Initiation Python

Xavier JUVIGNY

Septembre 2009

1 Les bases de Python

1.1 Introduction

1.1.1 Historique

Python est un langage créé par Guido Van Rossum fin 1989 au Centrum voor Wiskunde en Informatica (CWI) aux Pays-Bas. Le nom du langage fait référence à la série britannique *Monty Python flying circus* dont Guido Von Rossum était fan. L'année suivante (1990), une première version (0.9) est sortie dans le domaine publique. La dernière version sortie au CWI est la version 1.2 en 1994.

En 1999, avec la sortie de la version 1.6.1, Python devient compatible avec la GPL (Gnu Public License). La version actuelle du langage est la 2.6.2. Une version beta de la 3.0 est en cours de développement. Au contraire des autres versions, cette version beta casse la compatibilité ascendante pour épurer le langage et enlever les bibliothèques standards obsolètes ou redondants.

1.1.2 Aperçu

Python est un langage disponible sur un grand nombre de plateformes : Unix, Windows, Mac, etc. Les sources sont disponibles afin de l'adapter à un grand nombre de machines.

Le site `www.python.org` centralise les différentes versions et portage du langage. On y trouve également toute la documentation ainsi qu'un tutorial en anglais, consultables en ligne et en format électronique (pdf).

C'est un langage **universel** utilisé pour un large éventail d'applications :

- Infographie (Light and Magic utilise python, Poser, Blender)
- Internet (la bibliothèque twisted en particuliers)
- Calcul scientifique (grâce aux bibliothèques numpy et scipy)
- Jeu et loisir (via pygame ou pyKyra)

– Education (comme langage d'initiation à la programmation)

Un nombre toujours plus grand de produits utilise Python. Dans le domaine qui nous intéresse, des produits de visualisations scientifiques comme Tecplot (commercial), Paraview ou MayaVi (logiciels libres) possèdent une interface python.

L'ONERA, Airbus, EDF et le CEA ont développé des logiciels utilisant Python (**elsA** pour l'ONERA, Paramed pour EDF et le CEA).

Python est un langage interprété, ce qui lui permet une certaine souplesse par rapport à des langages compilés comme le C, le Fortran ou le C++. Néanmoins, cette souplesse se paye par une relative lenteur par rapport aux langages sus-cités (On peut avoir un rapport de 10 entre Python et un langage comme le C).

Heureusement, l'intérêt du langage, outre sa syntaxe simplifiée, réside dans sa possibilité d'extension via une interface en C bien fournie et étendue, permettant de rajouter des modules (bibliothèques) écrits en C directement utilisables en Python. Cela permet grâce à ces modules écrits en C de pouvoir obtenir au final une performance similaire au C ou au Fortran.

1.1.3 Syntaxe

Python a été conçu pour être un langage lisible. Il vise à être visuellement épuré, et utilise des mots anglais fréquemment là où d'autres langages utilisent de la ponctuation, et possède également moins de constructions syntaxiques que de nombreux langages structurés tels que C, Perl, ou Pascal. Les commentaires sont indiqués par le caractère #.

Les blocs sont identifiés par l'indentation, au lieu d'accolades comme en C/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. Les parenthèses sont facultatives dans les structures de contrôle :

```c
// Fonction factorielle en C
int factorielle(int x)
{
    if (x == 0)
      return 1;
    else
      return x*factorielle(x-1);
}
```

```python
# Fonction factorielle en python
def factorielle(x):
    if x == 0:
      return 1
    else:
      return x*factorielle(x-1)
```

Voici brièvement les autres caractéristiques :

– Python peut être lancé en mode interactif (comme un shell évolué)

