

# Cours POO M1: python

12/09 2023

# Introduction, modalités pratiques

# Description

- Cours de programmation
- 12 séances de cours de 1h30, le Mardi de 10h15 à 11h45,  
12 séances de TPs de 2h15, le Lundi de 15h15 à 17h30

# Description

- Cours de programmation
- 12 séances de cours de 1h30, le Mardi de 10h15 à 11h45, 12 séances de TPs de 2h15, le Lundi de 15h15 à 17h30
- Si vous n'avez pas d'ordinateur chez vous, **signalez-le aux enseignants.**

# Description

- Sur Moodle, se trouveront les supports de cours et corrections des TPs

# Description

- Sur Moodle, se trouveront les supports de cours et corrections des TPs
- Deux enseignants pour deux langages :
  - gilles.scarella@univ-cotedazur.fr (C++),
  - simon.girel@univ-cotedazur.fr (Python)

# Description

- Sur Moodle, se trouveront les supports de cours et corrections des TPs
- Deux enseignants pour deux langages :
  - gilles.scarella@univ-cotedazur.fr (C++),
  - simon.girel@univ-cotedazur.fr (Python)

# Description

- Sur Moodle, se trouveront les supports de cours et corrections des TP
- Deux enseignants pour deux langages :  
    gilles.scarella@univ-cotedazur.fr (C++),  
    simon.girel@univ-cotedazur.fr (Python)
- Objectifs: connaissance de base des langages, notions de Programmation Orientée Objet et Algorithmique



# Organisation C++ et python

- 5 séances de cours sur python, 7 sur C++
- 5 séances de TPs sur python, 7 sur C++
- 1 TP noté en python  
2 TP notés en C++  
Pas d'examen final
- C++ représente 60% de la note et python 40%.

# Plan du cours de python

- Syntaxe de base de python
- Modules *numpy*, *matplotlib*, *scipy*
- Classes en python

# Biblio

- Lien officiel python <https://www.python.org/doc>
- Liens [numpy](#), [scipy](#), [matplotlib](#),
- Liens [sympy](#) et [sagemath](#) seulement si vous êtes intéressés

Avec *python*, faites attention aux numéros de version que vous utilisez!

# Installation et utilisation de python

# Installation

- Voir sur Moodle le fichier *installation\_python.pdf* (dans le menu *python*) pour installer python et ce qui est nécessaire pour le cours, **sur votre machine perso**
- Déjà installé dans les salles de TP du Petit Valrose et les salle infos du bâtiment M.
- **Signalez** rapidement tout problème d'installation sur vos machines.

# Présentation

- python est un langage interprété contenant des phases de compilation.
- On peut l'exécuter via l'interpréteur (on voit immédiatement le résultat d'une commande) ou en compilant (délai)
- python (et ses modules) est un langage qui évolue
- possibilité d'interfacer python avec C, C++, Fortran
- python3 a remplacé python2. N'utilisez que python3!

# On peut utiliser python de différentes façons

- Sur Linux ou sur Mac, une possibilité est d'utiliser *python3* en ligne de commande. La commande *python3* donne accès à l'interpréteur, mais pas de manière interactive.

```
python3
```

On peut aussi exécuter directement un fichier (ou script)

```
python3 mon_fichier.py
```

# On peut utiliser python de différentes façons

- Sur Linux ou sur Mac, une possibilité est d'utiliser *python3* en ligne de commande. La commande *python3* donne accès à l'interpréteur, mais pas de manière interactive.

```
python3
```

On peut aussi exécuter directement un fichier (ou script)

```
python3 mon_fichier.py
```

- Sur Linux ou sur Mac et si *ipython3* est installé, utilisez *ipython3* qui donne accès à l'interpréteur python de manière interactive

```
ipython3
```



# Différences python3 et ipython3

- ipython3 est interactif et plus convivial.
- ipython3 donne accès à l'aide des fonctions (`? <mafonction>`)
- ipython3 donne accès à l'aide et au code des fonctions (`?? <mafonction>`)
- ipython3 permet la complétion automatique (pas le cas de python3)
- ipython3 conserve l'historique des commandes
- Utiliser `%run` avec ipython3 pour exécuter un fichier
- mais ipython3 n'est pas toujours installé

