Une petite référence Numpy

(mise à jour la plus récente: 15 octobre 2013)

Les nouveaux programmes des CPGE scientifiques (rentrée 2013) comportent un enseignement d'informatique *pour tous*, et prévoient notamment l'utilisation du langage `Python`.

Ce document fait suite à « Une petite référence Python », dont la version la plus récente est sur mathprepa.fr

Il est consacré à une présentation assez complète (mais qui ne saurait être exhaustive) du module `numpy`.

Un petit mot personnel, au sujet de ce document.
J'ai voulu l'écrire pour mettre mes propres idées en place au sujet de `numpy`. Mais comment avoir une vue d'ensemble sans entrer dans les détails ? Et comment rendre compte en une soixantaine de pages d'une entité dont le manuel de référence en fait plus de mille ?

J'ai fait de mon mieux, avec un plan qui je l'espère « tient la route », malgré un coté « catalogue » un peu inévitable. Après peut-être une lecture linéaire de ce document, on préfèrera sans doute des accès choisis via la table des matières. J'ai fait en sorte d'accompagner les fonctions d'exemples suggestifs. Dans ce document électronique, les noms des fonctions sont eux-mêmes des liens vers la documentation officielle (en anglais).

Pour toute suggestion, on peut me contacter à mon adresse académique (ci-dessous, et c'est la bonne orthographe).

J'apprécierai tout retour, même si c'est pour me signaler des oublis, des erreurs, des fautes d'orthographe !
Les messages amicaux font toujours plaisir, bien sûr.

Jean-Michel Ferrard
Mathématiques, lycée Saint-Louis
44 Boulevard Saint-Michel,
75006, Paris

jean-miche.ferrard@ac-paris.fr

mathprepa.fr

Table des matières

jean-miche.ferrard@ac-paris.fr mathprepa.fr

Introduction

Le module `numpy` est la boîte à outils indispensable pour faire du calcul scientifique avec Python.

Pour modéliser les vecteurs, matrices, et plus généralement les tableaux à n dimensions, `numpy` fournit le type `ndarray`.

Il y a des différences majeures avec les listes (resp. les listes de listes) qui pourraient elles aussi nous servir à représenter des vecteurs (resp. des matrices) :

- Les tableaux `numpy` sont *homogènes*, c'est-à-dire constitués d'éléments du même type.
  On trouvera donc des tableaux d'entiers, des tableaux de flottants, des tableaux de chaînes de caractères, etc.
- La taille des tableaux `numpy` est fixée à la création. On ne peut donc augmenter ou diminuer la taille d'un tableau comme le ferait pour une liste (à moins de créer un tout nouveau tableau, bien sûr).

Ces contraintes sont en fait des avantages :

- Le format d'un tableau `numpy` et la taille des objets qui le composent étant fixés, l'empreinte du tableau en mémoire est invariable et l'accès à ses éléments se fait en temps constant.
- Les opérations sur les tableaux sont optimisées en fonction du type des éléments, et sont beaucoup plus rapides qu'elles ne le seraient sur des listes équivalentes.

Traditionnellement, on charge la totalité du module `numpy`, mais en le renommant en `np`, de la manière suivante :

```
>>> import numpy as np
>>>
```

Les fonctions de `numpy` sont alors accessibles par leur *nom qualifié* « `np.nom_de_la_fonction` ».

Important : dans toute la suite, on supposera que le module `numpy` a été importé de cette manière.
Quand on évoquera la fonction « `array` » par exemple, on pensera toujours à l'utiliser avec la syntaxe « `np.array` »

On ne chargera jamais `numpy` par « `from numpy import *` » : le nombre de fonctions importées est en effet trop important, et avec lui le risque d'homonymie avec les définitions déjà présentes au moment de l'importation.

jean-miche.ferrard@ac-paris.fr mathprepa.fr

Chapitre 1

Fonction `array` et « data types »

1.1 Création de tableaux avec `array`

On utilise en général la fonction `array` pour former un tableau à partir de la liste (ou la liste des listes) de ses éléments.

Dans l'exemple ci-dessous, on forme une matrice a à coefficients entiers, à trois lignes et quatre colonnes, à partir d'une liste de trois sous-listes de longueur quatre.

La fonction `array` agit comme un mécanisme de conversion vers `'numpy.ndarray'` qui est le type commun à tous les tableaux `numpy`.

On ne confondra pas avec le résultat renvoyé par l'attribut `dtype` (abréviation de *data type*) de a, et qui indique le type commun à tous les éléments de celui-ci (en l'occurrence des entiers codés sur 32 bits, c'est-à-dire quatre octets ou *bytes*).

```
>>> a = np.array([[8,3,2,4],[5,1,6,0],[9,7,4,1]]); a
array([[8, 3, 2, 4],
       [5, 1, 6, 0],
       [9, 7, 4, 1]])
>>> type(a)
<class 'numpy.ndarray'>
>>> a.dtype
dtype('int32')
```

Remarque : la *méthode* `tolist` d'un tableau `numpy` le transforme en la liste (éventuellement une liste de listes) de ses éléments. Attention, ce c'est pas une *fonction* du module `numpy`. On n'écrira donc pas `np.tolist(a)` mais `a.tolist()`

```
>>> a = np.arange(15).reshape(3,5); a
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
>>> b = a.tolist(); b # convertit a en une liste de listes
[[0, 1, 2, 3, 4], [5, 6, 7, 8, 9], [10, 11, 12, 13, 14]]
```

```
>>> c = np.array(b) # retour tableau
>>> c
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
```

1.2 Attributs de dimension d'un tableau

Voici quelques attributs de dimension des tableaux `numpy`
(les exemples s'appuient sur la définition précédente de a).

- `shape` indique le *format* du tableau, sous la forme du tuple du nombre d'éléments dans chaque direction : ici le couple $(3, 4)$ indique qu'il y a trois lignes et quatre colonnes.
- la *fonction* `alen` (ce n'est pas un *attribut*) donne la première dimension d'un tableau (la taille pour un vecteur, le nombre de lignes pour une matrice).
- `size` donne le nombre total d'éléments, et par exemple np pour une matrice de type $n \times p$. Ici notre tableau a contient 12 éléments.

  variante : `np.size(a,0)` et `np.size(a,1)` donnent respectivement le nombre de lignes et le nombre de colonnes d'une matrice.
- `ndim` renvoie le nombre d'indices nécessaires au parcours du tableau (usuellement : 1 pour un vecteur, 2 pour une matrice).

```
>>> a.shape
(3, 4)
>>> np.alen(a)
3
>>> a.size
12
>>> np.size(a,0)
3
>>> np.size(a,1)
4
>>> a.ndim
2
```

Les fonctions `shape`, `size`, et `ndim` peuvent être évoquées à la fois comme des *attributs* d'un tableau et comme des *fonctions* du module `numpy` prenant un tableau en argument. Pour prendre l'exemple de `shape`, on peut donc écrire aussi bien « `a.shape` » que « `np.shape(a)` ».

1.3 Le « data type » d'un tableau

Le *data type* est automatiquement choisi par la fonction `array` au vu des coefficients.

En modifiant à peine l'exemple précédent, il suffit d'un seul flottant (on a changé 4 en 4.0) pour que le tableau tout entier soit considéré comme formé de flottants (ici codés sur 64 bits).

```
>>> b = np.array([[8,3,2,4],[5,1,6,0],[9,7,4.0,1]]); b
array([[ 8.,  3.,  2.,  4.],
       [ 5.,  1.,  6.,  0.],
       [ 9.,  7.,  4.,  1.]])
>>> b.dtype
dtype('float64')
```

Il est toujours possible de forcer le *data type* du tableau avec l'option `dtype` de la fonction `array` (à condition de faire du *upcasting*, c'est-à-dire de forcer le typage « vers le haut », basé sur les inclusions strictes $\mathbb{Z} \subsetneq \mathbb{R} \subsetneq \mathbb{C}$).

Dans l'exemple suivant, on force la création d'un tableau de nombres complexes, alors que la nature des coefficients devait conduire à un *data type* entier (on aurait pu aussi bien remplacer `8` par `8+0j`, par exemple) :

```
>>> c = np.array([[8,3,2,4],[5,1,6,0],[9,7,4,1]], dtype=complex); c
array([[ 8.+0.j,  3.+0.j,  2.+0.j,  4.+0.j],
       [ 5.+0.j,  1.+0.j,  6.+0.j,  0.+0.j],
       [ 9.+0.j,  7.+0.j,  4.+0.j,  1.+0.j]])
>>> c.dtype
dtype('complex128')
```

1.4 Copie d'un tableau avec conversion de « data type »

On peut créer une copie d'un tableau d'un *data type* à un autre en utilisant sa méthode `astype`.
Pour illustrer ces possibilités on reprend les définitions précédentes de a (*dtype*=int), b (*dtype*=float) et c (*dtype*=complex).
On retiendra que la méthode `astype` crée une *copie* du tableau (donc n'affecte pas le contenu de la variable).

```
>>> a.astype(float)
array([[ 0.,  1.,  2.,  3.,  4.],
       [ 5.,  6.,  7.,  8.,  9.],
       [10., 11., 12., 13., 14.]])
```

effectue une copie du tableau a en convertissant vers le type float

```
>>> b.astype(int)
array([[8, 3, 2, 4],
       [5, 1, 6, 0],
       [9, 7, 4, 1]])
```

effectue une copie du tableau b en convertissant vers le type int

```
>>> c.astype(float)
array([[ 8.,  3.,  2.,  4.],
       [ 5.,  1.,  6.,  0.],
       [ 9.,  7.,  4.,  1.]])
```

effectue une copie du tableau c en convertissant vers le type float

Un tableau `numpy` occupe une place constante en mémoire, égale au produit du nombre de ses éléments par la taille (fixe) d'un élément du *data type* (en principe : 4 octets pour un entier, 8 pour un flottant, 16 pour un nombre complexe).

Les attributs `itemsize` et `nbytes` d'un tableau `numpy` donnent la taille d'un élément du tableau et la taille totale de celui-ci (exprimées en octets).

En réutilisant nos tableaux a, b, c, on affiche ici les valeurs des attributs `itemsize` (taille d'un élément), puis `size` (le nombre d'éléments) puis `nbytes` (taille totale).

un octet = un caractère = un *byte* = 8 bits.

```
>>> (a.itemsize, a.size, a.nbytes)
(4, 15, 60)

>>> (b.itemsize, b.size, b.nbytes)
(8, 12, 96)

>>> (c.itemsize, c.size, c.nbytes)
(16, 12, 192)
```

1.5 Les différents « data types » disponibles

Les éléments d'un tableau `numpy` particulier sont du même type, et on a vu les types suivants : `int`, `float` et `complex`.
En fait il y a d'autres types possibles et en voici la liste :

`bool` : booléen (True ou False), codé sur un octet

`int` : entier, équivalent de `int32` ou de `int64`, suivant les implémentations

`int8` : entier signé sur 1 octet (8 bits), intervalle $[\![-2^7, 2^7-1]\!] = [\![-128, 127]\!]$

`int16` : entier signé sur 2 octets (16 bits), intervalle $[\![-2^{15}, 2^{15}-1]\!] = [\![-32768, 32767]\!]$

`int32` : entier signé sur 4 octects (32 bits), intervalle $[\![-2^{31}, 2^{31}-1]\!] = [\![-2147483648, 2147483647]\!]$

`int64` : entier signé, 8 octets (64 bits), intervalle $[\![-2^{63}, 2^{63}-1]\!] = [\![-9223372036854775808, 9223372036854775807]\!]$

`uint8` : entier non signé, 1 octet (8 bits), intervalle $[\![0, 2^8-1]\!] = [\![0, 255]\!]$

jean-miche.ferrard@ac-paris.fr mathprepa.fr

`uint16` : entier non signé, 2 octets (16 bits), intervalle $[\![0, 2^{16}-1]\!] = [\![0, 65535]\!]$

`uint32` : entier non signé, 4 octets (32 bits), intervalle $[\![0, 2^{32}-1]\!] = [\![0, 4294967295]\!]$

`uint64` : entier non signé, 8 octets (64 bits), intervalle $[\![0, 2^{64}-1]\!] = [\![0, 18446744073709551615]\!]$

`float` : synonyme de `float64`

`float16` : flottant en demi-précision (un bit de signe, 5 bits d'exposant, 10 bits de mantisse)

`float32` : flottant en simple-précision (un bit de signe, 8 bits d'exposant, 23 bits de mantisse)

`float64` : flottant en double-précision (un bit de signe, 11 bits d'exposant, 52 bits de mantisse)

`complex` : synonyme de `complex128`

`complex64` : nombre complexe sur 64 bits (32 bits pour la partie réelle, 32 bits pour la partie imaginaire)

`complex128` : nombre complexe sur 128 bits (64 bits pour la partie réelle, 64 bits pour la partie imaginaire)

On peut aussi former des tableaux de chaînes de caractères dont la longueur n'excède pas une valeur donnée :

```
>>> np.array(("123",'abcbde',"12XY5"))   # vecteur de chaînes de caractères unicode, pos maxi < 6
array(['123', 'abcbde', '12XY5'],
      dtype='<U6')
```

1.6 Tableaux de nombres complexes

Les méthodes `real` et `imag` permettent de séparer un tableau `numpy` de nombres complexes en sa partie réelle et sa partie imaginaire. On obtient le tableau conjugué de a en évaluant `a.conj()` ou `np.conj(a)`

De même, on forme le tableau des modules (resp. des arguments) par `abs(a)` et `angle(a)`.

Voici par exemple une matrice z, de format 2×4, constituée de nombres complexes.

```
>>> z = np.array([[1j,1+1j,2-3j,4-1j],[2j,3-1j,2+2j,1+5j]]); z
array([[ 0.+1.j,  1.+1.j,  2.-3.j,  4.-1.j],
       [ 0.+2.j,  3.-1.j,  2.+2.j,  1.+5.j]])
```

Dans l'exemple suivant, on met la partie réelle (resp. imaginaire) du tableau z dans le tableau x (resp. y). On voit comment l'expression `x+1j*y` permet de reconstituer le tableau complexe initial. Les fonctions qui agissent ainsi terme à terme sur les éléments d'un tableau seront étudiées dans le chapitre « Fonctions universelles ».

```
>>> x = z.real; x
array([[ 0.,  1.,  2.,  4.],
       [ 0.,  3.,  2.,  1.]])
>>> y = z.imag; y
array([[ 1.,  1., -3., -1.],
       [ 2., -1.,  2.,  5.]])
```

```
>>> x + y*1j
array([[ 0.+1.j,  1.+1.j,  2.-3.j,  4.-1.j],
       [ 0.+2.j,  3.-1.j,  2.+2.j,  1.+5.j]])
>>> z.conj()
array([[ 0.-1.j,  1.-1.j,  2.+3.j,  4.+1.j],
       [ 0.-2.j,  3.+1.j,  2.-2.j,  1.-5.j]])
```


jean-miche.ferrard@ac-paris.fr mathprepa.fr

Chapitre 2

Lecture et écriture dans un tableau

2.1 Lecture de valeurs dans un vecteur, « slicing »

La lecture d'éléments d'un tableau `numpy` procède par `coupes` (« slices » en anglais) suivant une ou éventuellement plusieurs des dimensions du tableau. C'est ce mécanisme qui est à l'œuvre dans l'accès (en lecture et en écriture) aux éléments d'une liste (ou d'une liste de liste). Voir la doc ici : http://docs.scipy.org/doc/numpy/reference/arrays.indexing.html

Une coupe est un mécanisme de sélection de positions régulièrement espacées dans un intervalle $I = [\![0, p-1]\!]$.

La spécification la plus générale d'une coupe est `début(`*inclus*`) : fin(`*exclue*`) : incrément` et désigne les positions de I à partir de la valeur de *début* (incluse), par incrémentations successives tant qu'on reste dans l'intervalle défini par les valeurs de *début* et de *fin* (cette dernière étant exclue : très important).

Les positions acceptées par l'interpréteur `Python` forment en fait l'intervalle $I' = [\![-p, \ldots, -2, -1, 0, 1, 2, \ldots, p-1]\!]$ et elles sont converties « modulo p » en une position effective dans $I = [\![0, p-1]\!]$.

L'intérêt de cette extension syntaxique est que les positions négatives successives $-1, -2, \ldots$, qui sont converties en les positions $p-1, p-2, \ldots$ (où p représente la longueur du vecteur v) permettent de lire les éléments de v à partir de la fin.

Par défaut, la position de début (incluse) est 0 et la position de fin (exclue) est la longueur p de I donc de v.

Par défaut également, la valeur d'incrémentation est 1.

Comme ça parait un peu compliqué de prime abord, on va prendre quelques exemples, d'abord sur un vecteur.

On choisit ici de former le vecteur v, de longueur $p = 16$, tel que $v[k] = 10k$ pour k dans $I = [\![0, 15]\!]$.

```
>>> v = np.array(range(0,160,10)); v
array([  0,  10,  20,  30,  40,  50,  60,  70,  80,  90, 100, 110, 120, 130, 140, 150])
```

On commence par lire des éléments individuels dans le vecteur v.

```
>>> v[0] # le 1er élément
0
>>> v[1] # le 2ième élément
10
>>> v[2] # le 3ième élément
20
```

ici on compte à partir du début

```
>>> v[-1] # le dernier élément
150
>>> v[-2] # l'avant-dernier
140
>>> v[-3] # l'avant-avant-dernier
130
```

ici on compte à partir de la fin

```
>>> v[v.size-1] # le dernier
150
>>> v[-16]      # le premier
0
>>> v[-v.size]  # le premier
0
```

formulations peu claires : à éviter

On continue, avec le même vecteur v, en effectuant quelques coupes de valeurs consécutives dans l'ordre des indices croissants, c'est-à-dire avec la valeur 1 d'incrémentation par défaut :

```
>>> v[4:11]    # de v[4] à v[10], donc 11-4 = 7 éléments consécutifs
array([ 40,  50,  60,  70,  80,  90, 100])

>>> v[:6]      # de v[0] à v[5], les 6 premières valeurs de v
array([ 0, 10, 20, 30, 40, 50])

>>> v[6:]      # (v étant de longueur p=16), de v[6] à v[p-1], les p-6 = 10 dernières valeurs de v
array([ 60,  70,  80,  90, 100, 110, 120, 130, 140, 150])

>>> v[:]       # la totalité du vecteur v, comme si on avait écrit v[0:p]
array([  0,  10,  20,  30,  40,  50,  60,  70,  80,  90, 100, 110, 120, 130, 140, 150])
```

Voici maintenant des coupes d'éléments consécutifs, mais à l'envers, et toujours avec le même vecteur v :

```
>>> v[11:4:-1]     # de v[11] à v[5], donc |4-11| = 7 éléments
array([110, 100,  90,  80,  70,  60,  50])
>>> v[:6:-1]       # de v[p-1=15] à v[7], donc |6-15| = 9 éléments
array([150, 140, 130, 120, 110, 100,  90,  80,  70])
>>> v[6::-1]       # de v[6] à v[0], donc 7 éléments
array([60, 50, 40, 30, 20, 10,  0])
>>> v[::-1]        # la totalité du vecteur, mais à l'envers
array([150, 140, 130, 120, 110, 100,  90,  80,  70,  60,  50,  40,  30, 20,  10,   0])
```

Et pour finir, des coupes du même vecteur v, mais avec un incrément différent de 1 ou -1 :

```
>>> v[::4]
array([  0,  40,  80, 120])
>>> v[1::4]
array([ 10,  50,  90, 130])
>>> v[::5]
array([  0,  50, 100, 150])
```

```
>>> v[:6:2]
array([ 0, 20, 40])
>>> v[:7:2]
array([ 0, 20, 40, 60])
>>> v[10::2]
array([100, 120, 140])
```

```
>>> v[:5:-3]
array([150, 120,  90,  60])
>>> v[:6:-3]
array([150, 120,  90])
>>> v[6::-3]
array([60, 30,  0])
```

2.2 Lecture de valeurs dans une matrice

On procède comme dans la sous-section précédente, en effectuant une coupe suivant la première et/ou suivant la deuxième dimension (ça se généralise bien sûr à des tableaux de trois dimensions ou plus).

On se contentera de quelques exemples avec cette matrice m d'ordre 5×8, de terme général $m[i,j] = 10i+j$ (en numérotant lignes et colonnes à partir de 0) :

```
>>> m = np.array([[10*i+j for j in range(8)] for i in range(5)]); m
array([[ 0,  1,  2,  3,  4,  5,  6,  7],
       [10, 11, 12, 13, 14, 15, 16, 17],
       [20, 21, 22, 23, 24, 25, 26, 27],
       [30, 31, 32, 33, 34, 35, 36, 37],
       [40, 41, 42, 43, 44, 45, 46, 47]])
```

```
>>> m[3,5]
35
```
élément en position (3,5)

```
>>> m[3]
array([30, 31, 32, 33, 34, 35, 36, 37])
```
vecteur-ligne en position 3

```
>>> m[:,5]
array([ 5, 15, 25, 35, 45])
```
vecteur-colonne en position 5

```
>>> m[1:4,2:6]
array([[12, 13, 14, 15],
       [22, 23, 24, 25],
       [32, 33, 34, 35]])
```
lignes 1 à 3, colonnes 2 à 5

```
>>> m[1::2,1::2]
array([[11, 13, 15, 17],
       [31, 33, 35, 37]])
```
lignes et colonnes impaires

```
>>> m[::2,::2]
array([[ 0,  2,  4,  6],
       [20, 22, 24, 26],
       [40, 42, 44, 46]])
```
lignes et colonnes paires

```
>>> m[:3,:2]
array([[ 0,  1],
       [10, 11],
       [20, 21]])
```
trois premières lignes
deux premières colonnes

```
>>> m[2:,4:]
array([[24, 25, 26, 27],
       [34, 35, 36, 37],
       [44, 45, 46, 47]])
```
à partir de la ligne 2
à partir de la colonne 4

```
>>> m[::2,::3]
array([[ 0,  3,  6],
       [20, 23, 26],
       [40, 43, 46]])
```
une ligne sur deux
une colonne sur trois

```
>>> m[::-1]
array([[40, 41, 42, 43, 44, 45, 46, 47],
       [30, 31, 32, 33, 34, 35, 36, 37],
       [20, 21, 22, 23, 24, 25, 26, 27],
       [10, 11, 12, 13, 14, 15, 16, 17],
       [ 0,  1,  2,  3,  4,  5,  6,  7]])
```
on inverse l'ordre des lignes

```
>>> m[:,::-1]
array([[ 7,  6,  5,  4,  3,  2,  1,  0],
       [17, 16, 15, 14, 13, 12, 11, 10],
       [27, 26, 25, 24, 23, 22, 21, 20],
       [37, 36, 35, 34, 33, 32, 31, 30],
       [47, 46, 45, 44, 43, 42, 41, 40]])
```
on inverse l'ordre des colonnes

jean-miche.ferrard@ac-paris.fr mathprepa.fr

2.3 Écriture de valeurs dans un vecteur

Si a est un vecteur `numpy` l'instruction « `a[n]=x` » écrit la valeur x en position n.

```
>>> a = np.arange(10); a                          # le vecteur des entiers de 0 à 9
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> a[7] = 999                                    # écrit la valeur 999 en position 7
>>> a                                             # affiche le nouveau contenu de a
array([  0, 1, 2, 3, 4, 5, 6, 999, 8, 9])
```

La valeur x écrite en position n du vecteur a doit a priori être compatible avec le « data type » de a.

Si on écrit un entier x dans un tableau a de flottants, pas de problème (x est converti en le flottant correspondant).

Mais si on écrit un flottant x dans un tableau a d'entiers, alors x est converti en un entier (par troncature, pas par arrondi) pour l'écriture : il n'y a donc pas de message d'erreur, ni même d'avertissement. Le « data type » et l'adresse du tableau a ne changent pas.

L'idée fondamentale avec les tableaux `numpy` est que leur « data type » et leur taille (le nombre total d'éléments) sont fixés lors de leur création. Par souci d'efficacité et d'économie, `Python` limite la création de copies du tableau initial, et donc privilégie les opérations réalisées « en place ».

Dans l'exemple ci-dessous, on voit que l'instruction `a[5]=12.99` (où a est un tableau d'entiers) a été traduite en `a[5]=12` : l'opération a été effectuée « en place » (l'adresse du tableau en mémoire n'a pas changé).

```
>>> a = np.arange(10);
>>> a, id(a)                          # demande le contenu du tableau a, et son adresse
(array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9]), 3957296)
>>> a[5] = 12.99                      # on essaie d'écrire le flottant 12.99 en position 5
>>> a, id(a)                          # en fait, on a écrit 12, et l'adresse de a est inchangée
(array([ 0,  1,  2,  3,  4, 12,  6,  7,  8,  9]), 3957296)
```

On peut écrire plusieurs valeurs simultanément dans un vecteur, en utilisant une syntaxe du genre `a[`*coupe*`]=...`. Il faut simplement veiller à ce que le membre de droite (la source) soit « array-like » (une liste, un tuple, un tableau) et que la coupe spécifiée a (donc la cible) soient de même taille. Ainsi, dans l'exemple ci-dessous, les instructions `a[4:7]=[111,222,333]`, `a[4:7]=(111,222,333)` ou encore `a[4:7]=np.array([111,222,333])` ont le même effet :

```
>>> a = np.arange(10); a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> a[4:7] = [111,222,333]; a          # place 111, 222, 333 en positions respectives 4, 5, 6
array([0, 1, 2, 3, 111, 222, 333, 7, 8, 9])
```

L'exemple suivant est peut-être moins évident. On commence par former un vecteur nul de longueur 10 (avec le data type par défaut, donc float), puis on évalue l'instruction `a[::2] = np.arange(1,6)`. On affecte donc respectivement les cinq valeurs du tableau `array([1, 2, 3, 4, 5])` aux positions paires du tableau a.

```
>>> a = np.zeros(10); a, id(a)         # un vecteur de 10 flottants égaux à 0, et son adresse
(array([ 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]), 19048976)
>>> a[::2] = np.arange(1,6); a         # place 1,2,3,4,5 aux positions 0,2,4,6,8. Adresse inchangée
(array([ 1., 0., 2., 0., 3., 0., 4., 0., 5., 0.]), 19048976)
```

Autre exemple un peu tordu, on copie le vecteur a à l'envers sur lui-même :

```
>>> a = np.arange(10); a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> a[::-1] = a; a
array([9, 8, 7, 6, 5, 4, 3, 2, 1, 0])
```

2.4 Écriture de valeurs dans une matrice

On procède comme pour les vecteurs, indice par indice.

