Python pour les scientifiques

Partie V – Les bibliothèques scientifiques

Cyril Desjouy

June, 2016
Updated : March 26, 2019

Conventions adoptée par la communauté[1] **:**

```
In  : import numpy as np
In  : import scipy as sp
In  : import matplotlib.pyplot as plt
```

Extrait de scipy.org :

- **NumPy**'s array type augments the Python language with an efficient data structure useful for numerical work, e.g., manipulating matrices. NumPy also provides basic numerical routines, such as tools for finding eigenvectors.
- **SciPy** contains additional routines needed in scientific work: for example, routines for computing integrals numerically, solving differential equations, optimization, and sparse matrices.
- The **Matplotlib** module produces high quality plots. With it you can turn your data or your models into figures for presentations or articles. No need to do the numerical work in one program, save the data, and plot it with another program.
- **SymPy**, for symbolic mathematics and computer algebra.

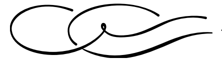
[1] Même si vous n'êtes pas obligés de suivre ces conventions, elles sont fortement recommandées.

01 Introduction à Numpy

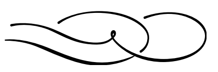
02 Introduction à Matplotlib

03 Introduction à Scipy

04 Introduction à Sympy



Introduction à Numpy



Syntaxe de la méthode array

```
# Chaque élément doit être du même type :  typage dynamique
In  : np.array([0, 1.])                    # Conversion de la liste [0,1.]...
Out : array([0., 1.])                      # ...en ndarray de float

# Forçage du type avec l'argument dtype
In  : np.array([0, 1.], dtype=int)         #
Out : array([0, 1])                        # ...une ndarray de int
In  : np.array([0, 1.], dtype=float)       #
Out : array([0., 1.])                      # ...une ndarray de float
In  : np.array([0, 1.], dtype=complex)     #
Out : array([0.+0.j, 1.+0.j])              # ...une ndarray de complex

# Conversion de liste de listes en array 2D
In  : np.array([[0, 1], [2, 3]])
Out : array([[0, 1],
             [2, 3]]

# Mix de types numériques, et aussi list/tuple/range de même dimensions :
In  : np.array([[1, 2.], (1 + 1j, 3.), range(2)])
Out : array([[ 1.+0.j, 2.+0.j],
             [ 1.+1.j, 3.+0.j]
             [ 0.+0.j, 1.+0.j]])           # ...Typage dynamique !
```

Les constructeurs ndarray: quelques exemples

```
# Les séquences numériques :
In  : np.arange(3)                          # Similaire à range...
Out : array([ 0, 1, 2])                     # ...mais retourne une array
In  : np.arange(0, 10, 5, dtype=complex)    # Le type peut être spécifié
Out : array([ 0.+0.j, 5.+0.j])

# Construction de matrices :
In  : np.zeros((2, 3))                      # Crée une array (n,m) de 0
Out : array([[ 0., 0., 0.],
             [ 0., 0., 0.]])
In  : np.zeros((2, 3), dtype=int)           # Idem, mais remplie de l'int 0
Out : array([[ 0, 0, 0],
             [ 0, 0, 0]])
In  : np.eye(4, k=1, dtype=float)           # Matrice diagonale...
Out : array([[ 0., 1., 0., 0.],
             [ 0., 0., 1., 0.],             # ...dont la diagonale est ...
             [ 0., 0., 0., 1.],             # ...décalée de k=1
             [ 0., 0., 0., 0.]])
```

- Numpy propose une grande collection de méthodes pour la construction de ndarray
- Tous les vecteurs et matrices classiques peuvent être construits simplement

Arrays of ones and zeros	
`empty()`	New array of given shape and type without initialization
`empty_like(a)`	New empty array with same shape as the array *a*
`zeros()`	New array of given shape and size filled with 0
`zeros_like(a)`	New zeros array with same shape as the array *a*
`ones()`	New array of given shape and size filled with 1
`ones_like(a)`	New ones array with same shape as the array *a*
`full()`	New array of given shape and size filled *val*
`full_like(a)`	New full array with same shape as the array *a*
`eye()`	Array with ones on a diagonal and zeros elsewhere
`identity()`	Identity array (special case of `eye()`)
Arrays from existing data	
`array()`	Create a new array
`copy(a)`	Return an array copy of *a*
Arrays from numerical ranges	
`arange()`	Evenly spaced values over a given interval
`linspace()`	Evenly spaced values over a given interval
`logspace()`	Evenly spaced values over a given interval on log scale
`meshgrid()`	Create array from coordinate vectors
Building classical matrices	
`diag()`	Extract a diagonal or construct a diagonal array
`tri()`	Tridiagonal array
`tril()`	Lower triangular array
`triu()`	Upper triangular array
`vander()`	Vandermonde array

