `break` permet de sortir d'une boucle. Lors d'une boucle dans une boucle, on sort de la boucle englobante.

L'instruction `continue` passe à la prochaine itération.

6.3. List comprehensions

Les *list comprehensions* offrent un mécanisme de boucles très compact, à condition que le code de la boucle soit simple. Par exemple pour multiplier chaque élément d'une liste par 2, on peut écrire:

```
b = [ i*2 for i in a ]
```

ce qui est équivalent à:

```
b = []  
for i in a:  
    b.append(i*2)
```

6.4. Itération sur deux listes simultanément

La fonction `zip` permet de faire une boucle sur deux listes simultanément:

```
vec1 = [1,2,3,4,5]  
vec2 = [2,4,6,8,10]  
vec12 = [(x,y) for x,y in zip(vec1, vec2)]
```

Si les listes n'ont pas la même longueur, `zip` arrête après la fin de la liste plus courte.

7. Lecture et écriture de fichiers

La fonction `open` ouvre un fichier et renvoie un objet permettant d'effectuer des opérations sur le fichier ouvert. Il faut donner un nom du fichier (en fait, le chemin d'accès), et le mode d'ouverture. Le mode par défaut est 'r' (lecture seul), il peut être 'w' pour l'écriture seul, 'a' pour ajouter des données à la fin du fichier, et 'r+' qui ouvre le fichier en lecture et écriture.

```
f = open('mon_fichier.txt')
```

Si le fichier `mon_fichier.txt` n'existe pas, il est créé en mode 'w'. Ouvrir un fichier non-existant en mode 'r' est une erreur.

Pour lire le contenu du fichier, il y a plusieurs possibilités:

```
s = f.read()          # Lit tout le fichier dans s
s = f.readline()      # Lit la ligne suivante dans s
s = f.readlines()     # Lit tout le fichier dans une liste,
                      # avec chaque ligne comme élément séparé
```

Si `readline` atteint la fin du fichier, une chaîne vide '' est retournée. Dans le cas d'une ligne vide, `s` contient toujours le signe de fin de ligne ('\n' sur Unix, '\r\n' sur Windows).

Il est aussi possible de lire le fichier ligne par ligne, simplement avec une boucle:

```
for s in f:
    print s
```

Une fois terminé avec le fichier, il faut le fermer:

```
f.close()
```

L'écriture d'un fichier n'est pas compliqué non plus:

```
a = 45.6
f = open('mon_fichier.txt', 'w')
f.write('Première ligne du fichier.\n')
f.write('La valeur de a est %f\n' % a)
f.close()
```

Notez le signe '\n' de fin de ligne à la fin de la commande. En effet, il faut insérer manuellement les sauts de lignes. Attention, sous Windows, les sauts de lignes sont différents des systèmes Unix ('\n' sous Unix, '\r\n' sous Windows).

8. Fonctions

La définition d'une fonction se fait avec le mot `def`, suivi du nom de la fonction, et puis le ou les arguments dans une parenthèse, séparés par des virgules. La fin de la ligne de définition est marquée par un double point. Le code de la fonction suit après avec une indentation d'un niveau. Le retour d'une valeur se fait avec `return`.

```
def ma_fonction(argument1, argument2):  
    print argument1  
    return argument2
```

Vous pouvez aussi retourner plusieurs arguments en les séparant avec une virgule (en fait, la valeur retournée est un seul tuple...). Il est également de définir des arguments optionnels en donnant une valeur par défaut. Les arguments optionnels doivent être après les arguments obligatoires.

```
def ma_fonction(argument1, argument2='ma_valeur'):  
    print argument1  
    return argument1, argument2
```

Il est une bonne pratique de documenter vos fonctions avec le docstring:

```
def ma_fonction(argument1, argument2='ma_valeur'):  
    """Cette fonction ne fait rien.  
    Vraiment rien."""  
    return argument1, argument2
```

L'appel de la fonction se fait après tout simplement:

```
valeur_de_retour = ma_fonction('a', 'b')
```

ou si la fonction retourne deux valeurs:

```
r1, r2 = ma_fonction('a', 'b')
```

Il est également possible dans Python d'utiliser les noms des arguments comme suit:

```
ma_fonction(argument2 = 'c', argument1 = 'n')
```

Du coup, il ne faut pas forcément respecter l'ordre des arguments. Par contre, il n'est pas possible d'utiliser un argument sans nom après un argument avec nom, dans le même appel de fonction:

```
ma_fonction(argument1 = 'a', 'b')    # Ceci donne une erreur!
```

Faites attention de ne jamais utiliser le nom d'une fonction pour un nom de variable, et vice-versa!

Un nom de fonction est juste une variable comme une autre, mais contenant une donnée de type fonction... Vous pouvez tester cela avec la fonction `type()` qui retourne le type d'une variable:

```
type(ma_fonction)    # Donne <type 'function'>
```

8.1. Les fonctions lambda

Les fonctions lambda sont mentionnées ici juste pour avoir entendu parler de ce concept. Une fonction lambda est une fonction anonyme, donc sans nom, contenant uniquement une ligne de code. Une telle fonction peut être écrite de manière très compacte:

```
lambda a, b : a + b
```

où *a* et *b* sont les arguments de la fonction, et *a+b* la valeur de retour. On pourrait réécrire la ligne ci-dessus dans une fonction avec nom:

```
def somme(a, b):  
    return a + b
```

8.2. Quelques outils de programmation fonctionnelle

Python connaît trois fonctions très pratiques pour manipuler des listes à l'aide d'une fonction quelconque: *filter*, *map*, *reduce*.

La fonction *filter* permet de filtrer une liste à l'aide d'une fonction. Chaque élément de la liste est passé à la fonction un à un, en appelant la fonction séparément avec chaque élément. Si la fonction retourne *True*, l'élément est gardé, et sinon écarté:

```
def est_nombre_pair(x):  
    return x % 2 == 0  
  
a = [1,2,3,4,5]          # équivalent à a = range(1,6)  
filter(est_nombre_pair, a) # retourne [2,4]
```

La fonction *map* permet d'appliquer une fonction quelconque à tous les éléments d'une liste, et retourne la valeur de retour de la fonction dans une nouvelle liste de la même longueur:

```
map(est_nombre_pair, a)  
# retourne [False, True, False, True, False]
```

La fonction *reduce* prend deux arguments et renvoie une seule valeur de synthèse. Au début, les deux premiers éléments de la liste sont passés à la fonction *reduce*. Par la suite, le résultat du précédent *reduce* et de l'élément suivant de la liste sont passés à la fonction *reduce*. Pour finir, une seule valeur de synthèse reste. Un exemple pour additionner les chiffres dans une liste:

```
def somme(a, b):  
    return a + b  
  
reduce(somme, [1,4,5,8,10])
```

Cet exemple est équivalent au code suivant:

```
reduce(lambda a,b: a+b, [1,4,5,8,10])
```

9. Classes

Nous ne donnons pas une introduction exhaustive aux classes dans Python ici, mais nous fournissons plutôt les bases nécessaires pour travailler un peu avec ce concept.

Une classe permet de créer une nouvelle structure de données qui encapsule des données et des fonctions à la fois. Une variable à l'intérieur d'une classe est appelée **attribut**, et une fonction dans une classe s'appelle **méthode**. Une classe peut être définie de cette manière:

```
class MaClasse:
    """Une classe pour montrer comment définir des classes."""
    i = 1234
    a = 'abcd'
    def f(self):
        return 'bonjour ' + self.a
```

Il y a donc un nom pour la classe, des attributs, et des méthodes. Une classe en tant que telle ne fait encore rien, c'est une définition d'une structure de données plutôt que la structure elle-même. Par la suite, il faut créer une **instance** d'une classe, ce qui donne un **objet**...

```
obj = MaClasse()
```

`obj` est donc une instance de la classe `MaClasse`, et souvent on dira simplement qu'il s'agit d'un objet.

Une fois l'objet instancié (initialisé), on peut accéder aux valeurs des attributs et exécuter les méthodes:

```
obj.i = 4567
s = obj.f()
print obj.a
```

Dans la définition de la classe ci-dessus, nous avons défini un argument pour notre méthode appelé `self`. Chaque méthode prend comme premier argument `self`, qui est simplement une référence à soi-même. Pour accéder à un attribut d'un objet depuis une méthode, il faut donc écrire `self.a` et non pas simplement `a`.

Dans une classe, il y a plusieurs méthodes spéciales, dont nous regardons deux de plus près: `__init__` et `__del__`.

