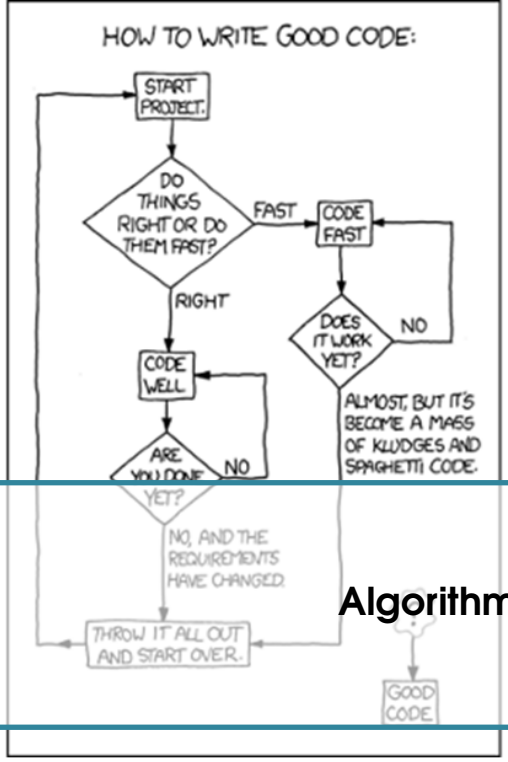
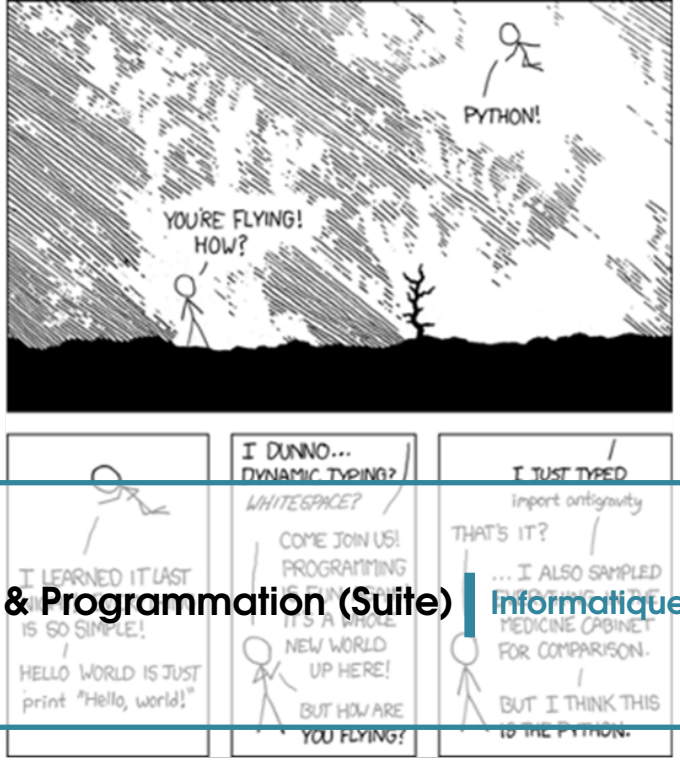




Cours

Chapitre 4
Utilisation de bibliothèques scientifiques

Savoirs et compétences :

Guillaume Haberer
PSI⋆

1 Pour commencer

1.1 Pourquoi un cours sur `numpy` ?

Le calcul scientifique

Il nécessite la manipulation de données en quantité souvent importante. Le module `numpy` offre la possibilité de faire du calcul de façon très efficace, reposant sur trois principes :

- les données sont stockées sous la forme de tableaux `numpy`, de type `ndarray` ;
- on utilise au maximum la manipulation de ces tableaux en évitant d'en faire des copies ;
- on évite si possible le recours aux boucles, préférant l'utilisation de fonctions *vectorialisées*.

Les contraintes liées à cette volonté d'optimisations sont que les tableaux `numpy` sont constitués d'éléments qui sont tous du même type (en général `np.float` ou `np.int64`), et que la taille des tableaux est fixée à la création. On ne peut donc pas augmenter ou diminuer la taille d'un tableau `numpy` comme on le ferait pour une liste.

