Python numérique: Numeric et NumPy

Konrad HINSEN

Centre de Biophysique Moléculaire (Orléans)

et

Synchrotron Soleil (St Aubin)

Numeric

Fonctionnalité de base pour le calcul numérique:

- Tableaux multidimensionnels
- Arithmétique et fonctions mathématiques sur les tableaux
- Algèbre linéaire (LAPACK)
- Transformée de Fourier (FFTPACK)
- Nombres aléatoires

➥ Implémentation efficace qui rend possible le calcul numérique en Python pur

Numeric, numarray, NumPy

Numeric 1996

Première implémentation de tableaux

numarray 2004

Meilleure performance pour les très grands tableaux

NumPy 2006

Evolution de Numeric avec intégration d'améliorations de numarray

➥ à long terme, seul NumPy va rester
mais à ce moment, c'est encore Numeric qui domine

Autres bibiliothèques scientifiques généralistes

ScientificPython

Modules Python d'intérêt général:

- **Géométrie**
  vecteurs, tenseurs, transformations linéaires
- **Fonctions**
  interpolation, dérivés automatiques, polynômes, fonctions rationnelles
- **Statistique**
  moments d'une distribution, histogramme
- **Fits**
  moindres carrés linéaire et non-linéaire
- **Unités**
  conversion, arithmétique
- **Visualisation**
  VRML, VPython, VMD
- **Parallélisme**
  calcul distribué, BSP, interface MPI
- **Entrées-Sorties**
  formatage compatible Fortran, tableaux, netCDF, PDB

SciPy

Interfaces à des nombreuses bibliothèques scientifiques:

- Statistique
- Optimisation
- Intégration numérique
- Algèbre linéaire
- Transformés de Fourier
- Traitement du signal
- Traitement d'images
- Algorithmes génétiques
- Équations différentielles partielles
- Fonctions spéciales

Les Tableaux

Tableaux

- multidimensionnels
- tous les élements du même type
- stockage compact des données, compatible C/Fortran
- opérations efficaces
- arithmétique
- indexation flexible
- indexation crée des références

Structure d'un tableau

a = arange(1, 7)