Commençons par l'écriture d'un coefficient :

```
>>> m
array([[ 0,  1,  2,  3],
       [10, 11, 12, 13],
       [20, 21, 22, 23]])
```

```
>>> m[1,2] = 999; m
array([[  0,   1,   2,   3],
       [ 10,  11, 999,  13],
       [ 20,  21,  22,  23]])
```

Continuons par l'écriture d'une ligne entière :

```
>>> m
array([[ 0,  1,  2,  3],
       [10, 11, 12, 13],
       [20, 21, 22, 23]])
```

```
>>> m[2] = [6, 7, 8, 9]; m
array([[ 0,  1,  2,  3],
       [10, 11, 12, 13],
       [ 6,  7,  8,  9]])
```

Terminons par l'écriture d'une colonne entière.

Notons qu'on pourrait ici, et avec le même effet, remplacer le tuple `(444,555,666)` par la liste `[444,555,666]`, ou par le vecteur `np.array([444,555,666])`, ou par la matrice-ligne `np.array([[444,555,666]])` mais (bizarrement) pas par la matrice-colonne `np.array([[444], [555], [666]])`.

```
>>> m
array([[ 0,  1,  2,  3],
       [10, 11, 12, 13],
       [20, 21, 22, 23]])
```

```
>>> m[:,1] = [444, 555, 666]; m
array([[  0, 444,   2,   3],
       [ 10, 555,  12,  13],
       [ 20, 666,  22,  23]])
```

Il y a énormément de possibilités, mais en dehors des trois précédentes, il faut vraiment en avoir l'utilité.

Juste pour réviser un peu l'utilisation de coupes à l'envers `::-1`, citons les exemples suivants :

```
>>> m
array([[ 0,  1,  2,  3],
       [10, 11, 12, 13],
       [20, 21, 22, 23]])
```

```
>>> m[::-1] = m; m
array([[20, 21, 22, 23],
       [10, 11, 12, 13],
       [ 0,  1,  2,  3]])
```

on a inversé l'ordre des lignes

```
>>> m[:,::-1] = m; m
array([[23, 22, 21, 20],
       [13, 12, 11, 10],
       [ 3,  2,  1,  0]])
```

puis inversé l'ordre des colonnes

2.5 Copies de tableaux, « vues » sur des tableaux

Le module `numpy` permet de travailler efficacement sur des tableaux possédant un nombre fixé d'éléments, ceux-ci ayant un type donné. Tout cela se passe à l'initialisation du tableau.

Avec ces tableaux, on applique donc un principe d'économie qui consiste à effectuer aussi peu que possible de recopies de tableaux en mémoire, sauf demande explicite. Tout dépend de la fonction utilisée sur le tableau.

L'exemple suivant est très important (parce que la situation évoquée ici revient souvent). Le vecteur a contenant un tableau `numpy`, la variable a pointe en fait sur la position de ce tableau en mémoire. L'instruction $b = a$ ne crée pas une nouvelle copie du tableau a, mais elle demande à la variable b de pointer sur le même tableau physique que a.

On exprime cette situation en disant que a et b constituent une même vue (« view » in english) d'un tableau `numpy`.

La conséquence immédiate est que toute modification apportée à cette *vue* (que ce soit par l'intermédiaire de la variable a ou de la variable b) affecte ce que « voient » et a et b. Ici, par exemple, après avoir posé $b = a$, l'instruction `b[2,1]=9` modifie le tableau vu par b, c'est-à-dire le tableau vu par a. C'est confirmé quand on évalue les expressions `id(a)` et `id(b)` qui donnent un résultat identique (l'adresse de ce que voient a et b).

```
>>> a
array([[ 0, 1, 2, 3],
       [10,11,12,13],
       [20,21,22,23]])
>>> b = a
```

```
>>> b[2,1] = 9; b
array([[ 0, 1, 2, 3],
       [10,11,12,13],
       [20, 9,22,23]])
```

on modifie `b[2,1]` dans le tableau b

```
>>> a
array([[ 0, 1, 2, 3],
       [10,11,12,13],
       [20, 9,22,23]])
```

la modification s'est répercutée sur a

```
>>> id(a)
19127280

>>> id(b)
19127280
```

c'est normal car a et b partagent une même vue

Il est bien sûr possible de « déconnecter » complètement les tableaux vus respectivement par deux identificateurs. L'expression `a.copy()` renvoie une copie « neuve » du tableau a, indépendante de l'original.

```
>>> a
array([[ 0, 1, 2, 3],
       [10,11,12,13],
       [20,21,22,23]])
>>> b = a.copy()
```

```
>>> b[2,1] = 9; b
array([[ 0, 1, 2, 3],
       [10,11,12,13],
       [20, 9,22,23]])
```

on modifie `b[2,1]` dans le tableau b

```
>>> a
array([[ 0, 1, 2, 3],
       [10,11,12,13],
       [20,21,22,23]])
```

la modification ne s'est par répercutée sur a

```
>>> id(a)
19134464

>>> id(b)
19096112
```

normal car a et b ne partagent pas la même vue

L'exemple suivant est beaucoup plus subtil et intéressant et instructif !

Au départ, le tableau a contient (en fait, « il voit ») une matrice d'entiers de type 5×8.

Ensuite on extrait une sous-matrice b de a (au moyen d'une coupe selon chacun des deux indices).

On décide ensuite d'annuler les éléments de b (avec la méthode `fill`, exposée plus loin, et qui travaille « en place »).

On demande ensuite le nouveau contenu de b (c'est-à-dire ce que *voit* b), et le résultat n'est pas surprenant.

```
>>> a
array([[ 0,  1,  2,  3,  4,  5,  6,  7],
       [10, 11, 12, 13, 14, 15, 16, 17],
       [20, 21, 22, 23, 24, 25, 26, 27],
       [30, 31, 32, 33, 34, 35, 36, 37],
       [40, 41, 42, 43, 44, 45, 46, 47]])
```

la matrice a initiale

```
>>> b = a[1:4,2:6]
>>> b
array([[12, 13, 14, 15],
       [22, 23, 24, 25],
       [32, 33, 34, 35]])
```

on en extrait une sous-matrice b

```
>>> b.fill(0)
>>> b
array([[0, 0, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 0]])
```

on annule les coefficients de b

Ce qui est plus inattendu, c'est que tous les coefficients de a et qui correspondant à l'emprunt que nous avons fait pour créer b sont annulés eux aussi ! ça se confirme après l'instruction `b.fill(1)` qui remplit de coefficients égaux à 1 le tableau vu par b, c'est-à-dire la sous-matrice de a correspondand à la définition de b.

```
>>> a
array([[ 0,  1,  2,  3,  4,  5,  6,  7],
       [10, 11,  0,  0,  0,  0, 16, 17],
       [20, 21,  0,  0,  0,  0, 26, 27],
       [30, 31,  0,  0,  0,  0, 36, 37],
       [40, 41, 42, 43, 44, 45, 46, 47]])
```

annuler b, c'est annuler une vue sur la partie du tableau a dont b est issue

```
>>> b.fill(1); a
array([[ 0,  1,  2,  3,  4,  5,  6,  7],
       [10, 11,  1,  1,  1,  1, 16, 17],
       [20, 21,  1,  1,  1,  1, 26, 27],
       [30, 31,  1,  1,  1,  1, 36, 37],
       [40, 41, 42, 43, 44, 45, 46, 47]])
```

c'est la même chose en remplissant la matrice b de coefficients égaux à 1

Important : on retiendra donc que lire une coupe d'un tableau `numpy` a, ça n'est pas créer un nouveau tableau, c'est renvoyer une « vue » sur un sous-tableau de a.

Si on reprend l'exemple précédent, le tableau a est en quelque sorte le père de b. Le terme exact est la base de b. D'ailleurs la méthode `base` du tableau b renvoie une vue sur le tableau a.

Il est intéressant de voir que modifier a, c'est également modifier b (exemple ci-dessous). Le lien entre les deux tableaux a et b est donc à double sens (c'est normal : ils partagent une même zone de données) :

```
>>> b.base
array([[ 0,  1,  2,  3,  4,  5,  6,  7],
       [10, 11,  1,  1,  1,  1, 16, 17],
       [20, 21,  1,  1,  1,  1, 26, 27],
       [30, 31,  1,  1,  1,  1, 36, 37],
       [40, 41, 42, 43, 44, 45, 46, 47]])
```

la base du tableau b c'est le tableau a

```
>>> a.fill(-1)
>>> b
array([[-1, -1, -1, -1],
       [-1, -1, -1, -1],
       [-1, -1, -1, -1]])
```

modifier a, c'est donc aussi modifier b

Pour plus de documentation sur le sujet : http://docs.scipy.org/doc/numpy/reference/arrays.ndarray.html

jean-miche.ferrard@ac-paris.fr mathprepa.fr

2.6 « Fancy indexing »

On peut accéder au contenu d'un tableau `numpy` tant en lecture qu'en écriture, en utilisant le « fancy indexing » (selon la terminologie officielle...). Ici les positions dans un vecteur ou dans une matrice ne procèdent plus nécessairement par des « coupes », mais peuvent être données dans un ordre tout à fait quelconque.

Si a est un tableau, et si I est un tableau d'indices (éventuellement une liste ou un tuple), alors $a[I]$ renvoie le tableau (de même format que le tableau I) formé des éléments $a[i]$, où i est dans I.

```
>>> a = np.arange(0,100,10); a
array([ 0, 10, 20, 30, 40, 50, 60, 70, 80, 90])
>>> b = a[[6,2,1,3]]; b
array([60, 20, 10, 30])
```

on demande le tableau $[a_6, a_2, a_1, a_3]$

```
>>> indices = np.array([[5,2,1],[3,8,7]])
>>> b = a[indices]; b
array([[50, 20, 10],
       [30, 80, 70]])
```

on demande le tableau $\begin{bmatrix}[a_5, a_2, a_1] \\ [a_3, a_8, a_7]\end{bmatrix}$

```
>>> a = np.arange(24).reshape(4,6); a
array([[ 0,  1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10, 11],
       [12, 13, 14, 15, 16, 17],
       [18, 19, 20, 21, 22, 23]])
```

```
>>> b = a[[3,1,0,1]]; b
array([[18, 19, 20, 21, 22, 23],
       [ 6,  7,  8,  9, 10, 11],
       [ 0,  1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10, 11]])
```

on demande le tableau des lignes a_3, a_1, a_0, a_1

Subtilité : les tableaux b formés ci-dessus (et qui ne sont pas obtenus pas « slicing ») ne sont pas de simples « vues » sur une partie de a (la notion a été discutée dans la sous-section précédente). Ils sont donc indépendants de a (modifier b n'affecte pas a et réciproquement).

Le « fancy indexing », ça fonctionne aussi pour des opérations d'écriture :

```
>>> a = np.arange(0,100,10); a
array([ 0, 10, 20, 30, 40, 50, 60, 70, 80, 90])
>>> a[[7, 2, 5, 3]] = (7777, 2222, 5555, 3333); a      # on modifie a[7], a[2], a[5] et a[3]
array([ 0, 10, 2222, 3333, 40, 5555, 60, 7777, 80, 90])
```

Comme si tout cela ne suffisait pas, on peut aussi utiliser les méthodes `take` et `put`.

Avec `a.put(positions,source)` on remplace, terme à terme, les valeurs de *source* dans les positions indiquées.

```
>>> a = np.arange(10); a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> a.put([6,1,5,2],[66,11,55,22]); a        # place 66 en position 6, 11 en position 1, etc.
array([ 0, 11, 22,  3,  4, 55, 66,  7,  8,  9])
```

Avec `a.take(positions)` on lit, terme à terme, les valeurs de a dans les positions indiquées.

```
>>> a = np.arange(0,100,10); a
array([ 0, 10, 20, 30, 40, 50, 60, 70, 80, 90])
>>> np.take(a,[6, 1, 5, 0])           # renvoie le tableau des valeurs a[6], a[1], a[5], a[0]
array([60, 10, 50,  0])
```

Pour une matrice, on utilisera un argument supplémentaire « `axis=...` » pour spécifier dans quel sens on fait la lecture (selon les lignes avec `axis=0`, selon les lignes avec `axis=1`). En l'absence de cette précision, la lecture s'effectue comme si le tableau avait été « aplati ».

```
>>> a = np.arange(0,160,10).reshape(4,4); a
array([[  0,  10,  20,  30],
       [ 40,  50,  60,  70],
       [ 80,  90, 100, 110],
       [120, 130, 140, 150]])
>>> i = np.array([3,0,2])  # trois indices
>>> a.take(i)   # lecture de a "aplati"
array([30,  0, 20])
```

```
>>> a.take(i,axis=0)     # lectures de lignes
array([[120, 130, 140, 150],
       [  0,  10,  20,  30],
       [ 80,  90, 100, 110]])
>>> a.take(i,axis=1)     # lectures de colonnes
array([[ 30,   0,  20],
       [ 70,  40,  60],
       [110,  80, 100],
       [150, 120, 140]])
```

On peut signaler également les fonctions suivantes :

`place(a,conds,b)` écrit dans a les valeurs de b aux positions de a spécifiées par le tableau de booléens *conds*

```
>>> a = np.arange(10); a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> np.place(a, np.mod(a,3) == 0, 0); a   # remplace toutes les valeurs multiples de 3 par des 0
array([0, 1, 2, 0, 4, 5, 0, 7, 8, 0])
```

La fonction `copyto` est presque synonyme de la fonction `place` (se reporter à la doc !)

```
>>> a = np.arange(10); a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> np.copyto(a,-1, where=np.mod(a,3) == 0); a   # remplace les valeurs multiples de 3 par des -1
array([-1,  1,  2, -1,  4,  5, -1,  7,  8, -1])
```

Chapitre 3

Dimensions d'un tableau

3.1 Redimensionnement par « reshape » ou « resize »

La principale méthode pour modifier la taille d'un tableau est `reshape`. Le redimensionnement d'un tableau est quand même soumis à la contrainte que le nombre total d'éléments doit rester constant. On pourra ainsi passer (dans les deux sens) d'un vecteur de taille n à une matrice de taille (p, q) ou à une matrice de taille (r, s) à condition que $n = pq = rs$.
Si a est un tableau `numpy`, l'expression `a.reshape(n,p)` renvoie une copie redimensionnée (le contenu de la variable initiale n'est donc pas affecté, comme on le constate ci-dessous).

```
>>> a = np.array(range(6)); a
array([0, 1, 2, 3, 4, 5])
>>> a.reshape(1,6)
array([[0, 1, 2, 3, 4, 5]])
>>> a
array([0, 1, 2, 3, 4, 5])
```

```
>>> a.reshape(2,3)
array([[0, 1, 2],
       [3, 4, 5]])
>>> a.reshape(3,2)
array([[0, 1],
       [2, 3],
       [4, 5]])
```

```
>>> a.reshape(6,1)
array([[0],
       [1],
       [2],
       [3],
       [4],
       [5]])
```

Il y a une autre possibilité qui consiste à réinitialiser *l'attribut* `shape` d'un tableau `numpy`. La différence avec ce qui précède est que le redimensionnement est fait « sur place », et donc que le contenu de la variable s'en trouve affecté.

```
>>> a = np.array(range(6));
>>> a
array([0, 1, 2, 3, 4, 5])
>>> a.shape = (2,3); a   # devient matrice 2 × 3
array([[0, 1, 2],
       [3, 4, 5]])
```

```
>>> a.shape = (3,2); a  # devient matrice 3 × 2
array([[0, 1],
       [2, 3],
       [4, 5]])
>>> a.shape = 6; a      # redevient un vecteur
array([0, 1, 2, 3, 4, 5])
```

Autre possibilité, la fonction `resize`, qui permet de redimensionner un bloc, en transformant (sur place) un tableau T (à n éléments) en un tableau T' (à n' éléments). Si $n' > n$, le tableau T' est rempli avec répétitions des éléments de t. Sinon T' est rempli par les n premiers éléments de T.

```
>>> a = np.array(range(6)); a
array([0, 1, 2, 3, 4, 5])
>>> a.resize(3,2); a
array([[0, 1],
       [2, 3],
       [4, 5]])
>>> a.resize(2,3); a
array([[0, 1, 2],
       [3, 4, 5]])
```

avec `a.resize` la modification s'effectue « en place »

```
>>> np.resize(a,(4,3))
array([[0, 1, 2],
       [3, 4, 5],
       [0, 1, 2],
       [3, 4, 5]])
>>> a
array([[0, 1, 2],
       [3, 4, 5]])
```

avec `np.resize(a,...)`, un nouveau tableau est créé

```
>>> np.resize(a,(1,7))
array([[0, 1, 2, 3, 4, 5, 0]])
>>> np.resize(a,(2,5))
array([[0, 1, 2, 3, 4],
       [5, 0, 1, 2, 3]])
>>> a
array([[0, 1, 2],
       [3, 4, 5]])
```

autres exemples avec `np.resize(a,...)`

La *fonction* `resize` permet également de créer des tableaux constants contenant une autre valeur que 0 et 1 (sinon on utiliserait les fonctions `zeros` ou `ones`) :

```
>>> np.resize(1.23, 4)
array([ 1.23,  1.23,  1.23,  1.23])
>>> np.resize(9, 7)
array([9, 9, 9, 9, 9, 9, 9])
```

```
>>> np.resize(5, (3,6))
array([[5, 5, 5, 5, 5, 5],
       [5, 5, 5, 5, 5, 5],
       [5, 5, 5, 5, 5, 5]])
```

3.2 Aplatissement d'un tableau

La méthode `flatten` d'un tableau `numpy` renvoie une copie « aplatie » de ce tableau.

Cette fonction est l'occasion de découvrir un argument facultatif souvent employé quand on est amené à parcourir un tableau. C'est l'argument `order=...` dont la valeur par défaut est `order='C'` et qui peut être modifié en `order='F'`.

Avec `order='C'` (où 'C' désigne le langage de programmation) le parcours s'effectue d'abord de gauche à droite sur une ligne avant de passer à la ligne suivante (donc l'indice de colonne varie prioritairement). Avec `order='F'` (où 'F' désigne le langage de programmation Fortran) le parcours s'effectue d'abord de haut en bas sur une colonne avant de passer à la colonne suivante (donc l'indice de ligne varie prioritairement).

```
>>> a = np.array([[1,5,3],[2,7,4]])
>>> a
array([[1, 5, 3],
       [2, 7, 4]])
```

```
>>> a.flatten()    # parcours par défaut (à droite d'abord)
array([1, 5, 3, 2, 7, 4])
>>> a.flatten('F') # parcours Fortran (en bas d'abord)
array([1, 2, 5, 7, 3, 4])
```

Remarque : la fonction `ravel` est peu ou prou synonyme de `flatten`.

Si a est un tableau `numpy`, alors `a.flat` renvoie un itérateur permet d'accéder séquentiellement à tous les éléments d'un tableau (dans l'ordre 'C' ou 'F' qui a présidé à la création du tableau). L'intérêt est qu'ici il n'y a pas (contrairement à la fonction `flatten`) de recopie du tableau.

```
>>> a
array([[9, 5, 6],
       [8, 7, 4]])
```

```
>>> a.flat[3]
8
```

l'élément en position 3 dans a

```
>>> sum(x for x in a.flat)
39
```

la somme des éléments de a

Il est intéressant de comparer avec le résultat suivant :

Quand on évalue `sum(x for x in a)`, la variable x prend les valeurs `a[0]=[9,5,6]` puis `a[1]=[8,7,4]`.
La somme est donc `[9,5,6]+[8,7,4]=[17, 12, 10]`.

```
>>> sum(x for x in a)
array([17, 12, 10])
```

3.3 Transposition d'une matrice

La fonction (ou la méthode) `transpose`, ou plus simplement la méthode `T`, renvoient la transposée d'une matrice :

```
>>> a
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12],
       [13, 14, 15, 16]])
```

```
>>> a.transpose()
array([[ 1,  5,  9, 13],
       [ 2,  6, 10, 14],
       [ 3,  7, 11, 15],
       [ 4,  8, 12, 16]])
```

```
>>> a.T
array([[ 1,  5,  9, 13],
       [ 2,  6, 10, 14],
       [ 3,  7, 11, 15],
       [ 4,  8, 12, 16]])
```

3.4 Suppressions/insertions de lignes, de colonnes

`delete(a,k,axis=0)` et `delete(a,k,axis=1)` suppriment respectivement la k-ième ligne et la k-ième colonne de a.

Le troisième argument indique donc le numéro d'indice (*l'axe*) selon lequel on pratique cette suppression.

L'argument `axis=...` est très souvent présent (de façon facultative) dans les fonctions du module `numpy` : disons pour simplifier, que pour des matrices (deux indices), `axis=0` signifie « selon l'indice de ligne » et `axis=1` signifie « selon l'indice de colonne ». Souvent, d'ailleurs, le mot `axis` peut être omis.

```
>>> a
array([[ 0,  1,  2,  3,  4],
       [10, 11, 12, 13, 14],
       [20, 21, 22, 23, 24],
       [30, 31, 32, 33, 34]])
```

une a matrice de type 4×5

```
>>> b = np.delete(a,2,0);
>>> b
array([[ 0,  1,  2,  3,  4],
       [10, 11, 12, 13, 14],
       [30, 31, 32, 33, 34]])
```

la matrice b est obtenue en supprimant la ligne d'indice 2 de a

```
>>> c = np.delete(b,3,1)
>>> c
array([[ 0,  1,  2,  4],
       [10, 11, 12, 14],
       [30, 31, 32, 34]])
```

la matrice c est obtenue en supprimant la colonne d'indice 3 de b

On peut également supprimer plusieurs colonnes (ou lignes) à la fois. Voici quelques exemples :

```
>>> np.delete(a,np.s_[0::2],1)
array([[ 1,  3],
       [11, 13],
       [21, 23],
       [31, 33]])
```

supprime les colonnes d'indice pair

```
>>> np.delete(a,np.s_[1::2],1)
array([[ 0,  2,  4],
       [10, 12, 14],
       [20, 22, 24],
       [30, 32, 34]])
```

supprime les colonnes d'indice impair

```
>>> np.delete(a,[0,2,3],1)
array([[ 1,  4],
       [11, 14],
       [21, 24],
       [31, 34]])
```

supprime les colonnes d'indice 0, 2, 3

Les expressions `insert(a,k,v,axis=0)` et `insert(a,k,v,axis=1)` permettent d'insérer la valeur v respectivement avant la k-ième ligne ou avant la k-ième colonne de a. Voici deux exemples :

```
>>> a
array([[ 0,  1,  2,  3],
       [10, 11, 12, 13],
       [20, 21, 22, 23]])
```

une matrice a de type 3×4

```
>>> np.insert(a,2,-1,axis=1)
array([[ 0,  1, -1,  2,  3],
       [10, 11, -1, 12, 13],
       [20, 21, -1, 22, 23]])
```

insère des -1 juste avant la colonne d'indice 2

```
>>> np.insert(a,1,0,axis=0)
array([[ 0,  1,  2,  3],
       [ 0,  0,  0,  0],
       [10, 11, 12, 13],
       [20, 21, 22, 23]])
```

insère des 0 juste avant la ligne d'indice 1

Les expressions `append(a,v,axis=0)` et `append(a,v,axis=1)` permettent d'ajouter une matrice-colonne C (respectivement une matrice V) après la dernière ligne (respectivement après la dernière colonne) de a. Il faut veiller à ce que la matrice ajoutée ait un format compatible avec celui de a. Voici deux exemples :

```
>>> a
array([[ 0,  1,  2,  3],
       [10, 11, 12, 13],
       [20, 21, 22, 23]])
```

une matrice a de type 3×4

```
>>> np.append(a,[[7],[8],[9]],1)
array([[ 0,  1,  2,  3,  7],
       [10, 11, 12, 13,  8],
       [20, 21, 22, 23,  9]])
```

ajoute une colonne [[7,8,9]] après la dernière colonne

```
>>> np.append(a,[[3,5,7,9]],0)
array([[ 0,  1,  2,  3],
       [10, 11, 12, 13],
       [20, 21, 22, 23],
       [ 3,  5,  7,  9]])
```

ajoute une ligne [3,5,7,9] après la dernière ligne

En reprenant l'exemple de la matrice précédente a, on voit comment ajouter la matrice identité à droite de la matrice a (l'argument `int` permet de faire en sorte que le résultat soit encore à coefficients entiers).

```
>>> np.append(a,np.identity(3),1)
array([[ 0.,  1.,  2.,  3.,  1.,  0.,  0.],
       [ 10., 11., 12., 13.,  0.,  1.,  0.],
       [ 20., 21., 22., 23.,  0.,  0.,  1.]])
```

```
>>> np.append(a,np.identity(3,int),1)
array([[ 0,  1,  2,  3,  1,  0,  0],
       [10, 11, 12, 13,  0,  1,  0],
       [20, 21, 22, 23,  0,  0,  1]])
```

3.5 Permutations/rotations de lignes, de colonnes

`fliplr(m)` inverse l'ordre des colonnes de m : ici « `lr` » est mis pour « `left right` ».

