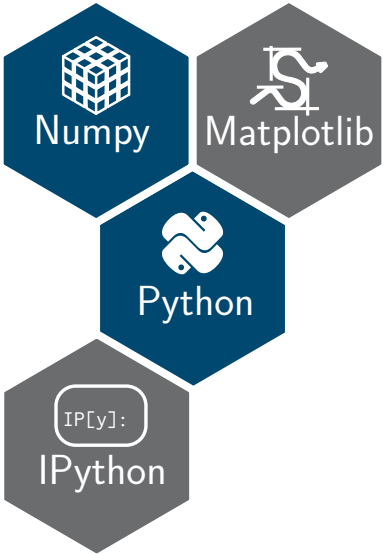


Option « Programmation en Python »

`numpy` : librairie pour le calcul scientifique

numpy ?

- Le module numpy est l'outil de base utilisé dans tous calculs scientifiques et donc numériques en Python
- numpy fournit en particulier des objets de type **vecteurs, matrices et plus généralement tableaux à *n* dimensions**
- numpy facilite et **optimise**[†] les opérations de **stockage et de manipulation** des données **numériques** notamment lorsque la taille des tableaux devient importante → *array oriented computing*

[†] les principales fonctions de numpy sont implémentées en C et en Fortran

Installation & importation de `numpy`

- Installation *via* `pip`

```
>_ pip install numpy
```

- Convention d'importation

```
In [1]: import numpy as np
```

Installation & importation de numpy

- Installation *via* pip

```
>_ pip install numpy
```

- Convention d'importation

```
In [1]: import numpy as np
```

- Documentation de référence du module http://docs.scipy.org/
- Aide interactive

```
In [1]: np.array?
     1 Docstring:
     2 array(object, dtype=None, copy=True, order=None, subok=False, ndmin=0)
     3
     4 Create an array.
     5 ...
```

```
In [2]: np.lookfor("create array")
     1 Search results for 'create array'
     2 ----------------------------------
     3 numpy.array
     4     Create an array.
     5 numpy.memmap
     6     Create a memory-map to an array stored in a *binary* file on disk.
```

- Documentation de référence du module http://docs.scipy.org/
- Aide interactive

```
In [1]: np.array?
     1 Docstring:
     2 array(object, dtype=None, copy=True, order=None, subok=False, ndmin=0)
     3
     4 Create an array.
     5 ...
```

```
In [2]: np.lookfor("create array")
     1 Search results for 'create array'
     2 --------------------------------
     3 numpy.array
     4     Create an array.
     5 numpy.memmap
     6     Create a memory-map to an array stored in a *binary* file on disk.
```

Création de vecteurs, matrices

- À partir d'une liste de valeurs
  - Vecteur

```
In [2]: v = np.array([0, 1, 2, 3])
In [3]: v
Out[3]: array([0, 1, 2, 3])
```

  - Matrice 2×2

```
In [4]: M = np.array([[0, 1], [2, 3]])
In [5]: M
Out[5]:
array([[0, 1],
       [2, 3]])
```

```
In [6]: type(v), type(M)
Out[6]: (numpy.ndarray, numpy.ndarray)

In [7]: v.ndim, M.ndim
Out[7]: (1, 2)

In [8]: v.shape, M.shape
Out[8]: ((4,), (2, 2))
```

Création de vecteurs, matrices

- À partir d'une liste de valeurs
  - Vecteur

```
In [2]: v = np.array([0, 1, 2, 3])
In [3]: v
Out[3]: array([0, 1, 2, 3])
```

  - Matrice 2×2

```
In [4]: M = np.array([[0, 1], [2, 3]])
In [5]: M
Out[5]:
array([[0, 1],
       [2, 3]])
```

```
In [6]: type(v), type(M)
Out[6]: (numpy.ndarray, numpy.ndarray)

In [7]: v.ndim, M.ndim
Out[7]: (1, 2)

In [8]: v.shape, M.shape
Out[8]: ((4,), (2, 2))
```

Création de vecteurs, matrices

- À partir d'une liste de valeurs
  - Vecteur

```
In [2]: v = np.array([0, 1, 2, 3])
In [3]: v
Out[3]: array([0, 1, 2, 3])
```

  - Matrice 2×2

```
In [4]: M = np.array([[0, 1], [2, 3]])
In [5]: M
Out[5]:
array([[0, 1],
       [2, 3]])
```

```
In [6]: type(v), type(M)
Out[6]: (numpy.ndarray, numpy.ndarray)

In [7]: v.ndim, M.ndim
Out[7]: (1, 2)

In [8]: v.shape, M.shape
Out[8]: ((4,), (2, 2))
```

Pourquoi `numpy` ?