# Exemple d'exécution avec ipython3

```
gscarella@math11 ~> ipython3
```

```
Python 3.6.8 (default, Aug 7 2019, 17:28:10)
```

```
Type 'copyright', 'credits' or 'license' for more information
```

```
IPython 7.5.0 -- An enhanced Interactive Python.
```

```
Type '?' for help.
```

```
In [1]: print('hello')
```

```
hello
```

```
In [2]: %run Hello.py
```

```
Hello avec Hello.py
```

# jupyter

- fournit un éditeur python dans un navigateur (doit d'abord être installé sur la machine)
- possibilité de mélanger texte et code
- On sauvegarde en général un fichier d'extension .ipynb
- sur Moodle, exemples de notebooks python contenant le code montré en cours



## python1

On reprend ici les exemples de morceaux de code donnés dans python1.pdf

### Affectation de variables

```
In [68]: i=3
         a, b, c = 5, -2, 14.1
         c = d = -5
```

```
In [69]: j = 4; c = 5*j
```

### Affichage de variables

```
In [70]: print(d)
         d # s'affiche si dans l'interpréteur
         -5
```

```
Out[70]: -5
```

### Chaînes de caractères

```
In [71]: print('Bonjour')
         print("Hello")
         print("il n'y a personne")

         Bonjour
         Hello
         il n'y a personne
```

# CoCalc

- CoCalc <https://cocalc.com> (Collaborative Calculation in the Cloud)
- Alternative à jupyter sur un site web dédié (moins bonnes performances, mais ne nécessite pas d'installation)

- 
- The image shows a Python IDE with a script on the left and a console on the right. The script defines a function `EulerExplicite` for numerical solution and `SolutionExacte` for the exact solution of a differential equation. The console displays the output of the script, showing the numerical solution and the exact solution for the initial value problem  $y' = -y$  with  $y(0) = 1$ .
- Script Content:**
- ```

1 # -*- coding: utf-8 -*-
2 ***
3 Created on Tue Sep 29 15:13:09 2020
4
5 @author: Simon Girel
6 ***
7
8 #%% Code pour les questions 1.a et 1.b
9 import numpy as np
10 import matplotlib.pyplot as plt
11
12 #1.a
13
14 def EulerExplicite(k,r,t0,tf,y0,N):
15     """ATTENTION : dans python, si a et b sont de type "integer", a/b ren
16     """
17     # Si votre fonction EulerExplicite compile bien mais vous renvoie une solution
18     # Il y a plusieurs facon de gérer ce problème.
19     # mais si vous décidez de définir h comme je le fais ci-dessous, le plus simple
20     t0mp=float(t0) # Ce serait logique de convertir tf et N, pour n'avoir que des float
21     ytmp.zeros(N)
22     ttmp.zeros(N) #ici je crée t itérativement, on pourrait très bien le définir direct
23     y[0]=y0
24     t[0]=t0
25     h=(tf-t0)/(N-1) # pour avoir N point, on subdivise [t0,tf] en (N-1) intervalles de
26     for i in range(0,N-1):
27         y[i+1]=y[i]+h*(-k*y[i]+r)
28         t[i+1]=t[i]+h
29     return [t,y]
30
31
32 #1.b
33 # On peut faire une fonction qui renvoie la valeur de la solution exacte,
34 def SolutionExacte(t):
35     ytmp.exp(-t)
36     return y
37
38
39
40 #ttmp,ytmp=EulerExplicite(1,0,0,1,1,100)
41 sol_exacte=sol_ttmp.exp(-ttmp)

```
- Console Output:**
- ```

Nom      Type      Taille      Valeur
erreur   Array of float64 (100,) [0.00000000e+00 5.08438671e-05 1.00663170e-04 ... 1.865546...
erreurMax Array of float64 (100,) [1.17079441e-01 1.01073345e-03 9.12475624e-04 ... 1.89597...
i         int       1          1
pente    float64    1          0.00000000e+00
sol_exacte Array of float64 (990,) [1.00000000e+00 0.99999902 0.99979804 ... 0.36795375 0.367919...
temps    Array of float64 (990,) [0.00000000e+00 1.00000000e-04 2.01979398e-04 ... 9.997980...

```
- The console also shows the execution of the script, including the calculation of the numerical solution and the exact solution, and the plotting of the results.