1.2 Comment charger le module `numpy` ?

On charge traditionnellement l'ensemble du module `numpy` en le renommant éventuellement `np` :

```
import numpy as np
```

2 Les tableaux `numpy`

2.1 Création avec `np.array`

On peut définir un tableau `numpy` à partir d'une liste, en utilisant la fonction `array` du module `numpy`, nommée `np.array` :

```
a = np.array( [1, 2, 3] )
# [1 2 3]
b = np.array( [[1, 2, 3],
               [2, 3, 4],
               [0, 1, 0]])
# [[1 2 3]
#  [2 3 4]
#  [0 1 0]]
```

Mais ce ne sont pas des listes :

```
print(type(a))
# <class 'numpy.ndarray'>
print(a.dtype)
# int64
```

L'attribut `dtype` indique le type commun à tous les éléments du tableau – ici des entiers codés sur 64 bits.

2.2 Création avec des fonctions spéciales

2.2.1 Vecteurs spéciaux (tableaux uni-dimensionnels)

On connaît les fonctions `np.arange` et `np.linspace` :

```
a = np.arange(0, 10, 2) # start, stop exclu, step
print(a)
# [0 2 4 6 8]
```

```
b = np.linspace(0, 10, 6) # start, stop inclus, nb
print(b)
# [0. 2. 4. 6. 8. 10.]
```

Ces fonctions sont en particulier utilisées pour le tracé des fonctions.

2.2.2 Matrices spéciales (tableaux bi-dimensionnels)

Découvrons les trois fonctions spéciales : `np.ones`, `np.zeros`, `np.eye`, ainsi que la fonction `np.diag` :

```
a = np.ones((3, 5)) # Matrice de 1 3x5
# [[ 1.  1.  1.  1.  1.]
#  [ 1.  1.  1.  1.  1.]
#  [ 1.  1.  1.  1.  1.]]
b = np.ones((3, 5), np.int)
# [[1 1 1 1 1]
#  [1 1 1 1 1]
#  [1 1 1 1 1]]
c = np.zeros((3, 5)) # Matrice de 0 3x5
# [[ 0.  0.  0.  0.  0.]
#  [ 0.  0.  0.  0.  0.]
#  [ 0.  0.  0.  0.  0.]]
d = np.zeros((3, 5), dtype=np.bool)
```

```
# [[False False False False False]
#  [False False False False False]
#  [False False False False False]]
e = np.eye(3) # Matrice diagonale de 1 de taille 3x3
# [[ 1.  0.  0.]
#  [ 0.  1.  0.]
#  [ 0.  0.  1.]]
f = np.diag([1,2,3]) # Création d'une matrice diagonale 3x3
#  définie par ... la diagonale
# [[1 0 0]
#  [0 2 0]
#  [0 0 3]]
```

2.2.3 Tableaux aléatoires

```
a = np.random.randint(0, 10, size=(2, 5)) # tirage uniforme d'entiers dans [0, 10[
print(a)
# [[8 2 6 2 7]
#  [4 3 5 5 5]]
b = np.random.binomial(10, .3, (2, 5)) # simule (2, 5) valeurs d'une vad suivant une binomiale(10,.3)
print(b)
# [[1 1 2 3 3]
#  [1 2 4 2 1]]
c = np.random.geometric(.3, 5) # simule 5 valeurs d'une vad suivant une géométrique(.3)
print(c)
# [1 1 1 3 3]
d = np.random.poisson(4.3, 5) # simule 5 valeurs d'une vad suivant une poisson(4.3)
print(d)
# [2 6 6 3 3]
```

Comment simuler dix lancers d'un dé équilibré à 6 faces ?

2.2.4 Lecture des données dans un fichier

Si des données sont stockées dans un fichier texte avec délimiteur (format CSV), on peut construire la matrice de ces données avec la fonction `np.genfromtxt`.

Supposons que le fichier `data.csv` contienne les données suivantes :

```
1,2,3,4,5
6,0,0,7,8
0,0,9,10,11
```

On peut alors définir une matrice de la façon suivante :

```
m = np.genfromtxt("./data.csv", delimiter = ',')
print (m)
# [[ 1.  2.  3.  4.  5.]
#  [ 6.  0.  0.  7.  8.]
#  [ 0.  0.  9. 10. 11.]]
```

2.3 Attributs d'un tableau

Les tableaux `numpy` sont des `objets`, qui possèdent certains `attributs` (des propriétés de ces objets). Modifier ces attributs est une action « en place », sans copie du tableau.

- `dtype` indique le type commun de ses éléments ;
- `size` indique le nombre d'éléments du tableau ;
- `shape` donne le tuple de son format (nombre le lignes, de colonnes...) ;
- `ndim` le nombre d'indices nécessaires au parcours du tableau c'est-à-dire le nombre d'éléments du tuple indiquant son format.