- Les objets de type `numpy.ndarray` $\equiv$ à une liste Python (ou liste de listes)
- **Pourquoi ne pas simplement utiliser les listes Python pour les calculs au lieu de créer un nouveau type de tableau ?**

Il existe plusieurs (très bonnes) raisons à cela:

- Les listes Python sont très générales (on parle également d'objet de haut niveau). **Elles peuvent contenir n'importe quel objet → typage dynamique**. Elles ne supportent pas les opérations mathématiques.
- Les tableaux ou *array* de numpy sont **statiquement typées et homogènes**[†]
  - Le type des éléments est déterminé lorsque le tableau est créé → **plus de typage dynamique**
  - De même la taille du tableau est fixée à la création → **stockage en mémoire optimisée**
- En raison du typage statique, les fonctions mathématiques telles que la multiplication et l'addition de matrices peuvent être mises en œuvre *via* un langage compilé (C et Fortran)

[†] pour plus de détails, *cf.* discussion

Il existe plusieurs (très bonnes) raisons à cela:

- Les listes Python sont très générales (on parle également d'objet de haut niveau). **Elles peuvent contenir n'importe quel objet → typage dynamique**. Elles ne supportent pas les opérations mathématiques.
- Les tableaux ou *array* de numpy sont **statiquement typées et homogènes**[†]
  - Le type des éléments est déterminé lorsque le tableau est créé → **plus de typage dynamique**
  - De même la taille du tableau est fixée à la création → **stockage en mémoire optimisée**
- En raison du typage statique, les fonctions mathématiques telles que la multiplication et l'addition de matrices peuvent être mises en œuvre *via* un langage compilé (C et Fortran)

[†] pour plus de détails, *cf.* discussion

Il existe plusieurs (très bonnes) raisons à cela:

- Les listes Python sont très générales (on parle également d'objet de haut niveau). **Elles peuvent contenir n'importe quel objet → typage dynamique**. Elles ne supportent pas les opérations mathématiques.
- Les tableaux ou *array* de numpy sont **statiquement typées et homogènes**[†]
  - Le type des éléments est déterminé lorsque le tableau est créé → **plus de typage dynamique**
  - De même la taille du tableau est fixée à la création → **stockage en mémoire optimisée**
- En raison du typage statique, les fonctions mathématiques telles que la multiplication et l'addition de matrices peuvent être mises en œuvre *via* un langage compilé (C et Fortran)

[†] pour plus de détails, *cf.* discussion

Il existe plusieurs (très bonnes) raisons à cela:

- Les listes Python sont très générales (on parle également d'objet de haut niveau). **Elles peuvent contenir n'importe quel objet → typage dynamique**. Elles ne supportent pas les opérations mathématiques.
- Les tableaux ou *array* de numpy sont **statiquement typées et homogènes**[†]
  - Le type des éléments est déterminé lorsque le tableau est créé → **plus de typage dynamique**
  - De même la taille du tableau est fixée à la création → **stockage en mémoire optimisée**
- En raison du typage statique, les fonctions mathématiques telles que la multiplication et l'addition de matrices peuvent être mises en œuvre *via* un langage compilé (C et Fortran)

[†] pour plus de détails, *cf.* discussion

Il existe plusieurs (très bonnes) raisons à cela:

- Les listes Python sont très générales (on parle également d'objet de haut niveau). **Elles peuvent contenir n'importe quel objet → typage dynamique**. Elles ne supportent pas les opérations mathématiques.
- Les tableaux ou *array* de numpy sont **statiquement typées et homogènes**[†]
  - Le type des éléments est déterminé lorsque le tableau est créé → **plus de typage dynamique**
  - De même la taille du tableau est fixée à la création → **stockage en mémoire optimisée**
- En raison du typage statique, les fonctions mathématiques telles que la multiplication et l'addition de matrices peuvent être mises en œuvre *via* un langage compilé (C et Fortran)

[†] pour plus de détails, *cf.* discussion

Pourquoi numpy ?

- Démonstration

```
In [1]: %timeit [i**2 for i in range(1000)]
1000 loops, best of 3: 403 us per loop
```

```
In [3]: a = np.arange(1000)
In [4]: %timeit a**2
100000 loops, best of 3: 12.7 us per loop
```

Pourquoi numpy ?