# Les bases de python

# Affectation de variables

- L'utilisateur ne déclare pas forcément de type pour les variables mais elles en possèdent un quand même. La commande *type* permet de connaître le type d'une variable.
- Exemples d'affectation de variables

```
k = 4  
c = 5*k
```

- ";" permet de séparer deux instructions sur une même ligne

```
a=5; b=5. ; c =14.1  
print(type(a), type(b), type(c))
```

On obtient:

int, float, float

- Les booléens sont notés True et False.



# Affichage de variables

- Quand on exécute un script (fichier) python, il faut utiliser la commande *print* dans le script pour que l'affichage soit visible.

```
d=3  
print(d)
```

- Depuis la console ou sur jupyter, on peut aussi afficher le contenu d'une variable directement en tapant son nom (ne fonctionne que si aucune autre instruction ne suit le nom de la variable).

```
d=3 ; d # le contenu est affiché
```

```
d=3 ; d ; b=2 # rien n'est affiché
```

# Affichage de variables

- Quand on exécute un script (fichier) python, il faut utiliser la commande *print* dans le script pour que l'affichage soit visible.

```
d=3  
print(d)
```

- Depuis la console ou sur jupyter, on peut aussi afficher le contenu d'une variable directement en tapant son nom (ne fonctionne que si aucune autre instruction ne suit le nom de la variable).

```
d=3 ; d # le contenu est affiché
```

```
d=3 ; d ; b=2 # rien n'est affiché
```

- Noter au passage que tout ce qui se trouve entre *#* et le prochain retour à la ligne est ignoré lors de l'exécution (**commentaires**).

# Opérateurs

- Opérateurs habituels:  $+$ ,  $-$ ,  $*$ ,  $/$ ,  $=$ , ...
- $//$  fournit le quotient de la division euclidienne
- $\%$  fournit le reste de la division euclidienne
- $**$  fournit l'élévation à la puissance
- opérateurs logiques (ils peuvent s'écrire en lettres):  
or :  $|$ , and :  $\&$ , not :  $!=$ , is :  $==$

# Chaînes de caractères

- On peut utiliser des guillemets ou des simples quotes

```
print('Bonjour'); print("Hello")  
print("il n'y a personne")
```

- On peut afficher plusieurs variables à la suite (sans formatage)

```
print("Bonsoir", 12+25)
```

On obtient:

Bonsoir 37

# Chaînes de caractères

- Concaténation simple ou multiple (+,\*). Utiliser `str()` pour une conversion

```
L="le temps"+" ne fait rien à l'affaire")  
print(3*"ect. ")  
print("Bienvenue en M"+str(18-17))
```

- `len` donne la taille de la chaîne de caractères
- On peut utiliser des méthodes pour une chaîne de caractères (*find*, *replace*, *split*, *strip* ...)

```
s = "Bonjour"; s.find("o") # renvoie 1  
s.replace("jou", "soi") # renvoie "Bonsoir"  
s1 = "une longue chaine de caractères"  
L = s1.split()
```

# Affichage formaté

```
"bla {0} bla {1} bla {2} bla".format(var0,var1,var2)
```

On place `{k:format}` là où le  $k$ -ème élément doit être inséré, éventuellement avec un format, la syntaxe `{k}` utilisera le format par défaut.

- Affichages mixtes

```
A=3.1415926535 ; B=384400
print('La valeur de A est {0}, une
      approximation serait {1:.4f}. La distance
      Terre-Lune est de {2:.2e} km'.format(A,A,B)
)
```

La valeur de A est 3.1415926535, une approximation serait 3.1416. La distance Terre-Lune est de 3.84e+05 km

- `:f` → 6 décimales    `:.n f` →  $n$  décimales    `:s` → chaîne de caractères  
`:.ne` → réel en écriture scientifique avec  $n$  décimales

# Les listes en python

# Listes

- Une liste est une suite d'éléments pouvant être de types différents.
- Concaténation avec '+', '\*' (pas d'opération arithmétique sur les listes !)

```
l1 = [1, 4, 5, 1] # liste d'entiers
l2 = ['maths', 'M', 1, '2023'] # mixte
l3= l1 + l2 # concatenation des deux
l4= l1* 3 # équivalent à l1+l1+l1
```

- La commande *list()* crée une liste vide (on peut aussi faire `l=[]`).
- Une liste est mutable (voir plus loin)
- Une liste est ordonnée, peut contenir plusieurs fois la même valeur et peut contenir des éléments de types différents.