Remarque : ça ne marche pas pour les vecteurs, car il faut qu'il y ait au moins deux dimensions.

```
>>> m
array([[ 0,  1,  2,  3],
       [10, 11, 12, 13],
       [20, 21, 22, 23]])
```

la matrice initiale m

```
>>> np.fliplr(m)
array([[ 3,  2,  1,  0],
       [13, 12, 11, 10],
       [23, 22, 21, 20]])
```

inverse l'ordre des colonnes de m

```
>>> m
array([[ 0,  1,  2,  3],
       [10, 11, 12, 13],
       [20, 21, 22, 23]])
```

le contenu initial de m est inchangé

`flipud(m)` inverse l'ordre des lignes de m : ici « `ud` » est mis pour « `up down` ».

```
>>> m
array([[ 0,  1,  2,  3,  4],
       [10, 11, 12, 13, 14],
       [20, 21, 22, 23, 24],
       [30, 31, 32, 33, 34]])
```
la matrice initiale m

```
>>> np.flipud(m)
array([[30, 31, 32, 33, 34],
       [20, 21, 22, 23, 24],
       [10, 11, 12, 13, 14],
       [ 0,  1,  2,  3,  4]])
```
inverse l'ordre des lignes de m

```
>>> m
array([[ 0,  1,  2,  3,  4],
       [10, 11, 12, 13, 14],
       [20, 21, 22, 23, 24],
       [30, 31, 32, 33, 34]])
```
le contenu initial de m est inchangé

Comme on le voit ici, c'est `flipud` qu'il faut utiliser pour l'inverser l'ordre des éléments des vecteurs :

```
>>> a = np.arange(1,10); a
array([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
>>> np.flipud(a)
array([9, 8, 7, 6, 5, 4, 3, 2, 1])
```

Entre autres variations possibles, on peut n'inverser que les k premiers éléments d'un vecteur :

```
>>> a = np.arange(1,10); a
array([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
>>> a[:7] = np.flipud(a[:7]); a
array([7, 6, 5, 4, 3, 2, 1, 8, 9])
```

Dans cet exemple, on inverse l'ordre des éléments de la ligne qui est en position 2 dans m :

```
>>> m
array([[ 0,  1,  2,  3,  4],
       [10, 11, 12, 13, 14],
       [20, 21, 22, 23, 24],
       [30, 31, 32, 33, 34]])
```

```
>>> m[2] = np.flipud(m[2]); m
array([[ 0,  1,  2,  3,  4],
       [10, 11, 12, 13, 14],
       [24, 23, 22, 21, 20],
       [30, 31, 32, 33, 34]])
```

Pour ceux que ça intéresse, `rot90(m,k=1)` renvoie une copie de la matrice m après k rotations d'angle $\pi/2$.

```
>>> m
array([[ 0,  1,  2,  3],
       [10, 11, 12, 13],
       [20, 21, 22, 23]]
```

```
>>> np.rot90(m)
array([[ 3, 13, 23],
       [ 2, 12, 22],
       [ 1, 11, 21],
       [ 0, 10, 20]])
```
rotation de 90 °

```
>>> np.rot90(m,-1)
array([[20, 10,  0],
       [21, 11,  1],
       [22, 12,  2],
       [23, 13,  3]])
```
rotation de -90 °

```
>>> np.rot90(m,2)
array([[23, 22, 21, 20],
       [13, 12, 11, 10],
       [ 3,  2,  1,  0]])
```
rotation de 180 °

L'expression `roll(a,k,axis=None)` renvoie une *copie* du tableau après k rotations d'une position (vers la droite si $k > 0$, vers la gauche si $k < 0$). Les éléments qui sortent d'un coté rentrent de l'autre. Un nouveau tableau est créé, il ne s'agit donc pas d'une rotation « en place ». Pour une matrice, l'argument `axis` est facultatif (par défaut c'est une rotation sur les lignes). Pour une rotation sur les colonnes d'une matrice, on utilisera l'argument `axis=1`.

```
>>> a = np.arange(10); a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> np.roll(a,1)
array([9, 0, 1, 2, 3, 4, 5, 6, 7, 8])
>>> np.roll(a,2)
array([8, 9, 0, 1, 2, 3, 4, 5, 6, 7])
>>> np.roll(a,-4)
array([4, 5, 6, 7, 8, 9, 0, 1, 2, 3])
```
diverses rotations d'un même vecteur

```
>>> m
array([[ 0,  1,  2,  3],
       [10, 11, 12, 13],
       [20, 21, 22, 23]])
>>> np.roll(m,1,axis=0)
array([[20, 21, 22, 23],
       [ 0,  1,  2,  3],
       [10, 11, 12, 13]])
```
rotation d'une ligne vers le bas

```
>>> np.roll(m,1,axis=1)
array([[ 3,  0,  1,  2],
       [13, 10, 11, 12],
       [23, 20, 21, 22]])
>>> np.roll(m,-2,axis=1)
array([[ 2,  3,  0,  1],
       [12, 13, 10, 11],
       [22, 23, 20, 21]])
```
une colonne vers la droite
ou deux colonnes vers la gauche

L'expression `swapaxes(a,axe,axe')` effectue un échange des axes sur le tableau a. Pour les vecteurs, c'est sans effet, pour les matrices c'est équivalent à la transposition. Ça ne peut donc avoir d'utilité pour les tableaux à $n \geq 3$ indices.

```
>>> a = np.arange(30).reshape(2,3,5); a
array([[[ 0,  1,  2,  3,  4],
        [ 5,  6,  7,  8,  9],
        [10, 11, 12, 13, 14]],

       [[15, 16, 17, 18, 19],
        [20, 21, 22, 23, 24],
        [25, 26, 27, 28, 29]]])
```
deux tableaux de format 3×5

```
>>> np.swapaxes(a,0,1)
array([[[ 0,  1,  2,  3,  4],
        [15, 16, 17, 18, 19]],

       [[ 5,  6,  7,  8,  9],
        [20, 21, 22, 23, 24]],

       [[10, 11, 12, 13, 14],
        [25, 26, 27, 28, 29]]])
```
trois tableaux de format 2×5

3.6 Opérations par blocs

`concatenate` permet d'accoler deux ou plusieurs tableaux (horizontalement avec `axis=1`, verticalement avec `axis=0`).
En cas de concaténation horizontale par exemple, toutes les colonnes doivent avoir la même hauteur.
Les différents tableaux à concaténer doivent être donnés sous la forme d'un tuple (voir exemples) :

```
>>> a = np.array(range(8)).reshape(2,4)
>>> a
array([[0, 1, 2, 3],
       [4, 5, 6, 7]])

>>> b = np.zeros(a.shape,int); b
array([[0, 0, 0, 0],
       [0, 0, 0, 0]])

>>> c = np.ones(a.shape,int); c
array([[1, 1, 1, 1],
       [1, 1, 1, 1]])
```

```
>>> np.concatenate((a,b),axis=1)
array([[0, 1, 2, 3, 0, 0, 0, 0],
       [4, 5, 6, 7, 0, 0, 0, 0]])
>>> np.concatenate((a,b,c),axis=1)
array([[0, 1, 2, 3, 0, 0, 0, 0, 1, 1, 1, 1],
       [4, 5, 6, 7, 0, 0, 0, 0, 1, 1, 1, 1]])
>>> np.concatenate((a,c),axis=0)
array([[0, 1, 2, 3],
       [4, 5, 6, 7],
       [1, 1, 1, 1],
       [1, 1, 1, 1]])
```

On peut aussi utiliser les fonctions `hstack` (concaténation horizontale) et `vstack` (concaténation verticale).
Voici quelques exemples en reprenant la signification précédente des tableaux a, b, c :

```
>>> np.vstack((a,c))
array([[0, 1, 2, 3],
       [4, 5, 6, 7],
       [1, 1, 1, 1],
       [1, 1, 1, 1]])
```

```
>>> np.hstack((a,c))
array([[0, 1, 2, 3, 1, 1, 1, 1],
       [4, 5, 6, 7, 1, 1, 1, 1]])
>>> np.hstack((a,b,c))
array([[0, 1, 2, 3, 0, 0, 0, 0, 1, 1, 1, 1],
       [4, 5, 6, 7, 0, 0, 0, 0, 1, 1, 1, 1]])
```

La fonction `column_stack` combine des vecteurs-lignes en les colonnes d'un tableau. On voit ici la différence des résultats obtenus en utilisant `hstack`, `vstack` ou `column_stack` sur un tuple de deux vecteurs lignes a et b.

```
>>> a = np.array((1,2,3))
>>> b = np.array((2,3,4))
>>> np.hstack((a,b))
array([1, 2, 3, 2, 3, 4])
```

```
>>> np.vstack((a,b))
array([[1, 2, 3],
       [2, 3, 4]])
```

```
>>> np.column_stack((a,b))
array([[1, 2],
       [2, 3],
       [3, 4]])
```

Si a est un tableau de n lignes, l'expression `vsplit(a,k)` renvoie un tuple de k tableaux de n/k lignes représentant un découpage du tableau a (le nombre n de lignes de a doit être un multiple de k).

```
>>> a = np.array(range(24)).reshape(4,6); a
array([[ 0,  1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10, 11],
       [12, 13, 14, 15, 16, 17],
       [18, 19, 20, 21, 22, 23]])
```

une matrice de type 4×6

```
>>> b, c = np.vsplit(a,2)
>>> b
array([[ 0,  1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10, 11]])
>>> c
array([[12, 13, 14, 15, 16, 17],
       [18, 19, 20, 21, 22, 23]])
```

on la partage en deux tableaux de deux lignes

Si a est un tableau de n colonnes, l'expression `hsplit(a,k)` renvoie un tuple de k tableaux de n/k colonnes représentant un découpage du tableau a (le nombre n de colonnes de a doit être un multiple de k).
Voici un exemple en réutilisant la définition précédente du tableau a. On a partargé a et trois tableaux x, y, z de deux colonnes, avant de les combiner avec `hstack` mais dans l'ordre z, x, y.

```
>>> x,y,z = np.hsplit(a,3)
>>> print(x)
[[ 0  1]
 [ 6  7]
 [12 13]
 [18 19]]
```

```
>>> print(y)
[[ 2  3]
 [ 8  9]
 [14 15]
 [20 21]]
```

```
>>> print(z)
[[ 4  5]
 [10 11]
 [16 17]
 [22 23]]
```

```
>>> print(np.hstack((z,x,y)))
[[ 4  5  0  1  2  3]
 [10 11  6  7  8  9]
 [16 17 12 13 14 15]
 [22 23 18 19 20 21]]
```

En fait, les fonctions `hsplit` et `vsplit` acceptent aussi un deuxième argument sous la forme d'une liste croissante d'indices, ce qui permet de choisir plus finement les séparations en lignes ou en colonnes.

jean-miche.ferrard@ac-paris.fr mathprepa.fr

On reprend à nouveau l'exemple du tableau a précédent (et qui possède 6 colonnes). L'argument $[2,5]$ signifie qu'on marque une séparation juste avant la colonne d'indice 2, puis juste avant la colonne d'indice 5. Il en résulte un partage de a en trois tableaux ayant respectivement 2, puis $5-2=3$, et enfin $6-5=1$ colonnes.

```
>>> x, y, z = np.hsplit(a,[2,5])
>>> x
array([[ 0,  1],
       [ 6,  7],
       [12, 13],
       [18, 19]])
```

```
>>> y
array([[ 2,  3,  4],
       [ 8,  9, 10],
       [14, 15, 16],
       [20, 21, 22]])
```

```
>>> z
array([[ 5],
       [11],
       [17],
       [23]])
```

`tile(a,[n,p])` construit un tableau en répétant n fois le tableau a dans le sens horizontal et p fois dans le sens vertical.

```
>>> a = [[1,2],[3,4]]
>>> np.tile(a,2)
array([[1, 2, 1, 2],
       [3, 4, 3, 4]])
>>> np.tile(a,[2,1])
array([[1, 2],
       [3, 4],
       [1, 2],
       [3, 4]])
```

```
>>> np.tile(a,[4,4])
array([[1, 2, 1, 2, 1, 2, 1, 2],
       [3, 4, 3, 4, 3, 4, 3, 4],
       [1, 2, 1, 2, 1, 2, 1, 2],
       [3, 4, 3, 4, 3, 4, 3, 4],
       [1, 2, 1, 2, 1, 2, 1, 2],
       [3, 4, 3, 4, 3, 4, 3, 4],
       [1, 2, 1, 2, 1, 2, 1, 2],
       [3, 4, 3, 4, 3, 4, 3, 4]])
```


jean-miche.ferrard@ac-paris.fr mathprepa.fr

Chapitre 4

Tableaux spécifiques

Il est bien commode de fabriquer des tableaux dont le terme général répond à une formule donnée ou au contraire est obtenu de façon pseudo-aléatoire. On voit ici les exemples les plus utiles.

4.1 Tableaux constants

La fonction `zeros` permet de former des tableaux dont tous les coefficients sont nuls.

– le premier argument, obligatoire, précise le format (`shape`) du tableau : un entier n pour un vecteur de longueur n, un couple (n, p) pour une matrice de type $n \times p$, ou encore un tuple $(n, p, q, r, \ldots)$ pour un tableau à plus de deux indices. Attention : la syntaxe `zeros(3,5)` est incorrecte, il faut écrire `zeros((3,5))`.

– un argument facultatif permet de fixer le type de données (c'est `float` par défaut).

```
>>> np.zeros(5)
array([ 0.,  0.,  0.,  0.,  0.])
>>> np.zeros((1,5))
array([[ 0.,  0.,  0.,  0.,  0.]])
>>> np.zeros(5,int)
array([0, 0, 0, 0, 0])
```

```
>>> np.zeros((5,3))
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
```

```
>>> np.zeros((5,1),complex)
array([[ 0.+0.j],
       [ 0.+0.j],
       [ 0.+0.j],
       [ 0.+0.j],
       [ 0.+0.j]])
```

Avec une syntaxe analogue, la fonction `ones` permet de former des tableaux dont tous les coefficients valent 1.

```
>>> np.ones((3,6))
array([[ 1.,  1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.,  1.]])
```

```
>>> np.ones((3,6),int)
array([[1, 1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1, 1]])
```

Rappelons que chez les booléens, false = 0 = « nul » = « vide », et True = 1 = « non nul » = « non vide » :

```
>>> np.zeros((3,3),bool)
array([[False, False, False],
       [False, False, False],
       [False, False, False]], dtype=bool)
```

```
>>> np.ones((3,3),bool)
array([[ True,  True,  True],
       [ True,  True,  True],
       [ True,  True,  True]], dtype=bool)
```

La méthode `fill` d'un tableau `numpy` remplit ce tableau par une valeur constante.

Comme le montrent les exemples suivants, le remplissage se fait « en place ». L'adresse du tableau en mémoire ne change pas. En particulier, il n'est pas possible de modifier le « data type » (voir la colonne de droite, ou la valeur flottante 3.14 a été convertie en la valeur entière 3 pour le remplissage).

```
>>> a = np.array([[1,5,3],[2,7,4]])
>>> a, id(a)
(array([[1, 5, 3],
       [2, 7, 4]]), 3899264)
```

```
>>> a.fill(1);
>>> a, id(a)
(array([[1, 1, 1],
       [1, 1, 1]]), 3899264)
```

```
>>> a.fill(3.14);
>>> a, id(a)
(array([[3, 3, 3],
       [3, 3, 3]]), 3899264)
```

On peut également noter les fonctions `ones_like` et `zeros_like` dont le rôle est former le tableau constant (valeurs 0 ou valeurs 1) ayant le même format que le tableau passé en argument.

```
>>> a = np.arange(10).reshape(2,5); a
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9]])
```

```
>>> np.zeros_like(a)
array([[0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0]])
```

```
>>> np.ones_like(a)
array([[1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1]])
```

Rappel : si ℓ est une liste, la syntaxe $\ell * n$ renvoie la concaténation de n exemplaires de ℓ.

On en déduit une façon très simple de former des tableaux constants, ou du moins contenant des répétitions :

```
>>> np.array([2]*5)
array([2, 2, 2, 2, 2])
>>> np.array([1,2]*3)
array([1, 2, 1, 2, 1, 2])
```

```
>>> np.array([[1,2]]*3)
array([[1, 2],
       [1, 2],
       [1, 2]])
```

```
>>> np.array([[1,2],[3,4]]*2)
array([[1, 2],
       [3, 4],
       [1, 2],
       [3, 4]])
```

4.2 Identité, matrices diagonales ou triangulaires

La fonction `identity` fabrique la matrice identité d'ordre n (donné comme premier argument).

```
>>> np.identity(4)
array([[ 1., 0., 0., 0.],
       [ 0., 1., 0., 0.],
       [ 0., 0., 1., 0.],
       [ 0., 0., 0., 1.]])
```

```
>>> np.identity(4,int)
array([[1, 0, 0, 0],
       [0, 1, 0, 0],
       [0, 0, 1, 0],
       [0, 0, 0, 1]])
```

```
>>> np.identity(4,complex)
array([[ 1.+0.j, 0.+0.j, 0.+0.j, 0.+0.j],
       [ 0.+0.j, 1.+0.j, 0.+0.j, 0.+0.j],
       [ 0.+0.j, 0.+0.j, 1.+0.j, 0.+0.j],
       [ 0.+0.j, 0.+0.j, 0.+0.j, 1.+0.j]])
```

La fonction `eye` permet de fabriquer la matrice identité ou plus généralement une matrice dont tous les coefficients sont nuls sauf ceux d'une certaine « parallèle » à la diagonale et qui valent 1.

Le premier argument (obligatoire) donne le nombre n de lignes. Le second argument (facultatif) donne le nombre p de colonnes (par défaut $p = n$ donc la matrice est carrée). Un argument facultatif nommé $k =$ .. permet de spécifier un décalage de la diagonale de 1 au dessus (si $k > 0$) ou en-dessous (si $k < 0$) de la diagonale principale.

Enfin, on peut ajouter un argument pour fixer le type des données (`float` par défaut).

```
>>> np.eye(5,dtype=int)
array([[1, 0, 0, 0, 0],
       [0, 1, 0, 0, 0],
       [0, 0, 1, 0, 0],
       [0, 0, 0, 1, 0],
       [0, 0, 0, 0, 1]])
```

```
>>> np.eye(5,6,dtype=int)
array([[1, 0, 0, 0, 0, 0],
       [0, 1, 0, 0, 0, 0],
       [0, 0, 1, 0, 0, 0],
       [0, 0, 0, 1, 0, 0],
       [0, 0, 0, 0, 1, 0]])
```

```
>>> np.eye(1,5)
array([[ 1.,  0.,  0.,  0.,  0.]])
>>> np.eye(1,5,1)
array([[ 0.,  1.,  0.,  0.,  0.]])
>>> np.eye(1,5,2)
array([[ 0.,  0.,  1.,  0.,  0.]])
```

```
>>> np.eye(4,k=-2)
array([[ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.],
       [ 1.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.]])
```

```
>>> np.eye(4,k=1)
array([[ 0.,  1.,  0.,  0.],
       [ 0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  1.],
       [ 0.,  0.,  0.,  0.]])
```

```
>>> np.eye(4,k=3)
array([[ 0.,  0.,  0.,  1.],
       [ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.]])
```

La fonction `diag` renvoie la diagonale d'une matrice. Avec un deuxième argument (facultatif) k, on obtient une sur-diagonale (si $k > 0$) ou une sous-diagonale (si $k < 0$).

```
>>> a
array([[ 0,  1,  2,  3],
       [10, 11, 12, 13],
       [20, 21, 22, 23],
       [30, 31, 32, 33]])
>>> np.diag(a)
array([ 0, 11, 22, 33])
```

```
>>> np.diag(a,1)
array([ 1, 12, 23])
>>> np.diag(a,2)
array([ 2, 13])
>>> np.diag(a,3)
array([3])
```

```
>>> np.diag(a,-1)
array([10, 21, 32])
>>> np.diag(a,-2)
array([20, 31])
>>> np.diag(a,-3)
array([30])
```

Inversement, la fonction `diag`, si elle est appliquée à un vecteur (ou à une liste, ou à un tuple) renvoie la matrice diagonale formée sur les coefficients de ce vecteur :

```
>>> np.diag([1,2,3])
array([[1, 0, 0],
       [0, 2, 0],
       [0, 0, 3]])
```

à partir d'une liste

```
>>> np.diag((1,2,3))
array([[1, 0, 0],
       [0, 2, 0],
       [0, 0, 3]])
```

à partir d'un tuple

```
>>> np.diag(np.array([1,2,3]))
array([[1, 0, 0],
       [0, 2, 0],
       [0, 0, 3]])
```

à partir d'un vecteur

Remarque : on signale les fonctions `diagonal` à la portée un peu plus générale que `diag` et `diagflat`, et le fait que `diagonal` est aussi un *attribut* des tableaux `numpy`.

```
>>> a.diagonal()
array([ 0, 11, 22, 33])
```
attribut `diagonal` de l'objet a

```
>>> np.diag(a)
array([ 0, 11, 22, 33])
```
fonction `diag` du module `numpy`

```
>>> np.diagonal(a)
array([ 0, 11, 22, 33])
```
fonction `diagonal` du module `numpy`

Signalons la fonction `fill_diagonal`, qui comme son nom l'indique permet d'écrire sur la diagonale d'une matrice.

```
>>> a
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

```
>>> np.fill_diagonal(a,0); a
array([[0, 2, 3],
       [4, 0, 6],
       [7, 8, 0]])
```

```
>>> np.fill_diagonal(a,[-1,-2,-3]); a
>>> array([[-1,  2,  3],
           [ 4, -2,  6],
           [ 7,  8, -3]])
```

La fonction `tri` forme une matrice dont les coefficients valent 1 le long et en-dessous d'une k-ème parallèle à la diagonale, et 0 au-dessus. On peut préciser le « dtype » (c'est float par défaut) :

```
>>> np.tri(4)
array([[ 1.,  0.,  0.,  0.],
       [ 1.,  1.,  0.,  0.],
       [ 1.,  1.,  1.,  0.],
       [ 1.,  1.,  1.,  1.]])
```

```
>>> np.tri(4,k=1)
array([[ 1.,  1.,  0.,  0.],
       [ 1.,  1.,  1.,  0.],
       [ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.]])
```

```
>>> np.tri(4,k=-1,dtype=int)
array([[0, 0, 0, 0],
       [1, 0, 0, 0],
       [1, 1, 0, 0],
       [1, 1, 1, 0]])
```

Appliquée à un tableau `numpy` les fonctions `tril` (L pour *low*) et `triu` (U pour *up*) renvoie une copie du tableau où tous les coefficients au-dessus ou en-dessous d'une certaine diagonale ont été annulés (voir exemples) :

```
>>> a
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12],
       [13, 14, 15, 16]])
```

```
>>> np.tril(a)
array([[ 1,  0,  0,  0],
       [ 5,  6,  0,  0],
       [ 9, 10, 11,  0],
       [13, 14, 15, 16]])
```

```
>>> np.triu(a)
array([[ 1,  2,  3,  4],
       [ 0,  6,  7,  8],
       [ 0,  0, 11, 12],
       [ 0,  0,  0, 16]])
```

```
>>> np.tril(a,1)
array([[ 1,  2,  0,  0],
       [ 5,  6,  7,  0],
       [ 9, 10, 11, 12],
       [13, 14, 15, 16]])
```

```
>>> np.tril(a,-1)
array([[ 0,  0,  0,  0],
       [ 5,  0,  0,  0],
       [ 9, 10,  0,  0],
       [13, 14, 15,  0]])
```

```
>>> np.triu(a,-1)
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 0, 10, 11, 12],
       [ 0,  0, 15, 16]])
```

4.3 Tableaux de valeurs échelonnées

La fonction `arange` crée des vecteurs de valeurs régulièrement espacées.
La syntaxe est `arange(d,f,h,dtype=...)`, et génère les valeurs de l'intervalle $[d, f[$ avec un pas de h.

```
>>> np.arange(10)
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> np.arange(100,200,20)
array([100, 120, 140, 160, 180])
>>> np.arange(100,201,20)
array([100, 120, 140, 160, 180, 200])
```

```
>>> np.arange(10,0,-1)
array([10, 9, 8, 7, 6, 5, 4, 3, 2, 1])
>>> np.arange(8,dtype=float)
array([ 0., 1., 2., 3., 4., 5., 6., 7.])
>>> np.arange(8.)
array([ 0., 1., 2., 3., 4., 5., 6., 7.])
```

Pour les valeurs dans le type `float` ou `complex`, et pour éviter une erreur d'arrondi qui pourrait affecter la dernière valeur de l'intervalle, il peut être préférable d'utiliser la fonction `linspace`.

La syntaxe est `linspace(a,b,n)` et le résultat est un vecteur de n valeurs régulièrement échelonnées du segment $[a, b]$ (donc ici les extrémités sont incluses). Par défaut, $n = 50$. Si $b < a$, les valeurs sont obtenues dans l'ordre décroissant.