- Démonstration

```
In [1]: %timeit [i**2 for i in range(1000)]
1000 loops, best of 3: 403 us per loop
```

```
In [3]: a = np.arange(1000)
In [4]: %timeit a**2
100000 loops, best of 3: 12.7 us per loop
```

Création de tableau (suite)

- Le type de données numériques est défini à la création du tableau
- Vecteur d'entiers

```
In [1]: v = np.array([0, 1, 2, 3])
In [2]: v
Out[2]: array([0, 1, 2, 3])

In [3]: v.dtype
Out[3]: dtype('int64')
```

- Vecteur de nombres flottants

```
In [1]: v = np.array([0., 1., 2., 3.])
In [2]: v.dtype
Out[2]: dtype('float64')
```

- ou en forçant le type de données (`float`, `int`, `bool`, 16, 32, 64 bits)

```
In [1]: v = np.array([0, 1, 2, 3], dtype=np.float)
In [2]: v.dtype
Out[2]: dtype('float64')
```

Création de tableau (suite)

- Le type de données numériques est défini à la création du tableau
- Vecteur d'entiers

```
In [1]: v = np.array([0, 1, 2, 3])
In [2]: v
Out[2]: array([0, 1, 2, 3])

In [3]: v.dtype
Out[3]: dtype('int64')
```

- Vecteur de nombres flottants

```
In [1]: v = np.array([0., 1., 2., 3.])
In [2]: v.dtype
Out[2]: dtype('float64')
```

- ou en forçant le type de données (`float`, `int`, `bool`, 16, 32, 64 bits)

```
In [1]: v = np.array([0, 1, 2, 3], dtype=np.float)
In [2]: v.dtype
Out[2]: dtype('float64')
```

Création de tableau (suite)

- Le type de données numériques est défini à la création du tableau
- Vecteur d'entiers

```
In [1]: v = np.array([0, 1, 2, 3])
In [2]: v
Out[2]: array([0, 1, 2, 3])

In [3]: v.dtype
Out[3]: dtype('int64')
```

- Vecteur de nombres flottants

```
In [1]: v = np.array([0., 1., 2., 3.])
In [2]: v.dtype
Out[2]: dtype('float64')
```

- ou en forçant le type de données (`float`, `int`, `bool`, 16, 32, 64 bits)

```
In [1]: v = np.array([0, 1, 2, 3], dtype=np.float)
In [2]: v.dtype
Out[2]: dtype('float64')
```

Création de tableau (suite)

- Le type de données numériques est défini à la création du tableau
- Vecteur d'entiers

```
In [1]: v = np.array([0, 1, 2, 3])
In [2]: v
Out[2]: array([0, 1, 2, 3])

In [3]: v.dtype
Out[3]: dtype('int64')
```

- Vecteur de nombres flottants

```
In [1]: v = np.array([0., 1., 2., 3.])
In [2]: v.dtype
Out[2]: dtype('float64')
```

- ou en forçant le type de données (`float`, `int`, `bool`, 16, 32, 64 bits)

```
In [1]: v = np.array([0, 1, 2, 3], dtype=np.float)
In [2]: v.dtype
Out[2]: dtype('float64')
```

Création de tableau (suite)

Dans la pratique, les valeurs d'un tableau sont rarement saisies une par une

- Fonction `arange` $\equiv$ `range`

```
In [1]: np.arange(10)
Out[1]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [2]: np.arange(0, 10, step=2)
Out[2]: array([0, 2, 4, 6, 8])
```

- Fonctions `linspace/logspace`

```
In [1]: np.linspace(0, 10, num=5)
Out[1: array([  0. ,   2.5,   5. ,   7.5,  10. ])

In [2]: np.logspace(0, 10, num=5)
Out[2]:
array([  1.00000000e+00,   3.16227766e+02,   1.00000000e+05,
         3.16227766e+07,   1.00000000e+10])
```

Création de tableau (suite)

Dans la pratique, les valeurs d'un tableau sont rarement saisies une par une

- Fonction `arange` ≡ `range`

```
In [1]: np.arange(10)
Out[1]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [2]: np.arange(0, 10, step=2)
Out[2]: array([0, 2, 4, 6, 8])
```

- Fonctions `linspace/logspace`

```
In [1]: np.linspace(0, 10, num=5)
Out[1: array([  0. ,   2.5,   5. ,   7.5,  10. ])

In [2]: np.logspace(0, 10, num=5)
Out[2]:
array([  1.00000000e+00,   3.16227766e+02,   1.00000000e+05,
         3.16227766e+07,   1.00000000e+10])
```

Création de tableau (suite)

- Vecteurs, matrices avec valeurs par défaut

```
In [1]: np.zeros(10)
Out[1]: array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])

In [2]: np.ones(shape=(3,3))
Out[2]:
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])

In [3]: np.full((3,3), 666, dtype=np.int)
Out[3]:
array([[666, 666, 666],
       [666, 666, 666],
       [666, 666, 666]])

In [4]: np.eye((3,3))
Out[4]:
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
```

