Université de Rouen
M1 IBIOM
Année 2012-2013

Mathématiques Appliquées
Python et les math (n°1.)

1. Python

Avertissement : l'auteur ne connaît pas Python et ce document a été élaboré très/trop vite [1]. De plus les exemples fonctionnent avec Python 2.6 ou 2.7 mais nécessitent d'être adaptés pour Python 3.0 ou supérieur.

Le but est d'utiliser Python comme Scilab, Matlab ou comme une puissante calculatrice. Initialement Python ne sait pas faire grand chose dans le domaine mathématique, comme tracer une fonction, calculer cos(1), additionner des matrices, etc. Cependant des extensions de grande qualité sont disponibles : Scipy, Numpy et Matplotlib pour citer les principales. Une recherche sur Internet permet de situer ces librairies et d'avoir des informations (en anglais, très rarement en français, voir par exemple www.dakarlug.org/pat/scientifique/html).

Il est possible d'utiliser Python de manière interactive, à la Scilab/Matlab, ce qui est pratique pour nos expériences numériques, plutôt que de créer un fichier `nom.py` et d'exécuter via `python nom.py`. L'exécutable `python` permet de le faire mais `ipython` («i» comme interactive !) propose des fonctionnalités supplémentaires (coloration syntaxique, indentation automatique, complétion et des choses plus pointues sur les objets, etc).

Toute distribution Linux récente propose Python, IPython, Matplotlib, Scipy et Numpy dans la liste des applications/paquets disponibles : à installer via le gestionnaire des paquets. Quant à Win$$ c'est possible aussi. Pour ne pas se casser la tête le projet `pythonxy` semble intéressant et propose un environnement agréable et toutes les librairies pour faire du calcul scientifique à la fois pour Win$$ et Linux : www.pythonxy.com et code.google.com/p/spyderlib/. L'auteur ne certifie pas que l'installation de ces deux programmes est facile. En cas de problème, se limiter à IPython, Matplotlib, Scipy et Numpy suffira.

2. Calcul numérique

Ce qui suit sera fait de façon interactive : `ipython`. `ipython` spécifie `[In]` pour l'entrée et `[Out]` pour la sortie. Pour faire simple `>>>` sera l'entrée en `IPython` pour nous. À vous devoir la sortie !

2.1. **Flottant et entier.** Comparer et expliquer

```
>>> 1/3
>>> float(1)/3
>>> 1./3
>>> 1./3.
et aussi
>>> 1.4
>>> print (1.4)
```

Comparer et expliquer

```
>>> v0 = 6
>>> g = 9.81
>>> t = 0.6
>>> y = v0*t - 0.5*g*t**2
>>> print y
>>> y
>>> a=1/3
>>> a
>>> a;
>>> print (a)
```

Comparer et expliquer

```
>>> X=1E30
>>> Y=1E10
>>> X+Y-X
>>> X-X+Y
```

Pour `Y=1` trouver la valeur minimale de `X` pour laquelle ce phénomène se produit.

2.2. **Fonctions mathématiques.** Tester `>>> cos(1)` : message d'erreur. La fonction cosinus n'est pas définie d'entrée de jeu dans Python. Pour disposer des fonctions mathématiques usuelles, la librairie (d'origine) est `math`. On peut alors d'importer juste les fonctions nécessaires par `from math import cos, log` ou toutes les fonctions mathématiques par `from math import *`. Dans le premier cas l'inconvénient est qu'il faut savoir à l'avance les fonctions utilisées par la suite, dans le deuxième cas on risque de surcharger –inutilement– la mémoire. Ceci est valable pour toutes les librairies/extensions : pour Scipy, Numpy on veillera à ne pas faire un «`from numpy import *`».

1. merci de bien vouloir signaler toute faute d'orthographe

Trouver comment avoir les définitions de π et e (indication : `pi, e`).

Calculer

$$3+\frac{8}{5}-(7\times 4),\quad \frac{3^2-8}{2^2}+\frac{3^2}{5}-\frac{6\times 11^2}{7^3},$$
$$\exp(\ln(5))\times(1-0.12^2)^{1/2},\quad \cosh^2 1.5-\sinh^2 1.5$$
$$\sin(\frac{\pi}{6}+\arccos 0.5),$$
$$\cos^2 4.770796326794897+\sin^2 11.05398163397448$$

Un dernier exemple sur l'approximation numérique. On définit le sinus hyperbolique par $\sinh(x)=\frac{1}{2}(\exp(x)-\exp(-x))$. On calcule des trois façons différentes $\sinh(2*\pi)$:

```
>>> from math import sinh, exp, e, pi
>>> x = 2*pi
>>> r1 = sinh(x)
>>> r2 = 0.5*(exp(x) - exp(-x))
>>> r3 = 0.5*(e**x - e**(-x))
>>> print r1, r2, r3
```

Expliquer. Dans le même style tester `print %.16f %.16f % (1/49.0*49, 1/51.0*51)`.