– C'est un langage typé (À chaque variable correspond un type), mais possédant un typage dynamique (et non statique comme en C ou C++). C'est à dire que le type d'une variable est définie à l'exécution

et qu'une variable peut changer de type durant l'exécution du programme.
- Python offre une gestion "transparente" de la mémoire. Un compteur de référence (un compteur comptabilisant le nombre de variables, fonctions, etc. accédant à une certaine variable) permet à Python de connaître le moment où il peut désallouer une variable.
- On accède aux variables automatiquement par adresse (comme en Fortran), et la notion de pointeur n'existe pas !
- Comme Python est un langage interprété, il est possible de modifier le code dynamiquement, voir d'interpréter une chaîne de caractère comme du code à exécuter.
- C'est un langage dit orienté objet. Python offre la possibilité de définir des classes, des exceptions et possède la notion d'héritage (dont l'héritage multiple). Nous y reviendrons.
- Contrairement au C++, tout est considéré comme des objets, les fonctions et les classes y compris (On peut très bien concevoir une classe générant comme instance des classes, ou manipuler des fonctions comme des variables ordinaires).
- Python effectue une compilation dynamique en bytecode afin d'accélérer l'exécutions de scripts
- Il existe des compilateurs externe (exemple, JIT par psyco) permettant de compiler les scripts avant leurs exécutions en optimisant le bytecode.
- Python est *case-sensitive*, c'est à dire qu'il est sensible aux majuscules et minuscules. La variable Ni et la variable NI sont des variables différentes !

1.2 Tour d'horizon du langage

Les commentaires dans Python se font de la même manière que pour le shell :

```
# Ceci est un commentaire
```

Pour afficher simplement un message ou des valeurs :

```
print '2+2 = ',2+2
```

La concaténation de chaînes de caractère se fait simplement :

```
animal = 'Drom'+"adaire"
```

On peut aussi répéter facilement un caractère grâce à l'opération de répétition :

```
trait = '-'*72 # trait represente une chaine contenant 72 symbols '-'
```

Pour une gestion plus fine des sorties, on peut comme en C formatter les sorties :

```
print 'Nb caracteres = %3d\n'%len(a)
```

Les opérations courantes de l'arithmétique entière sont fournies par défaut :

(46**157)% (37**156)

Remarquez que la taille des entiers est seulement limitée que par la mémoire disponible sur l'ordinateur.

La définition des fonctions se fait simplement :

```
def f(a,b,c) :
  d = a*b
  return d+c
```

On peut définir simplement des tableaux et accéder à ses éléments : :

```
inventaire=[3.14, 5, 3+2j, "Raton laveur"]
pi     = inventaire[0]}
raton = inventaire[-1]
numbers = inventaire[0:3]
```

Il est possible de supprimer un objet ou un module (considéré par Python comme un objet) par l'instruction **del** :

```
import math
inventaire=[3.14, 5, 3+2j, "Raton laveur", math.pi]
...
del inventaire
...
del math
```

2 Syntaxe du langage python

2.1 Définition des blocs

Un bloc est une partie du code encadrée par des délimiteurs (en C, le début d'un bloc est défini par { et la fin par }). En python, il n'y a pas vraiment de délimiteurs, mais c'est l'indentation qui sert de délimiteurs.

On utilise l'indentation pour définir des blocs pour les conditions, les boucles, les fonctions, les classes, etc.

Exemple :

```
if x<0 :
  print 'Attention : x est negatif'
  x = 0
y = math.sqrt(x)
```

2.2 Les opérateurs booléens

Les opérateurs logiques s'écrivent en toutes lettres. On a donc :

– L'opérateur **and** : c'est le `et` logique, on a donc :

True **and** True	= True
True **and** False	=False
False **and** False	=False

– L'opérateur **or** : c'est le `ou` logique, on a donc : True **or** True = True ; True **or** False = True ; False **or** False = False

– L'opérateur **is not** : C'est le `ou exclusif` logique, on a donc : True **is not** True = False ; True **is not** False = True ; False **is not** False = False

– L'opérateur **not** : C'est l'opérateur logique de `négation` : **not** True = False ; **not** False = True

De même, il existe des opérateurs logiques bit à bit (même symboles que le C sauf pour la négation) :

