INTRODUCTION AU MODULE NUMPY

Le langage Python dispose de nombreux modules qui réunissent des fonctions spécifiques propres à des domaines particuliers et programmées de façon «optimisée». Par exemple, on dispose de :

- `SciPy` : fonctions de calcul scientifique (polynômes, optimisation, etc.),
- `math` : fonctions utiles pour les opérations mathématiques (cosinus, sinus, exp, etc.),
- `MatlotLib` : tracé de courbes et autres représentations graphiques,
- `SymPy` : calcul symbolique (formel),
- `random` : manipulation d'objets à valeurs aléatoires,
- `os` : interaction avec le système d'exploitation,
- `time` : fonctions permettant de travailler avec le temps,
- `calendar` : exploitation de calendriers et dates,
- `urllib` et `urllib2` : récupération d'informations sur internet,
- `PyGame` : création de jeux 2D,
- `turtle` : interaction avec une “tortue" graphique,
- `Rpy` : calculs statistiques,
- etc.

Dans ce cours, on s'intéresse essentiellement au module `NumPy`. Sa finalité est de manipuler des objets numériques organisés en tableaux multidimensionnels, reflets des objets algébriques tels que les vecteurs, les matrices, etc.

Pour importer l'intégralité des fonctions du module `numpy` dans l'espace de travail du shell en cours, il suffit d'exécuter la commande suivante :

```
import numpy
```

Cela permet alors d'utiliser l'intégralité des fonctions du module en les faisant toutefois précéder du préfixe `numpy` séparé par un point :

```
a = numpy.cos(numpy.pi/4)
```

Pour alléger l'écriture du préfixe, on peut créer un alias lors de l'importation (usuellement `np`) :

```
import numpy as np
a = np.cos(np.pi/4)
```

Il se peut que l'on ait besoin de n'utiliser qu'un nombre restreint de fonctions du module. Dans ce cas, on peut les charger *individuellement* dans l'interpréteur par la commande :

```
from numpy import cos, sin, pi, ...
```

Elles sont alors accessibles *directement*, sans que l'on n'ait besoin de les faire précéder du nom du module. Par contre, s'il existait au préalable des fonctions du même nom dans l'espace de travail, celles-ci seraient écrasées par les fonctions importées. Il existe aussi la commande :

```
from numpy import *
```

qui importe toutes les fonctions du module "individuellement". L'intérêt est qu'il devient alors possible de se passer du préfixe du nom du module pour l'appel des fonctions.

```
from numpy import *
a = cos(pi/4)
```

Néanmoins, *il n'est pas recommandé d'utiliser cette commande* car elle écrase peut-être des objets de base. Ainsi, une personne extérieure qui lirait un programme utilisant les fonctions du module ne verrait pas clairement qu'il s'agit d'objets importés ; d'où risque de confusion. On privilégiera l'importation avec un alias.

I - Tableau

I.1 - Définition

Parmi ses nombreuses fonctionnalités, le module Numpy apporte un nouveau type d'objet qu'on appelle *tableau* (ou *array* en anglais) et dont la syntaxe de base est la suivante :

```
np.array(liste,dtype = typ)
```

où *liste* est une liste de valeurs et `typ` le type de données vers lequel on veut que soient converties les valeurs dans la liste (par exemple `float` ou `complex`). En effet, contrairement aux listes, les éléments d'un tableau doivent être tous du *même* type.
Le paramètre `dtype` est optionnel : par défaut, en cas d'hétérogénéité de type, toutes les données de la liste seront automatiquement converties vers le type le plus fort (des flottants par exemple si la liste contient des flottants et des entiers).

```
>>> a = np.array([1,2,3])
>>> a
array([1, 2, 3])
>>> b = np.array([1,2.5,'a'])
>>> b
array(['1','2.5','a'])
>>> c = np.array([1,2.5,1+2j])
>>> c
array([ 1.0+0.j,  2.5+0.j,  1.0+2.j])
>>> type(c[0])
numpy.complex128
```