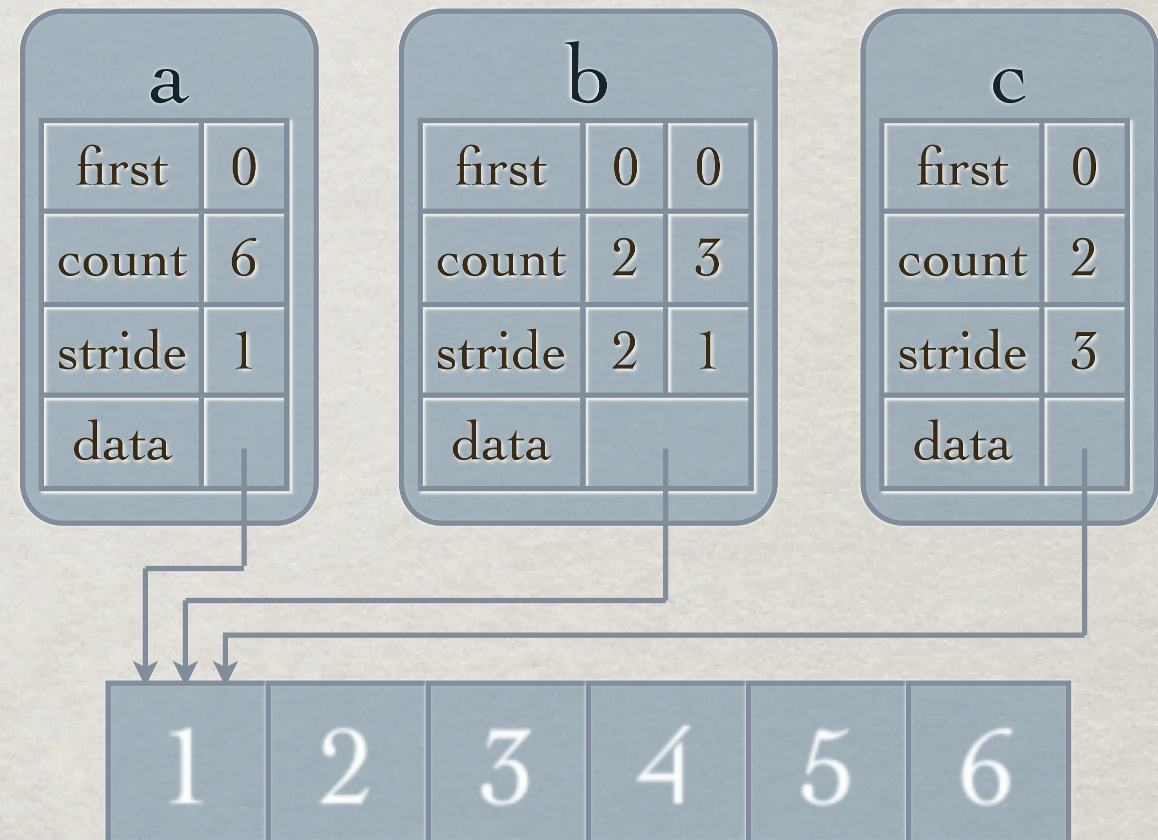
```
array([1, 2, 3, 4, 5, 6])
```

b = reshape(a, (3, 2))

```
array([[1, 2],
       [3, 4],
       [5, 6]])
```

c = a[::3]

```
array([1, 4])
```



Structure d'un tableau

a = arange(1, 7)