La méthode `__init__` (deux soulignés avant et après) est appelée **constructeur**. Cette méthode est exécutée lors de la création de l'objet. On peut aussi donner des arguments à la définition qui seront nécessaires pour la création de l'objet:

```
class UneAutreClasse:
    def __init__(self, a, b):
        self.a = a
        self.b = b

obj2 = UneAutreClasse(5,6)
```

La méthode `__del__` est appelée destructeur. Cette méthode est exécutée lorsque un objet est supprimé. Essayez le code suivant:

```
class Test:
    def __init__(self):
        print 'bonjour'
    def __del__(self):
        print 'au revoir'

t = Test()
del t          # t n'est plus défini après ça
```

10. Les modules

Un module est simplement un ensemble de code qui peut être chargé lors de l'exécution d'un script Python. La forme la plus simple d'un module est un fichier Python qu'on peut importer. Écrivez un fichier comme suit, et appelez-le `test.py`:

```
def hello():  
    return 'bonjour'
```

Si vous ouvrez maintenant une console Python (le dossier de travail doit être celui où se trouve le fichier `test.py`), vous pouvez écrire le code suivant:

```
import test          # Charger le module dans test.py  
print test.hello()
```

Lors de l'importation, il n'est pas nécessaire d'écrire l'extension `.py`, Python permet de charger uniquement des fichiers de ce type et ajoute automatiquement l'extension.

Toutes les fonctions et variables du fichier `test.py` doivent être accédées par `test.xxx`, pour éviter la perte d'une variable ou d'une fonction déjà existante avec le même nom.

Un module peut également être un dossier. Dans ce cas, il faut avoir au moins un fichier avec nom `__init__.py` à l'intérieur du dossier. Un module Python peut être en Python, ou du code compilé, écrit typiquement en C, C++, ou encore Fortran.

Python possède déjà un grand nombre de module pré-installés, et vous pouvez en installer d'autres. Ces modules se trouvent dans un ou plusieurs dossiers bien précis. La liste de ces dossiers où Python cherche des modules se trouve dans la variable `sys.path` du module `sys` (à charger avec `import sys`). Cette variable est initialisée au lancement de Python par la variable d'environnement `PYTHONPATH`, ou une valeur par défaut. Vous pouvez ajouter un dossier à l'aide de la commande `sys.path.append('/home/user/modules_python')` pour la durée d'une session Python. Le dossier de travail courant (symbolisé par un point sous Unix) figure également dans la liste des dossiers à parcourir.

L'instruction `dir` permet de lister toutes les fonctions et variables d'un module:

```
dir(sys)          # Liste des fonctions et variables du module sys
```

L'importation d'un module peut aussi se faire sous un autre nom, par exemple si le nom du module est long, ou si on a déjà une variable avec le nom du module:

```
import sys as s  
s.path          # au lieu de sys.path
```

Il est également possible de charger uniquement certaines parties d'un module, par exemple la fonction `hello()` de notre module `test`:

```
from test import hello  
print hello()      # au lieu de test.hello()
```

11. Aperçu de quelques modules Python

Python possède un nombre assez important de modules installés par défaut; il s'agit de la *librairie standard*. Nous allons présenter brièvement quelques-uns des plus importants.

11.1. Interface avec le système d'exploitation

Trois modules nous aident avec les opérations liées au système d'exploitation: `os`, `sys` et `shutil`. Il y a quelques différences mineures entre les différentes plateformes pour ces modules, pour des raisons évidentes.

Le module `os` offre un accès aux fonctions du système, p.ex. pour changer le dossier courant (`os.chdir()`), ou obtenir le dossier courant (`os.getcwd()`). La variable `os.sep` contient le caractère de séparation des dossier dans un chemin d'accès ('/' sous Unix, '\' sous Windows). D'autres fonctions utiles:

- `os.mkdir()` permet de créer un dossier (make directory)
- `os.rmdir()` permet d'enlever un dossier vide (utiliser `shutil.rmtree()` pour enlever un dossier avec tous les fichiers à l'intérieur)

Le module `sys` contient des fonctions et variables de base de l'interpréteur Python. Voici les plus importantes:

- `sys.argv` contient une liste des arguments du programme Python
- `sys.path` contient une liste des dossier à parcourir pour chercher des modules
- `sys.exit()` est une fonction permettant de quitter Python (équivalent à Ctrl-D)
- `sys.platform` donne le nom du système d'exploitation

Le module `shutil` (shell utilities) fournit des fonctions typiquement trouvée dans une console Unix. Il y a notamment les fonctions suivantes:

- `shutil.copytree()` pour copier un dossier avec tout son contenu
- `shutil.rmtree()` pour supprimer un dossier avec tout son contenu (à utiliser avec beaucoup de prudence, cette action n'est pas réversible!)
- `shutil.abspath()` convertit un chemin d'accès relatif en chemin absolu. Par exemple le chemin `'../../lib'` est converti en `'/lib'` (si votre dossier courant est le dossier de départ par défaut `/home/user`).
- `shutil.move()` permet de déplacer un fichier ou un dossier.

11.1. Maths

Le module `math` offre un grand nombre de fonctions et constantes mathématiques. Le module `random` donne accès à des générateurs pseudo-aléatoires.

Quelques fonctions et constantes du module `math`:

- `math.e` et `math.pi` sont deux constantes fréquentes
- `math.ceil()` arrondi vers le nombre entier supérieur
- `math.floor()` arrondi vers le nombre entier inférieur
- `math.log()` et `math.log10()` calculent le logarithme

- `math.sqrt()` calcule la racine carrée (`math.sqrt(4)` est équivalent à `4**0.5`)
- Il y a les fonctions trigonométriques aussi: `math.sin()`, `math.cos()`, `math.asin()`, etc.

La documentation complète se trouve à l'URL <http://docs.python.org/library/math.html> ou <http://localhost/pydoc/library/math.html> dans l'image virtuelle.

Le module `random` offre surtout deux fonctions utiles: `random.random()` pour générer une variable aléatoire entre 0 et 1 (1 non-inclut), et `random.randint(a, b)` génère un nombre entier entre a et b inclut.

11.1. Dates et heures

Le module `datetime` permet de travailler avec des dates et des heures. Voici juste quelques exemples utiles:

```
import datetime
aujourd'hui = datetime.date.today()
print aujourd'hui.year, aujourd'hui.month, aujourd'hui.day
maintenant = datetime.datetime.now()
print maintenant.day, maintenant.hour, maintenant.minute
fete_nationale = datetime.date(2011, 8, 1)
```

Le module `time` offre également quelques fonctions utiles:

- `time.time()` donne le temps en secondes depuis le 1.1.1970
- `time.sleep()` permet de faire «dormir» votre code pour le nombre de secondes spécifié

La documentation des modules se trouve aux adresses suivantes:

- <http://docs.python.org/library/datetime.html> ou <http://localhost/pydoc/library/datetime.html>
- <http://docs.python.org/library/time.html> ou <http://localhost/pydoc/library/time.html>

12. Introduction à Numpy

Numpy est une librairie puissante pour travailler avec des tableaux et matrices multi-dimensionnels. Grâce à Numpy, nous pouvons effectuer une grande gamme d'opérations sur des matrices et similaire. Un tableau ou une matrice à deux dimensions permet d'enregistrer des données raster provenant d'un SIG, avec la contrainte que la géoréférence n'est pas gérée.

Numpy est la bibliothèque de base pour le calcul scientifique. Il y a une grande communauté d'utilisateurs dans beaucoup de domaines différents.

L'objet principal de Numpy est l'**array**, aussi appelé **ndarray** (pour n-dimensional array). Un **ndarray** peut être créé simplement depuis une liste Python:

```
import numpy as np
a = np.array( [1, 2, 3] )
b = np.array( [[1,2], [3,4], [5,6]] )
```

Il est aussi possible de spécifier un type pour le **ndarray**:

```
c = np.array( [[1,2], [3,4], [5,6]], dtype=np.float64 )
```

Le type float64 est un nombre à virgule flottante avec une précision de 64 bits. D'autres longueurs fréquemment utilisés sont 32 ou 128 bits. Pour les nombres entiers, il y a des types similaires: int8, int16, int32, int64.

Le type d'un **ndarray** existant peut être obtenu avec **a.dtype** (pour l'array **a**).