– L'opérateur `&` : C'est le `et` bit à bit : 0b0011 & 0b0101 = 0b0001

– L'opérateur `|` : C'est le `ou` bit à bit : 0b0011 | 0b0101 = 0b0111

– L'opérateur `^` : C'est le `ou exclusif` bit à bit : 0b0011 ^ 0b0101 = 0b0110

– L'opérateur `~` : C'est le calcul du complément (négation bit à bit) ~0b0011 = −4

Les opérateurs **or** et **and** sont dits parresseux :

a **or** b **and** c **or** d ≡ (a **or** b) **and** (c **or** d)

2.3 Les conditions

Le branchement selon les conditions est proche de la syntaxe en C

La syntaxe de base : **if** condition : instruction

On peut rajouter une clause **else** : **if** condition : instruction1 **else** : instruction2

On peut également contracter **else if** en **elif** :

if condition1 : instruction1 **elif** condition2 : instruction2 **else** instruction3

Contrairement au C, il n'existe pas d'instruction `switch(...) case...` Soit on remplace cela par une série de **if** cond1 : instr1 **elif** cond2 : instr2 ... soit à l'aide de fonctions lambda (qu'on verra plus tard dans le cours).

Exemple :

```
# On calcul la valeurs absolue de x :
y = 0
if x < 0 :
  y = -x
else :
  y = x
```

2.4 Les boucles

2.4.1 La boucle while

Elle fonctionne comme en C. Sa syntaxe : **while** condition : instruction

Exemple :

```
# Calcul du factoriel :
def fact(n) :
```

```
  f = 1
  while n>0 :
    f *= n
    n = n-1
  return f
```

2.4.2 La boucle for

Dans le langage python, l'instruction **for** permet de parcourir un objet itérable, c'est à dire un objet contenant un ensemble de valeurs (ou d'autres objets) lesquels on peut accéder une à une. Par exemple, un tableau ou une liste sont des objets itérables.

La syntaxe de base est **for** var **in** iterable : instruction

Exemples :

```
# On parcourt tous les entiers de 0 a 9 :
# range(10) cree un tableau contenant tous les entiers entre 0 et 9 compris.
for i in range(10) :
  pass
# On parcourt et affiche les 5 premiers nombres premiers :
for i in [2,3,5,7,11] :
  print i;'\t',
print
```

2.5 Les types prédéfinis en Python

2.5.1 Les types unitaires

Les types simples en python sont :

- Les chaînes de caractère : `s = 'Toto'`
- Les entiers : `n = 10`
- Les flottants (double précision) : `p=3.1415`
- Les complexes (double précision) : `z=0.5+3.2j`
- Les booléens `True` et `False` : `f1 = True`
- Le type spécial `None` (équivalent au `NULL` du C)

Les types complexes sont :

- Le type fichier : `file ( fichier )`. Exemple :

```
f = file('~/.bashrc') # On ouvre le fichier
for l in f : # Pour chaque ligne contenu dans le fichier
  print l, # On affiche la ligne sans rajouter saut a la ligne (',')
f.close()  # On referme le fichier
```

- D'autres types définis par l'utilisateurs (les classes) ou issus d'un module

2.5.2 Les listes et les tuples

Les listes et les tuples sont vus comme des tableaux (des séquences d'éléments ordonnés). Seule différence : les listes sont modifiables, pas les

tuples.

Exemple de création d'une liste contenant 4 éléments :

```
tab = [2,3,6,7]
```

Exemple de création d'un tuple contenant 4 éléments :

```
tup = (2,3,6,7)
```

Pour accéder à un élément d'une liste ou d'un tuple :

```
p = tab[0]    # Premier element de tab
t = tup[1]    # Deuxieme element de tup
p2= tab[-1]   # Dernier element de tab
t2= tup[-2]   # Avant dernier element de tup
```

Comme en Fortran 90, on peut accéder à un sous-enseble de la liste ou du tuple :

```
tp = tab[2:5] # tp contient les elements de tab allant de 2 a 4
tp = tab[:4]  # tp contient les elements de tab allant de 0 a 3
tp = tab[2:]  # tp contient du 2ieme au dernier element de tab
tp = tab[:-1] # tp contient tous les elements de tab sauf le dernier
tp = tab[:]   # On copie dans tp tous les elements de tab
```

Remarquez que la dernière opération de l'exemple ci–dessus est équivalent à recopier la liste tab dans une nouvelle liste tp.