Création de tableau (suite)

- Générateurs aléatoires `rand/randint/randn`

```
In [1]: np.random.rand(3)
Out[1]: array([ 0.21401051,  0.19514481,  0.92647823])

In [2]: np.random.randint(0, 10, 3)
Out[2]: array([8, 8, 3])

In [3]: np.random.randn(3)
Out[3]: array([-0.4829445 , -1.05459848, -1.30539831])
```

⚠ Un générateur aléatoire n'est par définition pas aléatoire dans une machine déterministe qu'est un ordinateur !

```
In [1]: np.random.seed(1234)
```

Création de tableau (suite)

- Générateurs aléatoires `rand/randint/randn`

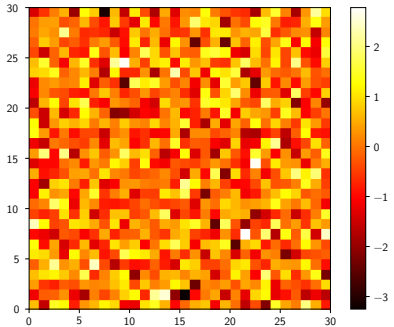
```
In [1]: np.random.rand(3)
Out[1]: array([ 0.21401051,  0.19514481,  0.92647823])

In [2]: np.random.randint(0, 10, 3)
Out[2]: array([8, 8, 3])

In [3]: np.random.randn(3)
Out[3]: array([-0.4829445 , -1.05459848, -1.30539831])
```

⚠ Un générateur aléatoire n'est par définition pas aléatoire dans une machine déterministe qu'est un ordinateur !

```
In [1]: np.random.seed(1234)
```

Création de tableau : intermède graphique

```
In [1]: import numpy as np
In [2]: import matplotlib.pyplot as plt
In [3]: %matplotlib

In [4]: img = np.random.randn(30, 30)
In [5]: plt.imshow(img, cmap=plt.cm.hot,
                   extent=(0,30,0,30))
In [6]: plt.colorbar()
```



Parcours par indice

- L'utilisation de l'opérateur `[]` est similaire à celle des listes

```
In [1]: x = np.random.randint(10, size=5)
In [2]: x
Out[2]: array([8, 0, 1, 6, 0])

In [3]: x[0], x[3], x[-1]
Out[3]: (8, 6, 0)
```

- Pour les tableaux à *n* dimensions

```
In [1]: x = np.random.randint(10, size=(3, 4))
In [2]: x
Out[2]:
array([[8, 3, 6, 4],
       [9, 8, 2, 0],
       [0, 5, 5, 4]])

In [3]: x[0, 0], x[2, 0], x[2, -1]
Out[3]: (8, 0, 4)
```

Parcours par indice

- L'utilisation de l'opérateur `[]` est similaire à celle des listes

```
In [1]: x = np.random.randint(10, size=5)
In [2]: x
Out[2]: array([8, 0, 1, 6, 0])

In [3]: x[0], x[3], x[-1]
Out[3]: (8, 6, 0)
```

- Pour les tableaux à *n* dimensions

```
In [1]: x = np.random.randint(10, size=(3, 4))
In [2]: x
Out[2]:
array([[8, 3, 6, 4],
       [9, 8, 2, 0],
       [0, 5, 5, 4]])

In [3]: x[0, 0], x[2, 0], x[2, -1]
Out[3]: (8, 0, 4)
```

- Comme pour les listes qui sont des objets *mutables*, il est possible d'assigner une valeur en spécifiant l'indice

```
In [4]: x[0, 0] = 12
In [5]: x
Out[5]:
array([[12, 3, 6, 4],
       [ 9, 8, 2, 0],
       [ 0, 5, 5, 4]])
```

⚠ Le type de données numériques stockées est fixé à la création du tableau

```
In [6]: x[0, 0] = 3.1415
In [7]: x
Out[7]:
array([[3, 3, 6, 4],
       [9, 8, 2, 0],
       [0, 5, 5, 4]])
```

- Comme pour les listes qui sont des objets *mutables*, il est possible d'assigner une valeur en spécifiant l'indice

```
In [4]: x[0, 0] = 12
In [5]: x
Out[5]:
array([[12, 3, 6, 4],
       [ 9, 8, 2, 0],
       [ 0, 5, 5, 4]])
```

⚠ Le type de données numériques stockées est fixé à la création du tableau

```
In [6]: x[0, 0] = 3.1415
In [7]: x
Out[7]:
array([[3, 3, 6, 4],
       [9, 8, 2, 0],
       [0, 5, 5, 4]])
```