```
>>> np.linspace(0,100,4)
array([ 0., 33.33333333, 66.66666667, 100.])
>>> np.linspace(0,100,7)     # 7 valeurs de [0,100] dans l'ordre croissant
array([ 0., 16.66666667, 33.33333333, 50., 66.66666667, 83.33333333, 100.])
>>> np.linspace(100,0,7)     # 7 valeurs de [0,100] dans l'ordre décroissant
array([ 100., 83.33333333, 66.66666667, 50., 33.33333333, 16.66666667, 0.])
```

On pourra consulter la fonction `logspace` qui fournit des valeurs régulièrement espacées sur une échelle logarithmique.

jean-miche.ferrard@ac-paris.fr mathprepa.fr

4.4 Tableaux répondant à une formule donnée

La fonction `fromfunction` permet de construire un tableau dont le terme général obéit à une formule donnée.

```
>>> def f(i,j): return 10*i+j
>>> np.fromfunction(f,(4,5))
array([[  0.,   1.,   2.,   3.,   4.],
       [ 10.,  11.,  12.,  13.,  14.],
       [ 20.,  21.,  22.,  23.,  24.],
       [ 30.,  31.,  32.,  33.,  34.]])
```

```
>>> # le même tableau, en forçant le type int
>>> np.fromfunction(f,(4,5),dtype=int)
array([[ 0,  1,  2,  3,  4],
       [10, 11, 12, 13, 14],
       [20, 21, 22, 23, 24],
       [30, 31, 32, 33, 34]])
```

Le même tableau peut être obtenu (et on a souvent utilisé cette possibilité) en convertissant une liste (ou une liste de listes) « en compréhension » avec `array` (qui choisit le « data type » du tableau à partir des éléments données ici explicitement). Cette méthode est plus « dépensière » que `fromfunction` car elle fabrique la liste avant conversion.

```
>>> np.array([[f(i,j) for j in range(5)]
              for i in range(4)])
array([[ 0,  1,  2,  3,  4],
       [10, 11, 12, 13, 14],
       [20, 21, 22, 23, 24],
       [30, 31, 32, 33, 34]])
```

On peut utiliser une « lambda fonction » pour créer le tableau.

Ici on crée une matrice de Hilbert, de terme général $H_{i,j} = 1/(i+j+1)$ avec $i, j \geq 0$.

```
>>> np.fromfunction(lambda i,j: 1/(i+j+1),(4,4))
array([[ 1.        ,  0.5       ,  0.33333333,  0.25      ],
       [ 0.5       ,  0.33333333,  0.25      ,  0.2       ],
       [ 0.33333333,  0.25      ,  0.2       ,  0.16666667],
       [ 0.25      ,  0.2       ,  0.16666667,  0.14285714]])
```

▷ Déterminants de Vandermonde

Soit $x = (x_i)_{1 \leq i \leq n}$ une famille de n scalaires, et soit A la matrice carrée d'ordre n, de terme général $a_{ij} = x_i^{j-1}$.

On sait que la valeur du déterminant de A est : $\det A = \prod_{i<j} (x_j - x_i)$.

Dans `numpy`, on dispose d'une fonction `vander`, mais attention : l'indexation n'est pas conforme à notre définition :

```
>>> a = np.vander([1,10,100]); print(a)
[[    1     1   1]
 [  100    10   1]
 [10000   100   1]]
>>> np.linalg.det(a)
-80190.000000000102
```

```
>>> a = np.vander(range(1,6)); print(a)
[[  1   1   1   1   1]
 [ 16   8   4   2   1]
 [ 81  27   9   3   1]
 [256  64  16   4   1]
 [625 125  25   5   1]]
```

Voici comment écrire notre propre fonction `vandermonde` :

```
>>> def vandermonde(x): # ici on attend une liste ou un intervalle
        import numpy as np; n = len(x); x = np.array(x)
        return np.vstack([x**i for i in range(n)])
```

```
>>> a = vandermonde([1,10,100]); print(a)
[[    1     1     1]
 [    1    10   100]
 [    1   100 10000]]
>>> np.linalg.det(a)
80189.999999999971
```

```
>>> vandermonde(range(1,6))
array([[  1,   1,   1,   1,   1],
       [  1,   2,   4,   8,  16],
       [  1,   3,   9,  27,  81],
       [  1,   4,  16,  64, 256],
       [  1,   5,  25, 125, 625]])
```

4.5 Tableaux pseudo-aléatoires

Les fonctions qui forment des tableaux pseudo-aléatoires sont présentes dans le sous-module `random` du module `numpy`.

`random.rand` renvoie un tableau de valeurs pseudo-aléatoires dans l'intervalle $[0,1]$ (au sens de la distribution uniforme).

Elle prend comme argument un entier n_1 (respectivement une séquence $n_1, n_2, \ldots$) et renvoie un vecteur de longueur n_1 (respectivement un tableau de format $n_1 \times n_2 \times \cdots$).

La fonction `random.sample` fait la même chose que `random.rand` mais l'argument est un tuple.

La fonction `random.seed` réinitialise le générateur de nombres aléatoires (argument entier).

```
>>> np.random.rand(6)
array([ 0.14243652,  0.77345108,  0.76627661,  0.41079884,  0.79899263, 0.95726012])
>>> np.random.seed(0)           # réinitialise le générateur de nombres aléatoires
>>> np.random.rand(2,6)         # noter la différence de syntaxe avec random.sample (ci-dessous)
array([[ 0.5488135 ,  0.71518937,  0.60276338,  0.54488318,  0.4236548 , 0.64589411],
       [ 0.43758721,  0.891773  ,  0.96366276,  0.38344152,  0.79172504, 0.52889492]])
```

La fonction `random.randn` suit la même syntaxe que `random.rand`, mais elle renvoie un échantillon de valeurs pseudo-aléatoires au sens de la loi normale réduite (c'est-à-dire d'espérance 0 et d'écart-type 1).

```
>>> np.random.randn(6)
array([-0.75931933,  0.49230711,  0.37242323,  1.17957196, -2.26637722, 1.05330359])
```

Appelées avec un argument vide, les fonctions `random.rand` et `random.randn` renvoie un nombre pseudo-aléatoire (respectivement pour la loi uniforme de $[0,1]$ ou pour la loi normale $\mathcal{N}(0,1)$).

La fonction `random.randint` renvoie une valeur ou un tableau de valeurs entières pseudo-aléatoires au sens de la distribution uniforme dans un intervalle semi-ouvert $[a,b[$.

La syntaxe est `random.randint(a,b)` pour une seule valeur, `random.randint(a,b,n)` pour un vecteur de longueur n, et `random.randint(a,b,(n,p))` pour une matrice de format $n \times p$.

Si on omet la valeur a, les entiers pseudo-aléatoires sont choisis dans $[0,b[$. Pour éviter toute ambiguité, on pourra nommer `size=...` l'argument de taille et `high=...` l'argument donnant la valeur maximum (exclue).

```
>>> np.random.randint(100)              # un seul entier pseudo-aléatoire dans [0,99[
72
>>> np.random.randint(1,7,size=10)
array([1, 3, 1, 5, 2, 6, 2, 4, 3, 6])   # dix lancers d'un dé à six faces
>>> np.random.randint(2,size=(2,8))     # un tableau de 2×8 entiers choisis dans {0,1}
array([[1, 1, 0, 0, 0, 0, 0, 0],
       [1, 1, 1, 0, 1, 0, 1, 1]])
```

La fonction `random.random_integers()` est très proche de la fonction `random.randint`.

La différence est `random.random_integers(a,b,(n,p))` renvoie des valeurs entières dans le segment $[a,b]$ et que si a est absent alors la valeur minimum est 1.

```
>>> np.random.random_integers(0,2,size=15)    # entiers choisis dans {0,1,2}
array([1, 1, 0, 0, 2, 1, 1, 2, 1, 2, 1, 1, 0, 1, 1])
```

La fonction `random.choice` renvoie un échantillon aléatoire de valeurs extraites d'un tableau a.

```
>>> a = np.array([1,8,3,1,7,4,5,3,5,4,1,7,2])   # un vecteur quelconque
>>> np.random.choice(a)                         # une valeur choisie dans a
4
>>> np.random.choice(a,10)                      # une succession de 10 valeurs choisies dans a
array([1, 1, 7, 3, 1, 7, 3, 3, 2, 3])
>>> np.random.choice(a,(2,10))                  # un tableau 2×10 de valeurs choisies dans a
array([[1, 1, 7, 4, 4, 5, 5, 7, 8, 7],
       [5, 3, 7, 3, 7, 1, 1, 4, 2, 4]])
```

La fonction `random.choice` accepte un argument nommé `replace=True/False` qui indique si le tirage de l'échantillon doit s'effectuer avec ou sans remise (`True` par défaut), et un argument nommé `p=...` qui permet de spécifier les probabilités avec lesquelles chacune des valeurs du tableau peut apparaître (par défaut : probabilité uniforme).

De plus, le premier argument (le tableau a) peut être remplacé par un entier n, et le tirage s'effectue dans $[\![0,n[\![$.

```
>>> np.random.choice(10,size=15)                 # 15 valeurs de [0,10[, tirage avec remise
array([0, 2, 7, 2, 9, 2, 3, 3, 2, 3, 4, 1, 2, 9, 1])
>>> np.random.choice(10,size=7,replace=False)    # 7 valeurs de [0,10[, tirage sans remise
array([1, 9, 8, 5, 2, 7, 0])
>>> probs = [0.1, 0.1, 0.6, 0.1, 0.1]            # une liste de 5 probabilités (somme = 1)
>>> np.random.choice(5,size=15)                  # 15 valeurs de [0,5[, avec équiprobabilité
array([3, 0, 2, 2, 0, 4, 3, 4, 0, 4, 3, 3, 4, 1, 3])
>>> np.random.choice(5,size=15,p=probs)          # 15 valeurs de [0,5[, avec probabilités probs
array([3, 2, 3, 2, 2, 2, 4, 2, 2, 2, 0, 2, 2, 2, 2], dtype=int32)
```

La fonction `random.shuffle` rebat aléatoirement (et « en place ») les éléments d'un vecteur.

```
>>> a = np.array([1,7,3,8,5,7,1,2,7])            # un vecteur de 9 valeurs
>>> np.random.shuffle(a); a                      # rebat les éléments de a, sur place
array([7, 1, 8, 7, 1, 5, 2, 7, 3])
>>> a = np.arange(15); a                         # le vecteur des entiers de 0 à 14
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
>>> np.random.shuffle(a); a                      # rebat les éléments de a, sur place
array([ 0,  8,  9, 14,  3,  4,  2, 12, 10,  1,  6, 11, 13,  7,  5])
```

La fonction `random.permutation` est analogue à `shuffle`, sauf que le résultat n'est pas calculé « en place » et qu'elle accepte un entier n comme argument (le résultat est alors une permutation de l'intervalle d'entiers $[\![0,n[\![$).

La fonction `permutation` accepte aussi un intervalle comme argument.

```
>>> np.random.permutation(15)                    # une permutation de [0,15[
array([ 7,  3,  4,  6, 14, 11, 10,  8,  1,  0,  2, 12,  9,  5, 13])
>>> np.random.permutation(range(10,21))          # une permutation de [10,21[
array([17, 11, 10, 15, 18, 13, 20, 19, 12, 14, 16])
>>> a = np.array([1,7,3,8,5,7,1,2,7])            # un vecteur de 9 valeurs
>>> np.random.permutation(a)                     # une permutation de ces neuf valeurs
array([3, 7, 5, 1, 8, 7, 2, 1, 7])
>>> a                                            # mais le contenu de a reste inchangé
array([1, 7, 3, 8, 5, 7, 1, 2, 7])
```

4.6 Probabilités, lois discrètes usuelles

Le module `numpy` offre des fonctions renvoyant des échantillons de valeurs répondant aux lois de probabilités usuelles.

▷ Loi binomiale

`random.binomial(n,p)` renvoie une valeur représentant une réalisation de la loi binomiale $\mathcal{B}(n,p)$.

Il s'agit donc du nombre de succès à l'issue de n tentatives avec une probabilité p de succès à chaque fois.

Un troisième argument faculatif `size=...` permet de définir un format de sortie sous la forme d'un tableau (vecteur, matrice) et donc de générer autant de valeurs que nécessaire pour remplir ce tableau. Le nom `size` est faculatif.

```
>>> np.random.binomial(100,1/2)           # nombre de succès après 100 essais avec p=1/2
46
>>> np.random.binomial(100,1/3,size=10)   # répéter 10 fois une série de 100 essais avec p=1/3
array([37, 36, 36, 42, 40, 31, 32, 26, 37, 32])
>>> np.random.binomial(1000,0.75,(3,10))  # répéter 3×10 fois une série de 1000 essais avec p=3/4
array([[758, 771, 743, 726, 761, 754, 748, 745, 762, 723],
       [738, 737, 758, 758, 740, 735, 766, 757, 724, 750],
       [723, 764, 743, 769, 749, 755, 745, 740, 740, 755]])
```

▷ Loi géométrique

`random.geometric(p)` renvoie une réalisation de la loi géométrique de paramètre p, c'est-à-dire le temps d'attente du premier succès (ou encore le nombre d'échecs avant celui-ci) dans une répétition de tentatives indépendantes ayant chacune une probabilité p de succès.

Le paramètre facultatif `size=...` permet de répéter cette expérience.

```
>>> e = np.random.geometric(1/10,size=10)
>>> e
array([15,  3,  2, 38, 13,  3,  1, 30, 15, 49])
```

10 fois de suite, on a attendu le premier succès, avec $p = 1/10$ à chaque tentative

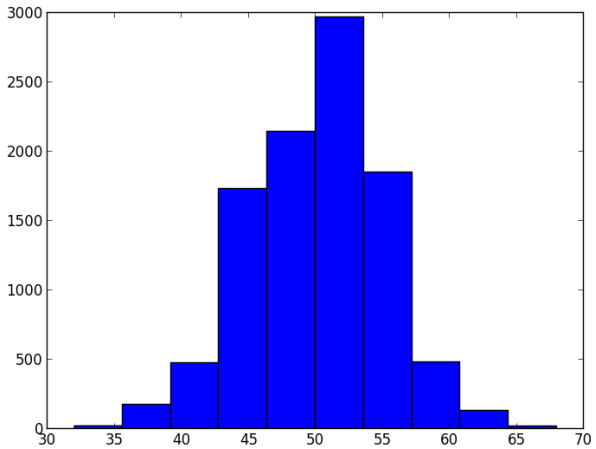
```
>>> e = np.random.geometric(1/4,size=10000)
>>> e.mean(), e.var()
(4.0233999999999996, 11.675052439999122)
```

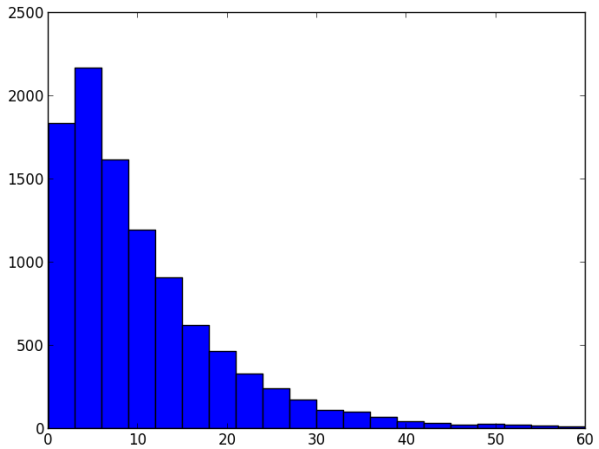
10000 fois de suite avec $p = 1/4$ moyenne et variance des observations

Pour illustrer les deux lois précédentes, on crée un échantillon de 10000 valeurs (à gauche suivant la loi binomiale de paramètre $1/2$, à droite suivant la loi géométrique de paramètre $1/10$), on forme un histogramme des valeurs obtenues en 10 intervalles égaux (à gauche, sur l'ensemble des valeurs obtenues) et en vingt intervalles (à droite, et sur $[0, 60]$) :

```
>>> import matplotlib.pyplot as plt
>>> a = np.random.binomial(100,1/2,size=10000)
>>> plt.show(plt.hist(a))
```

```
>>> a = np.random.geometric(1/10,size=10000)
>>> plt.show(plt.hist(a,bins=20,range=(0,60)))
>>>
```





▷ Loi binomiale négative

`random.negative_binomial(n,p)` renvoie une réalisation de la loi binomiale négative de paramètres (n,p).

Cette loi mesure le nombre d'échecs avant d'atteindre le n-ième succès dans une répétition de tentatives indépendantes ayant chacune une probabilité p de succès.

Là encore le paramètre facultatif `size=...` permet de répéter cette expérience.

```
>>> np.random.negative_binomial(100,1/2)  # avec p=1/2, on a connu 118 échecs avec le 100ème succès
118
>>> e = np.random.negative_binomial(100,1/3,size=10); e  # répéter 10 fois l'expérience avec p=1/3
array([204, 206, 206, 185, 218, 179, 181, 191, 214, 219])
>>> e.mean()  # on voit que la moyenne du nombre d'échecs avant le 100ème succès est proche de 200
200.30000000000001
```

▷ Loi hypergéométrique

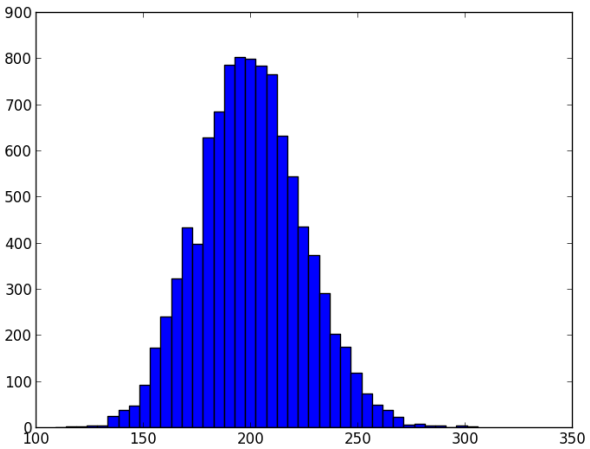
`random.hypergeometric(`n_{good}`,`n_{bad}`,N)` renvoie une réalisation de la loi hypergéométrique de paramètres (n_{good}, n_{bad}, N).

Si on considère le modèle classique d'une boîte contenant n_{good} jetons gagnants et n_{bad} jetons perdants, cette loi mesure le nombre de jetons gagnants obtenus après N tirages *sans remise* (on suppose donc que $N \leq n_{good} + n_{bad}$).

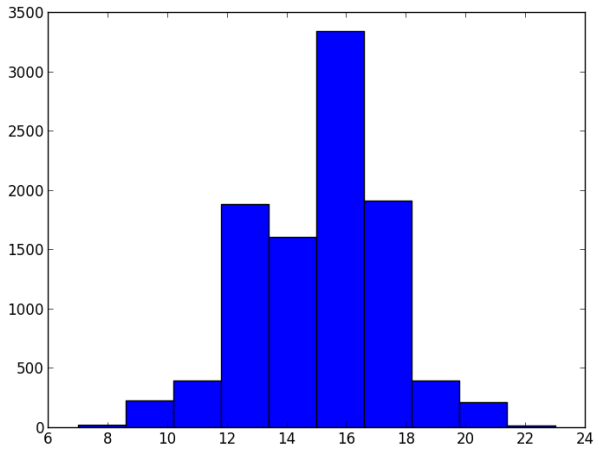
```
>>> e = np.random.hypergeometric(30,70,50); e  # 30 jetons ok, 70 "pas ok", 50 tirages sans remise
16                                             # ici on a obtenu 16 jetons "ok"
>>> e = np.random.hypergeometric(30,70,50,size=15); e  # répète 15 fois la même expérience
array([17, 17, 20, 15, 17, 13, 17, 13, 15, 13, 17, 14, 17, 14, 12])
>>> e.mean(), e.var()                          # moyenne et variance de l'ensemble des 15 résultats
(15.4, 4.6399999999999997)
```

Pour illustrer les deux lois précédentes, on crée un échantillon de 10000 valeurs (à gauche suivant la loi binomiale négative de paramètres $100, 1/3$, et à droite suivant la loi hypergéométrique de paramètres $30, 70, 50$).
On forme un histogramme des valeurs obtenues en 40 intervalles égaux (à gauche) et en dix intervalles (à droite), sur l'ensemble des valeurs obtenues :

```
>>> import matplotlib.pyplot as plt
>>> rnb = np.random.negative_binomial
>>> a = rnb(100,1/3,size=10000)
>>> plt.show(plt.hist(a,bins=40))
```




```
>>> rhyp = np.random.hypergeometric
>>> a = rhyp(30,70,50,size=10000)
>>> plt.show(plt.hist(a))
>>>
```



▷ Loi multinomiale

`multinomial(n,probs)` renvoie une réalisation de la loi multinomiale de paramètres $n, probs$. Ici p représente un vecteur $[p_1, p_2, \dots, p_k]$ de probabilités (donc les p_i sont dans $[0,1]$ et leur somme vaut 1).

Il s'agit ici de répéter n fois (indépendamment) une même expérience possédant k résultats possibles $r_1, r_2, \dots, r_k$ (avec p_i la probabilité du résultat r_i) et de lire le vecteur $[n_1, n_2, \dots, n_k]$ des occurences de chacun de ces résultats.

```
>>> np.random.multinomial(600,[1/6.]*6)     # on lance 600 fois un dé honnête
array([105,  89,  93,  92, 105, 116])       # nombre de fois où on a obtenu 1, ou 2, ou 3, etc.
>>> np.random.multinomial(600,[1/6.]*6,2)   # répète deux fois l'expérience précédente
array([[123,  82, 117, 102,  89,  87],
       [ 94, 108,  96, 111,  90, 101]])
```

Ici on effectue 6000 fois la même expérience aléatoire comportant trois résultats possibles avec les probabilités respectives 1/2, 1/3, 1/6. On obtient respectivement 3006 fois, 1932 fois, 1002 fois les trois résultats possibles :

```
>>> np.random.multinomial(6000,[1/2,1/3,1/6])
array([3066, 1932, 1002])
```

▷ Loi géométrique

`random.poisson(λ)` renvoie des réalisations de la loi géométrique de paramètre λ.

Cette loi, d'image $X(\Omega) = \mathbb{N}$ est définie par $p_\lambda(X = k) = \mathrm{e}^{-\lambda}\dfrac{\lambda^k}{k!}$ pour tout k de $\mathbb{N}$.

D'espérance λ, elle est une bonne approximation de la loi binomiale $\mathcal{B}\left(n, p = \dfrac{\lambda}{n}\right)$ avec « p petit et n grand ».

Si un événement rare est susceptible de se présenter avec une probabilité $p \ll 1$ (ou encore λ fois) dans un grand intervalle de temps, alors $p_\lambda(X, k)$ mesure la probabilité qu'il survienne k fois dans un tel intervalle de temps.

Dans l'exemple ci-dessous on considère un évènement susceptible de se présenter deux fois dans un intervalle de temps de longueur n (avec n grand). On répète 25 fois l'expérience qui consiste à observer combien de fois cet évènement s'est effectivement produit dans un intervalle de temps $[t_0, t_0 + n]$.

```
>>> np.random.poisson(lam=2,size=25)
array([1, 1, 3, 2, 0, 3, 2, 1, 4, 3, 3, 4, 2, 1, 1, 3, 6, 1, 2, 4, 2, 0, 4, 2, 1])
```

4.7 Probabilités, lois continues usuelles

`random.uniform([low, high, size])` renvoie une/des réalisation(s) de la loi uniforme sur $[low, high]$ (défaut $[0,1]$).

Comme avec toutes les lois, l'argument facultatif `size=...` permet de générer un tableau de résultats :

```
>>> np.random.uniform(10)                       # une réalisation de la loi uniforme sur [0,10]
6.022781406955742
>>> np.random.uniform(10,11)                    # une réalisation de la loi uniforme sur [10,11]
10.440442014479052
>>> np.random.uniform(100,101,5)                # loi uniforme sur [100,101], cinq fois
array([ 100.25736027,  100.77687524,  100.114918  ,  100.72635488, 100.92869012])
>>> np.random.uniform(low=-1,high=1,size=(2,6)) # loi uniforme sur [-1,1], 2 × 6 résultats
array([[ 0.97307878,  0.69721654,  0.78160556,  0.6497864 , -0.95157743, -0.7137485 ],
       [ 0.10387055,  0.04972031, -0.18853763,  0.87346586,  0.92676752, 0.96173206]])
```

`random.exponential([beta, size])` renvoie une/des réalisation(s) de la loi exponentielle de paramètre $\lambda = \dfrac{1}{\beta}$.

C'est une loi continue, définie sur $\mathbb{R}^+$, dont la densité s'écrit $f(x) = \lambda \mathrm{e}^{-\lambda x} = \dfrac{1}{\beta}\exp\left(-\dfrac{x}{\beta}\right)$. Son espérance est β.

Elle est « sans mémoire » : pour tous $t > 0$ et $h > 0$, on a : $p(X > t + h | X > t) = p(X > h)$.

```
>>> np.random.exponential(scale=10,size=7)   # 7 réalisations, loi exponentielle de paramètre 1/10
array([ 7.06399986, 10.55772426, 1.78660195, 0.82879292, 14.49713449, 9.4533084, 10.96488939])
```

`random.normal([m, σ, size])` renvoie une/des réalisation(s) de la loi normale d'espérance m, d'écart-type σ.

C'est une loi continue, définie sur $\mathbb{R}$, dont la densité s'écrit $f(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x-m)^2}{2\sigma^2}\right)$.

