Chap 9 : introduction à numpy et pylab.

1 Présentation de numpy

Le module `numpy` pour *numeric* `Python` est un *gros* module pour le calcul numérique en PYTHON. Avec `scipy` et `matplotlib` il permet de faire en Python l'analogue de ce qu'on peut faire avec `Matlab` ou `Scilab`, qui sont des logiciels de calculs numériques. Le module `pylab` regroupe `numpy` et `matplotlib`.

Fait essentiel :

> Aujourd'hui, en calcul scientifique, les objets de bases ne sont pas les nombres mais les *tableaux de nombres*. Ainsi la plupart des fonctions numpy vont agir directement sur des tableaux de nombres, comme on va le voir ci-après

On l'importe classiquement avec l'alias `np` :

```
import numpy as np
```

On peut voir toutes les fonctions `numpy` avec `dir(np)`. Notamment `np` contient toutes les fonctions mathématiques qu'on a déjà vu dans `math` mais dans une version plus puissante, car, comme on va le voir, elles s'appliquent aux tableaux.

1.1 L'objet de base de numpy : le `np.array`

On déclare un tableau (`array` en anglais) avec le mot clef `np.array`

```
>>>A=np.array([1,2,17])
>>>type(A)
numpy.ndarray
```

Le `nd` signifie *n-dimensional array*. Ici, le tableau est dit *1-dimensionnel*, par opposition à ce qu'on va voir ensuite. L'accès se fait comme pour les listes :

```
>>>A[2]
17
```

On peut déclarer, et cela sera très utile ensuite pour le calcul matriciel, des tableaux *bi-dimensionels*, qui ressemblent à nos listes de listes :

```
T=np.array([[1,2,3],[4,5,6]])
```

En `numpy` ce tableau est à interpréter comme tableau à deux lignes et trois colonnes, comme le montre la commande d'affichage :

```
>>>print(T)
[[1 2 3]
 [4 5 6]]
```

On accède alors aux entrées comme pour les listes de listes : avec `T[1][2]`, ou bien, et c'est spécifique à numpy, avec une syntaxe qui sera plus proche de l'usage matriciel, `T[1,2]`.

La commande `np.shape` renvoie la *forme* du tableau : nombre de lignes, nombres de colonnes.

```
>>>np.shape(T)
(2, 3)
```

Pour un tableau unidimensionnel elle renvoie simplement le nombre d'entrée

```
>>>np.shape(A)
(3,)
```

On peut faire des tableaux avec davantage de dimension, mais nous ne les utiliserons pas pour l'instant. Une différence entre `np.array` et `list` :

> Les données à l'intérieur de chaque entrée d'un `np.array` sont toutes de même type. La raison de cette contrainte sera expliquée au paragraphe 3 plus théorique.

On peut déclarer le type des données avec `dtype`.

```
A=np.array([1,2,3],dtype=int)
B=np.array([1,2,3],dtype=float)
C=np.array([1,2.4,3.2])
```

Pour `C` toutes les entrées seront considérées comme des `float` : structure de donnée *homogène*.

1.2 Des commandes natives sur les `np.array`

1.2.1 Les opérations usuelles

• **Additions entrée par entrée :** pour deux `np.array` $\mathtt{T} = [\mathtt{t}_0, \mathtt{t}_1, \ldots, \mathtt{t}_{n-1}]$ et $\mathtt{S} = [\mathtt{s}_0, \mathtt{s}_1, \ldots, \mathtt{s}_{n-1}]$ de même longueur, la commande `T+S` renvoie *non pas la concaténation comme pour les listes python*, mais le tableau obtenu en ajoutant entrée par entrée les éléments de `T` et `S`.

$$\mathtt{T} + \mathtt{S} = [\mathtt{t}_0 + \mathtt{s}_0, \ldots, \mathtt{t}_{n-1} + \mathtt{s}_{n-1}].$$

• **Multiplication entrée par entrée, division etc...** : même chose en remplaçant + par −, ∗, /.

