

TD2 Numpy & Matplotlib

1 Quelques rappels

Les tuples. Les tuples correspondent aux listes à la différence qu'ils sont non modifiables. Une valeur de type tuple est délimitée par des parenthèses comme par exemple `(1, 2, "a")`. Pour créer une variable `x` de type tuple, on écrit : `x = (1, 2, "a")`. Pour accéder à un élément d'un tuple, on utilise les crochets comme pour les listes. On ne peut pas modifier une valeur de type tuple : on ne peut pas changer un élément du tuple, ni ajouter, ni retirer un élément.

Les dictionnaires. Un dictionnaire est un ensemble énumérable de couples (*clé*, *valeur*) mais, à la différence de la liste, un dictionnaire contient des éléments non ordonnés. Plus clairement, il n'y a pas d'indice et on accède aux valeurs d'un dictionnaire par les clés. On définit un dictionnaire vide, appelé `dico`, avec l'instruction : `dico = {}`. Ensuite, on remplit le dictionnaire en définissant les différentes clés auxquelles on affecte des valeurs (une par clé) : `dico[cle] = valeur`. On peut ajouter autant de clés que nécessaire. On peut aussi initialiser toutes les clés d'un dictionnaire en une seule instruction :

```
dico={cle1 : valeur1, ..., clek : valeurk}
```

On accède à une valeur particulière du dictionnaire grâce à la clé associée : `dico[cle]`. Il est également possible de supprimer un élément d'un dictionnaire à l'aide de l'instruction : `del dico[cle]`. La méthode `keys()` appliquée à un dictionnaire retourne toutes les clés contenues dans ce dictionnaire. La méthode `values()` appliquée à un dictionnaire va retourner toutes les valeurs contenues dans ce dictionnaire.

La méthode `items()` fournit la liste des tuples (*cle*, *valeur*).

Exercice 1

Soit le dictionnaire suivant :

```
d = {'nom' : 'Dupuis',  
     'prenom' : 'Jacque',  
     'age' : 30}
```

Écrire en Python les instructions permettant de :

1. corriger l'erreur dans le prénom, la bonne valeur devant être `Jacques` ;
2. afficher la liste des clés du dictionnaire ;
3. afficher la liste des valeurs du dictionnaire ;
4. afficher la liste des paires clé/valeur du dictionnaire ;
5. écrire la phrase `"Jacques Dupuis a 30 ans"` à l'aide des éléments du dictionnaire.

2 Le module numpy

Le module `numpy` permet de faire du calcul scientifique : représenter des vecteurs et matrices, effectuer du calcul matriciel, des calculs statistiques... Traditionnellement on importe `numpy` de la façon suivante :

```
import numpy as np
```

Le fonction `array` permet de former un tableau `numpy` (de type `ndarray`) à partir d'une liste, ou d'une liste de listes.

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

crée une matrice `mat` de 3 lignes et 2 colonnes contenant des entiers. Tous les éléments d'un tableau `numpy` sont de même type. Ainsi,

```
v=np.array([1,5.0,3])
```

crée un vecteur de 3 float.

La méthode `tolist` appliquée à un tableau `numpy` renvoie une liste, ou une liste de listes, à partir du tableau. Ainsi, l'instruction

```
l=mat.tolist()
```

affecte à la variable `l` la valeur `[[1,2],[6,3],[4,5]]`.

Quelques fonctions utiles :

- `np.shape(mat)` renvoie (3,2), c'est-à-dire un tuple qui indique le nombre d'éléments dans chaque dimension,
- `np.size(mat)` renvoie 6, c'est-à-dire le nombre total d'éléments,
- `np.ndim(mat)` renvoie 2, c'est-à-dire le nombre de dimension.