Python

```
a = np.array( [1, 2, 3] )
print(a.size)
# 3
print(a.shape)
# (3,)
print(a.ndim)
# 1
```

```
b = np.array( [[1, 2, 3],
               [2, 3, 4]])
print(b.size)
# 6
print(b.shape)
# (2, 3)
print(b.ndim)
# 2
```

Il est remarquable que l'attribut `shape` soit mutable. On peut aussi utiliser la méthode `reshape`.

Python

```
b.shape = (3,2) # ou alors : b = b.reshape((3,2))
print(b)
# [[1 2]
#  [3 2]
#  [3 4]]
```

```
b.shape = (6, ) # ou alors : b.shape = (-1, )
print(b)
# [1 2 3 2 3 4]
```

En fait, les éléments d'un tableau `numpy` sont stockés consécutivement dans la mémoire, indépendamment du format. Ce n'est que lorsque c'est nécessaire (par exemple pour l'affichage) que l'attribut `shape` est utilisé. Modifier cet attribut est parfaitement négligeable en terme de complexité.

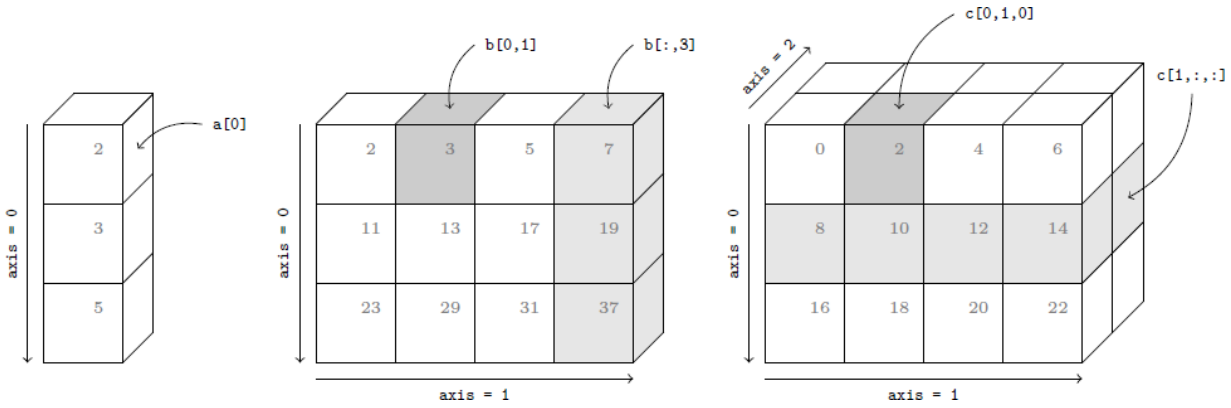
L'attribut `dtype` n'est pas modifiable, puisque le stockage d'un entier n'utilise pas le même nombre de bits que le stockage d'un flottant ou d'un complexe. Il peut cependant être imposé à la création du tableau `numpy`, pour éviter le casting automatique.

2.4 Lecture dans un tableau

2.4.1 Slicing

Python

```
a = np.array([2, 3, 5])
print(a.ndim)
# 1
print(a.shape)
# (3,)
print(a[0]) # 2
b = np.array(
 [ [2, 3, 5, 7],
 [11, 13, 17, 19],
 [23, 29, 31, 37] ])
print(b.ndim)
# 2
print(b.shape)
# (3, 4)
print(b[0,1])
# 3
print(b[:,3])
# [ 7 19 37]
c = np.array( range(24) )
c.shape = (3, 4, 2)
print (c)
# [[[ 0 1]
#   [ 2  3]
#   [ 4  5]
#   [ 6  7]]
#  [[ 8  9]
#   [10 11]
#   [12 13]
#   [14 15]]
#  [[16 17]
#   [18 19]
#   [20 21]
#   [22 23]]]
print(c[1,:,:])
# [[ 8 9]
#  [10 11]
#  [12 13]
#  [14 15]]
```



2.4.2 Les masques

Si `a` est un tableau `numpy`, et `b` un tableau de booléens de même format, alors `a[b]` met en relation un à un les éléments de `a` et ceux de `b`, en ne conservant que les éléments de `a` associés à la valeur `True` :

```
a = np.array(range(10))
a.shape = (2, 5)
print(a)
b = np.array([[ True, False, False, True, False],
              [False, True, False, False, True]])
print(a[b])
# [0 3 6 9]
```