La taille d'un array peut être obtenu avec **a.shape**, ce qui renvoie un tuple avec la taille pour chaque dimension:

```
a = np.array([[1,2], [3,4], [5,6]] )
a.shape()      # retourne (3,2)
```

Il est possible d'obtenir un nouvel array d'une taille différente, mais avec les mêmes éléments (il faut donc avoir au total le nombre d'éléments):

```
a.reshape((2,3))  # [[1, 2, 3], [4, 5, 6]]
a.reshape((6,))   # [1, 2, 3, 4, 5, 6]
```

Plusieurs autres fonctions permettent la création d'un array d'une taille à préciser:

```
np.zeros( (2,3) ) # Array rempli avec 0
np.ones( (2,3) )  # Array rempli avec 1
np.eye(3)         # Array carré avec 1 sur la diagonale, 0 sinon
```

Les fonctions suivantes permettent de créer un array avec une séquence de nombres:

```
np.arange(0, 2, 0.3)      # Séquence de 0 à <2 avec intervalle 0.3
                           # Retourne array([0., 0.3, 0.6, 0.9, 1.2, 1.5, 1.8])

np.arange(0,6).reshape( (2,3) )
    # Séquence de 0 à <6 avec intervalle 1,
    # avec une taille redéfinie à 2 lignes et 3 colonnes
```

Beaucoup d'opérations sont possibles sur les arrays:

```
a = np.arange(0,6).reshape( (2,3) )
    # array([[0, 1, 2],
    #       [3, 4, 5]])

a - 1      # Soustraction de 1 de chaque élément:
    # array([[ -1,  0,  1],
    #       [ 2,  3,  4]])

np.sum(a)   # La somme de tous les éléments
    # 15

np.sum(a, axis=0)  # Somme par colonne
    # array([3, 5, 7])

a % 2 == 0      # Donne True pour les nombres pairs
    # array([[ True, False,  True],
    #       [False,  True, False]], dtype=bool)

b = np.array( [[7, 8, 9], [9, 8, 7]] )
a * b          # Multiplication élément par élément
    # array([[ 0,  8, 18],
    #       [27, 32, 35]])
```

L'accès à un élément en particulier dans un array se fait très simplement; les différentes dimensions sont séparées par une virgule:

```
a[0, 2]      # L'élément sur la 1ère ligne de la 3ème colonne
a[:, 2]      # La 3ème colonne
a[-1, -1]    # L'élément sur la dernière ligne de
              # la dernière colonne
```

Numpy peut aussi travailler avec des matrices:

```
m = np.matrix(a)      # Une matrice à partir d'un array
n = np.matrix(b)
```

La différence entre un array et une matrice consiste uniquement dans la façon que certains opérateurs sont appliqués. Voici quelques exemples de calcul matriciel:

```
n2 = n.T              # Transposé de la matrice n
m * n2                # Multiplication matricielle et non
                      # élément par élément comme dans un array
s = np.matrix( [[4,5], [6,3]] )
s.I                   # Inverse de la matrice s ( $s * s' = I$ )
```

13. Introduction à GDAL

GDAL est une librairie écrite en C++ permettant de lire et écrire des nombreux formats de données raster. GDAL est utilisé à peu près dans tous les SIG comme librairie de base pour lire et écrire des fichiers raster, y compris QuantumGIS ou ArcGIS. Nous pouvons utiliser GDAL aussi directement depuis Python. La seule complication est que la documentation Python pour GDAL est très rudimentaire...

Des informations sur GDAL se trouvent sur le site Internet: <http://gdal.org>

La librairie GDAL est très bien intégrée avec la librairie OGR, l'équivalent pour les données vectorielles. OGR sera discuté plus tard.

GDAL et OGR mettent à disposition quelques programmes simples de ligne de commande (gdalwarp, ogr2ogr et autres).

13.1. Lire une image raster

L'ouverture d'un fichier raster se fait de cette manière:

```
from osgeo import gdal
dataset = gdal.Open(
    '/home/user/geodata/north_carolina/ncrast/elevation.tif',
    gdal.GA_ReadOnly
)
if dataset is None:
    print "Problème d'ouverture du fichier"
```

Si l'ouverture du fichier échoue, la valeur retournée par `gdal.Open` est `None`. Autrement, `dataset` contient une référence vers le fichier ouvert. `dataset` contient plusieurs informations utiles sur l'image raster ouverte:

```
dataset.RasterCount          # nombre de bandes
dataset.RasterXSize, dataset.RasterYSize  # la taille du raster
dataset.GetGeoTransform()     # la géoréférence
dataset.GetProjection()       # définition de la projection
```

La géoréférence est un tuple de 6 éléments; voici un exemple avec explication:

```
(630000.0,      # coordonnée x du coin supérieur gauche
 10.0,         # taille d'un pixel en x (ouest-est)
 0.0,          # rotation, 0 si l'image est droit
228500.0,      # coordonnée y du coin supérieur gauche
 0.0,          # rotation, 0 si l'image est droit
-10.0)         # taille d'un pixel en y (nord-sud);
               # négatif dans l'hémisphère Nord
```

L'image complète peut être lue dans un ndarray de manière très simple:

```
raster = dataset.ReadAsArray()
```

Si l'image raster contient uniquement une bande, le ndarray résultant possède deux dimensions. Si le raster a plusieurs bandes, l'array a 3 dimensions. On peut par la suite travailler comme avec un ndarray normal...

Le ndarray peut être visualisée rapidement depuis Python si le module `matplotlib` est installé:

```
import matplotlib.pyplot as plt
raster_plot = plt.imshow(raster)
raster_plot.figure.show()
```

Il est possible de lire uniquement une partie de l'image à l'aide de la méthode `ReadAsArray`:

```
raster2 = dataset.ReadAsArray(
    xoff=10, yoff=10, xsize=100, ysize=100
)
```

Une fois que nous avons terminé avec la lecture et/ou écriture du fichier, nous devons le fermer. Pour faire cela, il suffit de changer la valeur de `dataset` à `None`:

```
dataset = None
```

Avec GDAL, il est commun de travailler au niveau d'une bande plutôt que de l'ensemble du raster. Une fois le fichier ouvert, on peut simplement demander une bande en particulier:

```
dataset = gdal.Open(
    '/home/user/geodata/north_carolina/ncrast/elevation.tif',
    gdal.GA_ReadOnly
)
if dataset is None:
    print "Problème d'ouverture du fichier"

band1 = dataset.GetRasterBand(1)
```

A noter une particularité de GDAL: l'index des bandes commence à 1, et non pas à 0 comme d'habitude en Python...

Une bande offre plus de possibilités de manipulation que l'image raster elle-même. On peut également demander les données sous forme de ndarray avec `ReadAsArray`, comme par exemple un histogramme:

```
raster = band1.ReadAsArray()    # comme avant...
hist = band1.GetHistogram()
```

L'histogramme peut être dessiné comme suit avec `matplotlib`:

```
hist_plot = plt.vlines(x=range(256), ymin=0, ymax=hist)
hist_plot.figure.show()
```

13.2. Écrire une image raster

L'écriture est possible par bande:

```
dataset = gdal.Open(
    '/home/user/geodata/north_carolina/ncrast/elevation.tif',
    gdal.GA_Update
)
if dataset is None:
    print "Problème d'ouverture du fichier"

bandel = dataset.GetRasterBand(1)
bandel.WriteArray(raster)
```

La création d'une nouvelle image nécessite un pilote pour le bon format, et puis il faut définir manuellement la géoréférence et éventuellement la projection:

```
driver = gdal.GetDriverByName("HFA")
```

Une liste des formats peut être trouvée sur le site Web de GDAL. Parmi les plus importants: GTiff (GeoTiff), HFA (Erdas Imagine, .img), RST (Idrisi Raster), XYZ (ASCII Gridded XYZ).

Par la suite, il est aisé de copier un fichier complet:

```
source = gdal.Open(
    '/home/user/geodata/north_carolina/ncrast/elevation.tif'
)
dest = driver.CreateCopy(
    '/home/user/geodata/north_carolina/ncrast/elevation.img',
    source, strict=0
)
source = None          # Fermer les deux fichiers proprement
dest = None
```

Vous venez d'effectuer une conversion de format de fichier!

Évidemment, il est aussi possible de créer un nouveau fichier vide:

```
datasource2 = driver.Create(  
    name = '/home/user/test.img',  
    xsize = 800, ysize = 600,  
    bands = 1, eType = gdal.GDT_Float64  
)
```

Une fois le fichier créé, on peut travailler normalement en demandant une bande, et écrivant les données avec `WriteArray`.

A noter le dernier argument `eType` qui définit le type d'image raster. Les types les plus courants sont: `GDT_Byte` (8 bits), `GDT_Float32`, `GDT_Float64`, `GDT_Int16`, `GDT_Int32`, `GDT_UInt16` (unsigned integer; entier non-négatif), `GDT_UInt32`.