2.3. **Nombres complexes.** Python gère les nombres complexes. Cependant le «i» imaginaire sera `j`. Jouer avec les nombres complexes.

Du point de vue mathématique, certaines fonctions usuelles sont définies aussi sur l'ensemble des complexes, comme le sinus ou l'exponentielle. Faire un `sin(1+2j)` provoquera une erreur. Pour cela il est nécessaire d'utiliser l'extension `cmath`. Cependant l'inconvénient est que `sin(1)` retournera un nombre complexe (avec bien sûr une partie imaginaire nulle). `Numpy` apporte une solution :

```
>>> from numpy.lib.scimath import *
>>> sqrt(4)
>>> sqrt(-2)
```

2.4. **Écrire des fonctions.** Une fonction qui dépend de deux paramètres et retourne une valeur (un flottant) s'écrit de la façon suivante

```
def F2(x,t):
    return 3.*x+5*t-t**2
```

ou avec `IPython`

```
>>> def F2(x,t):
...     return 3.*x+5*t-t**2
...
```

Question : quel est l'impact de «`3.`»?

Il y a une question de variable locale/globale que l'on comprend à l'aide de l'exemple

```
a = 20; b = -2.5    #  variable globale
def f1(x):
    a = 21          # nouvelle variable locale
    return a*x + b # 21*x - 2.5
print a             # donne 20
def f2(x):
    global a
    a = 21          # la variable a globale est modifiée
    return a*x + b # 21*x - 2.5
f1(3); print a      # 20 est retourné
f2(3); print a      # 21 est retourné
```

Une fonction peut retourner plusieurs arguments. Dans l'exemple suivant on calcule le mouvement d'une masse ponctuelle soumise à la gravité :

$$\frac{dy}{dt}=v_0-gt.$$

```
>>> def yfunc(t, v0):
...     g = 9.81
...     y = v0*t - 0.5*g*t**2
...     dydt = v0 - g*t
...   return y, dydt
```

```
...
```

Tester

```
>>> position, vitesse = yfunc(0.6, 3)
```

Une fonction peut aussi ne rien retourner (afficher des résultats à l'écran, faire des graphiques, stocker des fichiers, etc). Dans ce cas il n'y a pas de `return`.

2.4.1. *Fonctions dans des fichiers.* Il est indispensable pour conserver, développer ses propres routines de les mettre dans un fichier annexe comme `tp1.py` par exemple édité avec votre éditeur préféré. Comme pour `import numpy as np` nous adopterons la méthode suivante. Supposons que le fichier `tp1.py` contient

```
def f(x):
    return 3   # une fonction
```

Pour bénéficier de `tp1.py` nous utiliserons

```
>>> import tp1 as tp
>>> tp.f(4)
```

En cas de modification de `tp1.py` pour charger les mises à jour

```
>>> reload(tp)   # nom du préfixe et pas du fichier
```

2.4.2. *Souvenir!*

Exercice. Calcul approché de $\exp(x)$. On rappelle/affirme que pour tout x réel

$$\exp(x) = \sum_{k=0}^{+\infty} \frac{x^k}{k!} \qquad (k! = k \times (k-1) \times \cdots \times 2 \times 1 \text{ et } 0! = 1)$$

(a) écrire une première fonction `cexpo1(x,n)`, `x` et `n` sont en entrée, qui retourne

$$\sum_{k=0}^{n} \frac{x^k}{k!}.$$

(b) écrire une deuxième fonction `cexpo2(x,n)`, `x` et `n` sont en entrée, qui calcule la même somme mais en commançant par la fin.

(c) Comparer ces deux procédures pour différentes valeurs de x et n avec la valeur de $\exp(x)$. Constatez et expliquez.

(d) Tester `cexpo2` avec $x = -20$ et $n = 100$. Que pensez vous de cette approximation ?

(e) En utilisant le fait que $\exp(-x) = 1/\exp(x)$, écrire `cexpo3(x,n)` qui donne des résultats satisfaisants avec $x \geq 0$ et $x < 0$

(f) Faire une version du calcul de $\exp(x)$, `cexpo4(x,n)`, avec une méthode quasi–Hörner, i.e. en utilisant le fait que

$$\sum_{k=0}^{n} \frac{x^k}{k!} = 1 + x\left(1 + \frac{x}{2}\left(1 + \frac{x}{3}\left(1 + \frac{x}{4}(\cdots + \frac{x}{n-1}(1 + \frac{x}{n})\cdots)\right)\right)\right)$$

et toujours en tenant compte de $\exp(-x) = 1/\exp(x)$ pour le calcul de $\exp(x)$ pour $x < 0$.