Sélection par indice

- Comme pour les listes, il est possible d'utiliser la syntaxe `[start:stop:step]` pour sélectionner un sous espace vectoriel

```
In [1]: x
Out[1]:
array([[3, 3, 6, 4],
       [9, 8, 2, 0],
       [0, 5, 5, 4]])
```

  - Sélection d'une ligne

```
In [2]: x[0]
Out[2]: array([[3, 3, 6, 4])
```

  - Sélection d'une colonne

```
In [2]: x[:, 0], x[:, 1]
Out[2]: (array([3, 9, 0]), array([3, 8, 5]))
```

- Comme pour les listes, il est possible d'utiliser la syntaxe `[start:stop:step]` pour sélectionner un sous espace vectoriel

```
In [1]: x
Out[1]:
array([[3, 3, 6, 4],
       [9, 8, 2, 0],
       [0, 5, 5, 4]])
```

  - Sélection d'une ligne

```
In [2]: x[0]
Out[2]: array([[3, 3, 6, 4])
```

  - **Sélection d'une colonne**

```
In [2]: x[:, 0], x[:, 1]
Out[2]: (array([3, 9, 0]), array([3, 8, 5]))
```

Sélection par indice

- À la différence des listes, les sous espaces vectoriels sélectionnés ne sont pas des copies mais **une vue réduite** de la matrice globale
- Toute modification opérée sur le sous espace vectoriel est reportée dans la matrice globale

```
In [1]: x
Out[1]:
array([[3, 3, 6, 4],
       [9, 8, 2, 0],
       [0, 5, 5, 4]])

In [2]: xx = x[:2, :2]
In [3]: xx
Out[3]:
array([[3, 3],
       [9, 8]])

In [4]: xx[0, 0] = 0
In [5]: x
Out[5]:
array([[0, 3, 6, 4],
       [9, 8, 2, 0],
       [0, 5, 5, 4]])
```

Sélection par indice

- À la différence des listes, les sous espaces vectoriels sélectionnés ne sont pas des copies mais **une vue réduite** de la matrice globale
- Toute modification opérée sur le sous espace vectoriel est reportée dans la matrice globale

```
In [1]: x
Out[1]:
array([[3, 3, 6, 4],
       [9, 8, 2, 0],
       [0, 5, 5, 4]])

In [2]: xx = x[:2, :2]
In [3]: xx
Out[3]:
array([[3, 3],
       [9, 8]])

In [4]: xx[0, 0] = 0
In [5]: x
Out[5]:
array([[0, 3, 6, 4],
       [9, 8, 2, 0],
       [0, 5, 5, 4]])
```

- Pour réaliser une copie d'un sous espace vectoriel, on utilisera la méthode copy()

```
In [2]: xx = x[:2, :2].copy()
In [3]: xx
Out[3]:
array([[0, 3],
       [9, 8]])

In [4]: xx[0, 0] = 666
In [5]: x
Out[5]:
array([[0, 3, 6, 4],
       [9, 8, 2, 0],
       [0, 5, 5, 4]])
```

- **Grâce à l'homogénéité des tableaux de `numpy`**, il est possible de réaliser des opérations mathématiques $\neq$ listes Python

```
In [1]: l = [1, 2, 3, 4]
In [2]: l+5
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-53-1cb32c2d071d> in <module>()
----> 1 l+5

TypeError: can only concatenate list (not "int") to list
```

- **Grâce à l'homogénéité des tableaux de `numpy`**, il est possible de réaliser des opérations mathématiques
- Opérateurs binaires

```
In [1]: x = np.arange(4)
In [2]: x
Out[2]: array([0, 1, 2, 3])

In [3]: x+5
Out[3]: array([5, 6, 7, 8])

In [4]: x-5
Out[4]: array([-5, -4, -3, -2])

In [5]: x*5
Out[5]: array([ 0,  5, 10, 15])

In [5]: x/5
Out[5]: array([ 0. ,  0.2,  0.4,  0.6])
```

- **Grâce à l'homogénéité des tableaux de `numpy`**, il est possible de réaliser des opérations mathématiques
- Opérateurs unaires

```
In [1]: x = np.arange(4)

In [2]: -x
Out[2]: array([0, -1, -2, -3])

In [3]: x**2
Out[3]: array([0, 1, 4, 9])

In [4]: x%2
Out[4]: array([0, 1, 0, 1])
```

- En plus des opérateurs usuels, `numpy` fournit un ensemble de **fonctions dites universelles** (ou *ufuncs*) opérant sur des tableaux
- Fonctions trigonométriques

```
In [1]: theta = np.linspace(0, np.pi, 3)

In [2]: np.cos(theta)
Out[2]: array([  1.00000000e+00,   6.12323400e-17,  -1.00000000e+00])

In [3]: np.sin(theta)
Out[3]: array([  0.00000000e+00,   1.00000000e+00,   1.22464680e-16])

In [4]: np.tan(theta)
Out[4]: array([  0.00000000e+00,   1.63312394e+16,  -1.22464680e-16])
```

- Autres fonctions : `np.exp()`, `np.power()`, `np.log()`, `np.log10()`,...