On peut connaître le nombre d'éléments d'une liste ou d'un tuple grâce à l'instruction len :

```
n = len(tab) # n contient le nombre d'elements de tab
```

Exemple de parcourt de tableau par les indices :

```
t = [2,3,5,7,11]
lgth = len(t)
for i  in xrange(lgth) :
  print  '%d prem.'%t[i]
```

De plus, on peut vérifier si un objet ou une valeurs appartient à un tableau (ou un tuple) :

```
3 in tab # Renvoie vrai
5 in tab # Renvoie Faux
```

L'opérateur + permet de concaténer deux listes :

```
tab1 = [2,3,6]
tab2 = [7,9,11]
tab  = tab1 + tab2 # tab = [2,3,6,7,9,11]
```

L'opérateur * permet de créer une liste par répétition :

```
tab = [1]*5 # Equivalent a tab = [1,1,1,1,1]
```

Il est également facile de trier une liste :

```
tab = [1,5,3,4,2]
tab.sort()
print tab # on affiche [1,2,3,4,5]
```

Pour inverser l'ordre des éléments :

```
tab = [1,5,3,4,2]
tab.sort()
tab.reverse()
print tab # on affiche [5,4,3,2,1]
```

Pour trouver le nombre d'occurence d'une valeurs ou d'un objet dans une liste (ou un tuple) :

```
tab = ["toto","titi","toto","tutu","toto"]
print tab.count("toto") # Affiche 3 (il y a trois fois "toto" dans tab)
```

Pour rajouter un élément à la fin d'une liste :

```
tab = ["toto","titi", "tutu"]
tab.append(3.1415) # Maintenant, tab vaut ["toto", "titi", "tutu", 3.1415]
```

Pour retirer un élément de la fin d'une liste et le retourner :

```
tab = ["toto","titi", "tutu"]
s = tab.pop()
print s   # Affiche "tutu"
print tab # Affiche ["toto","titi"]
```

Pour connaître l'indice minimum où se trouve un élément de la liste :

```
tab = ["toto","titi", "tutu", "titi"]
print tab.index("titi") # Affiche 1
```

Une application : le crible d'Erasthotene

```
# Creer un tableau d'entier allant de 2 a 99
tab = range(2,100)
for n in tab : # Pour chaque element de tab
  print '%2d premier'%n # Le premier element trouve est premier
  j = 2*n
  while j < 100 :
    # Si j multiple de n est dans tab
    if j in tab :
       # On l'enleve : pas premier
       tab.remove(j)
    # On passe au prochain multiple
    j += n
```

2.5.3 Les chaînes de caractères

En python, les chaînes de caractères sont délimitées par ' ou ". Il est également possible de définir une chaîne sur plusieurs lignes à l'aide des délimitateurs ''' ou """.

```
mot1 = "L'aile de la mouette"
mot2 = '"Courage fuyons", disez le general'
phrase = '''Il dit non avec la tete
mais il dit oui avec le coeur
il dit oui a ce qu'il aime
il dit non au professeur'''
```

Les chaînes de caractères ne sont pas des objets modifiables.
De nombreuses méthodes sont associées avec les chaînes de caractères :

– `lower` : renvoie la chaîne avec toutes les lettres en minuscule

```
print phrase.lower()
```

– `upper` : Renvoie la chaîne avec toutes les lettres en majuscule

```
print phrase.upper()
```

– `swapcase` : change les majuscules en minuscules et vice versa

```
print phrase.swapcase()
```

– `count` : Retourne le nombre d'occurence d'une sous-chaîne dans la chaîne

```
print phrase.count('il') # Renvoie 4 (le 'Il' avec majuscule au I ne compte pas)
```