Pour créer des tableaux spécifiques, on peut utiliser les fonctions :

- `np.zeros` renvoie un tableau dont tous les coefficients sont nuls : par exemple `np.zeros(5)` renvoie un vecteur de longueur 5 et `np.zeros((10,3))` renvoie une matrice de 10 lignes et 3 colonnes,
- `np.ones` renvoie un tableau dont tous les coefficients sont égaux à 1,
- la méthode `fill` appliquée à un tableau `numpy` permet d'attribuer à tous les éléments de ce tableau une valeur constante donnée en argument : par exemple, après l'exécution des deux instructions suivantes

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

mat vaut `[[1,1,1],[1,1,1]]`

- `np.identity(n)` renvoie la matrice identité d'ordre `n`
- `np.linspace(min,max,n)` renvoie un vecteur de `n` valeurs linéairement échelonnées dans le segment `[min, max]`

- `np.logspace(min, max, n)` renvoie un vecteur de n valeurs logarithmiquement échelonnées dans le segment $[\min, \max]$

Soient a et b deux tableaux `numpy`, l'expression $a+b$ (resp. $a*b$) est valide et elle réalise l'addition (resp. la multiplication) terme à terme des éléments du tableau. On peut aussi écrire : $a*10$ qui va alors multiplier chaque terme du tableau par 10. Pour calculer les 10 premières puissances de 2 on pourra exécuter l'instruction suivante :

```
2 ** np.array(range(10))
```

De façon générale, on peut appliquer des fonctions mathématiques usuelles f à chaque élément d'un tableau a avec la syntaxe `np.f(a)`. Par exemple, l'instruction

```
np.log(np.array(range(1, 11)))
```

va renvoyer :

```
array([ 0.          ,  0.69314718,  1.09861229,  1.38629436,  1.60943791,
        1.79175947,  1.94591015,  2.07944154,  2.19722458,  2.30258509])
```

Pour le calcul matriciel, on dispose des fonctions suivantes :

- `transpose(a)` qui renvoie la transposée d'une matrice a ,
- `trace(a)` qui calcule la trace d'une matrice a ,
- `dot(a, b)` qui effectue le produit matriciel de a par b ,
- `linalg.matrix_power(a, n)` qui calcule la puissance n ème de la matrice a ,
- `linalg.inv(a)` qui calcule l'inverse de la matrice carrée a ,
- `linalg.det(a)` qui calcule le déterminant de la matrice carrée a ,
- `linalg.solve(a, b)` qui résout le système $ax = b$,
- `linalg.norm(x, opt)` qui calcule la norme du vecteur x : norme euclidienne par défaut, norme 1 si `opt=1`, norme infinie si `opt=np.inf`,
- `linalg.eigvals(a)` renvoie le vecteur des valeurs propres de a .

Exercice 2:

Soit un entier $n \geq 0$ et a un réel non nul. On considère la matrice carrée d'ordre n à coefficients réels :

$$A_{n,a} = \begin{pmatrix} 0 & \frac{1}{a} & 0 & \dots & 0 \\ a & 0 & \frac{1}{a} & \dots & 0 \\ 0 & a & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & 0 & \frac{1}{a} \\ 0 & \dots & 0 & a & 0 \end{pmatrix}$$

1. écrire une fonction qui étant donnés un entier $n \geq 0$ et un réel a non nul renvoie la matrice $A_{n,a}$,

2. pour $3 \leq n \leq 8$ et $a \in \{-2, -1, 1, 2, 3\}$, écrire une fonction qui renvoie un dictionnaire dont les clés sont les déterminants de $A_{n,a}$ et les valeurs la liste des couples (n, a) ayant leur déterminant égal à la clé,
3. écrire le programme qui affiche les différentes valeurs de déterminants obtenus et, pour chacune d'entre elle, les couples (n, a) correspondants.

3 Le module `matplotlib`

Pour le simple tracé de courbes, nous n'utiliserons que le sous-module `pyplot` importé, avec alias, à l'aide de la commande :

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

La documentation est disponible à <http://www.matplotlib.org/>.

Les fonctions essentielles de `pyplot` sont :

- `plt.plot()` pour le tracé de points, de courbes ; le premier argument de `plot` est la liste des abscisses, le deuxième la liste des ordonnées et le troisième (optionnel) le motif des points :

- '.' pour un petit point,
- 'o' pour un gros point,
- '+' pour une croix,
- '*' pour une étoile,
- '-' points reliés par des segments,
- '-' points reliés par des segments en pointillés,
- '-o' gros points reliés par des segments (on peut combiner les options),
- 'b', 'r', 'g', 'y' pour de la couleur (bleu, rouge, vert, jaune, etc.).

- `plt.show()` pour afficher le graphique créé.

