Le module `numpy` : manipulation de tableaux

Les tableaux `numpy` sont fournis avec de nombreuses fonctions. En particuler, il est possible de faire des manipulations algébriques terme à terme ou des manipulations vectorielles rapidement.

Nous noterons `x` un tableau à une dimension et `A` un tableau à deux dimensions pour nos exemples.

In [27]:
```
import numpy as np

x = np.random.rand(5)  # ndarray de dimension 1 avec 5 éléments
A = np.random.rand(5, 4)  # ndarray de dimension 2 avec 5x4 élément
s

print(x)
print(A)
```

```
[0.4353689  0.86004402 0.68113537 0.75098234 0.70847761]
[[0.40296204 0.57118429 0.58225458 0.71689091]
 [0.73333866 0.04903131 0.9534291  0.81487278]
 [0.36907674 0.01568446 0.71536217 0.92957579]
 [0.67587119 0.2349833  0.81686407 0.04505474]
 [0.14043113 0.58567978 0.1844606  0.19235648]]
```

In [4]:
```
B = np.zeros(A.shape, dtype=int)
print(B.dtype)
B[:] = A
print(B)
```

```
int64
[[0 0 0 0]
 [0 0 0 0]
 [0 0 0 0]
 [0 0 0 0]
 [0 0 0 0]]
```

Caractéristiques des tableaux

Il est possible d'accéder facilement à la forme d'un tableau (attribut `shape`), au nombre total d'éléments (attribut `size`), au type des éléments (attribut `dtype`) et au nombre de dimensions (attribut `ndim`) :

In [2]:
```
for T in [x, A]:
    print(T.shape)  # forme du tableau (tuple)
    print(T.size)   # taille du tableau (int)
    print(T.dtype)  # type des éléments du tableau
    print(T.ndim)   # nombre de dimensions du tableau (taille de sh
ape)
```

```
(5,)
5
float64
1
(5, 4)
20
float64
2
```

Accès aux éléments en lecture et écriture : les slices

L'accès aux éléments en lecture et en écriture se fait grâce à l'opérateur `[]` .

Il est possible d'accéder à un seul élément : `x[i]` , pour `i` entre `0` et `x.size` , et `A[i, j]` , pour `i` entre `0` et `A.shape[0]` , `j` entre `0` et `A.shape[1]` .

In [15]:
```
print(x[1])
print(A[1, 2])
print(x[x.size-1])
print(A[1][2])  # solution interdite !!!
print(A[1, 2])
```

```
0.6434600715704256
0.6399495551622633
0.6635663333240868
0.6399495551622633
0.6399495551622633
```

In [16]:
```
print(A)
b = A[0]
print(type(b))
print(b.ndim, b.shape)
print(b[1])
b[1] = 0
print(A)
```

```
[[0.78623991 0.26053104 0.27598648 0.61229728]
 [0.70851122 0.66175222 0.63994956 0.0227666 ]
 [0.91054592 0.93936025 0.56463231 0.97984684]
 [0.33694979 0.09529412 0.94009188 0.98016555]
 [0.19446173 0.23792616 0.47398271 0.85189476]]
<class 'numpy.ndarray'>
1 (4,)
0.26053104120924075
[[0.78623991 0.         0.27598648 0.61229728]
 [0.70851122 0.66175222 0.63994956 0.0227666 ]
 [0.91054592 0.93936025 0.56463231 0.97984684]
 [0.33694979 0.09529412 0.94009188 0.98016555]
 [0.19446173 0.23792616 0.47398271 0.85189476]]
```

Il est possible d'accéder aux éléments de la fin du tableau en utilisant les nombres négatifs : le `-1` signifie le dernier élément, le `-2` l'avant-dernier, *etc*.

In [18]:
```
print(x[-1])
print(A[1, -2])
```

```
0.6635663333240868
0.0227665967052727
```

L'intérêt majeur des tableaux `numpy` est que l'on peut accéder directement à une *tranche* du tableau en utilisant les `:` . Voici quelques exemples :

In [24]:
```
print(x[:])     # tous les éléments
print(x[1:-1])  # tous les éléments sauf le premier et le dernier
print(list(x))
print(x[0])
```

```
[0.03605055 0.64346007 0.33159189 0.41639298 0.66356633]
[0.64346007 0.33159189 0.41639298]
[0.036050546021250574, 0.6434600715704256, 0.33159189430845337, 0.
41639298306462624, 0.6635663333240868]
0.036050546021250574
```