Illustration : Méthodes et attributs

```
# Manipulation de forme...
In  : c = np.arange(12)
In  : print(c)
Out : array([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11])
In  : c = c.reshape(3, 4)
In  : print(c)
Out : array([[ 0, 1, 2, 3],
             [ 4, 5, 6, 7],       # Peut s'écrire simplement en une ligne :
             [ 8, 9, 10, 11]])    # c = np.arange(12).reshape(3, 4)

# Évitez les fonctions stand-alone...
In  : %timeit np.mean(c)
Out : 7.01 µs per loop
In  : %timeit c.mean()
Out : 5.92 µs per loop            # ...préférez les méthodes héritées

# Les méthodes de calcul opèrent sur toutes des dimensions par défaut...
In  : c.mean()
Out : 5.5
In  : c.mean(axis=0)              # Elles peuvent également ...
Out : array([ 4., 5., 6., 7.])
In  : c.mean(axis=1)              # ...s'appliquer sur un axe particulier
Out : array([ 1.5, 5.5, 9.5])
```

Some Numpy array attributes	
ndarray.flags	Information about the memory layout of the array
ndarray.shape	Tuple of array dimensions
ndarray.strides	Tuple of bytes to step in each dimension when traversing an array
ndarray.ndim	Number of array dimensions
ndarray.size	Number of elements in the array
ndarray.itemsize	Length of one array element in bytes
ndarray.dtype	Data-type of the array's elements
ndarray.T	Transposed array
ndarray.real	Real part of the array
ndarray.imag	Imaginary part of the array

Some of the Numpy array methods	
ndarray.copy()	Return a copy of the array
ndarray.reshape()	Return array with a new shape
ndarray.transpose()	Return a view of the array with axes transposed
ndarray.repeat()	Repeat elements of an array
ndarray.sort()	Sort an array, in-place
ndarray.argsort()	Return the indices that would sort this array
ndarray.argmin()	Return indices of the min. values along given axis
ndarray.min()	Return the minimum along given axis
ndarray.conj()	Complex-conjugate all elements
ndarray.sum()	Return the sum over given axis
ndarray.mean()	Return the average along given axis
ndarray.std([])	Return the standard deviation along given axis

Illustration : Indexation et slicing

```
# Indexation des types built in :
In  : a = [[0, 1, 2], [3, 4, 5], [6, 7, 8]]
In  : a[0]                  # Retourne [0, 1, 2]
In  : a[0][0]               # Retourne 0

# Indexation des objets ndarray
In  : b = np.array(a)       # Conversion en ndarray
In  : b[0, 0]               # Retourne 0 !

# Slicing puissant et élégant avec les ndarray :
In  : a[:-1, :-1]
```

Illustration : Indexation et slicing

```
# Indexation des types built in :
In  : a = [[0, 1, 2], [3, 4, 5], [6, 7, 8]]
In  : a[0]                  # Retourne [0, 1, 2]
In  : a[0][0]               # Retourne 0

# Indexation des objets ndarray
In  : b = np.array(a)       # Conversion en ndarray
In  : b[0, 0]               # Retourne 0 !

# Slicing puissant et élégant avec les ndarray :
In  : a[:-1, :-1]           # Lignes de 0 à -1, colonnes de 0 à -1
Out : array([[0, 1],
             [3, 4]])
In  : a[:2, ::-1]
```

Illustration : Indexation et slicing

```
# Indexation des types built in :
In  : a = [[0, 1, 2], [3, 4, 5], [6, 7, 8]]
In  : a[0]                  # Retourne [0, 1, 2]
In  : a[0][0]               # Retourne 0

# Indexation des objets ndarray
In  : b = np.array(a)       # Conversion en ndarray
In  : b[0, 0]               # Retourne 0 !

# Slicing puissant et élégant avec les ndarray :
In  : a[:-1, :-1]           # Lignes de 0 à -1, colonnes de 0 à -1
Out : array([[0, 1],
             [3, 4]])
In  : a[:2, ::-1]           # Lignes de 0 à 2, colonnes inversées
Out : array([[2, 1, 0],
             [5, 4, 3]])
In  : a[:, ::-2]
```

