NumPy avancé

Konrad HINSEN

Centre de Biophysique Moléculaire, Orléans
et
Synchrotron SOLEIL, Saint Aubin

3 décembre 2013

Packaging avec NumPy

Vérifier si NumPy est installé (dans `setup.py`)

```
try:
    import numpy
except ImportError:
    numpy = None
```

Tester la version

```
if [int(s) for s in numpy.__version__.split('.')] \
  >= [1, 5, 0]:
    pass
```

Indiquer une dépendance de NumPy (setuptools/distribute)

```
setup(...
      install_requires = ['numpy >1.1'],
      ...)
```

Packaging avec NumPy

Où se trouve numpy/arrayobject.h ?

```
1 import numpy.distutils.misc_util
2 numpy_include_dirs = numpy.distutils.misc_util.\
3                         get_numpy_include_dirs()
```

Pour les modules C/Cython qui utilisent NumPy:

```
1 Extension(...
2           include_dirs=numpy_include_dirs,
3           ...)
```

Vues

Une *vue* est un tableau qui fait référence aux éléments d'un autre tableau au lieu d'avoir son propre espace mémoire. L'indexation crée des vues, tout comme `reshape` et quelques autres méthodes.

Vues

Une *vue* est un tableau qui fait référence aux éléments d'un autre tableau au lieu d'avoir son propre espace mémoire. L'indexation crée des vues, tout comme `reshape` et quelques autres méthodes.

Une vue a un attribut `base` dont la valeur est le tableau initial (qui peut lui-même être une vue). Pour un tableau qui n'est pas une vue, `base` est `None`.

Vues

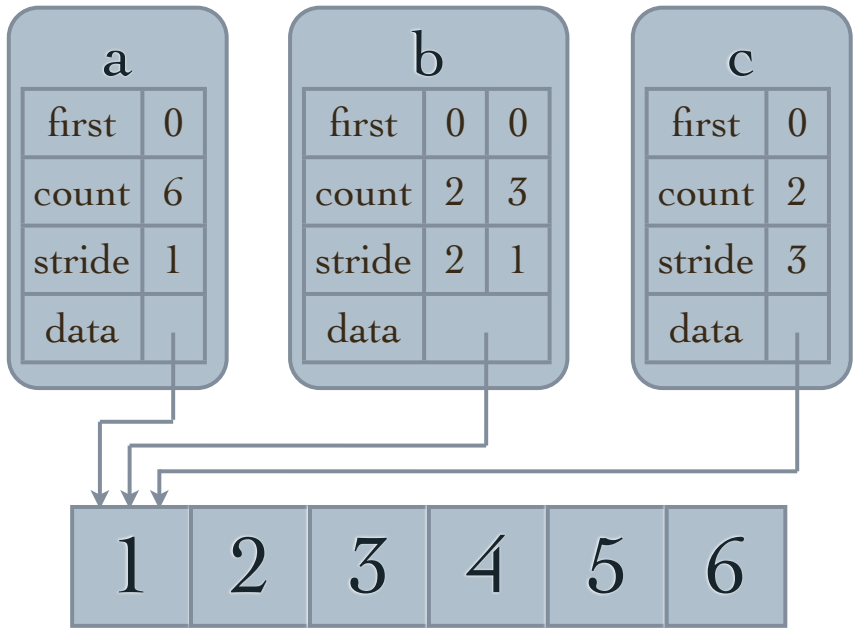
Une *vue* est un tableau qui fait référence aux éléments d'un autre tableau au lieu d'avoir son propre espace mémoire. L'indexation crée des vues, tout comme `reshape` et quelques autres méthodes.

Une vue a un attribut `base` dont la valeur est le tableau initial (qui peut lui-même être une vue). Pour un tableau qui n'est pas une vue, `base` est `None`.

Attention: Les éléments d'un tableau peuvent être modifiés en passant par n'importe quelle vue directe ou indirecte. Il n'est pas possible de vérifier s'il existe des vues pour un tableau donné.

La structure des tableaux dans la mémoire

```
>>> import numpy as np
>>> a = np.arange(1,7)
>>> a
array([1, 2, 3, 4, 5, 6])
>>> b = a.reshape((3, 2))
>>> b
array([[1, 2],
       [3, 4],
       [5, 6]])
>>> c = a[::3]
>>> c
array([1, 4])
```




```
>>> a[0] = 42
>>> c
array([42,  4])
```

Vues et gestion de la mémoire

Tant qu'au moins une vue existe, un tableau ne peut pas être détruit.

Après

```
gros = N.arange(100000000)
petit = gros[:5]
del gros
```

le gros tableau reste en mémoire jusqu'à la disparation du petit.