- En plus des opérateurs usuels, `numpy` fournit un ensemble de **fonctions dites universelles** (ou *ufuncs*) opérant sur des tableaux
- Fonctions trigonométriques

```
In [1]: theta = np.linspace(0, np.pi, 3)

In [2]: np.cos(theta)
Out[2]: array([  1.00000000e+00,   6.12323400e-17,  -1.00000000e+00])

In [3]: np.sin(theta)
Out[3]: array([  0.00000000e+00,   1.00000000e+00,   1.22464680e-16])

In [4]: np.tan(theta)
Out[4]: array([  0.00000000e+00,   1.63312394e+16,  -1.22464680e-16])
```

- Autres fonctions : `np.exp()`, `np.power()`, `np.log()`, `np.log10()`,...

- En plus des opérateurs usuels, `numpy` fournit un ensemble de **fonctions dites universelles** (ou *ufuncs*) opérant sur des tableaux
- Fonctions trigonométriques

```
In [1]: theta = np.linspace(0, np.pi, 3)

In [2]: np.cos(theta)
Out[2]: array([  1.00000000e+00,   6.12323400e-17,  -1.00000000e+00])

In [3]: np.sin(theta)
Out[3]: array([  0.00000000e+00,   1.00000000e+00,   1.22464680e-16])

In [4]: np.tan(theta)
Out[4]: array([  0.00000000e+00,   1.63312394e+16,  -1.22464680e-16])
```

- Autres fonctions : `np.exp()`, `np.power()`, `np.log()`, `np.log10()`,...

Opérations statistiques

- Somme des éléments d'un tableau

```
In [1]: x = np.random.rand(100)

In [2]: sum(x)
Out[2]: 50.394482884150314

In [3]: np.sum(x)
Out[3]: 50.394482884150314
```

- Toutefois, la formulation `np.sum()` propre à `numpy` présente l'avantage d'être nettement plus rapide (code compilé) en plus d'être plus générale

```
In [4]: big_array = np.random.rand(1000000)

In [5]: %timeit sum(big_array)
10 loops, best of 3: 82.9 ms per loop

In [6]: %timeit np.sum(big_array)
1000 loops, best of 3: 467 µs per loop
```

Opérations statistiques

- Somme des éléments d'un tableau

```
In [1]: x = np.random.rand(100)

In [2]: sum(x)
Out[2]: 50.394482884150314

In [3]: np.sum(x)
Out[3]: 50.394482884150314
```

- Toutefois, la formulation `np.sum()` propre à `numpy` présente l'avantage d'être nettement plus rapide (code compilé) en plus d'être plus générale

```
In [4]: big_array = np.random.rand(1000000)

In [5]: %timeit sum(big_array)
10 loops, best of 3: 82.9 ms per loop

In [6]: %timeit np.sum(big_array)
1000 loops, best of 3: 467 µs per loop
```

Opérations statistiques

- Somme des éléments d'un tableau : méthode `sum`

```
In [1]: M = np.random.randint(10, (3, 4))
In [2]: M
Out[2]:
array([[7, 0, 8, 4],
       [4, 7, 0, 5],
       [7, 0, 7, 6]])

In [3]: np.sum(M), M.sum()
Out[3]: (55, 55)
```

- Somme colonne par colonne

```
In [4]: M.sum(axis=0)
Out[4]: array([18,  7, 15, 15])
```

- Somme ligne par ligne

```
In [5]: M.sum(axis=1)
Out[5]: array([19, 16, 20])
```

Opérations statistiques

- Somme des éléments d'un tableau : méthode `sum`

```
In [1]: M = np.random.randint(10, (3, 4))
In [2]: M
Out[2]:
array([[7, 0, 8, 4],
       [4, 7, 0, 5],
       [7, 0, 7, 6]])

In [3]: np.sum(M), M.sum()
Out[3]: (55, 55)
```

- Somme colonne par colonne

```
In [4]: M.sum(axis=0)
Out[4]: array([18,  7, 15, 15])
```

- Somme ligne par ligne

```
In [5]: M.sum(axis=1)
Out[5]: array([19, 16, 20])
```