L'implémentation en mémoire des tableaux est optimisée par rapport à celle des listes. Cela permet d'accéder et/ou de modifier plus rapidement les valeurs d'un élément, ce qui se révèle essentiel dans les méthodes numériques de calcul où les tableaux contiennent souvent des dizaines de milliers de valeurs.
La désignation des éléments d'un tableau est identique à celle des listes. Comme ces dernières, les tableaux sont des objets *mutables* : on peut modifier un élément dans un tableau sans toucher aux autres.

```
>>> a = np.array([1,2,3,4,5])
>>> a[0]
1
>>> a[1] = 7
>>> a
array([1,7,3,4,5])
>>> a = np.array([1,2,3,4,5])
>>> a[2] = 8.5
>>> a
np.array([1,2,8,4,5])
>>> a[3] = 'pcsi'
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-28-5c6c611e4369> in <module>()
----> 1 a[3] = 'pcsi'
ValueError: invalid literal for int() with base 10: 'pcsi'
```

```
>>> a[3] = 1 + 2j
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-32-29cc5a54d2d3> in <module>()
----> 1 a[3] = 1 + 2j
TypeError: can't convert complex to int
```

I.2 - Création

Pour créer un tableau, on peut utiliser une syntaxe similaire à celle des listes.

```
>>> a = np.array([k for k in range(10)])
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

On peut d'ailleurs transformer une liste en tableau et inversement.

```
>>> a = np.array([k for k in range(10)])
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> b = list(a)
>>> b
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> c = np.array(b)
>>> c
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Voici quelques instructions permettant de créer des tableaux particuliers :

- `np.zeros(n,dtype=typ)` engendre un tableau de n zéros dont le type est `typ` (le type par défaut est `float64`);
- `np.arange(start,stop,step)` est analogue à `range` mais avec la possibilité d'utiliser des arguments de type `float` (la valeur `stop` n'est pas comprise);
- `np.linspace(start,stop,num,endpoint=bool)` engendre un tableau de `num` valeurs espacées régulièrement (par défaut 50 valeurs) allant de `start` à `stop` (la valeur `stop` étant comprisee si `endpoint` est `True` ce qui est le cas par défaut).

```
>>> np.zeros(10,dtype=int)
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
>>> np.arange(1,2,0.1)
array([1. , 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9])
>>> np.linspace(1,2,10,endpoint=False)
array([1. , 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9])
>>> np.linspace(1 , 2 , 10)
array([ 1. , 1.11111111, 1.22222222, 1.33333333, 1.44444444, 1.55555556, 1.66666667,
    1.77777778, 1.88888889, 2.  ] )
>>> np.linspace(1,2,11)
array ([1. , 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 2. ])
```

I.3 - Opérations

L'une des caractéristiques des tableaux – et cela constitue une différence essentielle par rapport aux listes – est leur comportement "classique" (*i.e.* case par case) par rapport aux opérateurs élémentaires (+,-,*,**,/), sous réserve de compatibilité au niveau des dimensions :

```
>>> a = np.array([0, 1, 2, 2, 3, 5])
>>> b = np.array([1, 2, 0, 2, 5, 3])
>>> a+b
array([1, 3, 2, 4, 8, 8])
>>> a*b
array([0,  2,  0,  4, 15, 15])
>>> a**2
array([0, 1, 4, 4, 9, 25])
>>> 1/(a+1)
array([1. , 0.5, 0.33333333, 0.33333333, 0.25, 0.16666667])
```

Par ailleurs, les fonctions définies par `numpy` sont vectorialisées c'est-à-dire qu'elles peuvent s'appliquer directement sur un tableau : l'application d'une fonction `f` à un tableau `T` produit le tableau `f(T)` constitué des images de chacun des éléments de `T` par `f`.

```
>>> a = np.array([0, 1, 2, 2, 3, 5])
>>> np.floor(np.exp(a))
array([1.  ,2.  , 7.  , 7.  , 20.  , 148. ])
```

Exercice 1

Soit la fonction f qui à $x > 0$ associe $\lfloor \ln(x) \rfloor$.
Définir une fonction python `images(n)` donnant le tableau des valeurs des $f(k)$ pour $k \in [\![1, n]\!]$.
Définir une fonction python `imagesbis(n)` donnant le tableau des valeurs des $\frac{f(k)}{k}$ pour $k \in [\![1, n]\!]$.