13.3. Accéder aux valeurs par coordonnées

Un problème de lire un raster dans un `ndarray` est que nous travaillons au niveau des pixels, et non avec des coordonnées géographiques. GDAL n'offre aucune fonction automatique pour convertir de la représentation pixels vers les coordonnées géographiques, et vice-versa. Nous devons faire cette conversion manuellement, avec la géoréférence comme base. La géoréférence est facile à obtenir avec

```
datasource.GetGeoTransform()  
# (600000.0, 2.5, 0.0, 198000.0, 0.0, -2.5)
```

Comme déjà expliqué, le 3ème et 5ème élément sont des paramètres de rotation. Du coup, la conversion de pixels en coordonnées et vice-versa implique un peu de maths... Pour cette raison, nous mettons à disposition le code dans un module à part. Le module peut être téléchargé aussi depuis <http://www.361degres.ch/pyspatial.zip>.

Voici un exemple d'utilisation:

```
import pyspatial.raster as psrast  
dataset = gdal.Open(  
    '/home/user/geodata/north_carolina/ncrast/elevation.tif',  
    gdal.GA_ReadOnly  
)  
b1 = dataset.GetRasterBand(1)  
georef = dataset.GetGeoTransform()  
rast = b1.ReadAsArray()  
# La coordonnée du centre du pixel en haut à gauche:  
c1 = psrast.pixel_to_world((0,0), georef)  
    # c1 est alors égal à (630005.0, 228495.0)  
# La coordonnée en haut à gauche (le coin, et non le centre):  
c2 = psrast.pixel_to_world((0,0), georef, exact=True)
```



```
# Trouver la valeur du raster à l'endroit (640422,222108)
px1 = psrast.world_to_pixel( (640422, 222108), georef)
rast[px1]      # 87.100197
```

Trouver l'étendu de la couche peut alors se faire comme suit:

```
etendu = [
    psrast.pixel_to_world((0, 0), georef, exact=True),
    psrast.pixel_to_world((b1.YSize, b1.XSize), georef, exact=True)
]
```

Il devient apparent ici que dans un ndarray, les axes sont inversés par rapport au système de référence géographique. Il faut donc référencer un ndarray avec les coordonnées y,x, puisque les matrices (et donc le ndarray) adoptent des coordonnées lignes / colonnes.

14. Manipulation de données vectorielles avec Shapely

À côté de GDAL et OGR, il y a une troisième librairie open-source écrite en C++ qui est très répandue et utilisée avec les SIG. Il s'agit de la librairie GEOS, qui a un équivalent Java avec JTS (Java Topology Suite). GEOS permet de faire des opérations topologiques sur les géométries, tel qu'intersection, union, zone tampon, etc. Avec Python, il est possible d'utiliser les fonctions GEOS soit avec le module GeoDjango, soit avec Shapely. GeoDjango est une extension géographique au environnement de développement Web Django. Pour des raisons de simplicité, nous allons nous concentrer sur Shapely.

Shapely travaille uniquement sur les géométries. Il est possible de faire des unions et intersections, une zone tampon, interroger si un point se situe à l'intérieur d'un polygone etc. La première chose à faire est d'importer, exporter et créer une géométrie dans Shapely. Par la suite, nous allons regarder quelques opérateurs utiles.

Shapely connaît toutes les types de géométries et opérateurs définis dans la *Simple Features Specification*. Ces types de géométrie sont:

- **Point**.
- **LineString**: une série de points connectés.
- **LinearRing**: un LineString avec le premier et dernier point identique.
- **Polygon**: est composé d'au moins un LinearRing externe, et potentiellement plusieurs LinearRing internes
- **MultiPoint**: une géométrie composée d'un ou plusieurs Point
- **MultiLineString**: une géométrie composée d'un ou plusieurs LineString
- **MultiPolygon**: une géométrie composée d'un ou plusieurs Polygon

Créer une géométrie est relativement simple:

```
from shapely.geometry import Point, LineString
p = Point(600.0, 200.0)
l = LineString([(535.0, 155.0), (600.0, 200.0)])
```

Après, il est facile d'obtenir quelques informations sur les géométries:

```
p.x, p.y, p.z      # les coordonnées du point
p.wkt              # représentation well-known text
p.wkb              # représentation well-known binary
l.envelope.wkt     # l'enveloppe de la ligne définie en haut
                  # (c'est un polygone)
l.bounds           # la même information, mais sous forme d'un tuple
l.length           # la longueur de la ligne
list(l.coords)     # liste des coordonnées
l.centroid         # le centroid (instance de Point)
l.geom_type        # ou l.geometryType(): retourne un string
l.area             # la surface, égale à 0 pour une ligne
```

Notez que ces informations sont des attributs du point, et non des fonctions...

14.1. Importation et exportation de géométries

Shapely peut importer et exporter des géométries en format WKT, WKB et GeoJSON:

```
from shapely import geometry, wkb, wkt

pt = wkt.loads('POINT (0.0 0.0)')
pt_wkt = pt.wkt
# pt_wkt = 'POINT (0.0000000000000000 0.0000000000000000)'

pt_wkb = pt.wkb.encode('hex')
# pt_wkb = '01010000000000000000000000000000000000000000'
pt2 = wkb.loads(pt_wkb.decode('hex'))

pt_json = pt.__geo_interface__
# dictionnaire compatible avec GeoJSON:
# {'coordinates': (0.0, 0.0), 'type': 'Point'}

# Le dictionnaire peut être transformé en string:
import json
pt_json_str = json.dumps(pt_json)

# Un JSON string peut être transformé en dictionnaire:
pt2_json = json.loads(pt_json_str)

# et Shapely peut charger cet objet de deux manières:
pt2a = geometry.shape(pt2_json)
pt2b = geometry.asShape(pt2_json)
```

La différence entre `geometry.shape` et `geometry.asShape` est que dans le cas de `geometry.shape`, les coordonnées restent stockées dans le dictionnaire `pt2_json`. Dans le cas de `geometry.asShape`, Shapely copie les coordonnées.

Il est également possible de transformer les coordonnées en Numpy array, et vice-versa:

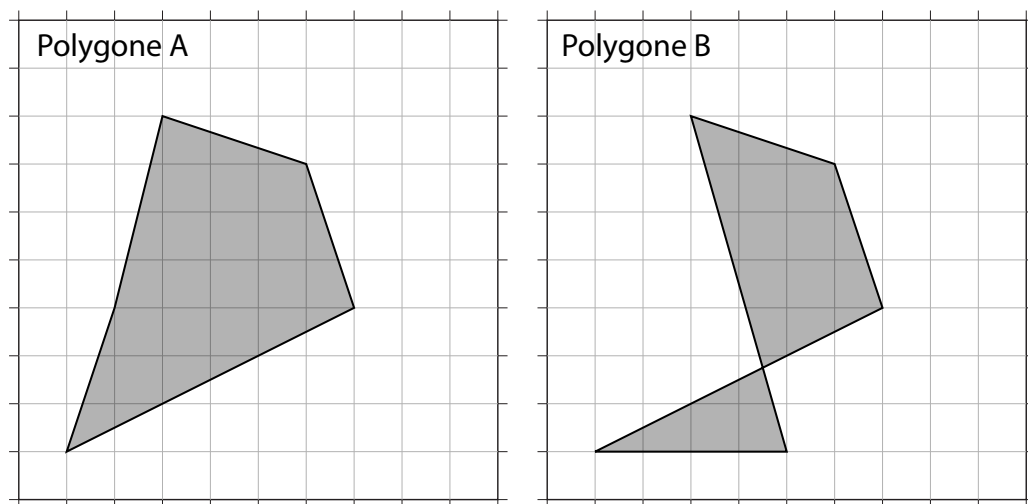
```
import numpy as np
line = wkt.loads('LINESTRING (0.0 0.0, 1.0 0.0, 1.5 1.0)')
line_coords = np.array(line)
line2 = geometry.asLineString(
    np.array([ [1.0, 0.5], [0.0, 0.0], [-1.0, -1.0] ])
)
```

14.2. Opérations géométriques

Shapely peut exécuter la plupart des opérations géométriques que l'on trouve dans un SIG. Voici les plus importantes.

Validité d'une géométrie

Il y a pour chaque type de géométrie quelques critères de validité. Une polygon ne doit par exemple pas s'intersecter elle-même (comme le polygon B).