- Somme des éléments d'un tableau : méthode `sum`

```
In [1]: M = np.random.randint(10, (3, 4))
In [2]: M
Out[2]:
array([[7, 0, 8, 4],
       [4, 7, 0, 5],
       [7, 0, 7, 6]])

In [3]: np.sum(M), M.sum()
Out[3]: (55, 55)
```

- Somme colonne par colonne

```
In [4]: M.sum(axis=0)
Out[4]: array([18,  7, 15, 15])
```

- Somme ligne par ligne

```
In [5]: M.sum(axis=1)
Out[5]: array([19, 16, 20])
```

Opérations statistiques

Fonction	Description
`np.sum`	Somme des éléments
`np.prod`	Produit des éléments
`np.mean`	Valeur moyenne
`np.std`	Standard déviation
`np.var`	Variance
`np.min`	Valeur minimale
`np.max`	Valeur maximale
`np.argmin`	Indice de la valeur minimale
`np.argmax`	Indice de la valeur maximale
`np.median`	Valeur médiane
`np.percentile`	Quantiles

- Multiplication de matrices

```
In [1]: M = np.ones(shape=(3,3))
In [2]: M
Out[2]:
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])

In [3]: M*M
Out[3]:
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])

In [4]: M.dot(M)
Out[4]:
array([[ 3.,  3.,  3.],
       [ 3.,  3.,  3.],
       [ 3.,  3.,  3.]])
```

- Transposition de matrices

```
In [1]: M = np.random.randint(5, size=(3,3))
In [2]: M
Out[2]:
array([[4, 1, 0],
       [2, 3, 0],
       [1, 0, 2]])

In [3]: M.transpose()
Out[3]:
array([[4, 2, 1],
       [1, 3, 0],
       [0, 0, 2]])
```

- Conversion d'un vecteur vers une matrice

```
In [1]: v = np.arange(4)
In [2]: v
Out[2]: array([0, 1, 2, 3])

In [3]: v[:, np.newaxis]
Out[3]:
array([[0],
       [1],
       [2],
       [3]])
```

Opérations algébriques : intermède graphique

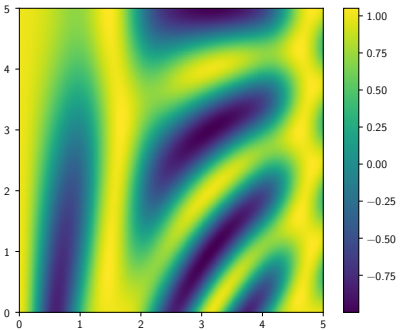
$$
\begin{aligned}
z = f(x, y) &= \sin^{10} x + \cos(x \cdot y) \cdot \cos x \\
&= \sin^{10} \begin{bmatrix} x_0 & \cdots \end{bmatrix} + \cos \left(\begin{bmatrix} x_0 & \cdots \end{bmatrix} \cdot \begin{bmatrix} y_0 \\ \vdots \end{bmatrix} \right) \cdot \cos \begin{bmatrix} x_0 & \cdots \end{bmatrix}
\end{aligned}
$$

```
In [1]: import numpy as np
In [2]: import matplotlib.pyplot as plt
In [3]: %matplotlib

In [4]: x = np.linspace(0, 5, 500)
In [5]: y = np.linspace(0, 5, 500)[:, np.newaxis]
In [6]: z = np.sin(x)**10 + np.cos(x*y)*np.cos(x)
In [7]: x.shape, y.shape, z.shape
Out[7]: ((500,), (500, 1), (500, 500))

In [8]: plt.imshow(z, extent=[0, 5, 0, 5],
                   cmap="viridis")
In [9]: plt.colorbar();
```

Opérations algébriques : intermède graphique

$$z = f(x, y) = \sin^{10} x + \cos(x \cdot y) \cdot \cos x$$
$$= \sin^{10} \begin{bmatrix} x_0 & \cdots \end{bmatrix} + \cos \left(\begin{bmatrix} x_0 & \cdots \end{bmatrix} \cdot \begin{bmatrix} y_0 \\ \vdots \end{bmatrix} \right) \cdot \cos \begin{bmatrix} x_0 & \cdots \end{bmatrix}$$

```
In [1]: import numpy as np
In [2]: import matplotlib.pyplot as plt
In [3]: %matplotlib

In [4]: x = np.linspace(0, 5, 500)
In [5]: y = np.linspace(0, 5, 500)[:, np.newaxis]
In [6]: z = np.sin(x)**10 + np.cos(x*y)*np.cos(x)
In [7]: x.shape, y.shape, z.shape
Out[7]: ((500,), (500, 1), (500, 500))

In [8]: plt.imshow(z, extent=[0, 5, 0, 5],
                   cmap="viridis")
In [9]: plt.colorbar();
```