I.4 - Méthodes et attributs

Les tableaux du module `numpy` possèdent leurs propres attributs et méthodes dont les plus classiques sont :

- ▷ `T.dtype` → type des données du tableau,
- ▷ `T.size` → taille du tableau, c'est-à-dire son nombre de cases (y compris dans le cas de tableaux multidimensionnels),
- ▷ `T.shape` → dimensions du tableau, c'est-à-dire tuple correspondant au nombre de lignes et au nombre de colonnes,
- ▷ `T.sum()` → somme de toutes les valeurs de `T`,
- ▷ `T.mean()` → moyenne de toutes les valeurs de `T`,
- ▷ `T.max()` → maximum de toutes les valeurs de `T` (idem avec `min()`),
- ▷ `T.argmax()` → plus petit indice `i` tel que `T[i]=T.max()` (idem avec `argmin()`),
- ▷ `T.round()` → arrondi de chaque valeur de `T` à la valeur entière la plus proche,
- ▷ `T.sort()` → tri des valeurs de `T` dans l'ordre croissant,
- ▷ `T.copy()` → copie totalement indépendante de `T`.

II - Tableaux multidimensionnels

II.1 - Création

Les tableaux bidimensionnels constituent la matérialisation des *matrices*. Pour créer un tableau bidimensionnel à m lignes et n colonnes, on peut utiliser la syntaxe suivante :

```
tab = np.array([[liste_1],...,[liste_m]])
```

où chacune des listes liste_k comporte n éléments. On rappelle que ces éléments doivent être tous du *même type*.
Par exemple :

```
>>> tab1 = np.array([[1,2,3,4],[5,6,7,8],[9,10,11,12]])
>>> tab1
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
```

On adapte cette idée pour des tableaux de tailles supérieures ; par exemple un tableau tridimensionnel :

```
>>> tab2 = np.array([[[1,2],[3,4]],[[5,6],[7,8]],[[9,10],[11,12]]])
>>> tab2
array([[[ 1,  2],
        [ 3,  4]],

       [[ 5,  6],
        [ 7,  8]],

       [[ 9, 10],
        [11, 12]]])
```

Pour désigner un élément d'un tableau multidimensionnel, on utilise une multi-indexation reprenant la hiérarchie de la définition précédente. Par exemple :

```
>>> tab1[1,2]   # ou : tab1[1][2]
7
>>> tab2[1,0,1] # ou : tab2[1][0][1]
6
```

II.2 - Slicing

La lecture ou l'écriture d'éléments d'un tableau peut se faire par coupes (*slices* en anglais), suivant une ou éventuellement plusieurs des dimensions du tableau. Le format d'une coupe dans un tableau `T` est le suivant :

`T[debut:fin:pas]`

où `debut` désigne le premier indice de position (compris), `fin` le dernier indice de position (non compris) et `pas` la période de coupe.

```
>>> tab3 = np.array([k for k in range(10)])
>>> tab3
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
>>> tab3[0:11:3]
array([0, 3, 6, 9])
>>> tab3[-1:0:-3]
 array([10,  7,  4,  1])
>>> tab1
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
>>> tab1[1:3,0:2]
array([[ 5,  6],
       [ 9, 10]])
>>> tab1[1:3,:]
array([[ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
>>> tab1[:,1:3]
array([[ 2,  3],
       [ 6,  7],
       [10, 11]])
```