– `find` : Retourne le plus petit indice où se trouve une sous-chaîne (-1 si la sous-chaîne n'existe pas dans la chaîne) :

```
print phrase.find('il') # Renvoie 30 (position du premier 'il')
```

– `replace` : Remplace dans chaîne toute occurence d'une sous-chaîne par une autre

```
print phrase.replace('il','elle')
```

– `center` : Retourne la chaîne de caractère centrée dans une chaîne de n caractères

```
print mot1.center(72)
```

Beaucoup d'autres méthodes existent. Pour savoir lesquelles, on peut taper la commande suivante :

```
dir(phrase) # Affiche toutes les methodes disponibles pour phrase
help(phrase.splitlines) # Affiche une aide pour la methode splitlines
```

Notons qu'en Python, comme pour l'instruction `print`, il est possible de formater une chaîne de caractère à l'aide de l'opérateur % :

```
an = 72
s  = "L'annee est %04d"%an
```

2.5.4 Les dictionnaires

Ce sont des tableaux associatifs (en réalité des tables de hashage) dont les clefs doivent être des objets non modifiables et les les valeurs des objets modifiables.

La création se fait à l'aide du symbol { et } :

```
dicovide = {} # Creer un dictionnaire vide
numbers = { 'rodeur' : 0, 'general' : 2, 'prisonnier' : 6}
```

Pour accéder ou créer un nouveau élément :

```
print numbers['general'] # affiche le num. du general
numbers['BigBrother'] = 1 # Rajoute un nouveau element
```

Pour accéder à l'ensemble des clefs, on utilise la méthode `keys` :

```
print numbers.keys()
```

Pour l'ensemble des valeurs :

```
print numbers.values()
```

Pour l'ensemble des paires (clef, valeurs) :

```
print numbers.items()
```

On peut tester directement si le dictionnaire contient une certaine clef :

```
print numbers.has_key('moi') # Mmmmh, renvoie faux...
```

Pour connaître là encore les méthodes associées à un dictionnaire, on peut utiliser les instructions `dir` et `help` (Comme pour tout objet python...).

2.6 Les fonctions

2.6.1 Les généralités

Les fonctions sont définies à l'aide de l'instruction `def`. La syntaxe est `def function(parameters) : bloc`.

Contrairement au C, on peut retourner plusieurs valeurs d'une fonction :

```
def divmod(a,b) :
  return a/b,a%b

quotient,reste = divmod(11,3)
print '11/3=%d, 11 mod 3 = %d'%(quotient,reste)
```

Il est possible de définir des valeurs par défaut pour les paramêtres d'une fonction :

```
def zoom(u,a=2) :
  return a*u

zoom(1.5,4)       # Classique
zoom(1.5)         # Avec valeurs par defaut pour a
zoom(u=3.5)       # On nomme le parametre dont on donne la valeurs
zoom(a=3,u=1./3) # Idem pour les deux parametres.
# A noter que dans ce cas,l'ordre n'a pas d'importance.
zoom(3.5,a=4)     # Passage mixte des parametres.
```

Les fonctions sont des variables comme les autres. Il est parfaitement possible de passer une fonction en tant que paramêtre d'une autre fonction, ou bien affecté une fonction à une variable.

Par exemple, il est possible de passer un paramêtre un produit matrice–vecteur à un solveur itératif.

2.6.2 Les fonctions anonymes

Ce sont des fonctions `lambda`, qui ne portent pas de nom. Elles ne peuvent contenir qu'une seule expression. Par exemple :

```
sort = lambda s1,s2 : cmp(len(s1),len(s2))
sort('tot', 'toto') # Renvoie -1
```

2.7 La documentation en ligne

Il est possible de fournir une documentation accessible par l'interpréteur Python pour une fonction, un module ou une classe. Elle se met en première ligne de la fonction/classe/module et peut être ensuite récupérée par la commande `help`.