Illustration : Indexation et slicing

```
# Indexation des types built in :
In  : a = [[0, 1, 2], [3, 4, 5], [6, 7, 8]]
In  : a[0]                   # Retourne [0, 1, 2]
In  : a[0][0]                # Retourne 0

# Indexation des objets ndarray
In  : b = np.array(a)        # Conversion en ndarray
In  : b[0, 0]                # Retourne 0 !

# Slicing puissant et élégant avec les ndarray :
In  : a[:-1, :-1]            # Lignes de 0 à -1, colonnes de 0 à -1
Out : array([[0, 1],
             [3, 4]])
In  : a[:2, ::-1]            # Lignes de 0 à 2, colonnes inversées
Out : array([[2, 1, 0],
             [5, 4, 3]])
In  : a[:, ::-2]             # Toutes les lignes, ...
Out : array([[2, 0],         # ...colonnes inversées...
             [5, 3],         # ...1 valeur sur 2
             [8, 6]])
In  : a[[0, -1], :]
```

Illustration : Indexation et slicing

```
# Indexation des types built in :
In  : a = [[0, 1, 2], [3, 4, 5], [6, 7, 8]]
In  : a[0]                  # Retourne [0, 1, 2]
In  : a[0][0]               # Retourne 0

# Indexation des objets ndarray
In  : b = np.array(a)       # Conversion en ndarray
In  : b[0, 0]               # Retourne 0 !

# Slicing puissant et élégant avec les ndarray :
In  : a[:-1, :-1]           # Lignes de 0 à -1, colonnes de 0 à -1
Out : array([[0, 1],
             [3, 4]])
In  : a[:2, ::-1]           # Lignes de 0 à 2, colonnes inversées
Out : array([[2, 1, 0],
             [5, 4, 3]])
In  : a[:, ::-2]            # Toutes les lignes, ...
Out : array([[2, 0],        # ...colonnes inversées...
             [5, 3],        # ...1 valeur sur 2
             [8, 6]])
In  : a[[0, -1], :]         # Ligne 0 et -1, toutes les colonnes
Out : array([[0, 1, 2],
             [6, 7, 8]])
```

Illustration

```
# Deux objets ndarray :
In  : a = np.arange(1, 4)                       # array([1, 2, 3])
In  : b = np.arange(4, 7)                       # array([4, 5, 6])

# Assemblage de ndarray :
In  : c = np.stack((a, b), axis=0)              # array([[1, 2, 3], [4, 5, 6]])
In  : d = np.stack((a, b), axis=1)              # array([[1, 4], [2, 5], [3, 6]])
In  : e = np.concatenate((a, b), axis=0)        # array([1, 2, 3, 4, 5, 6])

# Insertion et suppression :
In  : np.insert(d, 1, [0, 0, 0], axis=1)
Out : array([[1, 0, 4],
             [2, 0, 5],
             [3, 0, 6]])
In  : np.delete(d, -1, axis=0)
Out : array([[1, 4],
             [2, 5]])

# Sélection d'éléments sous condition :
In  : e[e>2]
Out : array([3, 4, 5, 6])
In  : e[np.logical_and(e>1, e<5)]
Out : array([2, 3, 4])
```

Some array manipulation routines	
Method	**Description**
Changing shape	
reshape(a, newshape)	Give a new shape to an array
ravel(a)	Return a contiguous flattened array
Joining arrays	
concatenate((a1, ...)[, axis])	Join a sequence of arrays along an existing axis
stack(tup [, axis])	Stack arrays in sequence along axis)
hstack(tup)	Stack arrays in sequence horizontally (column wise)
vstack(tup)	Stack arrays in sequence vertically (row wise)
Splitting arrays	
split(ary, idx [, axis])	Split *ary* into sub-arrays along axis
hsplit(ary, idx)	Split *ary* into sub-arrays horizontally (column-wise)
vsplit(ary, idx)	Split *ary* into sub-arrays vertically (row-wise)
Tiling arrays	
tile(A, n)	Construct an array by repeating A n times
repeat(a, repeats[, axis])	Repeat elements of an array
Adding and removing elements	
delete(arr, obj[, axis])	Return a new array without elements defined by *obj*
insert(arr, obj, values[, axis])	Insert values at the location defined by *obj*
append(arr, values[, axis])	Append values to the end of an array
Rearranging elements	
roll(a, shift[, axis])	Roll array elements along a given axis
rot90(m[, k, axes])	Rotate an array by 90 degrees

Tous les opérateurs peuvent être utilisés avec les `ndarray` :