Opérations logiques

- En plus des opérateurs et fonctions mathématiques, `numpy` fournit également les opérateurs de comparaison opérant sur les éléments d'un tableau

```
In [1]: x = np.array([1, 2, 3, 4, 5])

In [2]: x < 3
Out[2]: array([ True,  True, False, False, False], dtype=bool)

In [3]: x == 3
Out[3]: array([False, False,  True, False, False], dtype=bool)

In [4]: (x * 2) == (x**2)
Out[4]: array([False,  True, False, False, False], dtype=bool)
```

Opérations logiques

- numpy fournit également les méthodes any et all

```
In [5]: np.any(x > 10)
Out[5]: False

In [6]: np.all(x < 10)
Out[6]: True
```

Opérations logiques

- Il est finalement possible de dénombrer le nombre de valeurs d'un tableau satisfaisant à une ou des conditions

```
In [7]: np.sum(x > 3)
Out[7]: 2

In [8]: np.sum((x > 3) & (x < 5))
Out[8]: 1
```

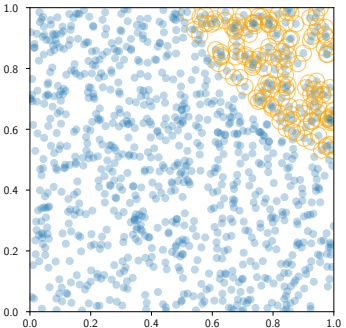
Sélection par masque

- Les opérations de comparaison sur des tableaux retournent un tableau de booléens qui peut servir à la sélection d'éléments du tableau

```
In [1]: x = np.random.randint(0, 10, 10)
In [2]: x
Out[2]: array([8, 9, 6, 2, 4, 5, 9, 4, 0, 7])

In [3]: x < 5
Out[3]: array([False, False, False,  True,  True, False, False,  True,  True, False], dtype=bool)

In [4]: x[x < 5]
Out[4]: array([2, 4, 4, 0])
```

Sélection par masque : intermède graphique

```
In [1]: import numpy as np
In [2]: import matplotlib.pyplot as plt
In [3]: %matplotlib

In [4]: x = np.random.rand(1000)
In [5]: y = np.random.rand(1000)
In [6]: plt.scatter(x, y, alpha=0.3)
In [7]: plt.axis("scaled"); plt.axis([0, 1, 0, 1])

In [8]: mask = (x*y > 0.5)
In [9]: plt.scatter(x[mask], y[mask], alpha=0.6,
                    edgecolors="orange", c="none",
                    s=200)
```



- numpy permet de charger un fichier texte dans un objet de type ndarray

```
In [1]: cat /tmp/results.tsv
# id        OPP    MQ1     MA
21606456   9.90  12.32  16.00
21402354  11.20  10.50  12.25

In [2]: results = np.loadtxt("/tmp/results.tsv")
In [3]: results
Out[3]:
array([[  2.16064560e+07,   9.90000000e+00,   1.23200000e+01,
          1.60000000e+01],
       [  2.14023540e+07,   1.12000000e+01,   1.05000000e+01,
          1.22500000e+01]])
```

Entrées/sorties de numpy

- numpy permet également de sauvegarder un tableau dans un fichier texte

```
In [4]: np.savetxt("/tmp/results2.tsv", results)
```

- le module pandas est toutefois bien mieux adapté à la lecture de fichier contenant des données numériques

Entrées/sorties de numpy

- numpy permet également de sauvegarder un tableau dans un fichier texte

```
In [4]: np.savetxt("/tmp/results2.tsv", results)
```

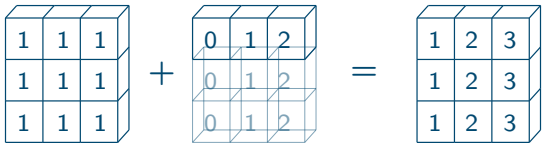
- le module pandas est toutefois bien mieux adapté à la lecture de fichier contenant des données numériques

Annexes

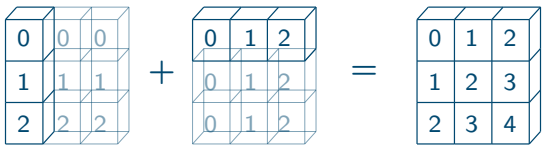
Opérations algébriques : *Broadcasting*†

`np.arange(3)+5`

`np.ones((3,3))+np.arange(3)`



`np.arange(3).reshape((3,1))+np.arange(3)`



† pour plus de détails, *cf.* discussion