On peut facilement tester la validité d'une géométrie avec la propriété `is_valid`:

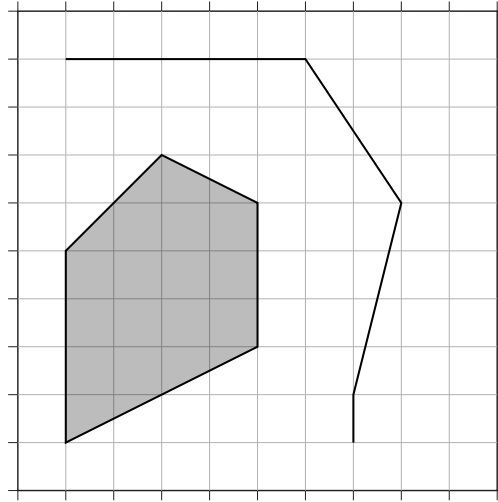
```
import shapely.geometry

polygon_a = shapely.geometry.Polygon([
    (1,1), (7,4), (6,7), (3,8), (2,4), (1,1)
])
polygon_a.is_valid          # retourne True

polygon_b = shapely.geometry.Polygon([
    (1,1), (7,4), (6,7), (3,8), (5,1), (1,1)
])
polygon_b.is_valid          # retourne False
```

Distance à un autre objet

Calculons la distance entre une ligne et un polygone:



```
import shapely.geometry
ligne = shapely.geometry.LineString([
    (1,9), (6,9), (8,6), (7,2), (7,1)
])
poly = shapely.geometry.Polygon([
    (1,1), (1,5), (3,7), (5,6), (5,3)
])
poly.distance(ligne)    # retourne 2.0
```

Buffer

Une zone tampon peut facilement être calculée pour toute géométrie:

```
pt = shapely.geometry.Point(0,0)
pt_buf = pt.buffer(1.0)
pt_buf.length    # retourne 6.28 (env. 2*pi)
```

Différence

Retourne les points d'une géométrie A qui ne superposent pas la géométrie B.

Intersection

Retourne l'intersection entre deux géométries (les parties en commun).

Différence symétrique

Retourne les points d'une géométrie A qui ne sont pas dans B, ainsi que les points dans B qui ne sont pas dans A. C'est le négatif de l'intersection, en quelque sort.

Union

Retourne l'ensemble des points des deux géométries.

Polygonize

Polygonize réfère à l'action de construire des polygones à partir d'un ensemble de lignes.

```
import shapely.ops
lines = [
    ((0, 0), (1, 1)),
    ((0, 0), (0, 1)),
    ((0, 1), (1, 1)),
    ((1, 1), (1, 0)),
    ((1, 0), (0, 0))
] # une liste de lignes faisant deux polygones
polygones = shapely.ops.polygonize(lines)
# le résultat peut être lu à l'intérieur d'une boucle
# extrait avec une liste:
poly_liste = list(polygones)
poly_liste[0].wkt
```

14.3. Tester la disposition de géométries

Shapely peut tester la position géométrique d'un objet par rapport à l'autre. La réponse de ces opérateurs sera simplement **True** ou **False**.

contains teste si une la géométrie B se trouve complètement à l'intérieur de la géométrie A, sans la toucher:

```
A = shapely.geometry.Polygon([(1,1),(7,4),(6,7),(3,8),(2,4)])
B = shapely.geometry.Point(4,5)
A.contains(B)
```

crosses s'applique uniquement aux unités linéaires (LineString, LinearRing, Polygon). Cette fonction teste si les deux lignes s'intersectent.

disjoint teste si les deux géométries sont complètement séparées.

equals détermine si les deux géométries sont au niveau géométrique identiques. A ne pas confondre avec un teste d'égalité au niveau informatique...

intersects retourne **True** si les deux géométries partagent au moins un point.

within est l'opérateur inverse de **contains**. Ces deux lignes sont donc strictement identiques:

```
A.contains(B)
B.within(A)
```

15. Introduction à OGR

OGR est une librairie permettant de lire et écrire des données vectorielles conformes à la «Simple Feature Specification». OGR permet de lire et écrire entre autres les formats SHP, GeoJSON, GML, Interlis*, CSV, KML, MapInfo, PostGIS*, PostGIS dump (les formats avec * ne sont pas disponibles par défaut). Une liste complète est disponible à l'URL http://www.gdal.org/ogr/ogr_formats.html.

OGR vient ensemble avec GDAL, et il est donc possible d'utiliser cette bibliothèque depuis Python. Avant de discuter l'API Python d'OGR, nous regardons rapidement un logiciel de ligne de commande venant avec OGR, `ogr2ogr`. Il s'agit d'un logiciel permettant de convertir un format vectoriel dans un autre, et de projeter un jeu de données en même temps. En lançant `ogr2ogr` sans aucun argument, nous pouvons avoir une aide pour l'utilisation du logiciel. Voici quelques exemples d'utilisation:

```
# Projection d'une couche de CH1903 (LV03) en WGS84:
ogr2ogr -s_srs "EPSG:21781" -t_srs "EPSG:4326" \
    -f "ESRI Shapefile" \
    /home/user/geodata/swissdata/Cantons_WGS84.shp \
    /home/user/geodata/swissdata/Cantons.shp

# Conversion de PostGIS vers ESRI Shapefile avec
# sélection spatiale et sur la base d'attributs
ogr2ogr -spat 475000.0 80000.0 625000.0 300000.0 \
    -where "annee > 1810" -f "ESRI Shapefile" \
    /home/user/geodata/swissdata/cantons_pg.shp \
    PG:'host=localhost user=user dbname=suisse password=user' \
    cantons

# Convertir un fichier Shape en GeoJSON
ogr2ogr -a_srs "EPSG:21781" -f "GeoJSON" \
    /home/user/geodata/swissdata/Cantons.json \
    /home/user/geodata/swissdata/Cantons.shp
```

Le format GeoJSON est très pratique car il est utilisable directement avec JavaScript (OpenLayers), et on peut le manipuler très facilement avec Python aussi. En plus, puisqu'il s'agit d'un format texte, il est possible de l'ouvrir dans un éditeur de texte, et d'inspecter le contenu, et de faire des modifications. Les géométries sont représentées en format WKT (Well-Known Text).

15.1. Ouvrir un jeu de données vectorielles

Voici un exemple simple comment ouvrir un fichier Shape avec OGR:

```
from osgeo import ogr
datasource = ogr.Open('/home/user/geodata/swissdata/Cantons.shp')
if datasource is None:
    print "Problème d'ouverture du fichier"
```

Une table de PostGIS peut également être ouverte d'une manière similaire:

```
datasource = ogr.Open(
    "PG: host=localhost dbname=suisse user=user password=user"
)
if datasource is None:
    print "Problème de connexion à la base de données"
```

A partir de la source de données, nous pouvons demander par la suite les couches contenues dans le jeu de données. Un fichier Shape contiendra typiquement une seule couche, tandis qu'une base de données PostGIS aura probablement plus qu'une couche. Le nombre de couche peut être obtenu avec:

```
datasource.GetLayerCount()
```

La couche elle-même peut être obtenue de deux manières:

```
lyr1 = datasource.GetLayerByIndex(0)
if lyr1 is None:
    print "Erreur de lecture de la couche"

cantons = datasource.GetLayerByName('cantons')
if cantons is None:
    print "Courche 'cantons' pas trouvée"
```

La deuxième méthode implique évidemment qu'on connaisse le nom de la couche. Pour obtenir tous les noms des couches, il faut faire une boucle pour obtenir chaque couche et en extraire le nom:

```
couches = []
for i in range(datasource.GetLayerCount()):
    lyr = datasource.GetLayerByIndex(i)
    couches.append(lyr.GetName())
```

Une fois que nous avons extrait une couche, nous pouvons extraire la définition de la couche, c'est-à-dire le nombre d'attributs, le type de géométries, les définitions de chaque attributs etc.:

```
lyr_def = lyr.GetLayerDefn()
lyr_def.GetFieldCount()
lyr_def.GetGeomType()      # C'est en fait juste un chiffre...
fdef = lyr_def.GetFieldDefn(0)
    # Retourne la définition du premier attribut
fdef.GetName()            # Le nom de l'attribut
fidx = lyr_def.GetFieldIndex('length')
    # Donne l'index de l'attribut avec nom 'length'
```

A partir de la couche, nous pouvons accéder aux **Features**, qui sont composés d'une géométrie et des attributs associés:

```
feat = lyr.GetFeature(1)    # 1 est le FID, peut varier...

# Il est aussi possible d'itérer à travers tous les Features
lyr.ResetReading()          # Remettre le curseur au début
for feat in lyr:
    # Travailler avec le Feature
    pass

# Ou d'extraire le Feature suivant:
lyr.ResetReading()
feat = lyr.GetNextFeature()

# Nous avons maintenant accès aux valeurs des champs:
feat.GetField(0)            # Premier attribut; nous devons demander
                            # l'index avec lyr_def.GetFieldIndex('...')

# Nous pouvons aussi demander la valeur dans un type de données
# spécifique:
feat.GetFieldAsDouble(0)
feat.GetFieldAsString(0)

# Il est aussi possible d'accéder depuis le Feature au nom des
# attributs:
feat.keys()
```

```
# Et aussi à un dictionnaire avec les attributs et les valeurs:  
feat.items()
```

Nous pouvons aussi accéder à la géométrie du Feature:

```
geom = feat.geometry()
```

Un Feature peut facilement être représenté en format GeoJSON. En principe, nous pouvons avoir ce format par Feature:

```
feat_json = feat.ExportToJson(as_object=True)
```

Certaines versions OGR ont un bug dans cette méthode, et vous avez une erreur plutôt que le format GeoJSON. Pour palier à ça, il faut un peu plus de code:

```
import json          # Pour transformer un string en objet Python  
feat_json = {  
    'type': 'Feature',  
    'geometry': json.loads(feat.geometry().ExportToJson()),  
    'properties': feat.items()  
}  
# Encore insérer le Feature ID:  
feat_json['properties']['id'] = feat.GetFID()
```

Le format GeoJSON nous sera très utile par la suite, car il nous permettra d'interagir avec le module Shapely que nous allons voir plus loin.