```
>>> np.random.normal(5,1,6)                    # 6 fois la loi normale d'espérance 5 d'écart-type 1
array([ 4.95536402,  5.06527611,  5.99306094,  5.63459564,  5.28943576,  4.28950257])
>>> np.random.normal(loc=5,scale=1,size=6)     # on recommence en nommant les arguments
array([ 3.29365791,  4.13874059,  3.45334375,  3.25639479,  6.81705112,  3.6577833 ])
>>> np.random.normal(size=6,scale=1,loc=1)     # arguments nommés => ordre quelconque
array([ 1.38367825,  -0.31248894,  3.60537642,  2.7389594 ,  4.29064175, 1.57216177])
```

`random.standard_normal([size])` renvoie une/des réalisation(s) de la loi normale d'espérance 0, d'écart-type 1.

```
>>> np.random.standard_normal(6)            # 6 réalisations de la loi normale centrée réduite
array([-0.17302555,  1.18449125, -1.23138598, -0.70714673,  0.07684741, -2.28847487])
>>> e = np.random.standard_normal(1000)     # 1000 réalisations (on n'affiche pas!)
>>> e.mean(), e.std()                       # moyenne et écart-type des résultats
(-0.021763362781015889, 0.95459943994848417)
```

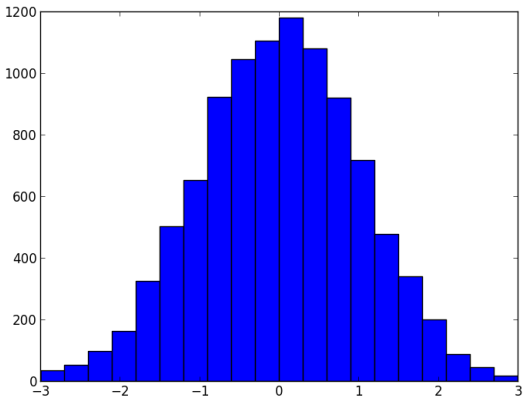
Bon, il y en a encore pas mal comme ça...

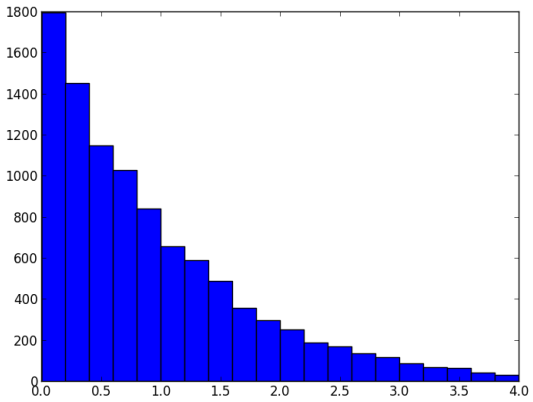
Le mieux est d'aller voir http://docs.scipy.org/doc/numpy/reference/routines.random.html

Mais pour finir, voici deux histogrammes.
On crée un échantillon de 10000 valeurs (à gauche suivant la loi normale centrée réduite, à droite suivant la loi exponentielle de paramètre 1), on forme un histogramme en 20 intervalles égaux (à gauche sur $[-3, 3]$, à droite sur $[0, 4]$) :

```
>>> import matplotlib.pyplot as plt
>>> a = np.random.standard_normal(10000)
>>> plt.show(plt.hist(a,bins=20,range=(-3,3)))
```

```
>>> a = np.random.exponential(1,10000)
>>> plt.show(plt.hist(a,bins=20,range=(0,4)))
>>>
```





Chapitre 5

Fonctions universelles

Pour la documentation officielle en anglais, voir : http://docs.scipy.org/doc/numpy/reference/ufuncs.html

Une *fonction universelle* (le terme exact est « ufunc », abréviation de *universal function*) est une fonction qui peut s'appliquer terme à terme aux éléments d'un tableau.

Si f est une *ufunc* et si $a = [a_0, a_1, \ldots, a_{n-1}]$ est un tableau, alors $f(a)$ renvoie le tableau $[f(a_0), f(a_1), \ldots, f(a_{n-1})]$.

Les a_k peuvent être des tableaux, par exemple les lignes d'une matrice m. La fonction f s'applique alors récursivement aux éléments de m. Un grand nombre de fonctions usuelles sont directement « universalisées » dans `numpy`.

Les fonctions mathématiques gardent le même nom (préfixé par `np` si on a importé `numpy` par `import numpy as np`).

Si on effectue une opération arithmétique entre un tableau a et un scalaire x, tout se passe comme si x était élevé au rang de tableau constant de même format que a.

Le mieux est de commencer par quelques opérations arithmétiques simples sur un vecteur ou une matrice.

```
>>> a = np.arange(1,10); a
array([1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> a + 1
array([ 2,  3,  4,  5,  6,  7,  8,  9, 10])
>>> 10 * a
array([10, 20, 30, 40, 50, 60, 70, 80, 90])
>>> a + a
array([ 2,  4,  6,  8, 10, 12, 14, 16, 18])
>>> a * a
array([ 1,  4,  9, 16, 25, 36, 49, 64, 81])
```

```
>>> m = np.arange(15).reshape(3,5); m
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
>>> m + 10
array([[10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19],
       [20, 21, 22, 23, 24]])
>>> m * m  # ce n'est pas le produit matriciel!!
array([[  0,   1,   4,   9,  16],
       [ 25,  36,  49,  64,  81],
       [100, 121, 144, 169, 196]])
```

Continuons avec des fonctions mathématiques usuelles appliquées à un vecteur a :

```
>>> a = np.arange(1,13); a
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12])
>>> 2 ** a      # 2 à la puissance chaque élément de a
array([ 2,  4,  8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096], dtype=int32)
>>> 1/a         # le vecteur des inverses
array([ 1.        ,  0.5       ,  0.33333333,  0.25      ,  0.2       ,  0.16666667,
        0.14285714,  0.125     ,  0.11111111,  0.1       ,  0.09090909,  0.08333333])
>>> np.log(a)   # le vecteur des logarithmes népériens
array([ 0.        ,  0.69314718,  1.09861229,  1.38629436,  1.60943791,  1.79175947,
        1.94591015,  2.07944154,  2.19722458,  2.30258509,  2.39789527,  2.48490665])
```

Deux remarques importantes :

- l'expression `a*b` renvoie le tableau des produits terme à terme des éléments de a et b : elle *ne désigne donc pas un produit matriciel* au sens où on l'entend habituellement (à suivre).
- rappelons que l'opérateur `^` désigne le « ou exclusif » sur les entiers, et pas l'élévation à une puissance.

```
>>> 2 ** np.arange(10)   # on calcule ici les dix premières puissances de 2
array([ 1, 2, 4, 8, 16, 32, 64, 128, 256, 512], dtype=int32)
```

5.1 Opérations arithmétiques

`add(a,b)` additionne terme à terme les éléments de a et b (autre syntaxe possible : $a+b$)

`subtract(a,b)` soustrait terme à terme les éléments de b à ceux de a (autre syntaxe possible : $a-b$)

`multiply(a,b)` multiplie terme à terme les éléments de a et b (autre syntaxe possible : $a*b$)

`divide(a,b)` quotients (flottants) terme à terme des éléments de a par ceux de b (autre syntaxe : a/b)

```
>>> a = np.arange(0,600,step=120);a
array([ 0, 120, 240, 360, 480])
>>> b  = np.array([10**k for k in range (5)]); b
array([ 1, 10, 100, 1000, 10000])
```

```
>>> np.divide(a,b)
array([ 0., 12., 2.4, 0.36, 0.048])
>>> a/b
array([ 0., 12., 2.4, 0.36, 0.048])
```

`floor_divide(a,b)` quotients (entiers) des divisions des éléments de a par ceux de b (autre syntaxe : $a//b$)

```
>>> a = np.arange(0,600,step=120);a
array([ 0, 120, 240, 360, 480])
>>> b  = np.array([10**k for k in range (5)]); b
array([ 1, 10, 100, 1000, 10000])
```

```
>>> np.floor_divide(a,b)
array([ 0, 12,  2,  0,  0], dtype=int32)
>>> a // b
array([ 0, 12,  2,  0,  0], dtype=int32)
```

`power(a,b)` élève les éléments de a à la puissance les éléments de b

`mod(a,b)` donne les restes dans les divisions des éléments de a par ceux de b

On dispose aussi des possibilités suivantes, qui prennent en argument un tableau a :

`negative(a)` (tableaux des opposés : autre syntaxe possible $-a$), `absolute(a)` (modules), `sign(a)` (signes)

Pour les arrondis :

`rint(a)` (à l'entier), `floor(a)`, `ceil(a)`, `trunc(a)` (troncature), `round_(a,n)` (arrondi à n décimales)

Pour la racine carrée et l'élévation au carré (toujours terme à terme) : `sqrt` et `square`.

5.2 Fonctions mathématiques usuelles

Toutes les fonctions mathématiques usuelles sont présentes sous forme universelle dans le module `numpy`. Elles gardent le même nom, mais comme nous l'avons déjà indiqué, il faut bien penser à les préfixer par `np`.

On dispose notamment des fonctions trigonométriques (circulaires et hyperboliques) directes et inverses :

`sin` `cos` `tan` `arcsin` `arccos` `arctan` `sinh` `cosh` `tanh` `arcsinh` `arccosh` `arctanh`

Et puisqu'on est dans la trigonométrie, et pour toute correspondance entre un x de a et un y de b :

`hypot(a,b)` renvoie les hypoténuses $\sqrt{x^2+y^2}$ `arctan2(a,b)` renvoie les angles polaires des points (y,x)

`degrees(a)` (ou encore `rad2deg`) convertit les angles x de radians en degrés

`radians(a)` (ou encore `deg2rad`) convertit les angles x de degrés en radians

On dispose bien sûr des fonctions exponentielles et logarithmiques :

`exp` `expm1` : `expm1(x)` signifie e^x-1 et est plus précis que `exp(x)-1` pour x proche de 0.

`log` `log10` `log2` : logarithme népérien (resp. de base 10, de base 2)

`log1p` : `log1p(x)` signifie $\ln(1+x)$ et est plus précis que `log(1+x)` pour x proche de 0).

`exp2` : `exp2(x)` signifie `2**x`, c'est-à-dire 2^x.

```
>>> x = 10**np.arange(-10,-15,-1.); x
array([  1.00000000e-10,   1.00000000e-11,   1.00000000e-12, 1.00000000e-13, 1.00000000e-14])
>>> np.log(1+x)/x
array([ 1.00000008,  1.00000008,  1.0000889 ,  0.99920072,  0.99920072])
>>> np.log1p(x)/x
array([ 1.,  1.,  1.,  1.,  1.])
```

```
>>> a = np.arange(1,6); a
array([1, 2, 3, 4, 5])
>>> np.sqrt(a)          # ne pas oublier le préfixe np
array([ 1.        ,  1.41421356,  1.73205081,  2.        ,  2.23606798])
>>> np.exp(a)
array([ 2.71828183,   7.3890561, 20.08553692, 54.59815003, 148.4131591])
>>> a                   # à ce stade le contenu de a n'a pas changé
array([1, 2, 3, 4, 5])
>>> np.exp(a,a)         # ici le résultat va dans a, mais le data-type est inchangé!!!
array([  2,   7,  20,  54, 148])
>>> a                   # la preuve
array([  2,   7,  20,  54, 148])
```

5.3 Variantes de syntaxe

Toutes les fonctions universelles acceptent un *argument supplémentaire facultatif* sous la forme d'un identificateur désignant lui-même un tableau `numpy` devant recevoir le résultat de l'opération (attention : le tableau cible conserve son *data type*, indépendamment du type du résultat).

Les « fonctions universelles » du module numpy sont intelligentes au point d'accepter des arguments du type « array like » (des listes, des tuples) et de les convertir en le tableau correspondant avant d'effectuer l'opération. Il est impossible de décrire le nombre de possibilités qui résultent de cette tolérance...

```
>>> a = np.array([10, 17, 5, 28])
>>> b = np.array([3, 5, 2, 8])
>>> np.add(a,b,a)   # a+b va dans a
array([13, 22,  7, 36])
>>> b
array([3, 5, 2, 8])
>>> a
array([13, 22,  7, 36])
```

```
>>> np.add([1,2,3],(4,5,6)) # une liste, un tuple
array([5, 7, 9])
>>> np.add([[1,2],[3,4]],10) # liste de liste, un scalaire
array([[11, 12],
       [13, 14]])
>>> np.power(2,[(1,2),(3,4)]) # improbable!
array([[ 2,  4],
       [ 8, 16]], dtype=int32)
```

Sur l'exemple suivant, on illustre la convergence vers $\ell = 2$ de la suite définie par $u_0 > 0$ et $u_{n+1} = \dfrac{u_n}{2} + \dfrac{2}{u_n}$

Pour cela, on forme l'échantillon $[1, 2, \ldots, 7]$ de valeurs de α.

On évalue alors plusieurs fois l'expression `np.add(a/2,2/a,a)` qui calcule le tableau des $x/2 + 2/x$ pour chaque x de a et qui, par « effet de bord », place le résultat dans le tableau a lui-même. On peut ainsi suivre la convergence (rapide car quadratique) de sept suites (u_n) en parallèle.

```
>>> a = np.arange(1,8,dtype=float); a
array([ 1.,  2.,  3.,  4.,  5.,  6.,  7.])
>>> np.add(a/2,2/a,a)
array([ 2.5       ,  2.        ,  2.16666667,  2.5       ,  2.9       ,  3.33333333,  3.78571429])
>>> np.add(a/2,2/a,a)
array([ 2.05      ,  2.        ,  2.00641026,  2.05      ,  2.13965517,  2.26666667,  2.42115903])
>>> np.add(a/2,2/a,a)
array([ 2.00060976,  2.        ,  2.00001024,  2.00060976,  2.00455764,  2.01568627,  2.03663017])
>>> np.add(a/2,2/a,a)
array([ 2.00000009,  2.        ,  2.        ,  2.00000009,  2.00000518,  2.00006104,  2.00032941])
```

Attention ! si on reprend l'exemple précédent, mais après avoir défini a par `a = np.arange(1,8)` (donc avec un type entier), alors a reste de type entier (la « convergence » vers 2 n'a ici plus du tout la même signification)

Ce comportement est normal, car tout cela s'effectue « en place » sans création d'un nouveau tableau en mémoire.

```
>>> a = np.arange(1,8); a
array([1, 2, 3, 4, 5, 6, 7])
>>> np.add(a/2,2/a,a)
array([2, 2, 2, 2, 2, 3, 3])
>>> np.add(a/2,2/a,a)
array([2, 2, 2, 2, 2, 2, 2])
```


jean-miche.ferrard@ac-paris.fr mathprepa.fr

5.4 Vectorisation d'une fonction

Pour qu'une fonction f puisse s'appliquer, terme à terme, à tous les éléments d'un ou de plusieurs tableaux (selon que f accepte un ou plusieurs arguments), il faut la « vectoriser ».

Considérons par exemple la fonction $f : x \mapsto \dfrac{x+\sin(x)}{x+\cos(x)}$.

```
>>> def f(x): return (x+sin(x))/(x+cos(x))
```

Tenter d'appliquer la fonction f à un tableau `numpy` conduit à une erreur :

```
>>> a = np.arange(1,7); a
array([1, 2, 3, 4, 5, 6])
```

```
>>> f(a)
TypeError: only length-1 arrays can be converted to Python scalars
```

Il faut donc utiliser la fonction `vectorize` pour rendre f universelle :

```
>>> vf = np.vectorize(f)          # la version vectorisée de f
>>> vf(a)                         # on peut applique f terme à terme aux éléments de a
array([ 1.1955257 ,  1.83684794,  1.56274044,  0.96917278,  0.76482477,  0.82190295])
```

L'exemple précédent ne nécessitait pas en fait qu'on vectorise f. Celle-ci est en fait une composée de fonctions et d'opérations usuelles (toutes déjà vectorisées), et on pouvait donc obtenir le même résultat de la manière suivante :

```
>>> (a+np.sin(a))/(a+np.cos(a))
array([ 1.1955257 ,  1.83684794,  1.56274044,  0.96917278,  0.76482477,  0.82190295])
```

Il n'est cependant pas certain que cette deuxième méthode soit la plus efficace, car elle occasionne plusieurs créations de tableaux avant de rendre son résultat (la méthode avec `vectorize(f)` se contenant d'une boucle sur le tableau initial).

Voici un autre exemple, où il est nécessaire de vectoriser l'application f car sa définition comporte un test. Ici $f(x,y)$ renvoie 0 si $x < y$ et elle renvoie y sinon :

```
>>> a = np.arange(1,7)
>>> a
array([1, 2, 3, 4, 5, 6])
>>> b = np.array([4,1,8,7,2,9])
>>> b
array([4, 1, 8, 7, 2, 9])
```

```
>>> def f(x,y): return 0 if x < y else y
>>> vf = np.vectorize(f)
>>> vf(a,b)
array([0, 1, 0, 0, 2, 0])
>>> vf(b,a)
array([1, 0, 3, 4, 0, 6])
```

En continuant sur cet exemple, on voit bien que la vectorisée de f agit comme une « ufunc » (fonction universelle) :

```
>>> vf(a,3)
array([0, 0, 3, 3, 3, 3])
>>> vf(3,b)
array([0, 1, 0, 0, 2, 0])
```

```
>>> vf(5,[[1,6,2],[3,4,8],[6,3,2]])
array([[1, 0, 2],
       [3, 4, 0],
       [0, 3, 2]])
```

`piecewise(a,[conds],[images])` permet de calculer l'image d'un tableau par une fonction f par morceaux. Chaque $f(x)$ est calculé en choisissant dans *images* ce qui correspond au premier test vérifié dans *conds*. Ici, par exemple, on associe -1 (resp. $0, 1$) suivant que l'élément x du tableau a vérifie $x < 30$ (resp. $30 \leq x < 70$, $x \geq 70$).

```
>>> a = np.random.random_integers(0,100,10); a
array([14, 35, 22, 55, 47, 71, 35, 65, 79, 61])
>>> np.piecewise(a, [a < 30, a < 70, a >= 70], [-1,0,1])
array([0, 0, 0, 0, 0, 1, 0, 0, 1, 0])
>>> a = np.random.random_integers(0,100,10); a
array([71, 74, 55, 15,  2, 44, 16, 41, 92, 75])
>>> np.piecewise(a, [a < 30, a < 70, a >= 70], [-1,0,1])
array([1, 1, 0, 0, 0, 0, 0, 0, 1, 1])
```

`apply_along_axis` permet d'appliquer une même fonction suivant les lignes, ou suivant les colonnes.

```
>>> m = np.arange(15).reshape(3,5); m
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
```

```
>>> np.apply_along_axis(np.mean,0,m) # moyenne lignes
array([ 5.,  6.,  7.,  8.,  9.])
>>> np.apply_along_axis(np.mean,1,m) # moyenne colonnes
array([ 2.,  7., 12.])
```

À voir aussi, les méthodes `reduce`, `accumulate` et `reduceat` des fonctions universelles.

5.5 Opérations logiques sur tableaux booléens

Si a est un tableau de booléens, l'expression `logical_not(a)` renvoie le tableau des valeurs booléennes contraires.

```
>>> a = np.array([False, True, False,  False,  True, False, True, True])
>>> print(a)
[False  True False False  True False  True  True]
>>> print(np.logical_not(a))
[ True False  True  True False  True False False]
```

Les fonctions `logical_and` `logical_or` `logical_xor` prennent en argument deux tableaux a et b de booléens. Elles appliquent une certaine fonction logique aux éléments de a et b, terme à terme, et renvoient le tableau des résultats. Avec un troisième argument, on peut spécifier le nom d'un tableau de booléens destiné à recevoir le résultat.

```
>>> a = np.array([False, True, False,  False,  True, False, True, True])
>>> b = np.array([True, True, False,  True,  True, False, False, True])
>>> np.logical_and(a,b)
array([False,  True, False, False,  True, False, False,  True], dtype=bool)
>>> np.logical_or(a,b)
array([ True,  True, False,  True,  True, False,  True,  True], dtype=bool)
```

5.6 Opérations binaires sur les tableaux d'entiers

Les fonctions `bitwise_and` `bitwise_or` `bitwise_xor` prennent en argument deux tableaux a et b d'entiers. Elles appliquent une fonction logique « bit à bit » aux éléments de a, b et renvoient le tableau des résultats terme à terme. Avec un troisième argument, on peut spécifier le nom d'un tableau de booléens destiné à recevoir le résultat.

```
>>> a = np.random.random_integers(128,255,size=8); a    # place dans a huit entiers dans [128,255]
array([128, 246, 220, 230, 252, 232, 216, 142])
>>> [np.binary_repr(x) for x in a]              # leur représentation sous forme de chaîne binaire
['10000000', '11110110', '11011100', '11100110', '11111100', '11101000', '11011000', '10001110']
>>> b = np.random.random_integers(128,255,size=8); b    # place dans b huit entiers dans [128,255]
array([187, 175, 237, 151, 152, 243, 187, 156])
>>> [np.binary_repr(x) for x in b]              # leur représentation sous forme de chaîne binaire
['10111011', '10101111', '11101101', '10010111', '10011000', '11110011', '10111011', '10011100']
>>> c = np.bitwise_and(a,b); c                  # place dans c le "et binaire" entre a et b
array([128, 166, 204, 134, 152, 224, 152, 140], dtype=int32)
>>> [np.binary_repr(x) for x in c]              # la représentation des résultats
['10000000', '10100110', '11001100', '10000110', '10011000', '11100000', '10011000', '10001100']
```

`invert(a)`, ou encore `np.bitwise_not(a)`, renvoie le tableau des négations binaires (donc les tableaux des $-x-1$, quand x parcourt a).

```
>>> a = np.random.random_integers(128,255,size=5); a
array([177, 216, 216, 135, 201])
>>> d = np.bitwise_not(a); d
array([-178, -217, -217, -136, -202], dtype=int32)
```

On pourra également considérer les fonctions `left_shift(x1, x2[, out])` et `right_shift(x1, x2[, out])`

Chapitre 6

Tests et comparaisons sur des tableaux

6.1 Comparaisons entre tableaux

`array_equal(a,b)` répond `True` si les tableaux a et b ont même format et même contenu, et `False` sinon.

```
>>> a = np.arange(0,12).reshape(3,4); a
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> b = np.copy(a)
>>> np.array_equal(a,b)  # mêmes tableaux
True
```

```
>>> b = b.reshape((4,3)); b # mêmes élts, forme ≠
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])
>>> np.array_equal(a,b)     # donc tableaux ≠
False
```

`greater(a,b)` renvoie le tableau des booléens $a_k > b_k$, où a_k et b_k se correspondent dans a et b.

```
>>> a = np.random.rand(7); a
array([ 0.45268328,  0.89699802,  0.47090844,  0.06040679,  0.92796843,  0.69654813,  0.64937467])
>>> b = np.random.rand(7); b
array([ 0.73789601,  0.70574489,  0.1792136 ,  0.48761371,  0.91307424,  0.54268789,  0.81540996])
>>> np.greater(a,b)    # pour chaque k, teste si a[k] > b[k]
array([False,  True,  True, False,  True,  True, False], dtype=bool)
```

On a des possibilités analogues avec :
`greater_equal(a,b)` (test $a_k \geq b_k$), `less(a,b)` (test $a_k < b_k$), `less_equal(a,b)` (test $a_k \leq b_k$),
`equal(a,b)` (test $a_k = b_k$) et `not_equal(a,b)` (test $a_k \neq b_k$) .

Les scalaires sont étendus en des tableaux constants pour permettre les comparaisons terme à terme :

```
>>> a = np.random.random_integers(100,1000,10); a
array([866, 305, 892, 849, 763, 433, 322, 442, 849, 239])
>>> np.less(a,500)              # teste si les a[k] vérifient a[k]<500
array([False,  True, False, False, False,  True,  True,  True, False,  True], dtype=bool)
>>> a = np.random.random_integers(10,size=12); a
array([ 6,  9,  4,  5,  5,  2,  3,  7,  9,  8,  4])
>>> np.equal(a,5)              # teste si les éléments sont égaux  à 5
array([False, False, False,  True,  True, False, False, False, False, False, False, dtype=bool)
```

On pourra consulter la fonction `allclose` qui teste si deux tableaux a et b sont identiques à une certaine précision près.
Autre fonction digne d'intérêt : `real_if_close`

6.2 Tris de tableau

L'expression `sort(a)` renvoie une *copie triée* du tableau a (l'original n'est donc pas modifié).

Mais attention : l'expression `a.sort()` trie le tableau a *sur place* (il y a donc une différence importante entre la fonction `sort` du module `numpy` et la méthode `sort` d'un objet `numpy` particulier).