In [22]:
```
help(np.__str__)
```

```
Help on method-wrapper object:

__str__ = class method-wrapper(object)
 |  Methods defined here:
 |
 |  __call__(self, /, *args, **kwargs)
 |      Call self as a function.
 |
 |  __eq__(self, value, /)
 |      Return self==value.
 |
 |  __ge__(self, value, /)
 |      Return self>=value.
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __gt__(self, value, /)
 |      Return self>value.
 |
 |  __hash__(self, /)
 |      Return hash(self).
 |
 |  __le__(self, value, /)
 |      Return self<=value.
 |
 |  __lt__(self, value, /)
 |      Return self<value.
 |
 |  __ne__(self, value, /)
 |      Return self!=value.
 |
 |  __reduce__(...)
 |      Helper for pickle.
 |
 |  __repr__(self, /)
 |      Return repr(self).
 |
 |  ----------------------------------------------------------------------
--------
 |  Data descriptors defined here:
 |
 |  __objclass__
 |
 |  __self__
 |
 |  __text_signature__
```

In [25]:
```
print(A[:, 0])        # la première colonne
print(A[1:-1, 1:-1])  # on enlève les premières et dernières lignes
/colonnes
```

```
[0.78623991 0.70851122 0.91054592 0.33694979 0.19446173]
[[0.66175222 0.63994956]
 [0.93936025 0.56463231]
 [0.09529412 0.94009188]]
```

In [26]:
```
print(A[::2, ::-1])  # on peut aussi faire varier le pas de la tran
che !!!
```

```
[[0.61229728 0.27598648 0.         0.78623991]
 [0.97984684 0.56463231 0.93936025 0.91054592]
 [0.85189476 0.47398271 0.23792616 0.19446173]]
```

Modification des éléments

Un point très important sur les tableaux `numpy` : l'affectation ne fait pas de copie... Pour faire une copie, il est nécessaire d'utiliser la fonction membre `copy()`. Sinon il y a un risque de modifier le tableau original...

In [37]:
```
B = A.copy()
C = np.ones(A.shape)
print(B)
B_in = B[1:-1, 1:-1]    # B_in est une sous-partie de B
B_in[:] = C[1:-1, 1:-1] # on copie les valeurs de C
print(B_in)
B_in[:] = 0
print(B)
print(C)
```

```
[[0.40296204 0.57118429 0.58225458 0.71689091]
 [0.73333866 0.04903131 0.9534291  0.81487278]
 [0.36907674 0.01568446 0.71536217 0.92957579]
 [0.67587119 0.2349833  0.81686407 0.04505474]
 [0.14043113 0.58567978 0.1844606  0.19235648]]
[[1. 1.]
 [1. 1.]
 [1. 1.]]
[[0.40296204 0.57118429 0.58225458 0.71689091]
 [0.73333866 0.         0.         0.81487278]
 [0.36907674 0.         0.         0.92957579]
 [0.67587119 0.         0.         0.04505474]
 [0.14043113 0.58567978 0.1844606  0.19235648]]
[[1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]]
```

Redimensionnement d'un tableau

L'attribut `shape` d'un tableau peut être modifié sans faire de copie. Pour faire simple, les tableaux `numpy` sont stockés de manière mono-dimensionnelle même si on peut les voir comme des matrices, ... Les dimensions supérieures à 2 ne servent en réalité qu'à accéder aux valeurs par d'autres vues.

Pour modifier la forme d'un tableau, on modifie l'attribut `shape`

In [40]:
```
x = np.arange(10)
print(x)
print(x.shape)
x.shape = (2, 5)
print(x.shape)
print(x)
x.shape = (5, 2)
print(x.shape)
print(x)
```

```
[0 1 2 3 4 5 6 7 8 9]
(10,)
(2, 5)
[[0 1 2 3 4]
 [5 6 7 8 9]]
(5, 2)
[[0 1]
 [2 3]
 [4 5]
 [6 7]
 [8 9]]
```

on peut aussi utiliser la commande `reshape` qui crée une vue (pas de copie)

In [41]:
```
y = x.reshape((10,))
y[::2] = -1
print(y)
print(x)
```

```
[-1  1 -1  3 -1  5 -1  7 -1  9]
[[-1  1]
 [-1  3]
 [-1  5]
 [-1  7]
 [-1  9]]
```

la fonction membre `flatten` fait une copie du tableau en écrasant toutes les dimensions