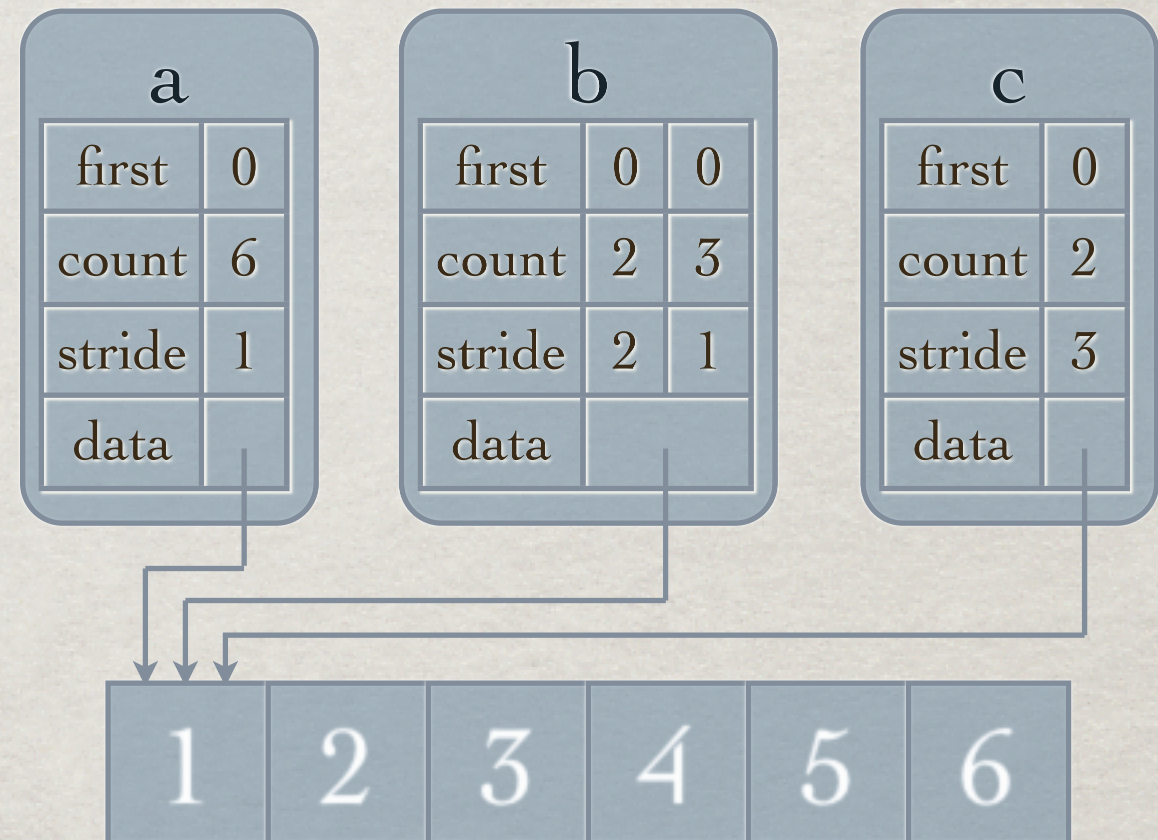
```
array([1, 2, 3, 4, 5, 6])
```

b = reshape(a, (3, 2))

```
array([[1, 2],
       [3, 4],
       [5, 6]])
```

c = a[::3]

```
array([1, 4])
```



Attention :

a[0] = 42

Structure d'un tableau

a = arange(1, 7)