Mieux vaut faire une copie:

```
gros = N.arange(100000000)
petit = gros[:5].copy()
del gros
```

Analyse d'un tableau

Tableau a

```
>>> a.dtype
dtype('int64')
>>> a.itemsize
8
>>> a.strides
(8,)
>>> a.__array_interface__ \
                ['data'][0]
4314532672
>>> a.flags
  C_CONTIGUOUS : True
  F_CONTIGUOUS : True
  OWNDATA : True
  WRITEABLE : True
  ALIGNED : True
  UPDATEIFCOPY : False
>>> a.base is None
True
```

Tableau b

```
>>> b.dtype
dtype('int64')
>>> b.itemsize
8
>>> b.strides
(16, 8)
>>> b.__array_interface__ \
                ['data'][0]
4314532672
>>> b.flags
  C_CONTIGUOUS : True
  F_CONTIGUOUS : False
  OWNDATA : False
  WRITEABLE : True
  ALIGNED : True
  UPDATEIFCOPY : False
>>> b.base is a
True
```

Tableau c

```
>>> c.dtype
dtype('int64')
>>> c.itemsize
8
>>> c.strides
(24,)
>>> c.__array_interface__ \
                ['data'][0]
4314532672
>>> c.flags
  C_CONTIGUOUS : False
  F_CONTIGUOUS : False
  OWNDATA : False
  WRITEABLE : True
  ALIGNED : True
  UPDATEIFCOPY : False
>>> c.base is a
True
```

Tableaux C et Fortran

C

```
>>> C = N.array([[1, 2],
                 [3, 4]],
                order='C')
>>> C
array([[1, 2],
       [3, 4]])
>>> C.strides
(16, 8)
>>> C.flags
  C_CONTIGUOUS : True
  F_CONTIGUOUS : False
  OWNDATA : True
  WRITEABLE : True
  ALIGNED : True
  UPDATEIFCOPY : False
```

Fortran

```
>>> F = N.array([[1, 2],
                 [3, 4]],
                order='F')
>>> F
array([[1, 2],
       [3, 4]])
>>> F.strides
(8, 16)
>>> F.flags
  C_CONTIGUOUS : False
  F_CONTIGUOUS : True
  OWNDATA : True
  WRITEABLE : True
  ALIGNED : True
  UPDATEIFCOPY : False
```

Utilisation de la mémoire

Un des inconvénients de NumPy est l'allocation d'un grand nombre de tableaux temporaires. Une expression simple comme `c = 2*a + 3*b` est en fait traitée comme

```
1 tmp1 = 2*a
2 tmp2 = 3*b
3 c = tmp1 + tmp2
```

La création de trois tableaux, dont deux sont presque immédiatement supprimés, est coûteuse en temps, surtout pour des petits tableaux, et coûteuse en espace mémoire, surtout pour les grand tableaux.

Une autre inconvénient est la mauvaise utilisation du cache du processeur pour ce genre d'opération.

Arithmétique en place

```
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> a *= 2
>>> a
array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18])
>>> np.multiply(a, 2, a)
array([ 0,  4,  8, 12, 16, 20, 24, 28, 32, 36])
>>> a
array([ 0,  4,  8, 12, 16, 20, 24, 28, 32, 36])
>>> b = np.zeros(a.shape, a.dtype)
>>> np.multiply(a, 2, b)
array([ 0,  8, 16, 24, 32, 40, 48, 56, 64, 72])
>>> a
array([ 0,  4,  8, 12, 16, 20, 24, 28, 32, 36])
>>> b
array([ 0,  8, 16, 24, 32, 40, 48, 56, 64, 72])
```

numexpr

La bibliothèque `numexpr` évalue des expressions contenant des tableaux de façon optimisée en les compilant en langage C:

- élimination des tableaux intermédiaires
- accès aux tableaux optimisé pour utilise le cache du processeur
- utilisation de processeurs multiples

Exemple

```
1 import numpy as np
2 import numexpr
3 >>> a = 3.*np.arange(10000)-2.
4 >>> b = 0.1*np.arange(10000)**2
5 >>> numexpr.evaluate("3*a**2-5.*b")
6 array([  1.20000000e+01,   2.50000000e+00,
  4.60000000e+01, ...,
7         2.64805036e+09,   2.64858019e+09,   2.64911007e+09])
```

Les tableaux structurés

Le type des éléments d'un tableau (`dtype`) est normalement un type "simple" (`numpy.int`, `numpy.float32`, `numpy.complex64`, ...). Mais le `dtype` peut aussi être:

- un tuple
- une structure avec des champs nommés
- un tableau

Applications:

- stockage compact de données hétérogènes
- implémenter des modèles de données plus structurés
- interfacer avec des `structs` en C
- lire/écrire des fichiers HDF5

Éléments tuples

```
>>> x = np.zeros((2,),dtype=('i4,f4,a10'))
>>> x[:] = [(1,2.,'Hello'),(2,3.,"World")]
>>> x
array([(1, 2.0, 'Hello'), (2, 3.0, 'World')],
      dtype=[('f0', '>i4'), ('f1', '>f4'), ('f2', '|S10')])
>>> x[1]
(2,3.,"World")
```

Éléments champs nommés

```
>>> x = np.zeros(3, dtype={'names':['col1', 'col2'],
                           'formats':['i4','f4']})
>>> x
array([(0, 0.0), (0, 0.0), (0, 0.0)],
      dtype=[('col1', '>i4'), ('col2', '>f4')])
>>> x[0]
(0, 0.0)
>>> x['col1']
array([0, 0, 0], dtype=int32)
```

Éléments tableaux

```
>>> x = np.zeros(3, dtype='3int8, float32, (2,3)float64')
>>> x['f0'] = array([1, 2, 3])
>>> x.shape
(3,)
>>> x['f0']
array([[1, 2, 3],
       [1, 2, 3],
       [1, 2, 3]], dtype=int8)
>>> x['f0'].shape
(3, 3)
>>> x[0]
([1, 2, 3], 0.0, [[0.0, 0.0, 0.0], [0.0, 0.0, 0.0]])
>>> x['f2'].shape
(3, 2, 3)
>>> x[1]['f2'].shape
(2, 3)
```