(g) Quelle est la procédure qui donne la meilleure approximation ?

3. Les vecteurs

Python gère les tableaux. Par exemple

```
>>> x=[1, 2, 4]
>>> y=range(5)
```

Grosso modo pour tracer des courbes de fonctions, on génère les points des abscisses x_i et les valeurs en ces points $f(x_i)$: la courbe sera la ligne brisée reliant les points du plan (x_i, y_i). On a donc besoin d'utiliser

```
>>> def f(x):
...     return x**3   # une fonction
...
>>> n = 5
# nbre de points sur l'axe des abcisses
>>> dx = 1.0/(n-1)  #pas de discrétisation
>>> xlist = [i*dx for i in range(n)]
>>> ylist = [f(x) for x in xlist]
```

ce qui n'est pas toujours très pratique. Ajoutons que qu'un tableau à deux indices est considéré comme une liste de liste et enfin que les éléments d'un tableau ne sont pas nécessairement tous du même type, ceci ne correspond pas à l'idée d'un vecteur, d'une matrice.

Numpy fournit la fonction `array` pour une gestion «à la mathématique», similaire à Scilab, Matlab dans lesquels tout est matrice. Tester

```
>>> from numpy import *
# évidemment ce n'est pas à faire !!
>>> r=[1, 2, 3.]
>>> a=array(r)
>>> print (a)
>>> a=zeros(10)
>>> print (a)
>>> a=linspace(1,2,6)
>>> print (a)
```

Pour raccourcir l'écriture on a commencé les instructions par `from numpy import *`, ce qui n'est pas du tout recommandé mais allège les noms de fonctions. Il faut normalement écrire `from numpy import as np` et accéder aux instructions de Numpy via `np.nomfonction`.

On accède à un élément `array` par `a[i]` et comme pour les listes on a la gestion «à la slice». Attention `a[1:-1]` sélectionne tous les éléments sauf le premier et le dernier mais ne fait pas une copie de `a`. Ainsi

```
>>> a=linspace(1,2,6)
>>> b=a[-1:1]
>>> print (a,b)
>>> b[2]=10
>>> print (a,b)
```

Il faudra selon la situation utiliser `copy` : `b=a.copy()`

Le grand intérêt de Numpy réside dans la vectorisation : il n'est pas nécessaire de faire une boucle pour évaluer une fonction pour chaque élément d'un vecteur.

```
>>> a=linspace(1,2,6)
>>> cos(a)
>>> x=array([, 2, 3.5])
>>> a = x.copy()
>>> a **= 4
>>> a *= 3
>>> a += 2*x
>>> a += 4
>>> a /= x + 1
```

Dans le dernier cas on a calculé `a = (3*x**4 + 2*x + 4)/(x + 1)`. L'intérêt vient d'une économie mémoire : avec la méthode `a = (3*x**4 + 2*x + 4)/(x + 1)` Python fait des calculs intermédiaires et a donc besoin de place pour stocker ces résultats intermédiaires, ce qui n'est pas le cas de la méthode –moins lisible– dans l'exemple.

Cela fonctionne généralement avec toutes les fonctions (de Python ou les vôtres), avec quelques restrictions cependant (test à l'intérieur de la fonction).

4. Les matrices (enfin)

Même remarque que pour les vecteurs :

```
>>> table=[[-30, -22.0], [-20, -4.0], [-10, 14.0]]
>>> table2 = array(table)
>>> print table2
>>> type(table2)
```

On accède aux éléments via

```
>>> table2[1][0]
>>> table2[1,0]
```

La taille est donnée par `table2.shape` (valide aussi pour les vecteurs). On peut aussi redimensionner un vecteur, un tableau (évidemment un vecteur de 5 lignes ne peut être converti en une matrice de taille 4×2 ou 2×2, les nombres d'éléments doivent être égaux). On peut aussi extraire des sous tableaux (ou sous matrices) :

```
>>> t = linspace(1, 30, 30).reshape(5, 6)
>>> t
>>> t[1:-1:2, 2:]
>>> t[:-2, :-1:2]
```

Attention pour faire le produit matriciel (matrice×vecteur, matrice×matrice), `table2*table2` fait juste un produit terme à terme ! Pour l'algèbre linéaire on peut soit utiliser `dot` ou utiliser les objets de type `mat` :

```
>>> x1 = array([1, 2, 3], float)
>>> x2 = matrix(x)     # ou mat(x)
>>> x2                 # vecteur ligne
>>> x3 = mat(x).transpose() # vecteur col
>>> x3
```

```
>>> type(x3)
>>> isinstance(x3, matrix)
```

et là nous sommes sauvés

```
>>> A = eye(3)  # matrice identité
>>> A
>>> A = mat(A)
>>> A
>>> y2 = x2*A  # produit vecteur matrice
>>> y2
>>> y3 = A*x3  # produit matrice vecteur
>>> y3
```

Écrire la même séquence sans utiliser le type `mat` mais `array` et la fonction `dot`.