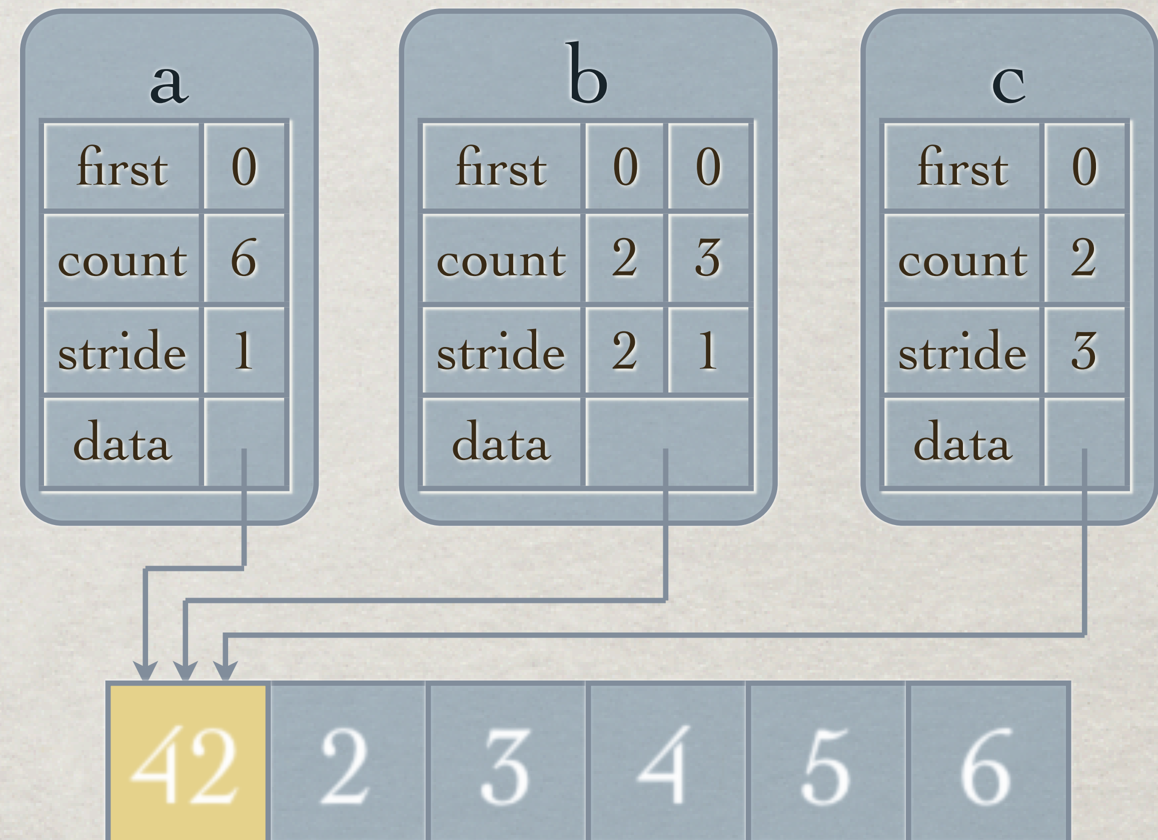
```
array([1, 2, 3, 4, 5, 6])
```

b = reshape(a, (3, 2))

```
array([[1, 2],
       [3, 4],
       [5, 6]])
```

c = a[::3]

```
array([1, 4])
```



Attention :

a[0] = 42

Structure d'un tableau

a = arange(1, 7)

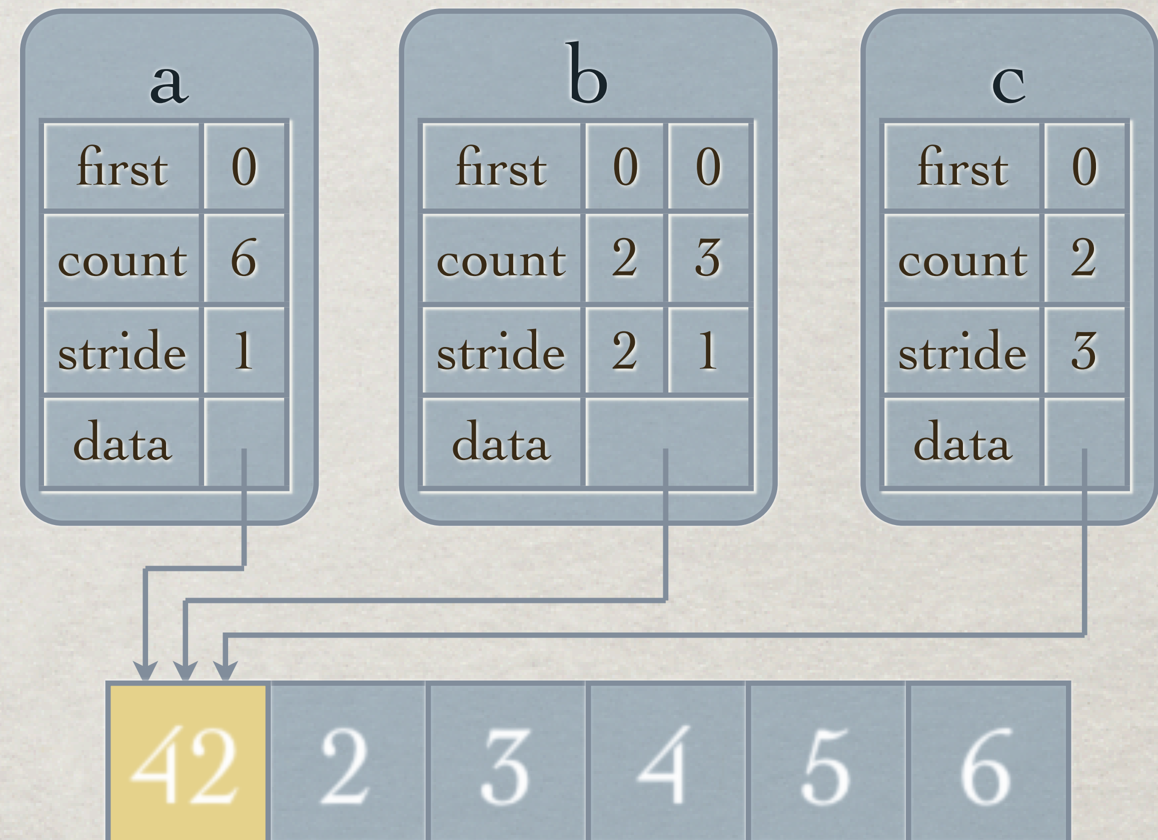
```
array([1, 2, 3, 4, 5, 6])
```

b = reshape(a, (3, 2))

```
array([[1, 2],
       [3, 4],
       [5, 6]])
```

c = a[::3]

```
array([1, 4])
```



Attention :