Une fois que nous avons fini avec notre couche OGR, nous devons fermer le jeu de données:

```
datasource = None
```

15.2. Manipuler les entités d'une couche

Nous pouvons obtenir un Feature, en modifier la géométrie ou les attributs, et enregistrer la version modifiée dans le jeu de données:

```
datasource = ogr.Open(  
    '/home/user/geodata/swissdata/Cantons.shp',  
    update = True  
)  
if datasource is None:  
    print "Problème d'ouverture du fichier"
```

```

lyr = datasource.GetLayerByIndex(0)
feat = lyr.GetNextFeature()

# Demander la valeur actuelle du premier attribut
feat.GetField(0)          # 25.715

# Donner une nouvelle valeur
feat.SetField(0, 25.777)

# Nous pouvons demander le nom du premier attribut à travers
# la définition du champ:
feat.GetFieldDefnRef(0).GetName()

# Écrire l'attribut
lyr.SetFeature(feat)      # Identifié avec le FID,
                          # donc pas besoin de passer l'index

# Être sûr d'écrire les changements
lyr.SyncToDisk()

# Close the datasource
datasource = None

```

15.3. Créer une nouvelle couche

Créer une nouvelle couche peut impliquer ou non la création d'une nouvelle source de données, dépendant sur le format de données utilisé. Un jeu de données PostGIS par exemple peut contenir plusieurs couches dans la même base de données, tandis qu'un fichier Shape peut contenir qu'une seule couche. Nous allons illustrer le processus de création d'un fichier Shape, avec la création d'attributs et de la géométrie (des points).

Création d'un nouveau fichier Shape à travers le «driver»:

```

import sys

driver = ogr.GetDriverByName('ESRI Shapefile')
if driver is None:
    print "Impossible de créer une couche en format Shapefile"
    sys.exit(1)

```

```

# Création du fichier Shape vide
datasource = driver.CreateDataSource("/home/user/random_pts.shp")
if datasource is None:
    print "Problème de création du fichier Shape"
    sys.exit(1)

# Le fichier Shape n'a toujours pas de couche. Nous devons la
# créer. Dans le cas d'un fichier Shape, le nom de la couche
# est identique au nom du fichier sans extension.
lyr = datasource.CreateLayer(
    name = "random_pts",
    srs = None,
    geom_type = ogr.wkbPoint
)
if lyr is None:
    print "Impossible de créer la couche"
    sys.exit(1)

# Créer un attribut ID, comme champ numérique entier
id_attr = ogr.FieldDefn('ID', ogr.OFTInteger)
id_attr.SetWidth(10)      # 10 chiffres au maximum
if lyr.CreateField(id_attr) is not 0:
    # CreateField retourne 0 si tout s'est bien passé...
    print "Problème de création de l'attribut ID"
    sys.exit(1)

# Créer un attribut Nom, comme champ de texte
nom_attr = ogr.FieldDefn('NOM', ogr.OFTString)
nom_attr.SetWidth(32)     # 32 caractères au maximum
if lyr.CreateField(nom_attr) is not 0:
    print "Problème de création de l'attribut Nom"
    sys.exit(1)

# Nous pouvons maintenant libérer la mémoire pour les attributs
# La couche en a fait une copie.
id_attr.Destroy()
nom_attr.Destroy()

```

```

# Nous pouvons maintenant commencer à créer les Features
# Nous créons 100 points dans un box de 0 à 100, la localisation
# est aléatoire.
import random          # pour les localisations aléatoires
for i in range(100):
    feat = ogr.Feature(lyr.GetLayerDefn())
    # Ecrire les valeurs des attributs
    feat.SetField('ID', i)
    feat.SetField('NOM', "Point %i" % (i+1,))
    # Créer la géométrie
    pt = ogr.Geometry(ogr.wkbPoint) # Créer un point "vide"
    pt.SetPoint_2D(
        point = 0,
        x = random.random() * 100.0,
        y = random.random() * 100.0
    )
    # Le premier argument est l'index du point à définir.
    # Ça ne fait pas beaucoup de sens pour un point, mais est
    # utilisé pour les lignes et polygones.

    # Finalement, nous pouvons définir la géométrie du Feature...
    feat.SetGeometry(pt)

    # ... et ajouter le Feature à la couche
    if lyr.CreateFeature(feat) != 0:
        # CreateFeature retourne 0 si tout s'est bien passé
        print "Problème de création du Feature"
        sys.exit(1)

    # Nous devons détruire chaque Feature créé avec CreateFeature
    # une fois que nous n'en avons plus besoin.
    # La couche tient toujours une copie du Feature, ça ne pose
    # donc pas de problème.
    feat.Destroy()

    pt.Destroy()          # et aussi la géométrie...

```

En principe, tout élément créé avec `ogr.Xxx` doit être détruit plus tard. Par contre, si vous détruisez un élément déjà détruit, votre programme va se terminer de manière abrupte, ensemble avec votre console Python (segmentation fault). En plus, après avoir fait `pt.Destroy()`, la variable `pt` contient toujours une référence vers l'objet, sauf que l'objet lui-même est parti... Il est donc une bonne pratique de faire quelque chose du style:

```
if pt is not None:
    pt.Destroy()
    pt = None
```

Nous pouvons maintenant fermer la couche:

```
datasource = None
```

Evidemment, il est aussi possible de supprimer un Feature d'une couche:

```
lyr.DeleteFeature(fid=0)
```

15.4. Lire et écrire du GeoJSON

Nous avons vu dans la section sur Shapely que nous pouvons utiliser directement des représentations GeoJSON pour Shapely, et aussi pour OGR. Par ailleurs, l'outil ogr2ogr permet de convertir des couches en format GeoJSON en lecture et écriture. En Python, il est très simple de lire et écrire des fichiers GeoJSON, beaucoup plus simple que de passer par OGR...

D'abord créer un fichier GeoJSON avec ogr2ogr (dans le Terminal):

```
ogr2ogr -f "GeoJSON" /home/user/geodata/swissdata/Cantons.json \  
/home/user/geodata/swissdata/Cantons.shp
```

Et puis dans Python:

```
import json  
f = open('/home/user/geodata/swissdata/Cantons.json')  
lyr = json.load(f)  
f.close()
```

Le résultat est un dictionnaire avec une structure bien définie. Vous pouvez facilement modifier ce dictionnaire; si vous respectez la structure, vous n'aurez aucune peine à l'écrire sur le disque à nouveau, et l'ouvrir dans un SIG.

La structure globale du dictionnaire est:

```
{ 'type': 'FeatureCollection', 'features': [  
    { 'type': 'Feature', 'geometry': {...}, 'properties': {...}  
  ] }
```

Évidemment, vous pouvez aussi construire une telle couche à partir de zéro.

L'écriture sur le disque n'est pas compliquée non plus:

```
f = open('/home/user/nouvelle_couche.json', 'w')  
json.dump(lyr, f)  
f.close()
```

C'est vraiment tout!

16. Se connecter à PostGIS avec Python

Pour se connecter à PostGIS, il suffit en principe de se connecter à PostgreSQL. Les géométries sont transférées en format WKB ou WKT. Puisque PostGIS intègre déjà une interface avec GEOS et la librairie de projection PROJ.4, la plupart des manipulations peuvent directement être exécutées dans PostGIS, à l'aide de quelques commandes SQL.