On verra au § 3.1 qu'il est aisé de construire le tableau des booléens traduisant par exemple la propriété : l'élément est divisible par 3.

```
b = a % 3 == 0
print(b)
[[ True False False True False]
 [False True False False True]]
```

Bref, pour obtenir tous les éléments de `a` qui sont divisibles par 3, on écrira donc :

```
a [ a % 3 == 0 ]
```

2.5 Écriture dans un tableau

Les tableaux `numpy` sont mutables, on peut donc modifier les valeurs qu'ils contiennent sans redéfinir l'objet lui-même. Et on peut modifier simultanément plusieurs valeurs, en utilisant les mêmes syntaxes qu'au paragraphe précédent.

Python

```
a = np.array(range(10))
a.shape = (2, 5)
print(a)
# [[0 1 2 3 4]
#  [5 6 7 8 9]]
a[0,0] = 7
print(a)
# [[7 1 2 3 4]
#  [5 6 7 8 9]]
a[0,:] = [8, 7, 6, 5, 4]
print(a)
# [[8 7 6 5 4]
#  [5 6 7 8 9]]
a[ a % 2 == 0 ] /= 2
print(a)
# [[4 7 3 5 2]
#  [5 3 7 4 9]]
```

Dans ce dernier exemple, les éléments pairs de `a` sont divisés par 2.

2.6 À propos des copies

Lorsque l'on manipule de grandes quantités de données, c'est souvent une mauvaise idée de vouloir copier ces données. C'est pourquoi la plupart des manipulations de tableau `numpy` se fait au travers de *vues* d'un même tableau.

```
a = np.array(range(10))
a.shape = (2, 5)
b = a
print(id(a), id(b))
# 4346543408 4346543408
```

Ici, comme on pouvait s'y attendre, `a` et `b` sont simplement deux noms pour un même objet.

R C'est en général une mauvaise idée, mais si on souhaite malgré tout faire une copie d'un tableau `numpy`, on utilise la méthode `copy()`, ou bien `astype(dtype)` déjà mentionnée.

3 Opérations sur les tableaux `numpy`

3.1 Les opérations se font terme à terme

Les opérations `+`, `*`, `/`, `==`, etc s'appliquent aux tableaux `numpy`, mais *TERME à TERME*. C'est cohérent avec la définition mathématique pour l'addition, mais ça ne l'est plus du tout pour la multiplication, les puissances etc.

Python

```
a = np.random.randint(0, 10, (2, 5))
b = np.random.randint(1, 10, (2, 5))
print(a)
# [[0 9 0 6 6]
#  [8 9 2 0 0]]
print(b)
# [[3 1 3 5 2]
#  [8 7 7 1 6]]
print(a + b)
# [[ 3 10 3 11  8]
#  [16 16 9  1  6]]
print(a * b)
# [[ 0  9  0 30 12]
#  [64 63 14 0  0]]
print(a // b)
# [[0 9 0 1 3]
#  [1 1 0 0 0]]
print(a / b)
# [[ 0.         9.         0.         1.2
3.        ]
#  [ 1.         1.28571429 0.28571429 0.
0.        ]]
print(a ** b)
# [[      0       9       0    7776      36]
#  [16777216 4782969     128       0       0]]
print(a == b)
# [[False False False False False]
#  [True False False False False]]
```

3.2 Les fonctions universelles s'appliquent terme à terme

Les fonctions définies dans le module `numpy` sont *universelles*, ou *vectorialisées*, c'est-à-dire qu'elles acceptent comme argument un tableau `numpy` et renvoient le tableau `numpy` des images. Utiliser des fonctions vectorialisées permet de faire gagner des facteurs énormes en temps de calcul.

```
x = np.linspace(-1, 1, 18)
y = np.arcsin(x)
print(y)
# [-1.57079633 -1.080839 -0.87058477 -0.70372051 -0.55790704 -0.42438971
#  -0.2985322 -0.1773996 -0.05885751 0.05885751 0.1773996 0.2985322
#   0.42438971 0.55790704 0.70372051 0.87058477 1.080839  1.57079633]
```

On peut vectoriser ses propres fonctions :

```
def f(x):
    k, s = 0, 0
    while s <= x:
        k += 1
        s += k
    return k

print(f(9.56))
# 4

x = np.arange(1, 10, 2)
# print(f(x)) # déclenche une erreur
vf = np.vectorize(f)
print(vf(x))
# [2 3 3 4 4]
```

3.3 Quelques méthodes sur les tableaux

On peut utiliser les méthodes (ou les fonctions associées) `a.max()`, `a.min()`, `a.sum()`, `a.prod()`, `a.mean()` (moyenne arithmétique), `a.var()` (variance), `a.std()` (écart-type).