# Quelques opérations sur les listes

- Opérations (attention!)

```
print(2*11)
```

```
[1, 4, 5, 1, 4, 5]
```

- Ajout ou suppression d'éléments

```
l3 = [2.1, 4, 2.1, 5.2]
l3.append(24) # ajout d'un élément
l3.remove(2.1) # retire la 1ère occurrence de
               2.1
l3.pop(2) # retire le 2ème élément
l3.sort() # ordonne la liste (si types
           comparables)
```

- Liste de listes

```
l4 = [[5, 6, 12], [7, 3]] # liste de listes
```

# Accès aux éléments d'une liste

La numérotation des indices commence à 0 en python!

Supposons que l est une liste avec suffisamment d'éléments:

```
l[0]          # 1er élément de l
l[:]          # tout le contenu de la liste
l[0:2]        # l[0] et l[1] mais pas l[2] !!!
l[-1]         # dernier élément de l
l[-2]         # avant-dernier élément de l
l[2:-1]       # l[2], l[3] jusqu'à l[-2] inclus
l[2:]         # l[2] jusqu'au dernier élt inclus
l[2:9:3]       # l[2], l[5], l[8] (pas=3)
l[2:8:3]       # l[2], l[5] mais pas l[8] (pas=3)
l[8:2:-2]     # l[8], l[6], l[4] mais pas l[2]
l[2::-3]       # l[2] jusqu'à l[-1] (pas=3)
```

# Objets mutables

Les listes (ainsi que sets et dictionnaires) sont des objets mutables, ils peuvent être modifiés et **toutes les variables pointant vers un objet mutable sont affectées par sa modification.**

```
l1 = [1, 5]
l2 = l1
l1[0]=6
print(l2); print(l1) # l1 et l2 identiques !
```

Ici l1 et l2 font référence au **même** objet

```
id(l1)==id(l2) #identifiant unique d'un objet
>>> True
```

# Objets mutables

Plusieurs syntaxes pour créer une copie, indépendante, d'un objet mutable :

```
l1 = [1, 5];  
l2 = 1*l1;  
l3 = l1.copy()  
l4= l1[:]  
l1.append(6)  
print(l1, l2, l3, l4) # seule l1 est modifiée
```

# Objets mutables

Un objet **non** mutable est donc un objet ne pouvant être modifié.

Exemple : int, float, string,...

```
c="Abracadraabra";  
c.replace("a","o") # renvoie la chaîne "  
    Abrocodrobro" mais ne modifie pas c.  
print(a)
```

Aucune fonction ou méthode ne saurait modifier l'objet référencé par la variable *a*.

# Les tuples

# Tuples

```
t = (5, 8, 9, 'abcd')
```

- Un tuple utilise des parenthèses
- Pour accéder à un élément du tuple, on utilise [], comme pour les listes

```
print(t[1]); print(t[3])
```

- Un tuple à un unique élément est (5,) - ne pas oublier ",," !
- Un tuple n'est pas mutable, il n'existe donc pas d'opérations comme *append*, *pop*, *remove*, *sort*...

# Tuples

- `tuple()` crée un tuple vide
- La fonction `tuple(.)` transforme son argument en tuple.

```
t1 = tuple([4, 'eeee', 9, 10])
```

- On peut seulement concaténer des tuples entre eux
- La méthode `count` permet de compter les occurrences d'une valeur

```
t1.count(10)    #renvoie 1
```

- opérations de concaténation sur les tuples: '+', '\*'

```
t1 = t1 + (67,); t1 = 4*t1
```

remarque : ici le tuple initialement défini n'est pas "modifié", on réaffecte la variable `t1` à un nouveau tuple.