Plusieurs modules permettent d'accéder à PostgreSQL. Nous montrons ici le module psycopg2 (<http://initd.org/psycopg/>). Faisons un exemple de lecture de la table `lacs` de la base de données `suisse`:

```
import psycopg2 as psql

# Se connecter à la base de données
conn = psql.connect("dbname=suisse user=user password=user")

# Nous avons besoin d'un curseur pour pouvoir faire des opérations
# sur la base; 2 curseurs sont nécessaire si nous voulons utiliser
# le résultat de 2 requêtes simultanément
cur = conn.cursor()

# Maintenant, nous pouvons exécuter des requêtes:
cur.execute("SELECT gid, st_astext(the_geom) FROM lacs;")

# Nous pouvons faire une boucle sur le résultat
# (possible qu'une fois):
for l in cur:
    id = l[0]
    geom = shapely.wkt.loads(l[1])
    print id, geom.area

# Fermer le curseur et la connexion
cur.close()
conn.close()
```

16.1. Passer des paramètres à la requête

Bien évidemment, il est aussi possible de faire des requêtes CREATE TABLE, INSERT et UPDATE, de la même manière. Cependant, si nous voulons un paramètre à la requête, nous devons être un peu prudents. Il s'agit d'éviter des «SQL injections», qui sont potentiellement un risque de sécurité. Il suffit de respecter quelques règles simples. Une simple requête INSERT s'écrit comme suit:

```
# Assumons que nous avons un curseur cur...
cur.execute("""INSERT INTO lacs (gid, the_geom)
            VALUES (%s, ST_GeomFromText(%s, 21781))""",
            (150, 'MULTIPOLYGON(((600 200, 620 210, 595 215, 600 200)))')
)
conn.commit()    # Pour enregistrer les modifications...
```

A l'intérieur du string SQL, utiliser uniquement %s comme format, même pour les nombres.

Les choses à ne pas faire:

- _ Passer une variable avec le contenu de la requête à cur.execute
- _ Utiliser les mécanismes Python pour construire un string, comme le + ou %

Donc, voici les choses à **éviter**:

```
sql = """INSERT INTO lacs (gid, the_geom)
        VALUES (%s, ST_GeomFromText(%s, 21781))"""
d = (150, 'MULTIPOLYGON(((600 200, 620 210, 595 215, 600 200)))')
cur.execute(sql % d)    # FAUX FAUX FAUX FAUX FAUX !!!!
```

Il est absolument crucial de respecter ces quelques règles. Voici un extrait de la documentation Python à ce sujet:

Warning: Never, **never**, **NEVER** use Python string concatenation (+) or string parameters interpolation (%) to pass variables to a SQL query string. Not even at gunpoint.

17. Quelques autres libraires Python intéressantes

Python possède beaucoup de libraires pour des tâches très variées. Tandis que les possibilités d'extension sont un point fort de Python, l'inconvénient est qu'on peut vite se perdre dans la multitude de modules. Voici une brève liste non exhaustive de quelques modules intéressants pour personnes travaillant avec données spatiales.

17.1. Mapnik

Mapnik (www.mapnik.org) est une puissante librairie de production cartographique, écrite en C++, avec une interface en Python. Il y a actuellement deux versions de Mapnik, la version 1 et 2, qui ne sont pas tout à fait compatibles l'un avec l'autre. Il y a donc 2 modules Python: `mapnik` et `mapnik2`.

L'outil cartographique TileMill (<http://mapbox.com/tilemill/>) est également basé sur Mapnik; TileMill est disponible uniquement pour Mac OS X et Linux).

17.2. Matplotlib

Matplotlib est un module pour faire des graphiques, il est inspiré par les capacités graphiques de Matlab, d'où le nom. Il est principalement utilisé pour la visualisation rapide de données notamment depuis Numpy. Matplotlib, comme Matlab, est beaucoup utilisé dans les milieux scientifiques.

Nous avons déjà vu que nous pouvons utiliser Matplotlib pour afficher des images raster.

Il est aussi possible de dessiner des polygones. Voici un exemple:

```
import shapely.geometry
import matplotlib.pyplot as plt
import numpy as np
import json

f = open('/home/user/geodata/swissdata/Cantons.json')
cantons = json.load(f)
f.close()

fig = plt.figure()

for feat in cantons['features']:
    geom = shapely.geometry.shape(feat['geometry'])
    coords = np.array(geom.exterior)
    plt.fill(coords[:,0], coords[:,1], 'g')

fig.show()
```

1. pyproj

pyproj est une interface Python pour la librairie PROJ.4, qui est utilisée dans a peu près tous les SIG pour la gestion de la projection. Généralement, nous n'avons pas besoin d'utiliser cette librairie directement.

2. PIL

PIL, ou Python Imaging Library, permet de lire et écrire des images (non géoréférencées). Site Web de PIL: <http://www.pythonware.com/products/pil/>

Voici un exemple comment ouvrir une image:

```
import Image
im = Image.open('image.jpg')
im.size      # La taille de l'image en pixels
im.show()    # affiche l'image
```

Beaucoup de fonctions permettent de modifier l'image par la suite:

```
im.crop(100, 100, 400, 400)    # rogner l'image
im.resize((200,200))           # changer la taille
im.rotate(60)                   # tourner l'image de 60 degrés
                                # (dans le sens anti-horaire)
```

L'enregistrement d'une image n'est pas compliqué non plus:

```
im.save('nouvelle_image.jpg')
```

Il est possible de transformer une image en array Numpy, et vice-versa:

```
import numpy as np
a = np.array(im)                # c'est tout...
im2 = Image.fromarray(a)        # c'est vraiment tout...
```

3. scipy

Scipy est un module Python destiné aux scientifiques, mathématiciens et ingénieurs. Il est construit par dessus Numpy, qui fournit les fonctions de base pour les tableaux, et l'algèbre linéaire, transformé de Fourier etc.

Scipy intègre des nombreuses fonctions de statistiques, optimisation, intégration numérique, traitement du signal et d'image etc.

Grâce à Scipy, vous pouvez par exemple appliquer des nombreux filtres à des images, appliquer des opérations de morphologie mathématique, ou faire une classification hiérarchique.

<http://www.scipy.org/>

4. pysal

PySAL est un module pour l'analyse spatiale, notamment la statistique spatiale. On peut par exemple calculer l'indice de Moran, estimer les limites optimales pour les classes d'une carte choroplèthe (carte thématique avec surfaces colorées), ou encore construire des modèles basés sur les chaînes de Markov.

www.pysal.org

5. NetworkX

NetworkX est un module pour l'analyse de réseaux (des graphes au sens mathématique). Un graphe est par définition un ensemble de noeuds et de arêtes ou arcs (liens entre les noeuds). Un graphe peut être construit de manière relativement simple:

```
import networkx as nx
g = nx.Graph()
g.add_node(n=1)    # n peut être un nombre, string, tuple;
                  # les mêmes règles que pour la clé d'un
                  # dictionnaire s'appliquent

# Il est aussi possible d'ajouter plusieurs noeuds à la fois:
g.add_nodes_from([2,3,4])

# Ajouter des arêtes (edge)
g.add_edge(1,2)
g.add_edges_from([(1,3), (2,3), (2,4), (3,4)])
```

Par la suite, il est possible d'obtenir quelques informations sur le graphe:

```
g.number_of_nodes()    # retourne 4
g.number_of_edges()    # retourne 5
g.neighbors(3)         # les voisins du noeud 3: [1,2,4]
```

Il est aussi possible de faire des opérations comme le chemin le plus court:

```
nx.shortest_path(g, source=1, target=4)
# retourne [1, 2, 4]
```

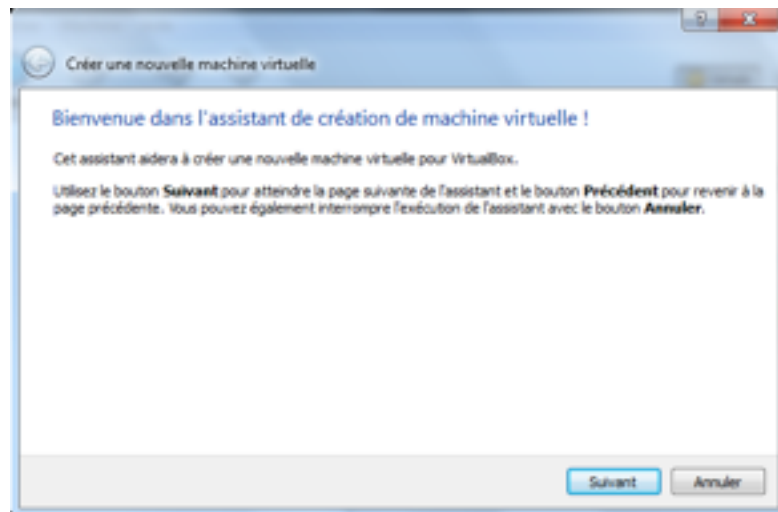
NetworkX permet aussi d'associer des attributs aux arêtes et de les utiliser comme poids (ou coût) pour le chemin le plus court. Il est aussi possible de créer des graphes orientés, où les arêtes sont des arcs (un arc est une arête orientée).