```
>>> a = np.random.randint(100,1000,size=13); a
array([514, 553, 400, 234, 449, 887, 240, 252, 317, 231, 522, 578, 548])
>>> np.sort(a)             # on utilise la fonction sort du module numpy.
array([231, 234, 240, 252, 317, 400, 449, 514, 522, 548, 553, 578, 887])
>>> a                      # le tableau a original n'est donc pas été modifié
array([514, 553, 400, 234, 449, 887, 240, 252, 317, 231, 522, 578, 548])
>>> a.sort()               # on utilise la méthode sort de l'objet
>>> a                      # cette fois-ci le tri a été effecuté sur place
array([231, 234, 240, 252, 317, 400, 449, 514, 522, 548, 553, 578, 887])
```

L'expression `argsort(a)` renvoie le tableau des indices i tel que $a_{i[k]}$ est la k-ème plus petite valeur de a :

```
>>> a = np.random.randint(100,1000,size=13); a
array([357, 136, 784, 999, 965, 172, 159, 935, 330, 609, 128, 518, 986])
>>> i = np.argsort(a); i    # le tableau des indices des éléments de a dans l'ordre croissant
array([10,  1,  6,  5,  8,  0, 11,  9,  2,  7,  4, 12,  3], dtype=int32)
>>> a[i[0]]                 # le plus petit élément de a
128
>>> a[i[5]]                 # le cinquième plus petit élément de a
357
>>> a[i]                    # permet de retrouver une copie triée de a
array([128, 136, 159, 172, 330, 357, 518, 609, 784, 935, 965, 986, 999])
```

`sort_complex(a)` trie le tableau a de nombres complexes, suivant les parties réelles puis les parties imaginaires :

```
>>> a = np.random.randint(1,9,size=13); a
array([8, 4, 2, 3, 2, 6, 5, 7, 1, 3, 8, 4, 3])
>>> b = np.random.randint(1,9,size=13); b
array([5, 5, 1, 2, 5, 6, 4, 6, 4, 4, 1, 1, 1])
>>> c = a + b*1j; c
array([ 8.+5.j,  4.+5.j,  2.+1.j,  3.+2.j,  2.+5.j,  6.+6.j,  5.+4.j,
        7.+6.j,  1.+4.j,  3.+4.j,  8.+1.j,  4.+1.j,  3.+1.j])
>>> d = np.sort_complex(c); d
array([ 1.+4.j,  2.+1.j,  2.+5.j,  3.+1.j,  3.+2.j,  3.+4.j,  4.+1.j,
        4.+5.j,  5.+4.j,  6.+6.j,  7.+6.j,  8.+1.j,  8.+5.j])
```


jean-miche.ferrard@ac-paris.fr mathprepa.fr

6.3 Minimum et maximum

`amin(a[,axis,out,keepdims])` renvoie le(s) minimum(s) d'un tableau, éventuellement selon un « axe » (une direction)

```
>>> a = np.random.random_integers(100,1000, size=(5,6)); a
array([[690, 266, 954, 806, 903, 473],
       [536, 351, 939, 331, 552, 162],
       [745, 981, 423, 224, 927, 840],
       [149, 849, 884, 961, 158, 494],
       [354, 318, 604, 214, 124, 253]])
```

```
>>> np.amin(a)                  # le minimum de a aplati
124
>>> np.amin(a,0)                # suivant le premier indice
array([149, 266, 423, 214, 124, 162])
>>> np.amin(a,1)
array([266, 162, 224, 149, 124]) # suivant le deuxième indice
```

```
>>> np.amin(a,1,keepdims=True)
array([[266],
       [162],
       [224],
       [149],
       [124]])
```

`amax(a[,axis,out,keepdims])` est l'analogue de `amin`, mais pour la recherche de maximum.

`ptp(a[,axis,out])`, où `ptp` signifie « peak to peak », donne l'écart maximum entre deux éléments du tableau (soit sur la version aplatie du tableau, soit suivant une direction). Il s'agit en fait de calculer `amax(a)-amin(a)`.
On reprend ici le tableau a de l'exemple précédent.

```
>>> np.amax(a), np.amin(a), np.ptp(a)
(981, 124, 857)
>>> np.amax(a,0), np.amin(a,0)
(array([745, 981, 954, 961, 927, 840]), array([149, 266, 423, 214, 124, 162]))
>>> np.ptp(a,0)
array([596, 715, 531, 747, 803, 678])
```

`argmax(a[,axis])` et `argmin(a[,axis])` renvoie les indices où se trouvent le ou les éléments maximum(s) (respectivement minimum(s)) dans un tableau. On reprend à nouveau le tableau a de l'exemple précédent.

```
>>> np.argmin(a), np.argmax(a)         # la position du min (resp du max) dans le tableau à plat
(28, 13)
>>> np.argmin(a,0)                     # n° de ligne des éléments minimuns, colonne par colonne
array([3, 0, 2, 4, 4, 1], dtype=int32)
>>> np.argmax(a,0)                     # n° de ligne des éléments maximums, colonne par colonne
array([2, 2, 0, 3, 2, 2], dtype=int32)
>>> np.argmin(a,1)                     # n° de colonne des éléments minimuns, ligne par ligne
array([1, 5, 3, 0, 4], dtype=int32)
```


jean-miche.ferrard@ac-paris.fr mathprepa.fr

6.4 Recherches dans un tableau

`any(a)` teste si le tableau a contient au moins un élément « vrai » (au sens booléen ou scalaire non nul).

```
>>> a = np.random.random_integers(10,99,size=20); a
array([19, 15, 34, 54, 76, 16, 75, 30, 90, 21, 25, 49, 47, 57, 94, 52, 39, 34, 21, 15])
>>> np.any(a>90)
True
>>> a = np.random.random_integers(10,99,size=(5,6)); a
array([[64, 13, 69, 42, 44, 46],
       [72, 97, 82, 27, 27, 34],
       [75, 14, 36, 11, 16, 86],
       [83, 23, 51, 57, 41, 38],
       [21, 97, 92, 85, 12, 41]])
>>> np.any(a>90,axis=0)   # colonne par colonne, teste s'il y a au moins un élément > 90
array([False,  True,  True, False, False, False], dtype=bool)
>>> np.any(a>90,axis=1)   # ligne par ligne, teste s'il y a au moins un élément > 90
array([False,  True, False, False,  True], dtype=bool)
```

`all(a)` teste si tous les éléments du tableau a sont « vrais » (au sens booléen ou scalaire non nul).

```
>>> a = np.random.random_integers(10,99,size=20); a
array([68, 50, 46, 51, 86, 12, 42, 45, 74, 36, 64, 88, 98, 87, 63, 18, 81, 99, 90, 54])
>>> np.all(a>10)   # est-ce que tous les éléments de a sont > 10 ?
True
>>> a = np.random.random_integers(10,99,size=(3,10)); a
array([[72, 37, 46, 28, 64, 30, 40, 22, 58, 46],
       [77, 74, 98, 21, 28, 90, 53, 56, 66, 38],
       [49, 75, 54, 28, 55, 90, 74, 68, 81, 71]])
>>> np.all(a % 2 == 0, axis=0)  # colonne par colonne, est-ce que tous les éléments sont pairs?
array([False, False,  True, False, False,  True, False,  True, False, False], dtype=bool)
>>> np.all(a % 17, axis=1)   # ligne par ligne, tous les éléments sont-ils non divisibles par 17 ?
array([ True,  True, False], dtype=bool)
```

`argwhere(a)` renvoie le tableau des positions où les éléments du tableau a sont « vrais » (au sens booléen ou scalaire non nul). On reprend le tableau a des exemples précédents, et on cherche la position des éléments de a qui sont strictement supérieurs à 500. Le résultat a été transposé pour plus de lisibilité.

```
>>> a = np.random.random_integers(100,1000, size=(5,6)); a
array([[690, 266, 954, 806, 903, 473],
       [536, 351, 939, 331, 552, 162],
       [745, 981, 423, 224, 927, 840],
       [149, 849, 884, 961, 158, 494],
       [354, 318, 604, 214, 124, 253]])
>>> np.argwhere(a>500).T   # les élts > 500 de a sont en position (0,0), (0,2), etc. (4,2)
array([[0, 0, 0, 0, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 4],
       [0, 2, 3, 4, 0, 2, 4, 0, 1, 4, 5, 1, 2, 3, 2]], dtype=int32)
```

L'expression `nonzero(a)` renvoie le tableau des positions des éléments non nuls de a.

```
>>> a = np.random.random_integers(-1,1,15); a
array([-1, -1,  1, -1,  0,  0, -1, -1,  0,  0,  1,  0,  0,  1, -1])
>>> i = np.nonzero(a); i    # tableaux des positions des éléments non nuls de a
(array([ 0,  1,  2,  3,  6,  7, 10, 13, 14], dtype=int32),)
>>> a[i]                    # récupère le tableau des éléments non nuls
array([-1, -1,  1, -1, -1, -1,  1,  1, -1])
```

Dans le cas d'une matrice, la fonction `nonzero` renvoie un tuple (ici un couple) formé des indices des éléments non nuls. Le résultat est donc sous la forme $[\ell_1, \ell_2, \ldots, \ell_n], [c_1, c_2, \ldots, c_p]$ où les $[\ell_i, c_j]$ sont les positions des éléments non nuls.

```
>>> a = np.random.random_integers(-1,1,(3,5)); a
array([[-1,  0, -1,  0,  1],
       [-1,  1,  0,  1, -1],
       [ 0,  0,  0,  1,  0]])
>>> i = np.nonzero(a); i
(array([0, 0, 0, 1, 1, 1, 1, 2], dtype=int32), array([0, 2, 4, 0, 1, 3, 4, 3], dtype=int32))
>>> a[i]
array([-1, -1,  1, -1,  1,  1, -1,  1])
```

`count_nonzero` renvoie seulement le nombre d'éléments non nuls du tableau

```
>>> a = np.random.random_integers(-1,1,15); a
array([ 0,  1,  0,  1,  0,  0,  1,  0,  1,  1,  0,  0,  0, -1,  0])
>>> np.count_nonzero(a)                # compte le nombre d'éléments non nuls du tableau a
6
```

`where(condition)` renvoie le tableau des positions des éléments d'un vecteur qui possèdent une propriété particulière.

```
>>> a = np.random.random_integers(100,1000,15); a
array([318, 789, 611, 310, 254, 453, 864, 155, 400, 342, 377, 117, 796, 297, 949])
>>> i = np.where(a % 2); i     # tableau des positions des éléments impairs
(array([ 1,  2,  5,  7, 10, 11, 13, 14], dtype=int32),)
>>> a[i]                       # récupère le tableau de ces éléments impairs
array([789, 611, 453, 155, 377, 117, 297, 949])
```

La fonction `where` admet deux arguments facultatifs sous la forme d'un tableau a et d'un tableau b. À chaque fois que la *condition* qui est le premier argument de `where` est vraie, c'est l'élément correspondant de a qui est renvoyé, sinon c'est l'élément de b. Tout ça demande un peu d'habitude mais peut conduire à des constructions assez élégantes.

Dans l'exemple ci-dessous, on crée deux tableaux pseudo-aléatoires d'entiers de l'intervalle $[\![1, 9]\!]$. On voit d'ailleurs comment la ligne `rdi = ...` nous permet de définir un raccourci pour désigner une fonction au nom un peu long...

```
>>> rdi = np.random.random_integers  # un raccourci commode
>>> a = rdi(1,9, size=(5,6)); a
array([[2, 1, 7, 1, 8, 2],
       [2, 3, 4, 8, 9, 3],
       [5, 6, 7, 6, 6, 7],
       [8, 2, 8, 6, 7, 8],
       [2, 1, 7, 3, 1, 3]])
```

```
>>> b = rdi(1,9, size=(5,6))
>>> b
array([[6, 4, 1, 6, 8, 7],
       [1, 3, 3, 8, 3, 1],
       [9, 4, 8, 2, 8, 3],
       [6, 8, 4, 4, 9, 5],
       [4, 2, 8, 3, 6, 4]])
```

```
>>> np.where(a<5,a,b)
array([[2, 1, 1, 1, 8, 2],
       [2, 3, 4, 8, 3, 3],
       [9, 4, 8, 2, 8, 3],
       [6, 2, 4, 4, 9, 5],
       [2, 1, 8, 3, 1, 3]])
```

```
>>> np.where(a<5,0,b)
array([[0, 0, 1, 0, 8, 0],
       [0, 0, 0, 8, 3, 0],
       [9, 4, 8, 2, 8, 3],
       [6, 0, 4, 4, 9, 5],
       [0, 0, 8, 0, 0, 0]])
```

```
>>> np.where(a<5,0,1)
array([[0, 0, 1, 0, 1, 0],
       [0, 0, 0, 1, 1, 0],
       [1, 1, 1, 1, 1, 1],
       [1, 0, 1, 1, 1, 1],
       [0, 0, 1, 0, 0, 0]])
```

On pourra également consulter `extract(condition, arr)` et `searchsorted(a, v[, side, sorter])`

6.5 Tableaux d'un point de vue ensembliste

L'expression `unique(a)` renvoie les éléments de a triés, avec suppression des doublons :

```
>>> a = np.array([5,8,3,2,4,2,3,5,7])
>>> np.unique(a)
array([2,3,4,5,7,8])
>>> np.unique(a,True)       # ajoute la position des éléments uniques sélectionnés
(array([2,3,4,5,7,8]), array([3,2,4,0,8,1], dtype=int32))
>>> np.unique(a,True,True)  # renvoie en plus les indices pour reconstruire le tableau initial
(array([2,3,4,5,7,8]), array([3,2,4,0,8,1], dtype=int32), array([3,5,1,0,2,0,1,3,4], dtype=int32))
```

La fonction `unique` accepte deux arguments supplémentaires sous forme booléenne. Reprenons l'exemple précédent :

```
>>> a = np.array([5,8,3,2,4,2,3,5,7])    # le tableau initial
>>> b, c, d = np.unique(a,True,True)
>>> b                                    # le tableau trié des valeurs uniques de a
array([2, 3, 4, 5, 7, 8])
>>> c                                    # les positions dans a des éléments uniques
array([3, 2, 4, 0, 8, 1], dtype=int32)
>>> np.take(a,c)                         # on reconstitue donc ici le tableau b
array([2, 3, 4, 5, 7, 8])
>>> d                                    # tableau des positions dans le tableau b...
array([3, 5, 1, 0, 2, 0, 1, 3, 4], dtype=int32)
>>> np.take(b,d)                         # ... permettant de reconstruire le tableau a
array([5, 8, 3, 2, 4, 2, 3, 5, 7])
```

`in1d(a,b)` renvoie le vecteur des tests de l'appartenance des éléments du vecteur a dans le vecteur b.

```
>>> a = np.arange(0,25,3); a
array([ 0,  3,  6,  9, 12, 15, 18, 21, 24])
>>> b = np.arange(0,25,6); b
array([ 0,  6, 12, 18, 24])
>>> np.in1d(a,b)                      # les éléments de a sont-ils dans b ?
array([ True, False,  True, False,  True, False,  True, False,  True], dtype=bool)
>>> np.in1d(b,a)                      # les éléments de b sont-ils dans a ?
array([ True,  True,  True,  True,  True], dtype=bool)
```

`intersect1d(a,b)` renvoie l'intersection des deux vecteurs a et b.

`union1d(a,b)` renvoie l'union ensembliste des deux vecteurs a et b.

`setdiff1d(a,b)` renvoie les éléments de a qui ne sont pas dans b.

`setxor1d(a,b)` renvoie la différence symétrique des vecteurs a et b.

```
>>> a = np.arange(0,25,2); a          # multiples de 2 dans l'intervalle [0,25[
array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18, 20, 22, 24])
>>> b = np.arange(24,-1,-3); b        # multiples de 3 dans l'intervalle [0,25[
array([24, 21, 18, 15, 12,  9,  6,  3,  0])
>>> np.intersect1d(a,b)               # multiples à la fois de 2 et de 3
array([ 0,  6, 12, 18, 24])
>>> np.union1d(a,b)                   # multiples de 2 ou de 3
array([ 0,  2,  3,  4,  6,  8,  9, 10, 12, 14, 15, 16, 18, 20, 21, 22, 24])
>>> np.setdiff1d(a,b)                 # multiples de 2, mais pas de 3
array([ 2,  4,  8, 10, 14, 16, 20, 22])
>>> np.setxor1d(a,b)                  # multiples de 2 ou de 3, mais pas de 6
array([ 2,  3,  4,  8,  9, 10, 14, 15, 16, 20, 21, 22])
```


jean-miche.ferrard@ac-paris.fr mathprepa.fr

6.6 Sommes, produits, différences

L'expression `a.sum()` renvoie la somme des éléments de a.

On dispose aussi de `a.cumsum()` (sommes cumulées), `a.prod()` (produit) et `a.cumprod()` (produits cumulés).

On dispose comme souvent de deux syntaxes possibles : utiliser la méthode du tableau a (par exemple `a.sum()`) ou une fonction du module numpy appliquée au tableau a (par exemple `np.sum(a)`).

Voici quelques exemples sur un vecteur :

```
>>> a = np.array([5,8,3,2,6]); a
array([5, 8, 3, 2, 6])
>>> np.sum(a)
24
>>> a.sum()
24
```

```
>>> np.cumsum(a)
array([ 5, 13, 16, 18, 24], dtype=int32)
>>> a.prod()
1440
>>> a.cumprod()
array([ 5, 40, 120, 240, 1440], dtype=int32)
```

`diff(a[,n,axis])` calcule les différences d'éléments consécutifs, éventuellement le long d'un axe (0 pour l'indice de ligne, 1 pour l'indice de colonne), cette opération étant éventuellement répétée n fois (par défaut $n = 1$).

```
>>> a = np.arange(1,15)**2; a        # les entiers a_k = k², avec 1 ≤ k < 15
array([  1,   4,   9,  16,  25,  36,  49,  64,  81, 100, 121, 144, 169, 196])
>>> b = np.diff(a); b
array([ 3,  5,  7,  9, 11, 13, 15, 17, 19, 21, 23, 25, 27])
>>> c = np.diff(b); c
array([2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
>>> d = np.diff(c); d
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

Voici par exemple un test élégant pour vérifier que les valeurs d'un tableau sont dans l'ordre strictement croissant :

```
>>> a = np.array([1,2,4,7,5,9,11,13,17,19])
>>> np.all(np.diff(a)>0)          # vérifie si les différences entre élts consécutifs sont > 0
False
```

On pourra également consulter `ediff1d(ary[, to_end, to_begin])` et `gradient(f, *varargs)`

`cross(a,b)` renvoie le produit vectoriel des deux vecteurs a et b (mais il y a beaucoup d'options !)

```
>>> a = np.array([1,5,2]); b = np.array([3,4,1]); np.cross(a,b)
array([ -3,   5, -11])
>>> xa, ya, za = tuple(a); xb, yb, zb = tuple(b)       # on refait le calcul 'à la main'
>>> ya*zb - za*yb, za*xb -xa*zb, xa*yb - ya*xb
(-3, 5, -11)
```

L'expression `trapz(y[,x,h,axis])` illustre la « méthode des trapèzes ».

Elle calcule la somme des aires des trapèzes $\sum_i (x_{i+1} - x_i)\frac{y_i + y_{i+1}}{2}$, ou $h\sum_i \frac{y_i + y_{i+1}}{2}$ suivant l'option.

```
>>> x = np.linspace(0,np.pi,100)    # 100 valeurs régulièrement espacées, de 0 à π
>>> np.trapz(np.sin(x),x)           # intègre la fonction sin par les trapèzes sur ces abscisses
1.9998321638939924
>>> x = np.linspace(0,np.pi,1000)   # on recommence, mais avec 1000 points
>>> np.trapz(np.sin(x),x)
1.9999983517708528
```

6.7 Calculs statistiques, histogrammes

On dispose des fonctions `mean` (moyenne arithmétique), `var` (variance) et `std` (écart-type)

```
>>> a.mean()          # moyenne arithmétique
4.7999999999999998
>>> a.var()           # variance
4.5600000000000005
```

```
>>> a.std()          # écart-type
2.1354156504062622
>>> a.std()**2       # (écart-type)² => variance
4.5599999999999996
```

Toutes ces fonctions s'appliquent à une matrice. On peut alors préciser un axe de calcul (`axis=0` pour un calcul sur les lignes et `axis=1` pour un calcul sur les colonnes). Si on ne précise pas cet argument, le calcul est fait « à plat ».

```
>>> a = np.arange(20).reshape(4,5)
>>> a
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19]])
```

```
>>> a.sum()        # somme du tableau « aplati »
190
>>> a.sum(axis=0)  # somme des lignes
array([30, 34, 38, 42, 46])
>>> a.sum(axis=1)  # somme des colonnes
array([10, 35, 60, 85])
```

On dispose également des fonctions `cov` (matrice de covariance), et `corrcoef` (coefficients de corrélation).

`average(a[,axis,weights,returned])` renvoie la moyenne, éventuellement pondérée, éventuellement suivant un axe. Si le dernier argument (booléen et facultatif) *returned* est présent et vaut *true*, la somme des poids est ajoutée au résultat.

```
>>> rdi = np.random.random_integers
>>> a = rdi(10,size=(5,6))
>>> a
array([[ 5,  2,  1,  6,  2,  5],
       [ 8,  6,  9,  8,  2, 10],
       [ 6,  6, 10,  7,  6,  4],
       [ 7,  6, 10,  3,  9,  7],
       [ 1,  1, 10,  5,  5,  1]])
```

```
>>> np.average(a)          # moyenne du tableau
5.5999999999999996
>>> np.sum(a)/np.size(a)   # on vérifie !
5.5999999999999996
>>> np.average(a,axis=0)   # moyenne des lignes
array([ 5.4,  4.2,  8. ,  5.8,  4.8,  5.4])
>>> np.average(a,axis=1)   # moyenne des colonnes
array([ 3.5,  7.16666667,  6.5,  7.,  3.83333333])
```

```
>>> a = rdi(10,size=5); a
array([5, 6, 8, 1, 2])
>>> p = np.array([1,4,6,4,1]); p
array([1, 4, 6, 4, 1])
```

```
>>> np.average(a,weights=p)
5.1875
>>> np.average(a,weights=p,returned=True)
(5.1875, 16.0)
```

`median(a,[axis])` calcule la valeur médiane (ou les valeurs médianes suivant un axe d'une matrice)

```
>>> a
array([[ 5,  2,  1,  6,  2,  5],
       [ 8,  6,  9,  8,  2, 10],
       [ 6,  6, 10,  7,  6,  4],
       [ 7,  6, 10,  3,  9,  7],
       [ 1,  1, 10,  5,  5,  1]])
```

```
>>> np.median(a)           # médiane du tableau aplati
6.0
>>> np.median(a,axis=0)    # médianes suivant les lignes
array([ 6.,   6.,  10.,  6.,  5.,  5. ])
>>> np.median(a,axis=1)    # médianes suivant les colonnes
array([ 3.5,  8.,   6.,  7.,  3. ])
```

`percentile(a,n[,axis])` calcule le(s) n-ième(s) percentile(s), éventuellement selon un axe dans le cas d'une matrice.

```
>>> a = np.random.rand(1000); a
array([ 2.26465441e-01, ...
        ...  5.15893347e-01,
        ...  2.28513526e-01,
        ...  2.70221350e-01,
        ...  8.83760443e-01])
```

```
>>> np.percentile(a,10)         # valeur qui atteint les 10%
0.10722367362854898
>>> np.percentile(a,50)         # valeur qui atteint les 50%
0.52090862254043635
>>> np.median(a)                # 50-ième percentile = médiane
0.52090862254043635
```

On se borne à signaler les fonctions suivantes, et à renvoyer à la documentation « officielle » :

`histogram(a[,bins,range,normed,weights,...])` calcule un histogramme de valeurs de a (traité « à plat »)

`histogram2d(x, y[, bins, range, normed, weights])` `histogramdd(sample[, bins, range, normed, ...])`

`bincount(x[, weights, minlength])` `digitize(x, bins[, right])`

jean-miche.ferrard@ac-paris.fr mathprepa.fr

Chapitre 7

Calcul matriciel

7.1 Opérations linéaires

Les combinaisons linéaires de vecteurs ou de matrices s'effectuent de façon très naturelle :

```
>>> x = 100
>>> y = -1
>>> a = np.arange(6); a
array([0, 1, 2, 3, 4, 5])
```

```
>>> b = np.arange(1,13,2); b
array([ 1,  3,  5,  7,  9, 11])
>>> x*a+y*b
array([ -1,  97, 195, 293, 391, 489])
```

On pourra aussi considérer la (curieuse) fonction `einsum`

`trace` calcule la trace (et plus que ça) d'une matrice.
Il y a plusieurs arguments facultatifs, mais on ne considérera ici que le deuxième (qu'on peut nommer `offset=...`) qui permet de sélectionner une sur-diagonale ou une sous-diagonale.
La fonction `trace` peut aussi être évoquée comme une *méthode* du tableau, donc `np.trace(m)` et `m.trace()` se valent.

```
>>> m = np.arange(36).reshape(6,6)
>>> m
array([[ 0,  1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10, 11],
       [12, 13, 14, 15, 16, 17],
       [18, 19, 20, 21, 22, 23],
       [24, 25, 26, 27, 28, 29],
       [30, 31, 32, 33, 34, 35]])
```

```
>>> m.trace()    # la trace comme méthode de m
60
>>> np.trace(m)  # avec la fonction numpy
60
>>> np.trace(m,1)  # 1ère sur-diagonale
40
>>> np.trace(m,-1) # 1ère sous-diagonale
56
```

7.2 Produits matriciels

`dot(a,b[,out])` effectue le produit matriciel ab, et (facultativement) place le résultat dans *out*.