# Fonctions len, sum

On peut utiliser *len* et *sum* sur tuples et listes pour obtenir la longueur et la somme des éléments.

```
t1 = tuple([4, 'eeee', 9, 10])
len(t1) # 4
sum(t1) #donne une erreur car non cohérence sur
        les types!
t2 = tuple([12, -14])
sum(t2) # retourne -2
l1 = [4, -2, -8] # l1 est une liste
len(l1) # 3
sum(l1) # donne -6
```

# Les sets

# Sets

- Un set est un ensemble d'éléments non ordonnés et uniques

```
s1 = {4, 3, 1, 2, 2, 1, "terre" , "mer"}  
print(s1) # renvoie {1, 2, 3, 4, 'mer', 'terre'  
          '}'  
s2 = {4, 2, 1, 3}  
s1 == s2 # True
```

- Contient les méthodes *add* (ajout d'un élément), *pop* (suppression du dernier élément), *remove*, *union*, *intersect*, ...
- Un set est un objet mutable (comme une liste)
- *set()* crée un set vide

# Les dictionnaires

# Dictionnaires

- Un dictionnaire associe une valeur à une clé. Par exemple

```
dpt = { 'Isere': 38, 'Drome': 26, 'Bouches-du-  
Rhône': 13, 'Finistere': 29 }
```

- On peut accéder à une valeur en utilisant sa clé. Par ex:

```
dpt [ 'Drome' ]
```

- La commande *dict()* crée un dictionnaire vide
- méthodes *keys()*, *values()*

```
dpt.keys()  
dpt.values()
```

# Autres commandes de base

# Suite de nombres (*range*)

- *range* renvoie une liste de nombres d'une manière semblable à une liste python

```
a = range(10)    # a contient 0, 1, ..., 9  
                 # /\ 10 n'y figure pas!
```

- On peut définir un pas.

```
b = range(2, 17, 3) #b contient 2,5,8,11,14  
b = range(2, 18, 3) #b contient 2,5,8,11,14,17
```

- *range* ne convient que pour des entiers!

# Indentation

En python, les blocs de codes sont définis par leur **indentation**.

- Deux points (:) et une indentation démarrent un bloc
- La désindentation le termine.
- Pas d'accolades, de crochets ou de mots clés spécifiques (begin, end,...).
- l'indentation se fait **automatiquement** lorsque l'on revient à la ligne après ":"



# Boucle for

- Un exemple:

```
for i in range(4):  
    print(i)  
print("et voilà !")
```

- Ne pas oublier l'indentation
- Mettre fin à l'indentation fait sortir de la boucle
- Autre exemple - boucle sur les éléments d'une liste

```
l = ['ABCD', 'fg', 'WXY']  
for mot in l:  
    print(mot) # affiche 'ABCD', 'fg', 'WXY'
```

# Boucle while

Attention ! La boucle *while* doit se terminer !

```
i= 0
while i<10:
    print(i)
    i = i + 1
```

# Condition avec if

- if ... else

```
i = 0
if (i==0):
    print('zero')
else:
    print('Non nul')
```

- if ... elif ... else (elif pour "else, if")

```
i = 0
if (i==0):
    print('zero')
elif (i==2):
    print('deux')
elif (i==3):
    print('trois')
else:
    print('Ni 0, ni 2, ni 3')
```

# Fonctions en python

- Le mot clé est *def*
- La fonction retourne le(s) objet(s) qui suivent l'instruction *return*
- Exemple: on veut définir la fonction  $g$

$$\begin{aligned} g : \mathbb{R} &\rightarrow \mathbb{R} \\ t &\mapsto t^2 + 1 \end{aligned}$$

Le code python est:

```
def g(t):  
    return t**2 + 1
```

# Fonctions mathématiques - lambda fonctions et arguments par défaut

- *lambda* permet une définition courte de fonctions.  
Par exemple, en une seule ligne

```
f = lambda x: x*x + 1  
print(f(13))
```

- Exemple d'une fonction avec un argument par défaut

```
def g(t=5):  
    return t**2 + 1  
print(g())      # renvoie 26  
print(g(2))     # renvoie 5
```

# Modules

- Un module python contient un ensemble de fonctions (et de variables) répondant à un objectif précis (par exemple: fonctions systèmes (informatique), fonctions mathématiques, manipulation de fichiers, ...)
- Importer un module (généralement au début du script)

```
import nom_module
```

- Exemples de modules: sys, os, math, numpy, scipy, matplotlib, sympy, pickle, random, ...