- Tous les opérateurs arithmétiques : `+`, `-`, `/`, `%`, `//`, `*`, `**`
- Tous les opérateurs de comparaison : `>(=)`, `<(=)`, `==`, `!=`
- Les opérateurs logiques : `logical_and`, `logical_or`, `logical_not`, `logical_xor`
- Toutes les fonctions mathématiques de Numpy :
  - Fonctions trigonométriques : `cos`, `sin`, `tan`, `arccos`, `arcsin`, `arctan`
  - Fonctions hyperboliques : `cosh`, `sinh`, `tanh`, `arccosh`, `arcsinh`, `arctanh`
  - Exponentiels et logarithmes : `exp`, `log`, `log10`

```
# Deux vecteurs...
In  : a = np.array([[1, 2], [3, 4]])
In  : b = np.array([[5, 6], [7, 8]])

# ...des opérateurs... agissant élément par élément (element-wise)
In  : a*(a + b)                    # ...arithmétiques...
Out : array([[ 6, 16], [30, 48]])
In  : a > b                        # ...de comparaison...
Out : array([[False, False], [False, False]], dtype=bool)
In  : np.logical_and(a>1, a<3)     # ...logiques...
Out : array([[False, True], [False, False]], dtype=bool)
In  : np.log(a)                    # ...mathématiques...
Out : array([[ 0., 0.30103 ], [ 0.47712125, 0.60205999]])
```

$$v_1 = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} \qquad v_2 = \begin{pmatrix} 4 \\ 5 \\ 6 \end{pmatrix}$$

Vector products

```
# Dot product or inner product :  v1^T · v2
In  : v1@v2                                 # Eq.  to np.inner(v1, v2)
Out : 32


# Outer product :  v1 ⊗ v2 = v1 · v2^T
In  : np.outer(v1, v2)                      # Eq.  to v1[:, np.newaxis]*v2
Out : array([[4, 5, 6],                     # or v2*v1[:, np.newaxis]
             [8, 10, 12],
             [12, 15, 18],


# Cross product :  v1 × v2
In  : np.cross(v1, v2)
Out : array([-3, 6, -3])
```

$$v_1 = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} \qquad m_1 = \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix} \qquad m_2 = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

Matrix products

```
In  : m2@m1                  # Eq.  to np.dot(m1, m2)
Out : array([[22, 28],       # or np.matmaul(m1, m2
             [49, 64],
In  : m1@m2
Out : array([[9, 12, 15],
             [19, 26, 33],
             [29, 40, 51],
In  : m2@v1
Out : array([14, 32])
```

- `np.dot` : fonction historique de Numpy pour les produits matrices/vecteurs
- `np.matmul` disponible depuis Numpy 1.10 :
  - similaire à `np.dot` mais multiplication scalaire interdite
  - règles de *broadcasting* différentes pour N>2 : voir doc pour plus de détails !
- `np.matmul` implémente la sémantique de l'opérateur `@` depuis Python 3.5
- L'opérateur `@` peut être utilisé *in-place* avec `@=`
- Voir aussi : `tensordot`, `vdot`, `einsum`, `inner`, `outer`, `kron`, `matrix_power`, `multi_dot`

Numpy linear algebra functions	
Function	**Description**
Matrix and vector products	
`dot/matmul`	Matrix product of two arrays
`linalg.multi_dot`	Dot product of two or more arrays in a single function call
`linalg.matrix_power`	Square matrix to the power (integer) n
`einsum`	Einstein summation convention on the operands
`kron`	Kronecker product of two arrays
Decomposition	
`linalg.choleski`	Choleski decomposition
`linalg.qr`	Compute the qr factorization of a matrix
`linalg.svd`	Singular Value Decomposition
Matrix eigenvalues	
`linalg.eig`	Eigenvalues and right eigenvectors of a square array
`linalg.eigvals`	Eigenvalues of a general matrix
Norms and other numbers	
`linalg.norm`	Matrix or vector norm
`linalg.det`	Compute the determinant of an array
`trace`	Return the sum along diagonals of the array
Solving equations and inverting matrices	
`linalg.solve`	Solution to a linear matrix equation
`linalg.lstsq`	Least-squares solution to a linear matrix equation
`linalg.inv`	Compute the (multiplicative) inverse of a matrix

Method	Description
`loadtxt()/savetxt()`	Read/Save an array to a text file
`load()/save()`	Read/Save an array to a `.npy` binary file
`load()/savez()`	Save several arrays into an `.npz` uncompressed archive
`load()/savez_compressed()`	Save several arrays into an `.npz` compressed archive