Beaucoup d'autres fonctions sont également disponibles, pour caractériser le graphe, et aussi pour visualiser le graphe ou une partie.

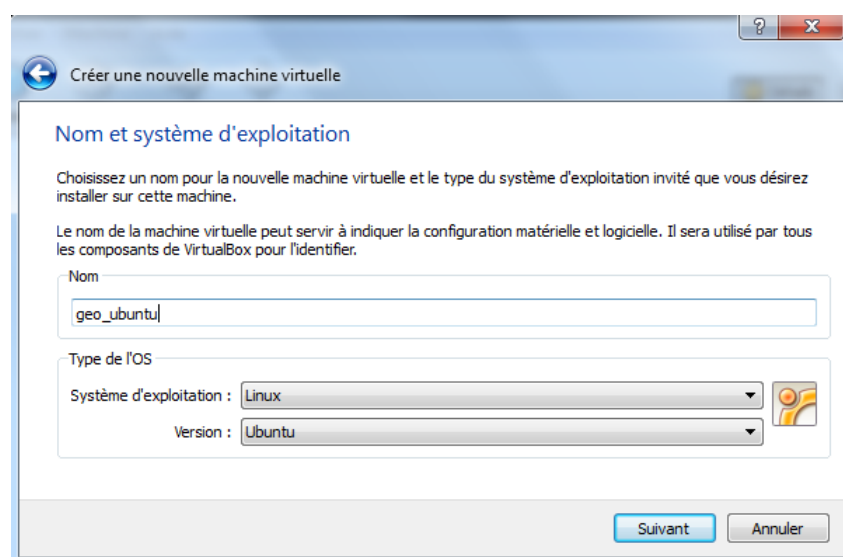
<http://networkx.lanl.gov>

Appendice A: Installer la machine virtuelle Linux

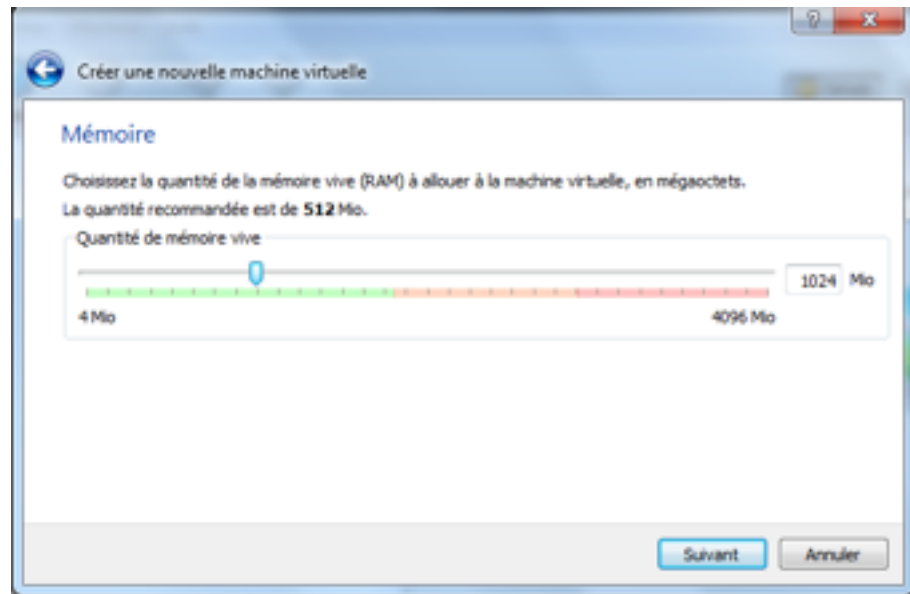
1. Installer VirtualBox. Nous avons testé l'image avec la version 4.1.6. Le logiciel d'installation pour cette version est fourni sur la clé USB.
2. Copier l'image de disque virtuel «geo_ubuntu.vdi» de la clé USB sur votre disque dur.
3. Lancer VirtualBox. Dans la fenêtre principal, cliquer sur le bouton «Créer», en haut à gauche. Un assistant vous guidera à travers la création d'une machine virtuelle.



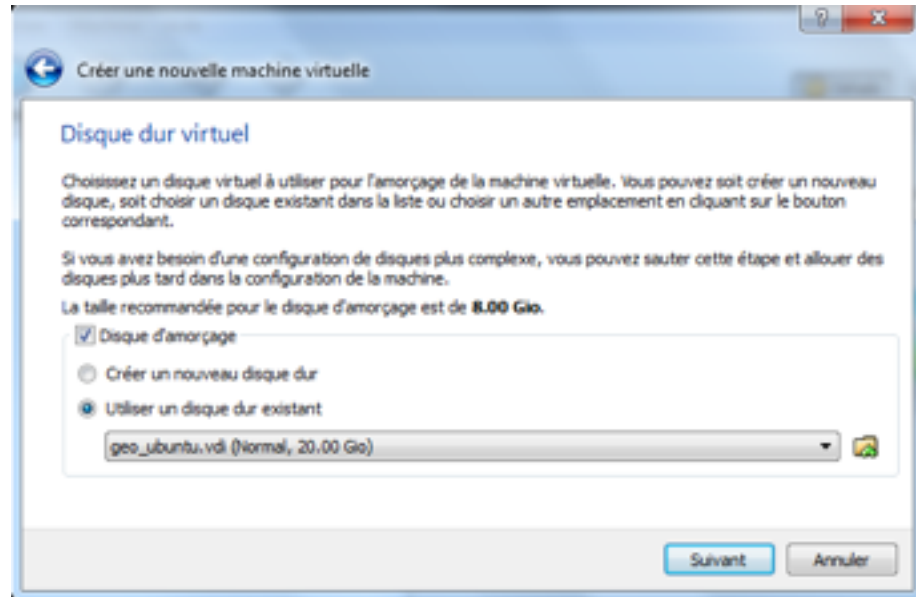
4. Saisir un nom de la machine virtuelle, p.ex. «geo_ubuntu». Vérifier que le système d'exploitation est «Linux» et la version «Ubuntu» dans les menus déroulants.



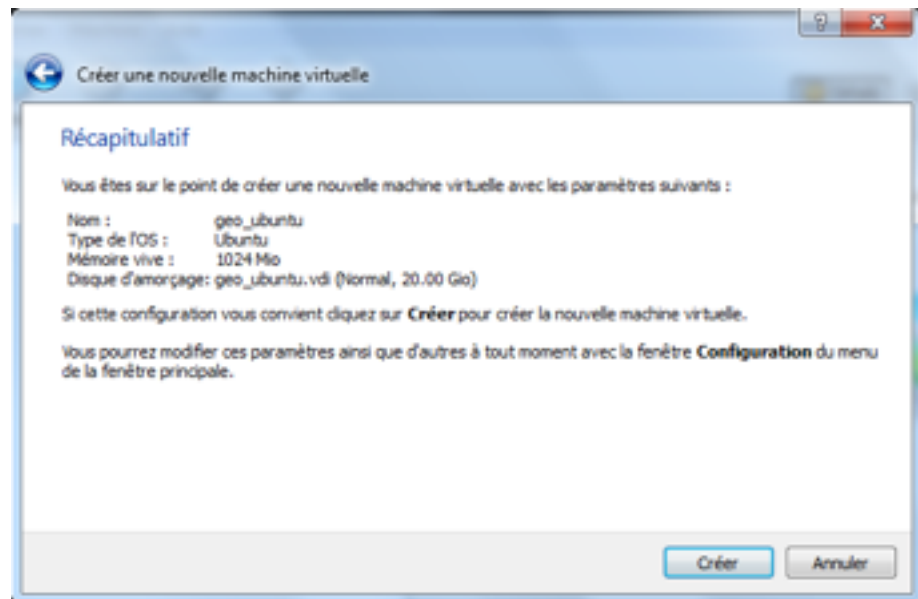
5. Attribuez de la mémoire vive à la machine virtuelle. 1Go devrait être suffisant.



6. Dans l'étape suivante, il faut sélectionner un disque dur. Ne créez pas un nouveau disque, mais sélectionnez l'image copiée depuis la clé USB.



7. La fenêtre suivante est une récapitulation des paramètres. Cliquez sur le bouton «Créer».



8. La création de la machine virtuelle est maintenant terminée. Vous pouvez démarrer la machine.