Exemple :

```
def add(a,b) :
  '''renvoie la somme de a et b'''
  return a+b

help(add)
```

3 Les classes

3.1 Définition

Une classe permet de définir un nouveau type permettant de créer des instances (des objets/variables) contenant des données et des méthodes (fonctions) associées à ces données.

Comme dans tout langage orienté objet, il est nécessaire de définir la manière dont on initialise un objet de cette classe, puis définir les différentes opérations/fonctions permises avec cet objet. Pour accéder aux donnés d'un objet dans une de ses méthodes, on utilise le mot clef self.

Exemple :

```
# On definit une nouvelle classe gerant des
# vecteurs en 3 dimensions :
import math
class Vecteur :
    '''Vecteur pour geometrie en trois dimensions'''
    # Constructeur : decrit comment initialiser un vecteur a trois dimensions :
    def __init__(self,x,y,z=0) :
      self.x = x
      self.y = y
      self.z = z
    # On definit la fonction calculant la norme cartesienne du vecteur :
    def norm(self) :
      'Retourne la norme cartesienne du vecteur'
      return math.sqrt(self.x**2 + self.y**2 + self.z**2)
    # Redefinition de l'operateur + pour les vecteurs
    def __add__(self,v) :
```

```
        return Vecteur(self.x+v.x,self.y+v.y,self.z+v.z)
    # Redefinition de l'operateur * pour le produit scalaire
    def __mul__(self,v) :
        return self.x*v.x+self.y*v.y+self.z*v.z
    # Permet d'acceder en lecture a x,y ou z a l'aide d'indice :
    def __getitem__( self, i ) :
        if i==0 : return self.x
        if i==1 : return self.y
        if i==2 : return self.z
    # Permet d'acceder en ecriture a x,y ou z a l'aide d'indice
    def __setitem__( self, i, v ) :
        if   i==0 : self.x = v
        elif i==1 : self.y = v
        elif i==2 : self.z = v
    # Permet d'afficher le vecteur a l'aide de print
    def __str__(self) :
        return '(%7.5g, %7.5g, %7.5g)'%(self.x,self.y,self.z)

u = Vecteur(1,3,5)
dir(u)
help(u)
help(u.norm)
print u.norm()
# change la composante y de u
u[1] = 2
# Affiche u
print u
print u.x # Affiche l'abscisse de u
v = Vecteur(2,4,7)
w = u + v # w recoit la somme de u et v
dot = u*v
print dot
```

Il est possible de vérifier à chaque instant si une variable est bien une instance d'une classe spécifique :

```
isinstance(u,Vecteur)   # Renvoie vrai
isinstance(dot,Vecteur) # Renvoie faux
```

3.2 Héritage

Comme tous les langages objets, Python accepte l'héritage (multiple également). L'héritage permet de spécialiser une classe pour un sous-type spécifique.

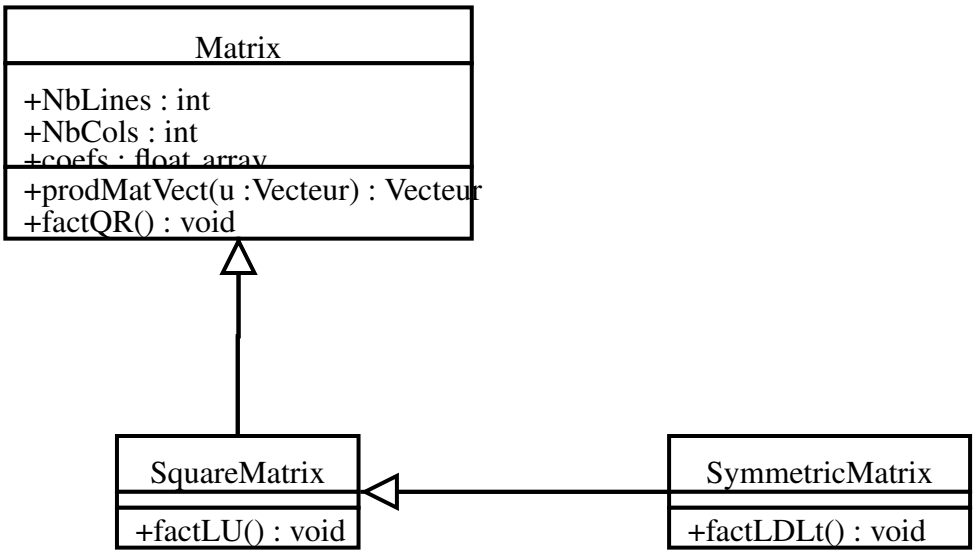
Par exemple, une matrice carrée est une matrice et une matrice symétrique est une matrice carrée.