Vérifier que si le produit de deux matrices n'est pas possible (question de taille) alors Python retourne un message d'erreur.

Résolution de système linéaire :

```
>>> n=10
>>> A = zeros((n,n))
>>> x = zeros(n)
>>> b = zeros(n)
>>> for i in range(n):
>>>     x[i] = i/2.0   # on définit x
>>>     for j in range(n):
>>>         A[i,j] = 2.0 + float(i+1)/float(j+i+1)
>>> b = dot(A, x)  # produit matrice-vecteur
>>> # résout le système linéaire A*y=b:
>>> y = linalg.solve(A, b)
```

Exercice. Comment créer les tableaux suivants (sans les définir élément par élément, on a tout de même le droit de définir 2, 3 mais pas plus éléments !)

```
[[ 0 1 2 3 4]
 [ 5 6 7 8 9]
 [10 11 12 13 0]
 [15 16 17 18 19]
 [20 21 22 23 24]]
```

```
[[ 1.  1.  1.  1.]
 [ 1.  1.  1.  1.]
 [ 1.  1.  1.  2.]
 [ 1.  6.  1.  1.]]
```

```
[[0 0 0 0 0]
 [2 0 0 0 0]
 [0 3 0 0 0]
 [0 0 4 0 0]
 [0 0 0 5 0]
 [0 0 0 0 6]]
```

Vérifier vos exercices calculatoires avec Python ! [`linalg.inv(A)` permet de calculer la matrice inverse] La librairie Scipy possède des fonctionnalités supplémentaires sur l'algèbre linéaire et d'autres outils d'analyse numérique (interpolation, minimisation, fft, etc.)

5. Quelques ajouts

Nous avons `np.linspace(a,b,n)` qui renvoie un `array` contenant une discrétisation équidistante de $[a,b]$ de n points. Il y a un équivalent avec

```
>>> np.r_[1:7:5j] # Notez bien le 'j'
array([ 1. ,  2.5,  4. ,  5.5,  7. ])
# Notez bien la différence
>>> np.r_[1:7:5]
array([1, 6])
>>> np.r_[1:4:.5]
array([ 1. ,  1.5,  2. ,  2.5,  3. ,  3.5])
```

En fait la commande `np.r_` fait plus que cela et permet aussi de concaténer (voir la commande `np.concatenate`). Par exemple

```
>>> np.r_[np.array([1,2,3]), 0, 0, np.array([4,5,6])]
array([1, 2, 3, 0, 0, 4, 5, 6])
```

```
>>> np.r_[-1:1:6j, [0]*3, 5, 6]
array([-1. , -0.6, -0.2,  0.2,  0.6,  1. ,  0. ,  0. ,  0. ,  5. ,  6. ])
```

On peut concaténer selon les lignes ou les colonnes

```
>>> a = np.array([[0, 1, 2], [3, 4, 5]])
>>> np.r_['-1', a, a] # selon les colonnes
array([[0, 1, 2, 0, 1, 2],
       [3, 4, 5, 3, 4, 5]])
>>> np.r_[a,a] # selon les lignes
array([[0, 1, 2],
       [3, 4, 5],
       [0, 1, 2],
       [3, 4, 5]])
>>> np.r_['0,2', [1,2,3], [4,5,6]] # selon les lignes mais avec un objet de  dim>=2
array([[1, 2, 3],
       [4, 5, 6]])
```

La commande accepte en réalité un argument du type `'n,m,p'`. Le `n` correspond à l'axe selon lequel on concatène, le `m` la dimension finale de l'objet tandis que `p` «which axis should contain the start of the arrays which are less than the specified number of dimensions. In other words the third integer allows you to specify where the 1s should be placed in the shape of the arrays that have their shapes upgraded. By default, they are placed in the front of the shape tuple. The third argument allows you to specify where the start of the array should be instead. Thus, a third argument of 0 would place the 1s at the end of the array shape. Negative integers specify where in the new shape tuple the last dimension of upgraded arrays should be placed, so the default is -1.» Merci de m'expliquer tous les détails !

```
>>> np.r_['0,2,0', [1,2,3], [4,5,6]]
array([[1],
       [2],
       [3],
       [4],
       [5],
       [6]])
>>> np.r_['1,2,0', [1,2,3], [4,5,6]]
array([[1, 4],
       [2, 5],
       [3, 6]])
```

Toujours à propos de concaténation, `np.concatenate( , axis=1)` est équivalent à `np.hstack`, de même `np.concatenate( , axis=0)` correspond à `np.concatenate( , axis=0)`. Cela permet de raccourcir.

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

```
       [3],
       [4]])
```

et

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