Numpy methods for File I/O

Syntaxe : `savetxt/loadtxt` et `save/load`

```
In  : import numpy as np
In  : arr = np.array([[1, 2, 3], [4, 5, 6]])

# Sauvegarde...
In  : np.savetxt("myfile.txt", arr)   # np.save('myfile.npy', arr)

# Chargement...
In  : np.loadtxt("myfile.txt")        # np.load('myfile.npy')
Out : array([[1., 2., 3.],
             [4., 5., 6.]])
```

Note : `Scipy` fournit aussi des fonctions pour gérer les fichiers `mat` de Matlab (Voir `loadmat` et `savemat` du sous module `scipy.io`)

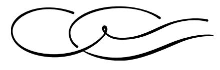
Syntaxe : les archives savez/savez_compressed

```
# Sauvegarde...
In  : arr1 = np.array([1, 2, 3])
In  : arr2 = np.array([4, 5, 6])
In  : np.savez("myfile", a1=arr1, a2=arr2)

# Chargement...
In  : data = np.load("myfile.npz")
In  : data.keys()        # Une archive contient plusieurs objets
Out : [a1, a2]
In  : print(data[a2])    # dont l'accès se fait par clé
Out : array([4, 5, 6])
```

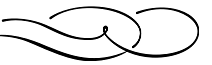
Note sur l'usage du disque et sur les temps de chargement/sauvegarde :

Saving/loading time for an array of 10^6 random elements

Method	Execution time	Disk usage
`loadtxt()/savetxt()`	900ms/700ms	24 Mo
`load()/save()`	2ms/25ms	7.7 Mo
`load()/savez()`	45μs/45ms	7.7 Mo
`load()/savez_compressed()`	45μs/325ms	7.2 Mo (7.4 Ko for np.zeros)



Introduction à Matplotlib



matplotlib

Une figure se trace en trois étapes :

- **L'initialisation :**
  - Créé un objet *figure* avec la méthode `figure`
  - Crée éventuellement un objet *axes* avec la méthode `subplots`

Illustration

```
In  : import matplotlib.pyplot as plt

In  : plt.figure()                    # ou :  fig, axes = plt.subplots()
```

Une figure se trace en trois étapes :

- **L'initialisation :**
  - Créé un objet *figure* avec la méthode `figure`
  - Crée éventuellement un objet *axes* avec la méthode `subplots`
- **Le remplissage :**
  - trace des données
  - change les propriétés de la figure, du tracé, des axes...

Illustration

```
In  : import matplotlib.pyplot as plt

In  : plt.figure()                      # ou :  fig, axes = plt.subplots()
In  : plt.plot(x, y, "color", label="label")
In  : plt.xlabel("x")
In  : plt.ylabel("y")
In  : plt.title("titre")
In  : plt.legend()
```

Une figure se trace en trois étapes :

- **L'initialisation :**
  - Créé un objet *figure* avec la méthode `figure`
  - Crée éventuellement un objet *axes* avec la méthode `subplots`
- **Le remplissage :**
  - trace des données
  - change les propriétés de la figure, du tracé, des axes...
- **L'affichage :** affiche l'objet fini à l'écran

Illustration

```
In  : import matplotlib.pyplot as plt

In  : plt.figure()                          # ou :  fig, axes = plt.subplots()
In  : plt.plot(x, y, "color", label="label")
In  : plt.xlabel("x")
In  : plt.ylabel("y")
In  : plt.title("titre")
In  : plt.legend()
In  : plt.show()
```

Pyplot API

```
import numpy as np
import matplotlib.pyplot as plt

x = np.arange(1000)
y = np.random.rand(1000)

plt.figure()
plt.plot(x, y, "k", label="c1")
plt.axis([0, 1000, 0, 1])
plt.xlabel("x")
plt.ylabel("y")
plt.title("My Title")
plt.grid()
plt.legend()
plt.show()
```

Object-oriented API

```
import numpy as np
import matplotlib.pyplot as plt

x = np.arange(1000)
y = np.random.rand(1000)

fig, ax = plt.subplots()
ax.plot(x, y, "k", label="c1")
ax.set_xlim([0, 1000])
ax.set_ylim([0, 1])
ax.set_xlabel("x")
ax.set_ylabel("y")
ax.set_title("My Title")
ax.grid()
ax.legend()
plt.show()
```

- Tracés rapides : API[2] pyplot ⇒ simple d'utilisation
- Tracés avancés : API orientée objet ⇒ contrôle total des propriétés des figures
- La méthode `subplots` est utilisée pour initialiser une figure et ses axes. Ne pas confondre avec la méthode `subplot` utilisé par l'API pyplot !