II.3 - Opérations usuelles

Comme pour les tableaux monodimensionnels, les opérations usuelles sont appliquées élément par élément.

```
>>> tab1*2
array([[ 2,  4,  6,  8],
       [10, 12, 14, 16],
       [18, 20, 22, 24]])
>>> tab2*tab2
array([[[  1,   4],
        [  9,  16]],
       [[ 25,  36],
        [ 49,  64]],
       [[ 81, 100],
        [121, 144]]])
>>> np.exp(tab2)
array([[[  2.71828183e+00,   7.38905610e+00],
        [  2.00855369e+01,   5.45981500e+01]],
       [[  1.48413159e+02,   4.03428793e+02],
        [  1.09663316e+03,   2.98095799e+03]],
       [[  8.10308393e+03,   2.20264658e+04],
        [  5.98741417e+04,   1.62754791e+05]]])
```

II.4 - Opérations «matricielles»

II.4.a - Transposition

La *transposition* consiste en une *inversion des indices* : au tableau dont l'élément courant est `tab[i,j,k,...]`, on associe le tableau dont l'élément courant est `tab[...,k,j,i]`. Cette opération est réalisée par la méthode `transpose()`. Par exemple :

```
>>> tab1
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
>>> tab1.transpose()    # ou : tab1.T
array([[ 1,  5,  9],
       [ 2,  6, 10],
       [ 3,  7, 11],
       [ 4,  8, 12]])
```

II.4.b - Concaténation

La *concaténation* consiste en une agrégation de tableaux de *tailles compatibles* : deux tableaux accolés verticalement doivent avoir le même nombre de colonnes ; deux tableaux accolés horizontalement doivent avoir le même nombre de lignes.

```
>>> tab3 = np.array([[13, 14, 15, 16]])
>>> tab3
array([[13, 14, 15, 16]])
>>> tab4 = np.concatenate((tab1,tab3),axis=0)  # noter la présence des parenthèses
>>> tab4                                       # axis=0 -> concaténation verticale
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12],
       [13, 14, 15, 16]])
```

```
>>> tab5 = np.concatenate((tab1,tab3),axis=1)  # axis=1 pour concaténation horizontale
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-95-66e734811591> in <module>()
----> 1 tab5 = np.concatenate((tab1,tab3),axis=1)
ValueError: all the input array dimensions except for the concatenation axis
must match exactly
```

L'erreur est due à une incompatibilité entre le nombre de lignes de `tab1` (3 lignes) et `tab3` (1 ligne).

```
>>> tab5 = np.concatenate((tab1.T,tab3.T),axis=1)
>>> tab5
array([[ 1,  5,  9, 13],
       [ 2,  6, 10, 14],
       [ 3,  7, 11, 15],
       [ 4,  8, 12, 16]])
```

II.4.c - Produit matriciel

Le produit matriciel de deux matrices A et B de termes génériques respectifs $a_{i,j}$ et $b_{i,j}$ est la matrice C dont le terme générique $c_{i,j}$ est donné par la relation :

$$c_{i,j} = \sum_{k=1}^{n} a_{i,k} b_{k,j}$$

où n représente le nombre de colonnes de A et le nombre de lignes de B.
Avec `numpy`, le produit matriciel entre tableaux aux dimensions compatibles peut être réalisé par l'intermédiaire de la méthode dot(). Par exemple :

```
>>> A = np.array([[1,2,3],[1,2,3]])
>>> A
array([[1, 2, 3],
       [1, 2, 3]])
>>> B = np.array([[1,3],[2,4],[1,3],[2,4]])
>>> B
array([[1, 3],
       [2, 4],
       [1, 3],
       [2, 4]])

>>> A.dot(B)
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-107-7fbaa337fd94> in <module>()
----> 1 A.dot(B)
ValueError: objects are not aligned
>>> B.dot(A)
array([[ 4,  8, 12],
       [ 6, 12, 18],
       [ 4,  8, 12],
       [ 6, 12, 18]])
```