Il existe un grand nombre d'opérations communes entre les matrices quelconques, les matrices carrées et les matrices symétriques (addition, multiplication, factorisation QR), mais certaines opérations ne peuvent se faire que sur les matrices carrées (factorisation LU) ou même symétriques (factorisation LDL^t).

Afin de ne pas redéfinir les opérations communes à ses trois types de matrice, on définit une hiérarchie telle que les matrices carrées puissent utilisées les méthodes définies dans les matrices quelconques et que les ma-

trices symétriques puissent utilisées les méthodes définies dans les matrices carrées et les matrices quelconques.

Pour cela on utilise le mécanisme d'héritage. On fait hériter le type matrice quelconque au type matrice carrée et le type matrice carrée au type matrice symétrique.


Matrix
+NbLines : int
+NbCols : int
+coefs : float array
+prodMatVect(u :Vecteur) : Vecteur
+factQR() : void
SquareMatrix
+factLU() : void
SymmetricMatrix
+factLDLt() : void

En python, cela se traduira par :

```
import numpy
class Matrix :
   "Definition d'une matrice rectangulaire quelconque"
   def __init__(self,nbLines, nbCols, coefs=None)
     self.nbLines = nbLines
     self.nbCols  = nbCols
     if coefs :
       self.coefs = coefs
     else :
       self.coefs = numpy.empty((nbLines,nbCols),numpy.double)

   def prodMatVect(self, u) :
     v = numpy.zeros((self.nbLines),numpy.double)
     for i in xrange(0,self.nbLines) :
       for j in xrange(0,self.nbCols) :
         v[i] += self.coefs[i,j]*u[j]
     return v

   def factQR(self) :
     # Normalement, ici factorisation QR
     # Pas encore programme ;)
     print 'Factorisation QR'

class SquareMatrix(Matrix) :
   "Definition d'une matrice carree"
   def __init__(self, dim, coefs) :
     Matrix.__init__(self,dim,dim,coefs)

   def factLU(self) :
     # Ici, algo de factorisation de Gauss
     print 'Factorisation LU'

class SymmetricMatrix(SquareMatrix) :
    """
       Definition d'une matrice symmetrique
       On ne stocke que la partie triangulaire inferieure
    """
    def __init__(self, dim, coefs) :
      self.nbLines = dim
      self.nbCols  = dim
      if coefs :
        self.coefs = coefs
      else :
        self.coefs = numpy.empty((dim*(dim+1)/2),numpy.double)

     def prodMatVect(self, u) :
       # On adapte le produit matrice vecteur
       # a la matrice symmetrique
       v = numpy.zeros((self.nbLines),numpy.double)
       for i in xrange(0,self.nbLines) :
         # Pour les coefficients hors diagonale
         for j in xrange(0,i) :
           v[i] += self.coefs[i*(i+1)/2+j]*u[j]
           v[j] += self.coefs[i*(i+1)/2+j]*u[i]
         # Prise en compte de la diagonale
         v[i] += self.coefs[i*(i+1)/2+i]*u[i]
       return v

     def factLDLt(self) :
       # Normalement, factorisation LDLt a programmer
       print 'Factorisation LDLt'
```

4 Les exceptions

4.1 Principe

Une exception se définit par :
- Une exception correspond à une erreur, ou une nécessité d'arrêter le traitement en cours
- Elle se propage, remontant la pile d'appel, jusqu'à être interceptée
- En Python, une exception est une classe héritant d'`Exception`