In [42]:
```
print(B)
C = B.flatten()  # shape = (size, )
print(C)
print(B.shape)
print(C.shape)
B[0, :] = 1
print(B)
print(C)
```

```
[[0.40296204 0.57118429 0.58225458 0.71689091]
 [0.73333866 0.         0.         0.81487278]
 [0.36907674 0.         0.         0.92957579]
 [0.67587119 0.         0.         0.04505474]
 [0.14043113 0.58567978 0.1844606  0.19235648]]
[0.40296204 0.57118429 0.58225458 0.71689091 0.73333866 0.
 0.         0.81487278 0.36907674 0.         0.         0.92957579
 0.67587119 0.         0.         0.04505474 0.14043113 0.58567978
 0.1844606  0.19235648]
(5, 4)
(20,)
[[1.         1.         1.         1.        ]
 [0.73333866 0.         0.         0.81487278]
 [0.36907674 0.         0.         0.92957579]
 [0.67587119 0.         0.         0.04505474]
 [0.14043113 0.58567978 0.1844606  0.19235648]]
[0.40296204 0.57118429 0.58225458 0.71689091 0.73333866 0.
 0.         0.81487278 0.36907674 0.         0.         0.92957579
 0.67587119 0.         0.         0.04505474 0.14043113 0.58567978
 0.1844606  0.19235648]
```

Boucle sur les tableaux

Il y a plusieurs façons de faire des boucles sur les tableaux : soit sur les indices, soit directement sur les éléments car un tableau peut être vu comme un itérable.

In [43]:
```
for i in range(A.shape[0]):
    for j in range(A.shape[1]):
        print(f"A[{i:1d}, {j:1d}] = {A[i, j]}")
```

```
A[0, 0] = 0.4029620436231577
A[0, 1] = 0.5711842918750226
A[0, 2] = 0.5822545811405262
A[0, 3] = 0.7168909148318624
A[1, 0] = 0.7333386577562792
A[1, 1] = 0.04903131283969531
A[1, 2] = 0.9534290983083085
A[1, 3] = 0.8148727841266159
A[2, 0] = 0.36907674145657765
A[2, 1] = 0.015684460895222352
A[2, 2] = 0.7153621725775243
A[2, 3] = 0.9295757850973615
A[3, 0] = 0.6758711947878449
A[3, 1] = 0.2349832954274067
A[3, 2] = 0.8168640664241296
A[3, 3] = 0.04505474363460582
A[4, 0] = 0.14043113199637214
A[4, 1] = 0.5856797844066353
A[4, 2] = 0.18446060330004188
A[4, 3] = 0.19235647620494312
```

In [44]:
```
for Ai in A:
    for Aij in Ai:
        print(Aij)
```

```
0.4029620436231577
0.5711842918750226
0.5822545811405262
0.7168909148318624
0.7333386577562792
0.04903131283969531
0.9534290983083085
0.8148727841266159
0.36907674145657765
0.015684460895222352
0.7153621725775243
0.9295757850973615
0.6758711947878449
0.2349832954274067
0.8168640664241296
0.04505474363460582
0.14043113199637214
0.5856797844066353
0.18446060330004188
0.19235647620494312
```

In [45]:
```
for i, Ai in enumerate(A):
    for j, Aij in enumerate(Ai):
        print(f"A[{i:1d}, {j:1d}] = {Aij}")
```

```
A[0, 0] = 0.4029620436231577
A[0, 1] = 0.5711842918750226
A[0, 2] = 0.5822545811405262
A[0, 3] = 0.7168909148318624
A[1, 0] = 0.7333386577562792
A[1, 1] = 0.04903131283969531
A[1, 2] = 0.9534290983083085
A[1, 3] = 0.8148727841266159
A[2, 0] = 0.36907674145657765
A[2, 1] = 0.015684460895222352
A[2, 2] = 0.7153621725775243
A[2, 3] = 0.9295757850973615
A[3, 0] = 0.6758711947878449
A[3, 1] = 0.2349832954274067
A[3, 2] = 0.8168640664241296
A[3, 3] = 0.04505474363460582
A[4, 0] = 0.14043113199637214
A[4, 1] = 0.5856797844066353
A[4, 2] = 0.18446060330004188
A[4, 3] = 0.19235647620494312
```

In [46]:
```
# accès en écriture car Ai est un ndarray
for Ai in A:
    Ai[:] = 0
print(A)

# accès en lecture seule (copie) car Aij est un float
for Ai in A:
    for Aij in Ai:
        Aij = 1
print(A)
```

```
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
```

In [ ]:
```
```