a[0] = 42

print c

```
array([42, 4])
```

Création de tableaux

- array([[1, 2], [3, 4]])

```
array([[1, 2],
       [3, 4]])
```

- zeros((2, 3), Float) NumPy: zeros((2, 3), float)

```
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
```

- arange(0, 10, 2)

```
array([0, 2, 4, 6, 8])
```

- arange(0., 0.5, 0.1)

```
array([ 0. ,  0.1,  0.2,  0.3,  0.4])
```

Attention aux effets d'arrondi !

Indexation

a = arange(6)

```
array([0, 1, 2, 3, 4, 5])
```

- a[2]

```
2
```

- a[2:4]

```
array([2, 3])
```

- a[1:-1]

```
array([1, 2, 3, 4])
```

- a[:4]

```
array([0, 1, 2, 3])
```

- a[1:4:2]

```
array([1, 3])
```

- a[::-1]

```
array([5, 4, 3, 2, 1, 0])
```

Indexation

a = array([[1, 2], [3, 4]])

```
array([[1, 2],
       [3, 4]])
```

- a[1, 0]

```
3
```

- a[1, :]   a[1]

```
array([3, 4])
```

- a[:, 1]

```
array([2, 4])
```

- a[:, :, NewAxis]   NumPy: a[:, :, newaxis]

```
array([[[1],
        [2]],

       [[3],
        [4]]])
```

Arithmétique

a = array([[1, 2], [3, 4]])   a.shape = (2, 2)

```
array([[1, 2],
       [3, 4]])
```

- a + a

```
array([[2, 4],
       [6, 8]])
```

- a + 1

```
array([[2, 3],
       [4, 5]])
```

- a + array([10, 20])   array([10, 20]).shape = (2,)

```
array([[11, 22],
       [13, 24]])
```

- a + array([[10], [20]])   array([[10], [20]]).shape = (2, 1)

```
array([[11, 12],
       [23, 24]])
```

Arithmétique

c = a + b avec a.shape==(2, 3, 1) et b.shape==(3, 2)

1) len(a.shape) > len(b.shape)

   ➥ b → b[NewAxis, :, :], b.shape → (1, 3, 2)

2) Comparer a.shape et b.shape élément par élément:
   - a.shape[i] == b.shape[i]: tout va bien
   - a.shape[i] == 1: répéter a b.shape[i] fois
   - b.shape[i] == 1: répéter b a.shape[i] fois
   - sinon : erreur

3) Faire la somme élément par élément

4) c.shape == (2, 3, 2)

Arithmétique

x = a + b + c + d

- temp1 = a + b
- temp2 = temp1 + c
- x = temp2 + d

➥ création de deux tableaux temporaires
et d'un tableau résultat

➥ perte de performance pour les grands tableaux

Plus efficace :

```
a += b
a += c
a += d
```

Optimisation à faire en cas de besoin seulement !

Arithmétique

a + b ⇔ add(a, b)

- add(a, b, c) met la somme dans le tableau c
  ➥ pas de création de tableau temporaire
- add.reduce(a) fait la somme sur la première dimension
- add.accumulate(a) fait les sommes intermédiaires

```
a = arange(5)
    array([0, 1, 2, 3, 4])
add.reduce(a)
    10
add.accumulate(a)
    array([ 0,  1,  3,  6, 10])
```