• **Multiplication ou addition d'un nombre à un tableau :** si `a=2` et `T=np.array([1,1,1])` alors `a*T` renverra le `np.array([2,2,2])` et `a+T` renverra `np.array([3,3,3])`.

1.2.2 Les fonctions `numpy` s'appliquent à chaque entrée d'un `np.array`

a) Les fonctions mathématiques usuelles sont dans `numpy` : par exemple `np.sin`.

> La différence entre `np.sin` et `math.sin` c'est que les fonctions `np` s'appliquent entrée par entrée à des tableaux. On dit qu'elles sont « vectorialisées ». Ceci permet une économie d'écriture car cela épargne l'écriture d'un certain nombre de boucles `for` pour parcourir le tableau.

Ainsi par exemple :

```
>>>L=np.array([0,np.pi/2,np.pi, 3*np.pi/2])
>>>M=np.sin(L)
>>>print(M)
[  0.00000000e+00   1.00000000e+00   1.22464680e-16  -1.00000000e+00]
```

Ce n'est peut-être pas très joli, mais c'est des braves flottants qui doit être considérés comme 0, 1, 0, -1.

b) Un laxisme typique de Python :

> Les fonctions `numpy` s'appliquent aussi aux listes Python, mais elles renvoient un `np.array`

Par exemple :

```
>>>T=[1,2,3]
>>>L=np.exp(T)
>>> L
array([  2.71828183,   7.3890561 ,  20.08553692])
```

c) Exemples utiles pour les TP sur les nombres complexes :

(i) si `T=np.array([complex(1,2),complex(3,4)])` est un tableau (ici de deux nombres complexes) alors `np.real(T)` renvoie le tableau des parties réelles ici `np.array([1,3])` et `np.imag(T)` le tableau des parties imaginaires.

(ii) La fonction `np.exp` gère aussi les exp. complexes ! Par exemple si `I=complex(0,1)`, `T=[2*I*np.pi,0]` alors :

```
>>> np.exp(T)
array([ 1. -2.44929360e-16j,  1. +0.00000000e+00j])
```

ce qu'il faut comprendre comme `[1,1]`.

1.3 Comment fabriquer agréablement des `np.array` ?

1.3.1 Il est très simple de fabriquer un tableau rempli de zéros

```
A=np.zeros(6)
print(A)
B=np.zeros((4,4))
print(B)
```

L'intérêt est qu'ensuite, on peut modifier les entrées... si on a plutôt besoin que ce tableau soit formé d'entiers :

```
A=np.zeros(6,dtype=int)
```

1.3.2 Alternative `numpy` au `range` : `np.arange`

`np.arange(a,b,p)` renvoie le `np.array([a, a+p, ..., a+kp])` jusqu'au plus grand `k` tel que `a+kp <b`.

• Une différence avec le `range` de Python est qu'il peut produire des flottants avec un pas qui peut être lui-même un flottant.

Par exemple :

```
>>>np.arange(0.1,1.1,0.1) # le troisième argument est le pas
 array([ 0.1,  0.2,  0.3,  0.4,  0.5,  0.6,  0.7,  0.8,  0.9,  1. ])
```

• Une autre différence est donc qu'il renvoie déjà un `np.array` et pas un itérateur.

1.3.3 Alternative `np.linspace`

`np.linspace(a,b,N)` renvoie le `np.array([a,a+(b-a)/(N-1),a+2(b-a)/(N-1),...,b]` : subdvision régulière du segment `[a,b]` en `N-1` intervalles, ce qui donne un tableau avec `N` entrées.