[2] *Application Programming Interface*

Pyplot API

```
import numpy as np
import matplotlib.pyplot as plt

x = np.arange(1000)
y = np.random.rand(1000)

plt.figure()
plt.subplot(211)
plt.plot(x, y, "k")
plt.xlabel("x")
plt.xlabel("y")
plt.subplot(212)
plt.plot(x, y**2, "k")
plt.xlabel(r"$x^2$")
plt.xlabel("y")
plt.show()
```

Object-oriented API

```
import numpy as np
import matplotlib.pyplot as plt

x = np.arange(1000)
y = np.random.rand(1000)

fig, ax = plt.subplots(2, 1)
ax[0].plot(x, y, "k")
ax[0].set_xlabel("x")
ax[0].set_.xlabel("y")
ax[1].plot(x, y**2, "k")
ax[1].set_xlabel(r"$x^2$")
ax[1].set_xlabel("y")
plt.show()
```

- L'instruction `subplots()` retourne par défaut un objet axe `ax`
- L'instruction `subplots(2, 1)` retourne une `ndarray` 1D d'objets axe (`ax[0]` et `ax[1]`)
- L'instruction `subplots(2, 2)` retournera une `ndarray` 2D d'objets axe (`ax[0, 0]`, `ax[0, 1]`, `ax[1, 0]`, et `ax[1, 1]`)

Un exemple de figure basique

```
import numpy as np
import matplotlib.pyplot as plt

plt.close("all")      # Close all Figs.

x = np.arange(0, 2*np.pi, np.pi/100)
y = np.sin(x)

plt.figure("My First Fig :)")
plt.plot(y, x, "k", linewidth=2, marker="+", label=r"$sin(x)$")
plt.axis([x.min(), x.max(), y.min(), y.max()])
plt.title(r"My great 1$^{st}$", fontsize=20)
plt.xlabel(r"$x$", fontsize=20)
plt.ylabel(r"$\int_0^1 xdx$", fontsize=20)
plt.legend(loc=2)
plt.grid()
plt.show()
```


- Matplotlib peut afficher du texte en utilisant LaTeX
- Préfixe "r" (*raw string*) avant un texte devant être interprété par LaTeX

Syntaxe : `subplot(ligne, colonne, indice)`

```
plt.figure()
plt.subplot(2, 1, 1)
plt.plot(x1, y1)
plt.xlabel(r"$x_1$")
plt.ylabel(r"$y_1$")
...
plt.subplot(2, 1, 2)
plt.plot(x2, y2)
plt.xlabel(r"$x_2$")
plt.ylabel(r"$y_2$")
...
plt.show()
```

1
2

Syntaxe : `subplot(ligne, colonne, indice)`

```
plt.figure()
plt.subplot(2, 1, 1)
plt.plot(x1, y1)
plt.xlabel(r"$x_1$")
plt.ylabel(r"$y_1$")
...
plt.subplot(2, 1, 2)
plt.plot(x2, y2)
plt.xlabel(r"$x_2$")
plt.ylabel(r"$y_2$")
...
plt.show()
```

1
2

```
plt.figure()
plt.subplot(1, 2, 1)
plt.plot(x, y)
...
plt.subplot(1, 2, 2)
plt.plot(x, y)
...
plt.show()
```

1	2

```
plt.figure()
plt.subplot(2, 2, 1)
...
plt.subplot(2, 2, 2)
...
plt.subplot(2, 2, 3)
...
plt.subplot(2, 2, 4)
...
plt.show()
```

1	2
3	4

```
plt.figure()
plt.subplot(2, 2, 1)
...
plt.subplot(2, 2, 2)
...
plt.subplot(2, 2, 3)
...
plt.subplot(2, 2, 4)
...
plt.show()
```

1	2
3	4

```
plt.figure()
plt.subplot(2, 3, 1)
...
plt.subplot(2, 3, 2)
...
plt.subplot(2, 3, 3)
...
plt.subplot(2, 3, 4)
...
plt.subplot(2, 3, 5)
...
plt.subplot(2, 3, 6)
...
plt.show()
```

1	2	3
4	5	6

Illustration : subplot

```
import numpy as np
import matplotlib.pyplot as plt

x = np.arange(0, 2*np.pi, np.pi/100)
z = y.repeat(int(y.size/2)).reshape(y.size, int(y.size/2))

plt.figure()
plt.subplot(121)
plt.imshow(z)
plt.xlabel("x")
plt.ylabel("y")
plt.colorbar()

plt.subplot(122)
plt.plot(z[:,0])
plt.grid()
plt.xlabel("x")
plt.ylabel("y")

plt.tight_layout()   # Ajuste automatiquement les subplots pour qu'ils ...
plt.show()           # ...soit correctement positionnés dans la figure.
```