Exercice 2

Soit la matrice carrée :

$$\mathrm{M}(n) = \begin{bmatrix} 0 & 1 & 2 & \vdots & n \\ 1 & 1 & 2 & \vdots & \vdots \\ 2 & 2 & 2 & \vdots & n \\ \dots & \dots & \dots & \ddots & \vdots \\ n & \cdots & n & \cdots & n \end{bmatrix}$$

Écrire la fonction `mat(n)` retournant le tableau semblable à la matrice ci-dessus. On proposera deux solutions fondées sur :

— une boucle ajoutant une nouvelle “couche” à chaque itération,

— une fonction récursive permettant de définir la matrice $\mathrm{M}(n)$ à partir de la matrice $\mathrm{M}(n-1)$.

III - Tracé de courbes

Pour tracer des courbes, on utilise le module `matplotlib.pyplot` importé ici avec l’alias `plt`.

```
import matplotlib.pyplot as plt
```

Par défaut, au moment de l’importation, Python crée une «feuille de figures» numérotée 1 dans laquelle on peut insérer une ou plusieurs figures. On peut créer d’autres pages par la commande `plt.figure(n)` où `n` est entier et on navigue entre les pages par la même commande. Par défaut, on est dans la feuille 1 et les tracés sont toujours faits dans la figure courante. On détruit la feuille courante par la commande `plt.close()`. On affiche les différentes pages de figures dans des fenêtres interactives extérieures par `plt.show()`.

Par exemple, on peut tracer la fonction cosinus ainsi :

```
plt.title("la fonction cos")
X = np.linspace(0, 2*np.pi)
plt.plot(X,np.cos(X),'r:s')
plt.show()
```

La commande standard pour insérer une courbe dans la feuille courante est :

```
plt.plot ( tabx, taby, ch_tracé , label = "nom_de_la_courbe" )
```

où `tabx, taby` constituent les deux tableaux donnant les coordonnées des points à relier, le paramètre optionnel `label` permet de donner un nom à la courbe pour la légende et `ch_tracé` est une chaîne de caractères précisant le type du tracé *via* trois informations :

- le premier caractère de `ch` indique la couleur via la première lettre du nom de la couleur en anglais b(lue), r(ed), g(reen), c(yan), m(agenta), y(ellow), k(black), w(hite).
- les caractères suivant indiquent la façon de relier les points ('-' pour une ligne, '- -' pour des pointillés tirets, ' :' pour des pointillés points).
- le dernier caractère indique comment sont signalés sur le graphique les points de la courbe donnés explicitement *via* les listes `listex` et `listey` ('o' pour des petits cercles, 's' pour des petits carrés, 'v' pour des triangles, des pixels par défaut).

Par défaut, le tracé se fait dans la couleur courante (qui change à chaque plot) et en trait plein. Les noms des courbes apparaissent dans la légende générée par la commande `plt.legend()`. On peut forcer le tracé d’une grille en filigrane dans la figure par `plt.grid(True)`.

Exercice 3

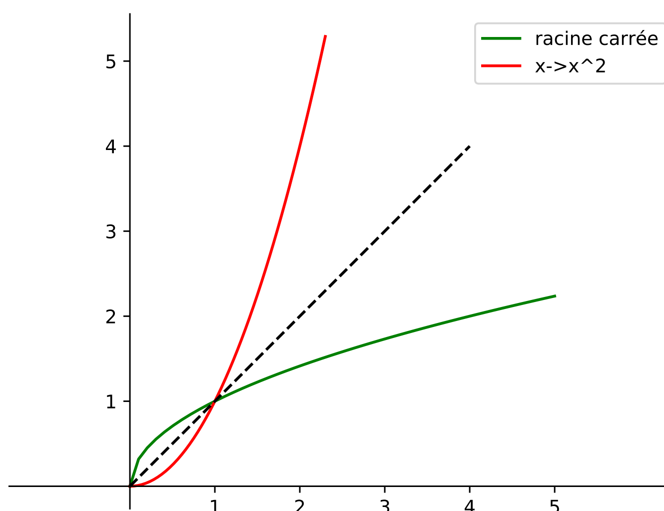
Tracer les représentations graphiques des fonctions exp, ln, racine carrée et de la fonction $x \mapsto x^3 + x - 2$.