Python

```
a = np.random.randint(0, 10, (2,5))
print(a)
# [[8 5 0 1 3]
#  [2 2 1 5 3]]
print(a.sum()) #ou encore np.sum(a)
# 30
print(a.sum(axis=0))
# [10  7  1  6  6]
print(a.sum(axis=1))
# [17 13]
print(a.mean())
# 3.0
print(a.mean(axis=0))
# [ 5.   3.5  0.5  3.   3. ]
print(a.mean(axis=1))
# [ 3.4 2.6]
print(a.max())
# 8
print(a.max(axis=0))
# [8 5 1 5 3]
print(a.max(axis=1))
# [8 5]
```

3.4 Opérations d'algèbre linéaire

Commençons par le produit matriciel : c'est la fonction `np.dot(a,b)` ou la méthode `a.dot(b)` :

Python

```
a = np.array([[1,2,3],
              [4,5,6],
              [7,8,9]])
b = np.array([[0,1,0],
              [1,0,0],
              [0,0,1]])
print(np.dot(a,b))
# ou alors print(a.dot(b))
# [[2 1 3]
#  [5 4 6]
#  [8 7 9]]
```

- Le produit scalaire canonique, entre deux vecteurs ou deux matrices, se calcule avec `np.vdot`.
- Le produit vectoriel se calcule avec `np.cross`.
- La transposée de la matrice `a` est `a.transpose()` ou plus simplement `a.T`.
- Le sous-module `linalg` fournit les fonctions : `np.linalg.det`, `np.linalg.inv`, `np.linalg.norm`, `np.linalg.solve`, `np.linalg.matrix_power`, `np.linalg.matrix_rank` ; dont les significations sont immédiates. Pour cette dernière, la matrice doit être inversible pour que le résultat soit cohérent.

```
a = np.array([[1,2,3],
              [0,2,2],
              [0,2,3]])
b = np.array([4,9,1])
print(np.linalg.solve(a,b))
# [ 3.   12.5  -8. ]
```

La fonction `np.linalg.eig` fournit un couple de deux matrices, la première étant le spectre de la matrice, donné sous forme d'un vecteur où les valeurs propres multiples apparaissent autant de fois que leurs multiplicités, et d'une matrice carrée dont les colonnes sont des vecteurs engendrant respectivement les espaces propres de la matrice. On utilisera cette fonction dans le chapitre de mathématique concernant la réduction des matrices.

```
a = np.array([[1,2,3],
              [0,1,2],
              [0,0,2]])
sp, p = np.linalg.eig(a)
print(sp)
# [ 1.  1.  2.]
print(p)
# [[ 1.00000000e+00 -1.00000000e+00  9.52579344e-01]
#  [ 0.00000000e+00  1.11022302e-16  2.72165527e-01]
#  [ 0.00000000e+00  0.00000000e+00  1.36082763e-01]]
```

3.5 Manipulation de polynômes

Une classe `Polynomial` est disponible, et permet de manipuler les polynômes, qui sont représentés à l'aide de la liste des coefficients classés par ordre croissant de degré. L'objet créé a un attribut `coef`, des méthodes `degree`, `roots`, `deriv`, `integ` permettant d'accéder respectivement au degré, à des valeurs approchées de racines, à la dérivée formelle, à une primitive formelle. On peut aussi utiliser l'objet pour évaluer la fonction polynomiale associée.

```
from numpy.polynomial import Polynomial

p = Polynomial([1, 0, 1])
# ou alors, plus commode pour travailler formellement :
X = Polynomial([1, 0])
p = X ** 2 + 1
print(p.coef)
# [ 1.  0.  1.]
print(p.degree())
# 2
print(p.roots())
# [ 0.-1.j 0.+1.j]
dp = p.deriv()
print(dp.coef)
# [ 0.  2.]
ip = p.integ()
print(ip.coef)
# [ 0.          1.          0.          0.33333333]
print(p(0))
# 1.0
print(p(0+1j))
# 0j
```

Références

[1] Guillaume Haberer, *Supports de cours de PSI⋆*, Lycée La Martinière Monplaisir, Lyon.