# Le module math

Il contient

- des fonctions mathématiques *math.sqrt*, *math.exp*, *math.sin*, *math.cos*, *math.log*, ...
- des constantes mathématiques: *math.pi*, ...
- Pour l'importer:

```
import math
```

La commande *import* permet d'importer le contenu d'une bibliothèque (module).

Sans *import math*, l'utilisateur ne peut pas utiliser les fonctions de ce module !

```
import math
print(math.log(17))
```

Importer un module et l'utiliser via un raccourci

```
import math as m
deuxpi = 2 * m.pi
```

Toutes les fonctions seront accessibles par *m.<fonction>*

On peut choisir de n'importer que les fonctions nécessaires

```
from math import sqrt, pi # seuls sqrt, pi
print(sqrt(5*pi))
print(log(16)) # !erreur
```

Pour tout importer du module *math*

```
from math import *
```



# numpy (début)

# numpy

*numpy* est un module pour le calcul scientifique.

- il contient des constantes et des fonctions mathématiques comme dans le module *math*: `numpy.pi`, `numpy.e`, `numpy.sqrt`, `numpy.exp`, `numpy.sin`, `numpy.cos`, `numpy.log`, ...
- par exemple

```
from numpy import sqrt, pi, sin
print([sqrt(73), sin(7*pi/2.)])
```

# tableaux (1D) avec numpy.array

Le principal intérêt de *numpy* est la manipulation des tableaux multidimensionnels grâce au type *numpy array*.

```
import numpy as np
t = np.array([np.pi/4, np.pi/2, np.pi])
```

- On peut appliquer les fonctions mathématiques de numpy directement sur un array ! (ne fonctionne pas avec le package *math*)

```
print np.sin(t)
>>> [7.07106781e-01  1.0    1.22464680e-16]
```

- La somme de deux arrays et la multiplication par un scalaire fonctionne comme pour les matrices en algèbre.

```
print(3*t) ; print(t+np.array([1,2,3]))
```

**remarque** : fonctionne également sur scalaires et arrays 2D, 3D...

# Vecteurs (1D) avec numpy.array

- L'accès à un élément d'un tableau numpy fonctionne comme pour les listes

```
t[0]    # 1er element de t  
t[[1,2,0]] # array contenant t[1], t[2], t[0]  
t[0:2]  # array contenant t[0] et t[1]  
t[-1]   # dernier element de t  
t[1:]   # array des t[i] t à partir de i=1
```

- Tous les éléments d'un array doivent être du même type ! Si on essaie de mélanger les types, python convertit automatiquement tous les éléments vers un même type → source d'erreur.
- Un tableau numpy est mutable et possède des attributs comme shape, size et des méthodes (sum, sort, ...)

# numpy.arange

- Tableau de type *numpy.array* contenant une suite de nombres avec un pas constant
- *numpy.arange(b)* contient tous les entiers de 0 à (b-1) par pas de 1
- Attention, la dernière valeur est exclue du tableau!

```
import numpy as np  
t = np.arange(7) # tableau de 0 à 6
```

- *numpy.arange(a,b)* contient tous les entiers de a à (b-1) par pas de 1

```
t = np.arange(3, 7) # t contient 3, 4, 5, 6
```

- *numpy.arange(a, b, h)* contient tous les nombres  $< b$  à partir de a, par pas de h

```
t = np.arange(3, 9, 2) # t contient 3, 5, 7  
t = np.arange(0, 1.25, 0.25) # renvoie [0,  
    0.25, 0.5, 0.75, 1]
```

# numpy.linspace

- *numpy.linspace* ressemble à *numpy.arange*.
- rappel : *numpy.arange(a, b, h)* contient tous les nombres  $< b$  à partir de  $a$ , **par pas de  $h$** .
- *numpy.linspace(a, b, m)* renvoie un tableau numpy qui subdivise l'intervalle  $[a, b]$  **en  $m$  points**.
- La borne supérieure est incluse dans le résultat (à la différence de *numpy.arange*)

```
import numpy as np
t = np.linspace(0, 1, 5) # renvoie [0, 0.25, 0.5,
                        0.75, 1]
```

# Matrices (2D) avec numpy.array

- Définition d'une matrice ligne par ligne avec *numpy.array*

```
import numpy as np
a = np.array([[1,2,3],[4,5,6],[7,8,9]]) #
      matrice 3x3
print(a)
```