Python adapte les axes aux valeurs des abscisses fournies pour la première courbe.
On peut modifier les limites du tracé par `plt.xlim(val_min,val_max)` et `plt.ylim(val_min,val_max)` ou par `plt.axis([valx_min,valx_max,valy_min,valy_max])`.
On donne un nom aux axes par `plt.xlabel('nom_abscisse')` et `plt.xlabel('nom_ordonnées')`. On peut aussi modifier les graduations placées automatiquement sur l'axe avec `plt.xticks` et `plt.yticks`.
Le tracé s'effectue dans un cadre formé de quatre bords `('right','top','bottom','left')` ; les bords bas et gauche indiquent respectivement les abscisses et les ordonnées. Cette configuration de base n'est pas adaptée pour le tracé traditionnel des courbes en mathématiques où l'on trace des courbes dans un repère orthonormé avec des axes passant par le point (0,0). Pour cela if faut modifier l'objet "cadre", d'abord en le récupérant dans une variable puis en effaçant les bords inutiles (haut et droite par exemple), en limitant les graduations sur les deux bords qu'on conserve (gauche et bas) et enfin en déplaçant ces bords sur l'abscisse ou l'ordonnée 0. Voici les commandes adaptées :

```
ax = plt.gca()
ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')
ax.xaxis.set_ticks_position('bottom')
ax.yaxis.set_ticks_position('left')
ax.spines['left'].set_position(('data',0))
ax.spines['bottom'].set_position(('data',0))
```

Exercice 4

Reproduire la figure suivante :



On peut tracer sur une même feuille plusieurs figures différentes (on parle de sous-figures). Il faut pour cela les organiser en lignes et colonnes (par exemple, trois lignes de deux figures). Les différentes figures sont alors numérotées dans l'ordre en commençant par la première ligne. Par exemple, pour six figures organisées en trois lignes et deux colonnes,

$$\begin{matrix} F_1 & F_2 \\ F_3 & F_4 \\ F_5 & F_6 \end{matrix}$$

Pour passer une sous-figure en figure courante, on utilise la commande

```
plt.subplot( nbre_lignes , nbre_colonnes , numéro )
```

Le tracé se fait alors comme d'habitude avec `plt.plot`. La commande `plt.suptitle(nom)`, où `nom` est une chaîne de caractères, permet de donner un nom global à la page de figure courante tandis que la commande `plt.title(nom)` donne un nom à la figure (ou sous-figure) en cours.

Par exemple, le code suivant engendre la page de figures qui suit :

```
X = np.linspace(-np.pi, np.pi)
XX = np.linspace(0, 2*np.pi)
plt.suptitle("les fonctions trigonométriques fondamentales")

plt.subplot(1, 2, 1)
ax = plt.gca()
ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')
ax.xaxis.set_ticks_position('bottom')
ax.yaxis.set_ticks_position('left')
ax.spines['left'].set_position(('data', 0))
ax.spines['bottom'].set_position(('data', 0))
plt.xticks([-np.pi, np.pi])
plt.yticks([-1, 1])
plt.axis([-3.5, 3.5, -1.2, 1.2])
plt.plot(X, np.sin(X))
plt.title("sinus")
```

```
plt.subplot(1, 2, 2)
plt.plot(XX, np.cos(XX))
ax = plt.gca()
ax.spines['top'].set_color('none')
ax.spines['right'].set_color('none')
ax.xaxis.set_ticks_position('bottom')
ax.yaxis.set_ticks_position('left')
ax.spines['bottom'].set_position(('data', 0))
plt.xticks([-np.pi, np.pi])
plt.yticks([-1, 1])
plt.axis([0, 6.5, -1.2, 1.2])
plt.title("cosinus")

plt.show()
```