Opérations structurales

a = (1 + arange(4))**2

```
array([ 1,  4,  9, 16])
```

- take(a, [2, 2, 0, 1])

```
array([9, 9, 1, 4])
```

- where(a >= 2, a, -1)

```
array([-1,  4,  9, 16])
```

- reshape(a, (2, 2))

```
array([[ 1,  4],
       [ 9, 16]])
```

- resize(a, (3, 5))

```
array([[ 1,  4,  9, 16,  1],
       [ 4,  9, 16,  1,  4],
       [ 9, 16,  1,  4,  9]])
```

- repeat(a, [2, 0, 2, 1])

```
array([ 1,  1,  9,  9, 16])
```

Autres fonctionnalités

Fonctions

arccos, arcsin, arctan, arctan2, ceil, cos, cosh, exp, fabs, floor, fmod, hypot, log, log10, sin, sinh, sqrt, tan, tanh

Constantes : π, e

Trois sources :

- Module `math` : réel seulement
- Module `cmath` : réel et complexe
- Module `Numeric/numpy` :

  réel, complexe, tableaux, instances de classes

Fonctions

```
from Numeric import *

class Nombre:

    def __init__(self, valeur):
        self.valeur = valeur

    def __repr__(self):
        return ”Nombre(%s)” % repr(self.valeur)

    def sqrt(self):
        return Nombre(sqrt(self.valeur))

print sqrt(Nombre(25.))
```

```
Nombre(5.0)
```

Applications:

- Scientific.Functions.Interpolation (fonction sur une grille)
- Scientific.Functions.Derivatives (dérivés automatiques)

Algèbre Linéaire

Opérations:

- équations linéaires, inversion
- valeurs propres, vecteurs propres
- norme, déterminant
- valeurs singulières
- moindres carrés, pseudo-inverse

Matrices générales seulement

Pour en savoir plus :

pydoc LinearAlgebra

ou

pydoc numpy.linalg

Transformées de Fourier

Opérations:

- FFT 1d complexe générale
- FFT 1d réelle
- FFT 1d hermite
- FFT 2d complexe et réelle
- FFT Nd complexe et réelle

Pour en savoir plus :

pydoc FFT

ou

pydoc numpy.fft

Nombres aléatoire

Opérations:

- nombre aléatoire individuel ou tableau
- distribution uniforme
- distribution gaussienne
- distribution exponentielle
- beaucoup plus de distributions dans `numpy`

Pour en savoir plus :

pydoc RNG

ou

pydoc numpy.random

Optimisation : Les Tableaux en Pyrex

Pyrex

- compilateur Python qui génère des modules d'extension en C
  ➥ accélération de 5% au mieux !

- extensions au langage Python pour écrire du C en syntaxe Python
  ➥ programmation mixte Python/C

Applications :

- optimiser une fonction Python
  en la traduisant progressivement en C
- écrire des modules d'extension
- écrire des interfaces à des bibliothèques C à la main

Exemple : Python

```python
def exp(x, termes = 50):
    somme = 0.
    puissance = 1.
    fact = 1.
    for i in range(termes):
        somme = somme + puissance/fact
        puissance = puissance*x
        fact = fact*(i+1)
    return somme
```

Exemple : Pyrex

Conversion automatique Python->C

```
def exp(double x, int termes = 50):
    cdef double somme
    cdef double puissance
    cdef double fact
    cdef int i
    somme = 0.
    puissance = 1.
    fact = 1.
    for i from 0 <= i < termes:
        somme = somme + puissance/fact
        puissance = puissance*x
        fact = fact*(i+1)
    return somme
```

Déclarations de variables en C

Boucle en C

Conversion automatique C->Python

Tableaux en Pyrex

```
include 'numeric.pxi'

def array_sum(ArrayType a):
    cdef double *data
    cdef double sum
    cdef int i
    assert a.nd == 1
    data = <double *>a.data
    sum = 0.
    for i from 0 <= i < a.dimensions[0]:
        sum = sum + data[i]
    return sum
```

Vérification du type Python

Déclarations de variables en C

Conversion automatique C->Python

Boucle en C

Conversion automatique C->Python