Méthodes principales pour le tracé 2D	
`subplot()`	Crée des sous figures
`plot()`	Trace des lignes/marqueurs
`loglog()`	Trace des lignes/marqueurs en échelle logarithmique
`semilogx()/semilogy`	Trace des lignes/marqueurs en échelle log. suivant x/y
`polar()`	Trace des lignes/marqueurs en coordonnées polaires
`imshow()`	Trace une image
`scatter()`	Trace des points (marqueurs)
`quiver()`	Trace des vecteurs
`pcolor()`	Tracé en couleur
`pcolormesh()`	Tracé en couleur (plus rapide avec un maillage régulier)
`contour()`	Trace les lignes de contours
`contourf()`	Trace des contours remplis
`text()`	Écrit du texte

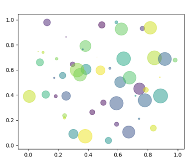
Méthodes principales pour gérer les propriétés de la figure	
`axis()`	Fixe les limites des axes des abscisses et ordonnées
`xlabel()/ylabel()`	Labels des axes des abscisses et ordonnées
`title()`	Titre de la figure
`grid()`	Affiche la grille
`legend()`	Affiche la légende (si les labels des tracés sont définis)
`colorbar()`	Affiche la colorbar (`imshow`, `pcolormesh`, `contour`, `contourf`)
`tight_layout()`	Ajuste les subplots dans la figure
`savefig()`	Sauvegarde la figure

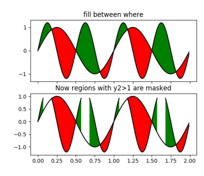
Un exemple de figure avancée

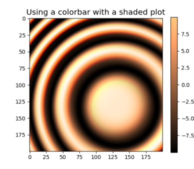
```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

X = np.arange(-3, 3, 0.15)
Y = np.arange(-3, 3, 0.15)
X, Y = np.meshgrid(X, Y)
R = np.cos(X**2 + Y**2) \
    / (1 + X**2 + Y**2)
Z = np.sin(R)
fig = plt.figure(figsize=(8, 6))
ax = Axes3D(fig)
ax.plot_surface(X, Y, Z, \
    rstride=2, cstride=2, alpha=0.75, linewidth=0.25, cmap=plt.cm.jet)
ax.contourf(X, Y, Z, 100, offset=-2, color="k", linewidth=0.25)
ax.set_zlim(-2, 1)
ax.set_xlabel(r"$x$", fontsize=24)
ax.set_ylabel(r"$y$", fontsize=24)
ax.set_zlabel(r"$z$", fontsize=24)
plt.show()
```

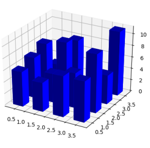
Un autre exemple

```
import numpy as np
import matplotlib.pyplot as plt

x = np.arange(0, 2*np.pi, np.pi/1000)
y = np.sinc(x) + np.random.rand(x.size)/75

fig, ax = plt.subplots(figsize=(8, 6))       # Create a new figure with a default 111 subplot
ax.plot(x, y, label=r"$sinc(x)$")
ax.legend(loc=4)
ax.grid()
ax.set_xlabel(r"$x$", fontsize=20)
ax.set_ylabel(r"$y$", fontsize=20)

from mpl_toolkits.axes_grid1.inset_locator import zoomed_inset_axes
axins = zoomed_inset_axes(ax, 4, loc=1)       # Zoom-factor=4, location=upper-right
axins.plot(x, y)

x1, x2, y1, y2 = 2.1, 2.8, 0., 0.2            # Specify the limits
axins.set_xlim(x1, x2)                        # Set the x-limits
axins.set_ylim(y1, y2)                        # Set the y-limits
plt.yticks(visible=False)                     # Hide the xticks
plt.xticks(visible=False)                     # Hide the yticks

from mpl_toolkits.axes_grid1.inset_locator import mark_inset
mark_inset(ax, axins, loc1=2, loc2=4, fc="none", ec="0.5")
# loc1 and loc2 are locations of the nodes, fc and ec, the colors
plt.tight_layout()
plt.show()
```

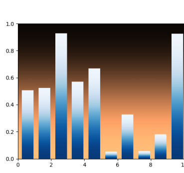


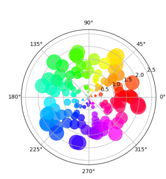


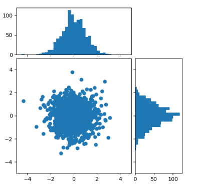


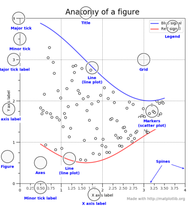


Bibliothèque graphique complète : Tracés, animations, widget, . . .
Allez voir la galerie d'exemple de Matplotlib : **matplotlib.org**