Plusieurs types d'exceptions (d'erreurs) sont fournis avec Python :
- Génériques : `RuntimeError`
- Liées au code : `SyntaxError, NameError, AttributeError`
- Liées au système : `SystemExit, IOError, KeyboardInterrupt`

4.2 Récupérer les exceptions

Le bloc `try -- except` :
- Du code pouvant générer (directement ou non) une exception peut être entouré d'un bloc `try :` et d'un bloc `except :`
- Le code présent dans le bloc de `try` sera exécuté jusqu'à ce qu'une exception arrive
- Le code se trouvant dans le bloc de `except` ne sera exécuté que si une exception arrive
- Le `except` stoppe la propagation de l'exception.

Nettoyer après une erreur – La clause `finally` :
- Lorsqu'une erreur survient, on peut avoir besoin de nettoyer (par exemple, effacer un fichier temporaire)
- Le code de la clause `finally` sera exécuté systématiquement après le reste
- La clause `finally` ne stoppe pas la propogation des exceptions

Autres modes de récupération - La clause `except` avec un type :
- Un type peut être précisé dans la clause `except`, pour ne récupérer qu'un type d'exceptions
- Toutes les exceptions héritant de ce type seront aussi interceptées
- Il est possible d'avoir plusieurs blocs `except` de suite

Récupérer l'objet exception :
- Une syntaxe du type `except IoError, exc :` permet de récupérer l'objet exception
- Des méthodes du module `sys` le permettent aussi.

4.3 Déclencher une exception

La clause `raise` :

– Déclencher une exception se fait avec `raise`
– Exemple :

```
if i < 0 :
  raise RuntimeError, 'i doit etre positif'
```

– Dans la clause `except`, un `raise` sans paramêtre relance l'exception courante.

Définir une exception personalisée – se fait en héritant de la classe `Exception`.

Un exemple :

```
try:
  z = x/y
except ZeroDivisionError :
  z = 0
except Exception, e:
  log("Erreur inattendue : "+str(e))
  raise
```

5 Modules et packages

5.1 Utilisation de modules et packages

La clause `import` :

– Syntaxe : `import package`
– Permet d'importer un module externe
– On peut renommer le package à l'aide de `as`

Exemples :

```
import os
import os.path as ospath
print os.getenv('PATH')
print ospath.join('home', '.bashrc')
```

La clause `from` :

– Permet d'importer directement des symboles dans l'espace courant
– Permet de n'importer que ce dont on a besoin
– Exemple : `from os import path`
– Il est possible, mais déconseillé d'utiliser $\star$

Importation avancée :

Les variables `syspath` et `PYTHONPATH` permettent de définir les chemins où on peut trouver les modules.

5.2 Création de modules et de packages

5.2.1 Création de modules

– Un module est un fichier Python définissant des symboles
– Le code au début de ce fichier est exécuté à l'importation

– Un module peut en importer d'autres, mais pas de manière circulaire

5.2.2 Création de packages

– Un package est un répertoire contenant plusieurs modules (et éventuellement d'autres packages)
– Il doit contenir un fichier `__init__.py`.

5.3 La bibliothèque standard

5.3.1 Aperçu

Une bibliothèque fournie :
– Python contient une bibliothèque standard très riche
– Des extensions très riches sont proposés en libre : modules pour le calcul numérique (numpy, scipy, . . .), traitement d'images (PIL, . . .), etc.

Des ressources en ligne :
– Sur le site de python : `www.python.org` (et non .com !) :
– Des liens vers de nombreux modules
– De la documentation en ligne

Quelques packages de la bibliothèque stantard :
– `sys, imp, gc` : pour modifier le comportement interne de Python
– `string, re` : manipulation de chaînes et expressions régulières
– `os, glob, commands` : Pour l'accès à l'OS
– `math, random` : Fonctions mathématiques et nombres aléatoires
– Des modules pour le réseau
– Sous Windows, des modules pour appeler des applications externes (Word, etc.)