Pour deux vecteurs $a = [a_1, \ldots, a_n]$ et $b = [b_1, \ldots, b_n]$, on calcule ici le produit scalaire réel $\sum_{k=1}^{n} a_k b_k$.
Un vecteur v est considéré comme une ligne à gauche, et une colonne à droite.

```
>>> a = np.arange(1,5); a
array([1, 2, 3, 4])
>>> b = np.arange(11,15); b
array([11, 12, 13, 14])
>>> c = np.arange(21,26); c
array([21, 22, 23, 24, 25])
>>> m = np.arange(20).reshape(4,5); m
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19]])
```

```
>>> np.dot(a,b)   # produit scalaire a.b
130
>>> np.dot(a,m)   # vecteur-ligne par matrice
array([100, 110, 120, 130, 140])
>>> np.dot(m,c)   # matrice par vecteur-colonne
array([ 240,  815, 1390, 1965])
>>> np.dot(m,m.T) # calcule m.m^t
array([[  30,   80,  130,  180],
       [  80,  255,  430,  605],
       [ 130,  430,  730, 1030],
       [ 180,  605, 1030, 1455]])
```

`vdot(a,b)` effectue le produit scalaire *hermitien* de deux *vecteurs* (le tableau a est conjugué).

Si $a=[a_1,\dots,a_n]$ et $b=[b_1,\dots,b_n]$, on calcule donc ici la somme $\sum_{k=1}^{n}\overline{a_k}\,b_k$.

```
>>> a = np.array([1+4j,2-1j,-1j]); a
array([ 1.+4.j,  2.-1.j, -0.-1.j])
>>> b = np.array([3-1j,1+3j,1j]); b
array([ 3.-1.j,  1.+3.j,  0.+1.j])
>>> np.dot(a,b)     # ici ∑a[k]b[k]
(13+16j)
```

```
>>> np.vdot(a,b)  # ici ∑ conj(a[k]) b[k]
(-3-6j)
>>> np.vdot(b,a)  # ici ∑ conj(b[k]) a[k]
(-3+6j)
>>> np.vdot(a,a)  # carré norme hermitienne
(23+0j)
```

Si a,b sont deux vecteurs, `outer(a,b)` calcule la matrice de terme général $m_{i,j}=a_i\,b_j$.

```
>>> a = np.array([1,10,100])
>>> b = np.array([1,2,3,4,5])
>>> np.outer(a,b)
array([[  1,   2,   3,   4,   5],
       [ 10,  20,  30,  40,  50],
       [100, 200, 300, 400, 500]])
```

```
>>> np.outer(b,a)
array([[  1,  10, 100],
       [  2,  20, 200],
       [  3,  30, 300],
       [  4,  40, 400],
       [  5,  50, 500]])
```

Il y a une variante, où `outer` est une méthode de la classe `ufunc` : si a et b sont deux vecteurs, l'expression `f.outer(a,b)` calcule la matrice des $f(x,y)$ où f est une fonction universelle, et où (x,y) parcourt le produit cartésien $a\times b$.

```
>>> np.add.outer([1,2,3],[40,50])
array([[41, 51],
       [42, 52],
       [43, 53]])
>>> np.add.outer([40,50],[1,2,3])
array([[41, 42, 43],
       [51, 52, 53]])
```

```
>>> np.multiply.outer([1,2,3],[40,50])
array([[ 40,  50],
       [ 80, 100],
       [120, 150]])
>>> np.multiply.outer([40,50],[1,2,3])
array([[ 40,  80, 120],
       [ 50, 100, 150]])
```

`linalg.matrix_power(M,n)` calcule la puissance n-ième de la matrice M.

```
>>> m = np.random.rand(3,3); m
array([[ 0.87036519, 0.19566762, 0.44566847],
       [ 0.78488071, 0.03240106, 0.01279585],
       [ 0.56204791, 0.3552536 , 0.48591225]])
>>> np.linalg.matrix_power(m,2)
array([[ 1.16159833, 0.33496745, 0.60695382],
       [ 0.71575569, 0.15917134, 0.35642884],
       [ 1.0411246 , 0.29410724, 0.49114351]])
>>> np.dot(m,m)
array([[ 1.16159833, 0.33496745, 0.60695382],
       [ 0.71575569, 0.15917134, 0.35642884],
       [ 1.0411246 , 0.29410724, 0.49114351]])
```

```
>>> np.linalg.matrix_power(m,5)
array([[ 3.05133098, 0.85347532, 1.54321714],
       [ 1.81056991, 0.50624713, 0.91515234],
       [ 2.6543686 , 0.74279929, 1.34279836]])
>>> print(np.linalg.matrix_power(m,-3))
[[   2.75837736  -26.07305165   15.05139821]
 [ 195.03228433  -91.77058847 -161.33223914]
 [-112.56378676  102.52375103   59.85757815]]
>>> np.linalg.matrix_power(m,0)
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
```

`kron(a,b)` effectue le produit de Kronecker de deux tableaux a et b :

```
>>> a = np.array([[1,2],[3,4]]); a;
array([[1, 2],
       [3, 4]])
>>> b = np.array([[1,5],[10,15]]); b
array([[ 1,  5],
       [10, 15]])
```

```
>>> np.kron(a,b)
array([[ 1,  5,  2, 10],
       [10, 15, 20, 30],
       [ 3, 15,  4, 20],
       [30, 45, 40, 60]])
```

on reconnaît $\begin{pmatrix} b & 2b \\ 3b & 4b \end{pmatrix}$

```
>>> np.kron(b,a)
array([[ 1,  2,  5, 10],
       [ 3,  4, 15, 20],
       [10, 20, 15, 30],
       [30, 40, 45, 60]])
```

on reconnaît $\begin{pmatrix} a & 5a \\ 10a & 15a \end{pmatrix}$

On pourra également considérer les fonctions `inner` et `tensordot`.

7.3 Inversion de matrices, résolution de systèmes

`linalg.inv(a)` calcule l'inverse de la matrice carrée a :

```
>>> a = np.fromfunction(lambda i,j:1/(i+j+1),shape=(4,4)); a
array([[ 1.        , 0.5       , 0.33333333, 0.25      ],
       [ 0.5       , 0.33333333, 0.25      , 0.2       ],
       [ 0.33333333, 0.25      , 0.2       , 0.16666667],
       [ 0.25      , 0.2       , 0.16666667, 0.14285714]])
```

une matrice de Hilbert

```
>>> print(np.linalg.inv(a))
[[   16.  -120.   240.  -140.]
 [ -120.  1200. -2700.  1680.]
 [  240. -2700.  6480. -4200.]
 [ -140.  1680. -4200.  2800.]]
```

son inverse est à coefficients entiers

Pour les amateurs, on pourra consulter `linalg.tensorinv(a[,ind])`

`linalg.det(a)` calcule le déterminant de la matrice carrée a.

```
>>> a = np.vander([1,2,3,4,5]); a
array([[  1,   1,   1,   1,   1],
       [ 16,   8,   4,   2,   1],
       [ 81,  27,   9,   3,   1],
       [256,  64,  16,   4,   1],
       [625, 125,  25,   5,   1]])
```

une matrice de Vandermonde

```
>>> np.linalg.det(a)
287.99999999999494
>>> d = [j-i for j in range(5) for i in range(j)]; d
[1, 2, 1, 3, 2, 1, 4, 3, 2, 1]
>>> np.multiply.reduce(d)
288
```

on calcule son déterminant, et on le retrouve

`linalg.solve(a,b)` résout le système $ax = b$.

```
>>> a = np.array([[1,-2,3],[2,1,4],[5,-1,2]]); a
array([[ 1, -2,  3],
       [ 2,  1,  4],
       [ 5, -1,  2]])
```

```
>>> b = np.array([14,12,13]); b
array([14, 12, 13])
>>> np.linalg.solve(a,b)
array([ 1., -2.,  3.])
```

`linalg.lstsq(a,b[,rcond])` calcule la meilleure solution du système $ax = b$ au sens des moindes carrés.
On se reportera à la documentation officielle pour les détails de syntaxe.

```
>>> a = np.array([[1,-2],[2,1],[5,-1]]); a
array([[ 1, -2],
       [ 2,  1],
       [ 5, -1]])
>>> b = np.array([14,12,13]); b
array([14, 12, 13])
```

une matrice a de type 3×2
un vecteur b de taille 3

```
>>> x = np.linalg.lstsq(a,b)[0]; x
array([ 3.0516129 , -2.29032258])
>>> y = np.dot(a,x); y
array([  7.63225806, 3.81290323, 17.5483871 ])
>>> np.dot(y-b,a)
array([ -3.55271368e-15, 1.95399252e-14])
```

x la solution « bestfit » du système $ax = b$.
on vérifie que $y = ax$ est orthogonal à l'image de a

Ceux que ça intéresse pourront consulter la fonction `linalg.tensorsolve(a,b[,axes])`

7.4 Normes matricielles et vectorielles

`linalg.norm(x[,p])` calcule une norme d'un vecteur ou d'une matrice.
Le deuxième argument `p` détermine le type de norme (essentiellement 'fro', ou `np.inf`, ou 1)
Par défaut, il s'agit de la norme de Frobenius : $\|M\| = \sqrt{\sum |m_{i,j}|^2} = \sqrt{\mathrm{tr}(M\,{}^t M)}$

```
>>> m = np.arange(-10,10).reshape(5,4); m
array([[-10,  -9,  -8,  -7],
       [ -6,  -5,  -4,  -3],
       [ -2,  -1,   0,   1],
       [  2,   3,   4,   5],
       [  6,   7,   8,   9]])
```

une matrice de taille 4×5

```
>>> x = np.linalg.norm(m); x
25.88435821108957
>>> np.trace(np.dot(m,m.T))
670
>>> x**2
670.0
```

on calcule $x = \|m\|_2$ et on vérifie $x = \sqrt{\mathrm{tr}(M\,{}^t M)}$

Reprenons la matrice de l'exemple précédent :

```
>>> np.linalg.norm(m,1)
26
>>> np.sum(np.abs(m),axis=0)
array([26, 25, 24, 25])
```

on calcule $\|m\|_1$ et on vérifie que c'est le maximum de la somme des $|m_{i,j}|$ le long d'une colonne

```
>>> np.linalg.norm(m,np.inf)
34
>>> np.sum(np.abs(m),axis=1)
array([34, 18,  4, 14, 30])
```

on calcule $\|m\|_\infty$ et on vérifie que c'est le maximum de la somme des $|m_{i,j}|$ le long d'une ligne

`linalg.cond(x[,p])` calcule le conditionnement d'une matrice.

Le deuxième argument `p` détermine le type de norme (essentiellement 'fro', `np.inf`, 1)

Par défaut, il s'agit de la norme de Frobenius : $\|M\| = \sqrt{\sum |m_{i,j}|^2} = \sqrt{\mathrm{tr}(M\,{}^t M)}$

Le conditionnement d'une matrice m est un indicateur de la précision avec laquelle ou peut résoudre des systèmes linéaires $mx = b$: s'il est « élevé » (très supérieur à 1), une faible perturbation sur les coefficients de b ou de m peut entraîner une forte variation sur la solution x (tout ça mérite évidemment des explications plus sérieuses) :

```
>>> m
array([[10,  7,  8,  7],
       [ 7,  5,  6,  5],
       [ 8,  6, 10,  9],
       [ 7,  5,  9, 10]])
>>> b = np.array([32,23,33,31]); b
array([32, 23, 33, 31])
```

une matrice m dont on va voir qu'elle est très mal conditionnée, et un vecteur b.

```
>>> np.linalg.solve(m,b)
array([ 1.,  1.,  1.,  1.])
>>> np.linalg.cond(m)
2984.0927016757551
>>> b2 = np.array([32.1,22.9,33.1,30.9])
>>> np.linalg.solve(m,b2)
array([  9.2, -12.6,   4.5,  -1.1])
```

résout $mx = b$, et calcule le conditionnement de m
on légèrement perturbe b : solution fortement déviée

Soit m_2 la matrice obtenue en perturbant légèrement notre matrice m mal conditionnée.
On constate que la solution du système $m_2 x = b$ est très éloignée de celle de $mx = b$.

```
>>> m2 = m + np.random.randn(4,4)*1e-2; m2              # on perturbe légèrement la matrice m
array([[  9.99139341,   7.00645197,   8.00176469,  6.97831664],
       [  7.01449654,   5.0015422 ,   5.99109133,  4.9888675 ],
       [  8.01091316,   6.00359532,  10.0033713 ,  8.98368763],
       [  6.98199067,   5.01979424,   9.00508571,  0.01918998]])
>>> np.linalg.solve(m2,b)                               # solution fortement déviée
array([ 1.85278963, -0.39366723,  1.32302275,  0.81104713])
```

On consultera aussi la fonction `linalg.pinv` qui calcule la pseudo-inverse de Moore-Penrose de a.

7.5 Valeurs et vecteurs propres

`linalg.eigvals(a)` renvoie le vecteur des valeurs propres de a (éventuellement répétées).

Voici un exemple d'une matrice 5×5 avec une valeur propre triple et une valeur propre double :

```
>>> m
array([[ 1,  1, -1,  2, -1],
       [ 2,  0,  1, -4, -1],
       [ 0,  1,  1,  1,  1],
       [ 0,  1,  2,  0,  1],
       [ 0,  0, -3,  3, -1]])
>>> valp = np.linalg.eigvals(m); valp   # les valeurs propres, réelles aux erreurs d'arrondis près
array([ 1.00001556 +0.00000000e+00j,  0.99999222 +1.34784311e-05j, 0.99999222 -1.34784311e-05j,
       -1.00000000 +0.00000000e+00j, -1.00000000 +0.00000000e+00j])
>>> valp = np.round(valp.real,4); valp  # ici on a arrondi les résultats
array([ 1.,  1.,  1., -1., -1.])
>>> np.linalg.det(m-np.identity(5))     # on vérifie que m-Id n'est pas inversible
0.0
>>> np.linalg.det(m+np.identity(5))     # on vérifie que m+Id n'est pas inversible
0.0
```

Si le calcul des valeurs propres de la matrice précédente était entaché d'une erreur d'arrondi certaine, c'était à cause des multiplicités élevées. Voici une matrice 5×5 dont toutes les valeurs propres sont distinctes :

```
>>> m
array([[ 5,  4,  8, -4],
       [-2, -1,  6,  0],
       [-2, -1,  0,  1],
       [ 2,  4, 20, -5]])
>>> valp = np.linalg.eigvals(m); valp     # on trouve les valeurs propres distinctes -3, 2, 1, -1
array([-3.,  2.,  1., -1.])
>>> [np.linalg.det(m-x*np.identity(4)) for x in valp]      # vérifie que les det(m − λI) sont nuls
[2.1760371282653116e-13, -2.66453525910037e-14, -7.105427357600993e-14, 2.6645352591003795e-14]
```

`linalg.eig(a)` fait comme `linalg.eigvals` mais en plus elle renvoie les vecteurs propres (unitaires).

Le résultat est un couple (valeurs propres, matrice de passage).

```
>>> m
array([[ 5,  4,  8, -4],
       [-2, -1,  6,  0],
       [-2, -1,  0,  1],
       [ 2,  4, 20, -5]])
>>> valp, vectp = np.linalg.eig(m); valp
array([-3.,  2.,  1., -1.])
>>> print(vectp)
[[ -3.01511345e-01   1.06787647e-16   1.79605302e-01   2.71947991e-15]
 [ -3.01511345e-01   4.36435780e-01   3.59210604e-01  -7.07106781e-01]
 [  1.40906305e-16   2.18217890e-01   1.79605302e-01   6.79869978e-16]
 [ -9.04534034e-01   8.72871561e-01   8.98026510e-01  -7.07106781e-01]]
```

`linalg.eigvalsh(a)` renvoie le vecteur des valeurs propres d'une matrice symétrique (réelle ou hermitienne).

Aucune vérification n'est faite pour savoir si a est bien symétrique. En fait, le second argument, facultatif et nommé `UPLO=...` indique si la fonction doit prendre en compte la partie sous-diagonale (`UPLO='L'`) ou surdiagonale (`UPLO='U'`).

```
>>> np.linalg.eigvalsh([[1,3],[3,1]])
array([-2.,  4.])
```

```
>>> np.linalg.eigvalsh([[1,1,3],[1,2,2],[3,2,4]])
array([-0.87298335,  1.        ,  6.87298335])
```

`linalg.eigh(a)` fait comme `linalg.eigvalsh` mais en plus renvoie les vecteurs propres (unitaires).

Le résultat est un couple (valeurs propres, matrice de passage).

```
>>> m
array([[-1,  1,  7, -3],
       [ 1, -1, -3,  7],
       [ 7, -3, -1,  1],
       [-3,  7,  1, -1]])
>>> valp, vectp = np.linalg.eigh(m)
```

```
>>> valp     # les valeurs propres, toutes simples
array([-12.,  -4.,   4.,   8.])
>>> vectp    # une matrice de passage orthogonale
array([[-0.5, -0.5, -0.5,  0.5],
       [ 0.5, -0.5, -0.5, -0.5],
       [ 0.5,  0.5, -0.5,  0.5],
       [-0.5,  0.5, -0.5, -0.5]])
```

Même en cas de valeur propre multiple, la matrice de passage obtenue est orthogonale :

```
>>> m
array([[ 1,  3,  3, -3],
       [ 3,  1, -3,  3],
       [ 3, -3,  1,  3],
       [-3,  3,  3,  1]])
>>> valp, vectp = np.linalg.eigh(m)
>>> valp
array([-8.,  4.,  4.,  4.])
```

matrice symétrique avec une valeur propre triple

```
>>> vectp
array([[-0.5,  0.74166396,  0.44714044, 0.        ],
       [ 0.5, -0.05087231,  0.64348945, 0.57735027],
       [ 0.5,  0.65442479, -0.52637439, 0.21132487],
       [-0.5, -0.13811148, -0.33002538, 0.78867513]])
>>> np.dot(vectp,vectp.T).round(8)
array([[ 1.,  0.,  0., -0.],
       [ 0.,  1.,  0.,  0.],
       [ 0.,  0.,  1.,  0.],
       [-0.,  0.,  0.,  1.]])
```

la famille des vecteurs propres est bien orthonormée

jean-miche.ferrard@ac-paris.fr mathprepa.fr

7.6 Décompositions matricielles

`linalg.cholesky(a)` calcule la décomposition $A = L\,{}^tL$ d'une matrice symétrique définie positive.

Ici la matrice L est triangulaire inférieure (L pour *Low*), à coefficients diagonaux strictement positifs.

```
>>> M
array([[ 4, -1, -1,  0],
       [-1,  4,  0, -1],
       [-1,  0,  4, -1],
       [ 0, -1, -1,  4]])
>>> np.linalg.eigvals(M)
array([ 2.,  4.,  6.,  4.])
```

une matrice symétrique définie positive

```
>>> L = np.linalg.cholesky(m); L
array([[  2. , -0.        , -0.        ,  0.        ],
       [ -0.5,  1.93649167,  0.        , -0.        ],
       [ -0.5, -0.12909944,  1.93218357, -0.        ],
       [  0. , -0.51639778, -0.55205245,  1.8516402 ]])
>>> np.allclose(np.dot(L,L.T),M)
True
```

vérifie que $L\,{}^tL = M$ aux erreurs d'arrondi près

`linalg.qr(a[,mode])` calcule la décomposition $M = QR$ d'une matrice inversible.

Ici Q est orthogonale et R est triangulaire supérieure à coefficients positifs.

Le résulat est obtenu sous la forme du couple (Q, R).

```
>>> M
array([[ 1,  5,  5, 11],
       [ 1,  5,  3,  3],
       [ 1, -1,  3, -1],
       [ 1, -1,  1,  3]])
>>> Q, R = np.linalg.qr(M)
```

```
>>> Q
array([[ 0.5,  0.5,  0.5,  0.5],
       [ 0.5,  0.5, -0.5, -0.5],
       [ 0.5, -0.5,  0.5, -0.5],
       [ 0.5, -0.5, -0.5,  0.5]])
```

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

Signalons enfin l'existence de la fonction `linalg.svd` (*singular value decomposition*)

Chapitre 8

Calcul polynomial

8.1 La classe `poly1d`

La classe `poly1d` offre des outils communs pour travailler sur des polynômes $P = a_n X^n + a_{n-1} X^{n-1} + \cdots + a_1 X + a_0$ définis et manipulés à travers le vecteur $[a_n, a_{n-1}, \ldots, a_0]$ de leurs coefficients (dans l'ordre des degrés décroissants).
Dans l'exemple suivant, on forme le polynôme $P = X^5 + 4X^4 + 2X^2 + 7X + 6$.

```
>>> p = np.poly1d([1,4,0,2,7,6])
>>> p
poly1d([1, 4, 0, 2, 7, 6])
>>> type(p)
<class 'numpy.lib.polynomial.poly1d'>
>>> print(p)   # affichage rudimentaire de p
   5     4     2
1 x + 4 x + 2 x + 7 x + 6
```

```
>>> p.order     # le degré de p
5
>>> p.c         # les coefficients de p
array([1, 4, 0, 2, 7, 6])
>>> p[1]        # coeff du terme de degré 1
7
>>> p(10)       # la valeur de p en 10
140276
```

La méthode `r` d'un objet `poly1d` renvoie les racines, réelles ou complexes, de ce polynôme.

```
>>> p = np.poly1d([1,0,3,1,-1])          # le polynôme p = X⁴ + 3X² + X − 1
>>> p.r                                  # deux racines réelles, deux complexes conjuguées
array([ 0.13632751+1.83095723j,  0.13632751-1.83095723j,  -0.69778453+0.j,  0.42512951+0.j ])
>>> p = np.poly1d([1, -4, -1, 16, -12])  # le polynôme p = X⁴ − 4X³ − X² + 16X − 12
>>> p.r                                  # quatre racines réelles distinctes
array([-2.,  3.,  2.,  1.])
```

Attention : en cas de racines multiples, le résultat obtenu peut souffrir d'une certaine imprécision, due à la méthode de calcul (`numpy` forme la matrice compagnon avant d'appliquer un algorithme de recherche des valeurs propres).

```
>>> p = np.poly1d([1, -4,  6,  -4, 1])   # le polynôme p = X⁴ − 4X³ + 6X² − 4X + 1
>>> p.r                                  # racine quadruple 1, grosse imprécision
array([ 1.00021716+0.j, 0.99999997+0.00021713j, 0.99999997-0.00021713j, 0.99978290+0.j ])
```

Le constructeur `poly1d` accepte un deuxième argument `r=...`, de type booléen (avec la valeur `False` par défaut).
S'il a la valeur `True`, on forme un polynôme par la liste de ses racines.
Ce même constructeur admet aussi un argument facultatif `variable=...` qui permet de spécifier une variable à afficher avec `print` (par défaut c'est `x`).

```
>>> p = np.poly1d([1,2,3,4],r=True)
>>> print(p)
   4      3      2
1 x - 10 x + 35 x - 50 x + 24
>>> p.r
array([ 4.,  3.,  2.,  1.])
```

```
>>> p = np.poly1d([1,2,3,4]); print(p)
   3     2
1 x + 2 x + 3 x + 4
>>> p = np.poly1d([1,2,3,4],variable='y'); print(p)
   3     2
1 y + 2 y + 3 y + 4
```

Remarque : on peut simplifier la saisie d'un polynôme en écrivant, par exemple, `P = np.poly1d`.

▷ Opérations usuelles sur les polynômes

Pour les polynômes de la classe `poly1d`, les opérations arithmétiques usuelles (addition, produit, quotient, reste) sont directement disponibles sous leur forme infixée habituelle.

Attention quand même : les opérateurs `//` et `%` ne sont pas disponibles pour la classe `poly1d`.
Il faut utiliser l'opérateur `/` qui renvoie le couple (quotient,reste) dans la division euclidienne.

```
>>> a = np.poly1d([1, 4, 2, 3, 5, 6])
>>> b = np.poly1d([1, 1, 2, 3])
>>> a + b
poly1d([1, 4, 3, 4, 7, 9])
>>> 2*a-3*b      # combinaison linéaire
poly1d([2, 8, 1, 3, 4, 3])
>>> a*b          # produit des deux polynômes
poly1d([ 1,  5,  8, 16, 24, 23, 25, 27, 18])
```

```
>>> b**2           # élévation au carré
poly1d([ 1,  2,  5, 10, 10, 12,  9])
>>> q, r = a / b   # division euclidienne
>>> q
poly1d([ 1.,  3., -3.])
>>> r
poly1d([ -3.,   2.,  15.])
>>> q*b+r          # on retrouve le polynôme a
poly1d([ 1.,  4.,  2.,  3.,  5.,  6.])
```

Un polynôme de la classe `poly1d` peut être dérivé ou intégré au moyen de ses méthodes `deriv` et `integ`.

Ces deux méthodes acceptent un argument (facultatif) précisant l'ordre de dérivation ou de primitivation (par défaut 1).

La méthode `integ` accepte un deuxième argument `k=...`, facultatif, précisant la valeur en 0 de la primitive demandée.

```
>>> a = np.poly1d([1,5,0,2,3,1])
>>> a.deriv()          # dérivée première
poly1d([ 5, 20,  0,  4,  3])
>>> b = a.deriv(2); b   # dérivée seconde
poly1d([20, 60,  0,  4])
```

```
>>> b.integ()      # primitive = 0 à l'origine
poly1d([  5.,  20.,   0.,   4.,   0.])
>>> b.integ(k=1)   # primitive = 1 à l'origine
poly1d([  5.,  20.,   0.,   4.,   1.])
>>> b.integ(2,k=-1)
poly1d([ 1.,  5.,  0.,  2., -1., -1.])
```

Il nous semble préférable d'utiliser les méthodes des objets de la classe `poly1d`, plutôt que d'appliquer des fonctions de `numpy` sur des polynômes ou des listes de coefficients.