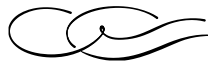




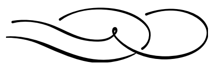




Note : Matplotlib is the brainchild of John Hunter (1968-2012), who, along with its many contributors, have put an immeasurable amount of time and effort into producing a piece of software utilized by thousands of scientists worldwide. If Matplotlib contributes to a project that leads to a scientific publication, please acknowledge this work by citing the project. You can use this ready-made bibteX citation entry.



Brève présentation de Scipy



- Énorme collection de fonctions pour le calcul scientifique
- `SciPy` fournit entre autres[3] :
  - ***Fonctions spéciales*** : Bessel, Hankel, Struve ...
  - ***Traitement du signal*** : convolution, corrélation, filtrage, wavelets ...
  - ***FFTpack*** : FFTs, opérateurs différentiel et pseudo-différentiels
  - ***Algèbre linéaire*** : problèmes aux valeurs propres, décompositions ...
  - ***Optimisation*** : *curve fitting*, recherche de racines, ...
  - ***Matrices creuses*** : opérations basiques et algèbre linéaire
  - ***Intégration*** : fonctions intégratrice et intégrateur pour *ODE*
  - ***Statistiques*** : distributions continues, multidimensionnelles et discrètes, ...
  - ***Clustering*** : théorie de l'information, détection ciblée, compression, ...
  - ***Traitement d'image*** : filtres, interpolation, mesure, morphologie, ...
- `Scipy` fournit des implémentations alternatives de certaines fonctions `Numpy`.
  Lisez les documentations respectives et choisissez selon vos besoins !

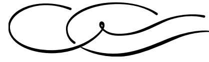
[3] Voir **docs.scipy.org** pour plus d'informations

Subpackage	Description
cluster	Clustering algorithms
constants	Physical and mathematical constants
fftpack	Fast Fourier Transform routines
integrate	Integration and ordinary differential equation solvers
interpolate	Interpolation and smoothing splines
io	Input and Output
linalg	Linear algebra
ndimage	N-dimensional image processing
odr	Orthogonal distance regression
optimize	Optimization and root-finding routines
signal	Signal processing
sparse	Sparse matrices and associated routines
spatial	Spatial data structures and algorithms
special	Special functions
stats	Statistical distributions and functions

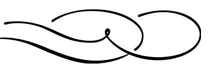
Scipy subpackages

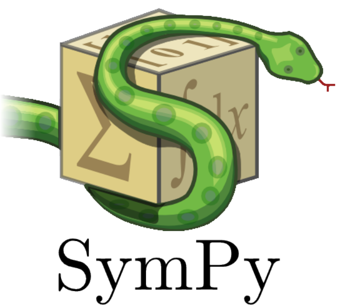
- Organisé en sous modules couvrant différents domaines du calcul scientifique
- Chaque sous module de `Scipy` doit être importé séparément, par exemple :

```
In : from scipy import fftpack                # Importe tout un sous module
In : import scipy.signal as sps               # Raccourci vers sous module
In : from scipy.special import struve         # Importe une fonction seule
```



Courte introduction à Sympy





Sympy pour le calcul symbolique

```
In  : import sympy
In  : x, a, b, c = sympy.symbols("x a b c")

In  : y1 = a*x**2 + b*x + c
In  : sympy.solve(y1, x)
Out : [(-b + sqrt(-4*a*c + b**2))/(2*a), -(b + sqrt(-4*a*c + b**2))/(2*a)]
In  : sympy.solve(y1.subs([(a, 1), (b, 2), (c, 3)]), x)
Out : [-1 - sqrt(2)*I, -1 + sqrt(2)*I]

In  : y2 = sympy.cos(x)**2
In  : sympy.integrate(y2, x)
Out : x/2 + sin(x)*cos(x)/2
```

- Sympy fournit une large collection d'outils pour :
  - Les opérations symboliques avec des scalaires et des matrices
  - La simplification et le développement d'expressions symboliques
  - L'intégration et la dérivation
  - Le calcul de limites
  - Le développement en séries et les approximations par différences finies
  - La résolution d'équations algébriques et différentielles
- Sympy propose également un affichage mathématique grâce à l'instruction :

```
In  : sympy.init_printing(use_unicode=True)
```