Par exemple :

```
>>> np.linspace(2,3,11)
array([ 2. ,  2.1,  2.2,  2.3,  2.4,  2.5,  2.6,  2.7,  2.8,  2.9,  3. ])
```

1.3.4 Redimensionner un tableau `np.reshape`

```
>>>A=array([1, 2, 3, 4, 5, 6])
>>> np.reshape(A,(2,3))
array([[1, 2, 3],
       [4, 5, 6]])
```

1.3.5 Concatenation des `np.array`

Bien sûr le `+` ne marche plus, pourquoi ? Fondamentalement, les tableaux ne sont pas fait pour faire de la concaténation, cf. § 3 pour comprendre pourquoi.

Il n'y a donc pas de *méthode* `append` sur les `numpy.ndarray` mais :

a) Une *commande* `np.append` avec la syntaxe qui n'est donc pas celle d'une méthode, qui ne *modifie pas son argument* mais fabrique un nouveau tableau :

```
A=np.array([1,2])
B=np.array([2,3])
C=np.append(A,B)
print(C)
```

b) Une commande `np.concatenate` qui ressemble beaucoup : *attention aux parenthèses !*

```
A=np.array([1,2])
B=np.array([2,3])
C=np.concatenate((A,B))
print(C)
```

2 Application des np.array pour un tracé efficace de graphes de fonctions

On a vu en T.P. comme tracer un graphe de fonction avec le `plot` de `matplotlib` et des listes python. Ici, on va aller un peu plus vite avec les tableaux numpy.

Par exemple si on veut tracer le graphe de sinus sur $[0, 2\pi]$, avec `linspace` on créé immédiatement un tableau de valeurs en abscisses `X` et avec `np.sin(X)` un tableau des images et donc :

```
import numpy as np
import matplotlib.pyplot as plt
X=np.linspace(0,2*np.pi,50)
Y=np.sin(X)
plt.plot(X,Y)
plt.show()
```

Variante plus commode : avec le module `pylab` qui contient les deux modules précédents :

```
import pylab as pl
X=pl.linspace(0,2*np.pi,50)
Y=pl.sin(X)
pl.plot(X,Y)
pl.show()
```

3 De l'usage du mot *tableau* en informatique

En informatique, on appelle, *au sens strict*, *tableau* `T` un certain type de donnée contenant possiblement plusieurs entrées modifiables, auxquelles on accède par leur numéro d'indice `T[0]`, ..., `T[i]` ... avec la propriété cruciale suivante :

l'accès au contenu de chaque entrée `T[i]` doit être à cout constant.

Techniquement, dans la plupart des langages, cette possibilité de coût constant est assurée par le fait que le contenu de chaque entrée est du *même type* et donc *codé sur le même nombre de bits*, et les entrées successives du tableau sont représentées par des cellules *contiguës* dans l'espace mémoire.

Autrement dit : un tableau de `int8` sera pointera vers une liste d'adresse mémoire successives, chaque adresse contenant un `int8`, et pour accéder `T[17]` à partir de la lecture seulement de

l'adresse de `T[0]`, il suffira de faire disons `+17` (ou `+17*8` par exemple si on a des entiers sur 64 bits) pour avoir l'adresse de `T[17]`.
Conséquence évidente :

ces contraintes ne permettent pas d'insérer ou de supprimer un élément au milieu du tableau.

Si on veut insérer un élément, on doit refabriquer un tableau, en copiant les éléments successivement...
Conséquence aussi : les listes python ne sont *pas* des tableaux au sens précédent. Et effectivement, on peut montrer que le coût d'accès aux entrées d'une liste python est constant jusqu'à une certaine longueur de listes, puis augmente subitement .... On parle d'accès à *coût constant amorti*.

Néanmoins, le plus souvent, quand on parlera de *tableau* dans les énoncés en I.P.T., on les codera par des listes en Python!!
Mais les tableaux numpy, eux, sont davantage des tableaux au sens précédent, d'où la contrainte d'homogénéité par exemple et l'absence d'un append qui modifierait le tableau.

En CamL, vous verrez la différences plus claire entre *tableaux* au sens ci-dessous, et *listes chaînées* où le `append` est naturel, mais l'accès n'est pas à coût constant.