Par exemple, on préfère écrire `a.deriv()` pour dériver un polynôme a de la classe `poly1d` plutôt que d'écrire `polyder(a)`.

Pour plus d'information, voir : http://docs.scipy.org/doc/numpy/reference/routines.polynomials.poly1d.html

▷ Interpolation polynomiale

L'expression `interp(x, xp, yp[, left, right])` renvoie les images y des abscisses x, obtenues par interpolation sur la base des points xp, yp.

```
>>> xp = [0, 1, 2, 3, 4]            # abscisses des données connues
>>> yp = [4, 1, 3, 2, 5]            # ordonnées yp des connées connues
>>> x = np.linspace(0,4,num=17); x  # 17 abscisses à interpoler (16 intervalles de longueur 0.25)
array([ 0.  ,  0.25,  0.5 ,  0.75,  1.  ,  1.25,  1.5 ,  1.75,  2.  ,
        2.25,  2.5 ,  2.75,  3.  ,  3.25,  3.5 ,  3.75,  4.  ])
>>> y = np.interp(x,xp,yp); y       # les 17 valeurs interpolées correspondantes
array([ 4.  ,  3.25,  2.5 ,  1.75,  1.  ,  1.5 ,  2.  ,  2.5 ,  3.  ,
        2.75,  2.5 ,  2.25,  2.  ,  2.75,  3.5 ,  4.25,  5.  ])
```

▷ Approximation au sens des moindres carrés

`polyfit(x,y,d)` renvoie le polynôme de meilleure approximation, au sens des moindres carrés, des points (x, y).

```
>>> x = [0,1,2,3,4]; y = [4,1,3,2,5]  # abscisses et ordonnées du nuage de points
>>> np.polyfit(x,y,1)                 # [a,b] => droite y = ax+b des moindres carrés
array([ 0.3,  2.4])
>>> np.polyfit(x,y,2)                 # [a,b,c] => approximation y = ax²+bx+c
array([ 0.64285714, -2.27142857,  3.68571429])
```

Pour cette fonction `polyfit`, il y a plusieurs options possibles, et on se reportera à la documentation officielle.

8.2 Le package numpy.polynomial

Attention, ce paragraphe contient des détails un peu techniques, qui peuvent être largement ignorés en première lecture.
Il suffira de se plier aux deux suggestions encadrées ci-dessous pour travailler avec les exemples.

À l'intérieur de **numpy**, on trouve un *package* nommé polynomial.
Il offre des outils communs pour travailler sur des polynômes définis et manipulés à travers la liste de leurs coefficients.
Mais ces coefficients sont des *coordonnées* dans une *base*, et il faut décider de la base utilisée.

La base la plus courante, dite *canonique*, est formée des monômes $1, X, X^2, \dots, X^n, \dots$.
Le polynôme $A(X) = 1 + 3X + 5X^2 + 2X^4$, par exemple, sera identifié par le vecteur $[1, 3, 5, 0, 2]$ de ses coefficients.
La base canonique suffit largement à traiter tout ce qui relève du « calcul polynomial » au sens général.

Mais les polynômes ça sert également à réaliser des approximations.
Et selon l'intervalle où on doit travailler, certaines bases sont beaucoup plus intéressantes que la base canonique.
Ces bases particulières consistent en une suite $(P_n(X))_{n\geq 0}$ à degrés échelonnés (c'est-à-dire $\deg(P_k) = k$ pour tout k).
Dans ce cas, un polynôme $A(X) = a_0P_0(X) + a_1P_1(X) + \cdots + a_nP_n(X)$ est décrit par le vecteur $[a_0, a_1, \dots, a_n]$.

Pour prévoir tous les cas de figure, le package **numpy.polynomial** contient plusieurs modules :

- le module **numpy.polynomial.polynomial** pour travailler avec la base canonique
- le module **numpy.polynomial.chebyshev** pour travailler avec la base des polynômes de Chebyshev.
- le module **numpy.polynomial.legendre** pour travailler avec la base des polynômes de Legendre.
- le module **numpy.polynomial.laguerre** pour travailler avec la base des polynômes de Laguerre.
- le module **numpy.polynomial.hermite** pour travailler avec la base des polynômes de Hermite.
  (on trouve également le module **numpy.polynomial.hermite_e**)

Le *package* **numpy.polynomial** contient aussi des *classes* pour former des « objets polynômes dans une base donnée ».
Le nom d'une classe est celui du module correspondant, à ceci près qu'il commence par une majuscule.
Chaque objet est créé (on dit *instancié*) par un constructeur qui porte lui aussi ce nom à majuscule.
On trouvera par exemple la classe (le constructeur) Polynomial pour fabriquer (*instancier*) des polynômes représentés par le vecteur de leurs coefficients dans la base canonique.
On dispose également des classes (constructeurs) Chebyshev, Legendre, Laguerre, Hermite, HermiteE.

Quelle que soit la base utilisée, les polynômes sont toujours définis par leur coefficients dans le sens des degrés croissants (c'est là une différence essentielle avec les polynômes de la classe **poly1d**).

Pour débuter, il est prudent de se limiter aux polynômes exprimés dans la base canonique $1, X, X^2, \dots, X^n, \dots$.
On propose le principe suivant, qui permettra d'éviter d'utiliser des noms trop longs :

On importe la *classe* **Polynomial** du package **numpy.polynomial** et on lui donne le nom **P**

```
from numpy.polynomial import Polynomial as P
```

▷ La classe poly1d ou le package polynomial ?

Laissons parler la documentation officielle : *prior to NumPy 1.4, numpy.poly1d was the class of choice and it is still available in order to maintain backward compatibility. However, the newer Polynomial package is more complete than numpy.poly1d and its convenience classes are better behaved in the numpy environment. Therefore Polynomial is recommended for new coding.*

8.3 La classe « Polynomial »

Le titre de cette section devrait être, on le sait : la classe « numpy.polynomial.polynomial.Polynomial »....
La classe la plus naturelle, « Polynomial », permet de former des combinaisons linéaires $A(X) = a_0 + a_1X + \cdots + a_nX^n$.
Formons par exemple le polynôme $a = -1 + 2X + X^3 - 4X^5 + 2X^6$.

```
>>> from numpy.polynomial import Polynomial as P     # IMPORTANT, ON SUPPOSERA QUE C'EST FAIT
>>> a = P([-1, 2, 0, 1, -4, 2]); a
Polynomial([-1.,  2.,  0.,  1., -4.,  2.], [-1.,  1.], [-1.,  1.])
```

▷ Quelques remarques à faire très vite

Il y a trois parties dans un polynôme, la première étant le vecteur numpy des coefficients (selon les degrés croissants).
Dans notre exemple, les coefficients, au départ entiers, ont automatiquement été convertis au format float.

La seconde partie est le « domaine » du polynôme, et la troisième partie est sa « fenêtre ».
Ce sont toutes deux des intervalles, par défaut $[-1, 1]$. Le domaine et la fenêtre d'un polynôme n'interviennent que lorsqu'on en vient aux problèmes d'approximation, et pas dans les opérations algébriques.
Le « domaine » est l'intervalle où pourront se trouver les abscisses des données devant faire l'objet de l'approximation, et la « fenêtre » est l'intervalle sur lequel on pourra tracer le polynôme résultant de cette approximation.

On peut extraire chacune des trois parties par la méthode adéquate (ici notre polynôme a).

```
>>> a.coef
array([-1.,  2.,  0.,  1., -4.,  2.])
```

```
>>> a.domain
array([-1.,  1.]
```

```
>>> a.window
array([-1.,  1.])
```

La méthode print permet de n'afficher que le vecteur des coefficients. Le préfixe poly est là pour nous indiquer qu'il s'agit bien d'un polynôme. Quant au type de notre polynôme a, on voit qu'il est assez étonnant.

```
>>> print(a)
poly([-1.  2.  0.  1. -4.  2.])
```

```
>>> type(a)
<class 'numpy.polynomial.polynomial.Polynomial'>
```

On peut définir un polynôme à coefficients complexes.
Ses attributs « domain » et « window » sont encore des intervalles de $\mathbb{R}$ (et toujours $[-1, 1]$ par défaut) :

```
>>> P([1,1+1j,2j,0,3-1j])       # le polynôme 1 + (1 + i)X + 2iX² + (3 − i)X⁴
Polynomial([ 1.+0.j,  1.+1.j,  0.+2.j,  0.+0.j,  3.-1.j], [-1.+0.j,  1.+0.j], [-1.+0.j,  1.+0.j])
```

▷ Opérations arithmétiques usuelles

Pour les polynômes formés à l'aide du constructeur Polynomial (abrégé en P), les principaux opérateurs arithmétiques (addition, produit, quotient, reste) sont directement disponibles sous leur forme infixée habituelle.
Pour voir de quoi il s'agit, rappelons la définition de notre polynôme a et formons un deuxième polynôme b :

```
>>> a = P([-1, 2, 0, 1, -4, 2]); a
Polynomial([-1.,  2.,  0.,  1., -4.,  2.], [-1.,  1.], [-1.,  1.])
>>> b = P([1, -1, 2, 1]); b
Polynomial([ 1., -1.,  2.,  1.], [-1.,  1.], [-1.,  1.])
```

Dès lors, on peut additionner les polynômes, les multiplier, etc. de façon très naturelle.
Dans les exemples qui suivent, on utilisera print pour éviter d'être distrait par les intervalles de domaine et de fenêtre.

```
>>> print(a)        # le polynôme a
poly([-1.  2.  0.  1. -4.  2.])
>>> print(b)        # le polynôme b
poly([ 1. -1.  2.  1.])
>>> print(a+b)      # le polynôme a + b
poly([ 0.  1.  2.  2. -4.  2.])
>>> print(2*a+3*b)  # le polynôme 2a + 3b
poly([ 1.  1.  6.  5. -8.  4.])
```

```
>>> print(a*b)      # le polynôme ab
poly([-1.  3. -4.  4. -3.  8. -9.  0.  2.])
>>> print(a // b)   # quotient dans div euclidienne
poly([ 19.  -8.   2.])
>>> print(a % b)    # reste dans division euclidienne
poly([-20.  29. -48.])
>>> print(P([1,1])**4) # le polynôme (X + 1)⁴
poly([ 1.  4.  6.  4.  1.])
```


jean-miche.ferrard@ac-paris.fr mathprepa.fr

Python est très flexible quand il s'agit d'opérations élémentaires entre polynômes. Pourvu que l'un des argument ait été formé avec le constructeur `Polynomial`, l'autre peut être une liste (ou un tuple, ou un tableau `numpy`) de coefficients.

```
>>> a = P([-1, 2, 0, 1, -4, 2]); print(a)
poly([-1.  2.  0.  1. -4.  2.])
>>> print(a + [1,1,1])
poly([ 0.  3.  1.  1. -4.  2.])
>>> print(a - (1,1,1,1,1,2))
poly([-2.  1. -1.  0. -5.])
```

```
>>> print(a // [1,1,1])
poly([ 1.  5. -6.  2.])
>>> print(a % (1,1,1))
poly([-2. -4.])
>>> print(a + np.arange(4))
poly([-1.  3.  2.  4. -4.  2.])
```

▷ Évaluations de polynômes

Les polynômes sont considérés comme des fonctions d'une variable, et il est extrêmement simple de calculer la valeur de A en un point, et même tout un échantillon de valeurs (on reprend l'exemple de $A = -1 + 2X + X^3 - 4X^5 + 2X^6$).

```
>>> a(1)        # on voit que a(1) est nul
0.0
>>> a([1,2,3])  # valeurs de a en 1, 2, et 3
array([   0.,   11.,  194.])
```

```
>>> m= [[1,2,3],[4,5,6]]
>>> a(m)  # un tableau de valeurs de a
array([[    0.,    11.,   194.],
       [ 1095.,  3884.,  10595.]])
```

Il est même possible de composer deux polynômes entre eux.

Ainsi, si $\begin{cases} A(X) = -1 + 2X + X^3 - 4X^5 + 2X^6 \\ B(X) = 1 - X + 2X^2 + X^3 \end{cases}$, on définit : $\begin{cases} A(B) = -1 + 2B + B^3 - 4B^5 + 2B^6 \\ B(A) = 1 - A + 2A^2 + B^3 \end{cases}$

```
>>> a = P([-1, 2, 0, 1, -4, 2])   # le polynôme A
>>> b = P([1, -1, 2, 1])          # le polynôme B
>>> print(a(b))                   # le polynôme A(B)
poly([  0.   1.  -3.  -2.  34. -99. 119. -73. -46. 113. -16. -12.  86.  70.  20.   2.])
>>> print(b(a))                   # le polynôme B(A)
poly([  3.  -4.  -4.   6.   4.  24. -57.  38. -68. 137. -112.  78. -112.  108. -48.   8.])
```

▷ Incompatibilités de domaine ou de fenêtre

On peut définir un polynôme ayant des attributs de domaine ou de fenêtre différents de [-1,1] :

```
>>> P([1,3,0,1],domain=[0,10],window=[-2,2]);
Polynomial([ 1.,  3.,  0.,  1.], [  0.,  10.], [-2.,  2.])
```

Mais deux polynômes qui n'ont pas le même attribut de *domaine* ou de *fenêtre* ne peuvent pas être les arguments d'une opération arithmétique (somme, produit, division euclidienne, exponentiation).

```
>>> a  = P([1,2,3],domain=[0,1]); b = P([2,5,1,4],domain=[-3,3]); a+b
TypeError: Domains differ
```

▷ Dérivées et primitives de polynômes

Reprenons notre polynôme $A(X) = -1 + 2X + X^3 - 4X^5 + 2X^6$

Il peut être dérivé (une ou plusieurs fois) par sa méthode `deriv` :

```
>>> a = P([-1, 2, 0, 1, -4, 2]); a
Polynomial([-1.,2.,0.,1.,-4.,2.],[-1.,1.],[-1.,1.])
>>> a.deriv()
Polynomial([2.,0.,3.,-16.,10.],[-1.,1.],[-1.,1.])
>>> a.deriv(0)
Polynomial([-1.,2.,0.,1.,-4.,2.],[-1.,1.],[-1.,1.])
```

```
>>> a.deriv(4)
Polynomial([ -96.,240.],[-1.,1.],[-1.,1.])
>>> a.deriv(5)
Polynomial([240.],[-1.,1.],[-1.,1.])
>>> a.deriv(6)
Polynomial([-0.],[-1.,1.],[-1.,1.])
```

Un polynôme peut être primitivé par sa méthode `integ`. Celle-ci accepte deux paramètres facultatifs : d'une part `k=...` indiquant le nombre de primitivations à effectuer (une, par défaut), et d'autre part `lwbd=...` indiquant à partir de quelle borne on intègre (c'est-à-dire où s'annule la primitive, par défaut c'est en 0)

jean-miche.ferrard@ac-paris.fr mathprepa.fr

```
>>> a = P([-1, 2, 0, 1, -4, 3]); a
Polynomial([-1.,  2.,  0.,  1., -4.,  2.], [-1.,  1.], [-1.,  1.])
>>> a.integ()                          # la primitive de a qui s'annule en 0
Polynomial([ 0., -1.,  1.,  0.,  0.25, -0.8,  0.5 ], [-1.,  1.], [-1.,  1.])
>>> print(a.integ(3))                  # on a primitivé trois fois, toujours à partir de 0
poly([ 0.   0.   0. -0.16666667  0.08333333  0.  0.00833333  0.01904762  0.00892857])
>>> b = a.integ(lbnd=1); b             # la primitive b de a qui s'annule en 1
Polynomial([ 0.05, -1.  ,  1.  ,  0.  ,  0.25, -0.8 ,  0.5 ], [-1.,  1.], [-1.,  1.])
>>> b(1)                               # on vérifie effectivement que b(1) est nul
0.0
```

▷ Racines de polynômes

Les racines (réelles ou complexes) d'un polynôme sont obtenues par la méthode `roots`.

```
>>> a = P([6,7,-3,-3,1]); a
Polynomial([ 6.,  7., -3., -3.,  1.], [-1.,  1.], [-1.,  1.])
>>> r = a.roots(); r
array([-1.00000002, -0.99999998,  2.,  3.])
```

deux racines simples, une racine double

```
>>> np.round(a(r),10)
array([ 0.,  0., -0., -0.])
>>> np.round_(a.deriv()(r),6)
array([ -0.,   0.,  -9.,  16.])
```

calcule $a(\lambda)$ et $a'(\lambda)$ sur les racines

Voici un exemple où le polynôme, à coefficients réels, a des racines non réelles :

```
>>> a = P([1,1,1,1,1,1]); a      # le polynôme A = 1 + X + X² + X³ + X⁴ + X⁵
Polynomial([ 1.,  1.,  1.,  1.,  1.,  1.], [-1.,  1.], [-1.,  1.])
>>> a.roots()                    # les racines sixièmes de l'unité sauf 1
array([-1.0+0.j, -0.5-0.8660254j, -0.5+0.8660254j, 0.5-0.8660254j, 0.5+0.8660254j])
```

Réciproquement, on peut former un polynôme (unitaire) à partir de ses racines :

```
>>> from numpy.polynomial import Polynomial as P     # comme d'habitude
>>> a = P.fromroots([1,2,3,4]); a    # polynôme de degré 4 dont les racines sont 1,2,3,4
Polynomial([ 24., -50.,  35., -10.,   1.], [-1.,  1.], [-1.,  1.])
>>> a.roots()                        # on vérifie
array([ 1.,  2.,  3.,  4.])
```

▷ Autres méthodes de la classe « Polynomial »

`basis(n)` renvoie le polynôme d'indice n dans la base utilisée, donc ici X^n.

```
>>> P.basis(6)                     # le polynôme X⁶
Polynomial([ 0.,  0.,  0.,  0.,  0.,  0.,  1.], [-1.,  1.], [-1.,  1.])
```

`degree()` donne le degré d'un polynôme, et `cutdeg(n)` ne garde que les termes de degré $\leq n$.

```
>>> a = P([1,5,6,8,7]); a.degree()     # le polynôme A = 1 + 5X + 6X² + 8X³ + 7X⁴
4
>>> b = a.cutdeg(2); b                 # ne garde que les termes de degré ≤ 2
Polynomial([ 1.,  5.,  6.], [-1.,  1.], [-1.,  1.])
```

`trim(tol=0)` supprime les termes dominants s'ils sont inférieurs à une certaine tolérance (nulle par défaut).

```
>>> a = P((1,6,1e-5,-1.e-7,0)); print(a)
poly([  1.00000000e+00   6.00000000e+00   1.00000000e-05  -1.00000000e-07   0.00000000e+00])
>>> print(a.trim())
poly([  1.00000000e+00   6.00000000e+00   1.00000000e-05  -1.00000000e-07])
>>> print(a.trim(1e-6))
poly([  1.00000000e+00   6.00000000e+00   1.00000000e-05])
```


jean-miche.ferrard@ac-paris.fr mathprepa.fr

8.4 Les classes « Chebyshev », etc

On peut importer les constructeurs qui correspondent aux familles de polynômes classiques, comme on l'a fait avec le constructeur `Polynomial`.

Voici quelques suggestions de raccourcis pour l'importation.

`from numpy.polynomial import Chebyshev as T` : polynômes de Chebyshev

l'utilisation de la lettre `T` est liée à une certaine habitude de notation (et de prononciation).

`from numpy.polynomial import Legendre as Le` : polynômes de Legendre

`from numpy.polynomial import Laguerre as La` : polynômes de Laguerre

`from numpy.polynomial import Hermite as H` : polynômes de Hermite

`from numpy.polynomial import HermiteE as He` : polynômes de Hermite

Ce qui est très rassurant, c'est que ces différentes classes possèdent essentiellement les mêmes méthodes, avec les mêmes noms (la seule différence étant le préfixe utilisé : `P`, `C`, etc.). Par exemple la méthode `T.basis(n)` renvoie le n-ième polynôme dans la base de Chebyshev.

Ce qui également très sympathique, dans les constructeurs `P`, `C`, etc, c'est qu'ils fabriquement des polynômes qui peuvent, entre objets d'une même famille, être manipulés de façon très naturelle (opérateurs infixes d'addition, de produit, de division euclidienne, d'exponentiation).

C'est la raison pour laquelle nous avons un peu détaillé l'utilisation des méthodes la classe `Polynomial`. La généralisation aux autres classes de polynômes est immédiate.

▷ L'exemple de la classe `Chebyshev`

```
>>> from numpy.polynomial import Chebyshev as T
>>> a = T([1,3,2]); a       # le polynôme a = T_0 + 3 T_1 + 2 T_2
Chebyshev([ 1.,  3.,  2.], [-1.,  1.], [-1.,  1.])
>>> type(a)
<class 'numpy.polynomial.chebyshev.Chebyshev'>
```

On ne constate pas beaucoup de changement par rapport ce qu'on a vu avec la classe `Polynomial` : le polynôme est toujours identifié par la liste de ses coefficients (dans le sens des degrés croissants) et il est encore muni des attributs *domaine* et *fenêtre* (par défaut à nouveau égaux à $[-1, 1]$).

D'ailleurs, juste pour voir, on va également importer la classe `Polynomial`.

```
>>> from numpy.polynomial import Polynomial as P
>>> b = P([1,3,2]); b       # le polynôme b = 1 + 3X + 2X²
Polynomial([ 1.,  3.,  2.], [-1.,  1.], [-1.,  1.])
>>> type(b)
<class 'numpy.polynomial.polynomial.Polynomial'>
```

À ce stade, il faut imprimer le contenu de a et b pour voir qu'il ne s'agit pas du tout du même objet.

Et il est heureusement impossible d'additionner (par exemple) ces deux polynômes !

```
>>> print(a)
cheb([ 1.  3.  2.])
>>> print(b)
poly([ 1.  3.  2.])
```

```
>>> a + b
Traceback (most recent call last):
[...]
TypeError: Polynomial types differ
```

Pour comparer deux polynômes, il faut les écrire dans une même base. Et pour cela, il faut être capable de convertir un polynôme écrit dans une base en un polynôme écrit dans une autre base.

C'est ce que fait la méthode `convert` de chacune des classes `Polynomial`, `Chebyshev`, etc.

Par exemple : `P.convert(p,kind=T)` renvoie la conversion d'un polynôme p dans la base de Chebyshev.

Inversement, `T.convert(q,kind=P)` renvoie la conversion d'un polynôme q dans la base canonique (tout cela suppose bien sûr qu'on a importé ces classes avec les noms courts `P` et `T`).

Dans l'exemple suivant, on convertit les six premiers polynômes de la base de Chebyshev dans la base canonique (on voit à cette occasion que ce sont les polynômes de Chebyshev dits « de première espèce »).

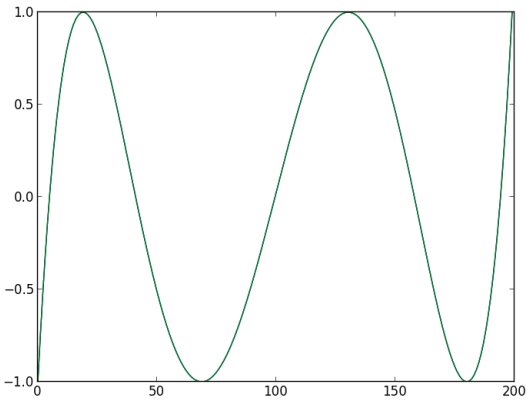
```
>>> for k in range(8): print(T.convert(T.basis(k),kind=P))
poly([ 1.])                                # le polynôme T_0 = 1
poly([ 0.  1.])                            # le polynôme T_1 = X
poly([-1.  0.  2.])                        # le polynôme T_2 = -1 + 2X^2
poly([ 0. -3.  0.  4.])                    # le polynôme T_3 = -3X + 4X^3
poly([ 1.  0. -8.  0.  8.])                # le polynôme T_4 = 1 - 8X^2 + 8X^4
poly([  0.   5.   0. -20.   0.  16.])      # le polynôme T_5 = 5X - 20X^3 + 16X^5
```

Inversement, voici les polynômes $1, X, \dots, X^7$ exprimés en fonction des polynômes $T_0, T_1, \dots, T_7$.

```
>>> for k in range(8): print(P.convert(P.basis(k),kind=T))
cheb([ 1.])                                                    # 1   =  T_0
cheb([ 0.     1.])                                             # X   =  T_1
cheb([ 0.5    0.     0.5])                                     # X^2 =  (T_0 + T_2)/2
cheb([ 0.     0.75   0.     0.25])                            # X^3 =  (3T_1 + T_3)/4
cheb([ 0.375  0.     0.5    0.     0.125])                   # X^4 =  (3T_0 + 4T_2 + T_4)/8
cheb([ 0.     0.625  0.     0.3125 0.     0.0625])          # X^5 =  (10T_1 + 5T_3 + T_5)/16
cheb([ 0.3125 0.     0.46875 0.     0.1875 0.     0.03125])
cheb([ 0.     0.546875 0.    0.328125 0.    0.109375 0.     0.015625])
```

Il reste bien sûr la possibilité d'effectuer un tracé, comme on le voit ici avec l'exemple du polynôme T_5, dont on a tracé les valeurs sur un échantillon de 200 points du segment $[-1, 1]$:

```
>>> import matplotlib.pyplot as plt
>>> plt.plot(T.basis(5)(np.linspace(-1,1,200)))
[<matplotlib.lines.Line2D object at 0x7dde890>]
>>> plt.show()
```



Mises à jour

La version la plus récente de ce document est disponible sur le site mathprepa.fr

Auteur

Jean-Michel Ferrard, jean-miche.ferrard@ac-paris.fr