- Accès aux éléments

$a[i,j]$  renvoie l'élément de la  $i$ ème ligne,  $j$ ème colonne (numérotation commence à 0!)

```
print(a[0,1]) # renvoie 2 ici
print(a[:,2]) # renvoie la colonne 2
print(a[2,:]) # renvoie la ligne 2
print(a[[0,2],:]) # renvoie les lignes 0 et 2
```

- $a.T$  renvoie la transposée de  $a$

# Matrices (2D) avec numpy.array

- **Attention** Le produit de deux *numpy.array* avec `*` correspond au produit terme à terme !!!
- Utiliser *np.dot* (ou encore `@` pour versions récentes de python) pour le produit matriciel

```
a = np.array([[1,2],[3,4]])  
print(np.dot(a,a))
```



# Somme des valeurs sur un tableau numpy

- `numpy.sum(t)` → somme des éléments de `t`, sur toutes les dimensions (lignes et colonnes)
- `numpy.sum(t,0)` → somme des éléments de `t`, somme sur les lignes
- `numpy.sum(t,1)` → somme des éléments de `t`, somme sur les colonnes

```
import numpy as np
t = np.array([[1, 2, 3],[4, 5, 6]])
np.sum(t)      # donne 21,
np.sum(t,0)    # donne [5, 7, 9],
np.sum(t,1)    # donne [6, 15],
```

# Initialisation d'un tableau numpy

- `numpy.zeros(n)` → vecteur de taille  $n$  ne contenant que des 0
- `numpy.zeros(( $n_1$ ,  $n_2$ ))` → matrice de dimension  $(n_1, n_2)$  ne contenant que des 0
- `numpy.ones(n)` → vecteur de taille  $n$  ne contenant que des 1
- `numpy.ones(( $n_1$ ,  $n_2$ ))` → matrice de dimension  $(n_1, n_2)$  ne contenant que des 1
- `numpy.eye(n,)` → matrice identité de taille  $n$
- `numpy.eye(n,k=c)` → matrice carrée de taille  $n$  avec des 1 sur la  $c$ -ième diagonale et 0 ailleurs.
- `numpy.diag(v,k=c)` → matrice carrée dont la  $c$ -ième diagonale contient les éléments du array  $V$  et 0 ailleurs.

# Dimensions d'un tableau numpy

- `numpy.shape` → donne la dimension d'un *numpy.array* dans un tuple
- `numpy.size` → donne le nombre d'éléments du tableau

```
import numpy as np
t = np.array([[1, 2, 3],[4, 5, 6]])
t.shape # ou np.shape(t)
t.size  # ou np.size(t)
np.size(t, 0) # nb d'elts dans la dimension 0
np.size(t, 1) # nb d'elts dans la dimension 1
```

# Manipulation des tableaux numpy

*numpy* contient beaucoup de fonctions agissant sur les tableaux

- tri d'un tableau
- suppression des doublons dans un tableau
- ajout d'éléments dans un tableau

```
import numpy as np
a = np.array([5, 7, 6, 5, 6, 2, 1, 13, 11])
at = np.sort(a)
a.sort() # tri de a avec modification de a
au = np.unique(a) # suppression des doublons
b = np.append(au, np.array([-12, -10])) #
      concaténation de tableaux numpy
```

# numpy par rapport aux listes et tuples

- Les fonctions standard de numpy acceptent listes et tuples comme argument (par exemple mean, std, median)

```
import numpy as np
L = [4, 5, 6]
print(np.mean(L)) # ok 5
```

- Des fonctions de numpy acceptent listes ou tuples comme argument mais renvoient un tableau numpy

```
print(np.append([4, 5, 6], [22, 25]))
```

- Dans certains cas, on peut avoir besoin de convertir une liste ou un tuple. Utiliser *numpy.array*

```
tL = np.array(L)
```

# Rappel - objets mutables

Attention, les objets *array*, comme les *list*, sont mutables.

```
t = np.array([1, 2, 3])
v=t
t[0]=7
print(v) # renvoie [7 2 3]
```

Si on veut une copie indépendante d'un array, on a trois syntaxes possibles

```
t = np.array([1, 2, 3])
v=1*t
w= np.copy(t)
y=t.copy()
t[0]=12
print(v,w,y) # affiche trois fois [1 2 3]
```