Voici deux exemples d'utilisation :

```
import numpy as np
import matplotlib.pyplot as plt
x=np.array([1,3,4,6])
y=np.array([2,3,5,1])
plt.plot(x,y)
plt.show()
```

```
import numpy as np
import matplotlib.pyplot as plt
x=np.linspace(0,2*np.pi,30)
y=np.cos(x)
plt.figure('Fonction cosinus')
plt.plot(x,y,label="cos(x)")
plt.title('Fonction cosinus')
plt.show()
```

En effet, on peut améliorer le tracé en remplissant quelques options avant de le sauvegarder (au format `.png` dans le répertoire utilisateur).

```
>>> plt.grid(True) # Affiche la grille
>>> plt.legend('cos', 'upper right', shadow = True) # Légende
>>> plt.xlabel('axe des x') # Label de l'axe des abscisses
>>> plt.ylabel('axe des y') # Label de l'axe des ordonnées
>>> plt.title('Fonction cosinus') # Titre
>>> plt.savefig('figCos') # sauvegarde du fichier figCos.png
```

Exercice 3

1. Tester les deux exemples.
2. Représenter la fonction $y = x^3 - 3x^2 + 2x + 12$ pour x variant de -10 à 15 à l'aide des modules `numpy` et `matplotlib`.
3. Soit un cercle de rayon $r = 1$ inscrit dans un carré de côté $l = 2$. L'aire du carré vaut 4 et l'aire du cercle vaut π . En choisissant N points aléatoires (à l'aide d'une distribution uniforme) à l'intérieur du carré, la probabilité que chacun de ces points se trouvent aussi dans le cercle est

$$p = \frac{\text{aire du cercle}}{\text{aire du carré}} = \frac{\pi}{4}$$

Soit n , le nombre de points tirés aléatoirement se trouvant effectivement dans le cercle, on a :

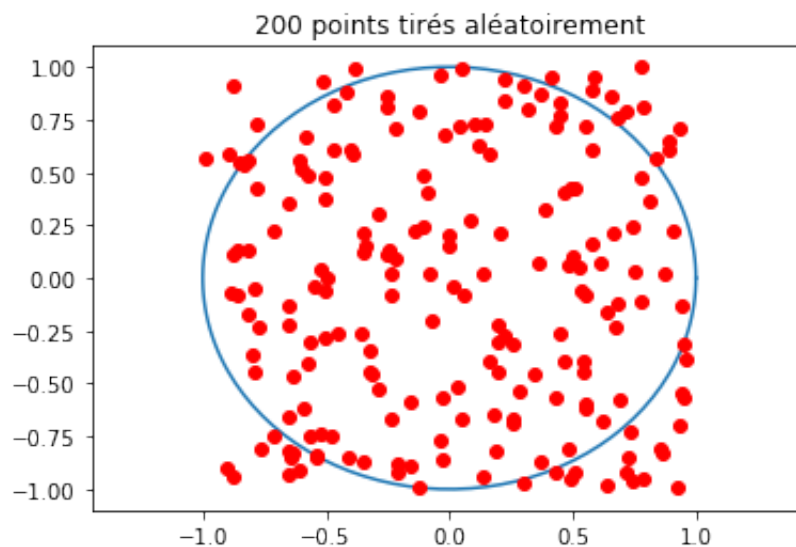
$$p = \frac{n}{N} = \frac{\pi}{4}$$

d'où

$$\pi = 4 \times \frac{n}{N}$$

Ecrire une fonction python qui prend comme argument N , et qui retourne une approximation de π par la méthode suivante : procéder en N itérations et, à chaque itération, choisir aléatoirement les coordonnées d'un point entre -1 et 1 (fonction `uniform` du module `random`), calculer la distance entre ce point et le centre du cercle, déterminer si cette distance est inférieure au rayon du cercle égal à 1, et si c'est le cas, incrémenter le compteur n de 1.

Représenter graphiquement la liste des N points tirés aléatoirement sous le format suivant :



4. On s'intéresse aux densités de probabilité impliquées dans la procédure précédente. Ecrire une fonction qui, comme dans la question précédente, tire N coordonnées (X, Y) avec X, Y uniformes dans $[-1; 1]$. On prendra ici N beaucoup plus grand (~ 50000). Vérifier que la distribution de X et Y est bien uniforme en comparant l'histogramme des valeurs prises par X et Y avec la densité $\rho_X(x) = \rho_Y(x) = 1/2$. On utilisera pour cela la fonction `hist` du module `matplotlib` avec 30 bandes :

```
plt.hist(x, bins=30, density=True)
```

5. On note $R^2 = X^2 + Y^2$ et $R = \sqrt{X^2 + Y^2}$. Vérifier graphiquement que sur $[0; 1]$, on a pour les densités ρ_{R^2} et ρ_R de R^2 et R :

$$\begin{aligned}\rho_{R^2}(0 \leq r \leq 1) &= \frac{\pi}{4} \\ \rho_R(0 \leq r \leq 1) &= \frac{\pi r}{2}\end{aligned}\tag{1}$$

6. *Bonus* : On veut nous-mêmes construire un histogramme sans utiliser la fonction `hist`. Nous allons le faire pour la distribution $Z \sim \mathcal{N}(0, 0.25)$. Pour cela, on partitionne $[-1; 1]$ en un ensemble de $2/b$ bandes de largeur b : $[-1; -1+b], [-1+b; -1+2b], [-1+2b; -1+3b], \dots, [1-2b; 1-b], [1-b; 1]$ (on prend b tel que $2/b$ soit entier).

Ecrire une fonction qui prend un argument un entier N et une largeur b et qui crée l'histogramme des tirages de Z . Pour cela, générer N tirages de Z en utilisant la fonction `gauss(μ, σ)` du module `random` avec $\mu = 0, \sigma = 0.25$. Utiliser un dictionnaire dont les clés sont les bords gauches des bandes $-1, -1+b, \dots, 1-2b, 1-b$ et les valeurs associées sont les nombre de tirages compris dans les bandes correspondantes aux clés. Enfin, utiliser la fonction `bar` de `matplotlib` sur la liste des bords gauches des bandes (liste des clés `edges`) et sur la liste des nombres de tirages associés (liste des valeurs `vals`) pour tracer l'histogramme :

```
plt.bar(edges, vals, width=b, align='edge')
```

4 Le module `numpy.random`

`numpy` vient avec son propre module `random`. Les fonctions de ce module sont sensiblement les mêmes que celles du module `random` de Python (`uniform, binomial, exponential...`), à l'exception de quelques-unes comme la loi normale/Gaussienne (`random` utilise `gauss` tandis que `numpy.random` utilise `normal`). On pourra les comparer dans leurs documentations respectives :

<https://docs.scipy.org/doc/numpy-1.17.0/reference/random/index.html>

<https://docs.scipy.org/doc/numpy-1.17.0/reference/random/generator.html#numpy.random.Generator>

De manière générale, les arguments des fonctions de `numpy.random` sont les mêmes que ceux de `random`, avec cependant l'argument supplémentaire `size`. Celui-ci permet de directement générer des valeurs aléatoires issues d'une distribution donnée dans un tableau `numpy` de taille désirée (type `numpy.ndarray`). Par exemple, pour la distribution uniforme :

- `random.uniform(0, 1)` retourne une unique valeur uniformément tirée entre 0 et 1.
- `np.random.uniform(0, 1, size=10000)` retourne un array de taille 10000 rempli de valeurs uniformément tirées entre 0 et 1.

- `np.random.uniform(0, 1, size=(10,3))` retourne une matrice de taille 10x3 remplie de valeurs uniformément tirées entre 0 et 1.

L'avantage de `numpy.random` est donc de pouvoir générer d'un coup un grand nombre de données aléatoires dans un `numpy.ndarray` de taille choisie. En plus d'être souvent plus rapide que de générer 1 à 1 des valeurs avec `random`, on peut directement utiliser toutes les opérations de `numpy` sur les matrices obtenues. Par exemple, si on veut mettre au carré 1000 tirages d'une distribution uniforme sur `[0;1]`, on écrit avec `random` :

```
x = []
for k in range(1000):
    x.append(random.uniform(0,1)**2)
```

Tandis qu'avec `numpy.random` :

```
x = np.random.uniform(0, 1, size=1000)**2
```

Exercice 4

Réécrire les programmes des questions 3, 4 et 5 de l'exercice 3 en utilisant le module `numpy.random`.