Révisions de Python : objets, syntaxe

1 Calculs élementaires

On affecte une valeur numérique à une variable par une simple affectation = :

```
a = 2
```

On dispose des opérations habituelles : `+,-,*,/`[1]

Attention, la puissance se note `**` et non `^` (ce dernier caractère est un XOR sur les représentations binaires des nombres en jeu... donc on oublie).
Ainsi `2**3` renverra 8 comme espéré, alors que `2^3` renvoie... `1`.

2 Listes

C'est une structure de données essentielle. En Python, une liste est constituée d'objets quelconques (nombres entiers, nombres à virgule, chaînes de caractère, listes...) et une même liste peut contenir des objets de type différents.
Par exemple :

```
L=[1.33,'hello',[1,2],1]
```

est une déclaration valable.
Dans ce qui suit, pour simplifier, les exemples seront construits avec des listes de nombres.

2.1 Définition

2.1.1 Définition explicite et indexation

En Python, on peut créer des listes assez naturellement :

```
L=[2,4,-4]
```

Attention, **l'indexation commence à 0** : dans cette liste `L[0]` vaut 2, `L[1]` vaut 4, `L[2]` vaut -4. `L[3]` n'existe pas : le programme renverra une erreur.
On peut aussi compter à partir de la fin avec des indices négatifs : ainsi `L[-1]` est bien défini, et vaut -4.

On peut considérer des sous-listes de la liste originale avec la syntaxe suivante : si `L` est une liste :

- `L[a:b]` renvoie la liste des éléments de `L` d'indice a, $a+1$, ..., $b-1$.
- `L[a:]` renvoie la liste des éléments de `L` d'indice a, $a+1$, ..., jusqu'à la fin de la liste
- `L[:b]` renvoie la liste des éléments de `L` d'indice 0, 1, ..., $b-1$.

Par exemple,

```
>>> L=[4,2,7,-2,3,1,10,5,0]

>>> L[3:6]
[-2, 3, 1]
```

[1] Attention, si vous utilisez Python 2.xx, la division est par défaut une *division entière,* et renvoie la partie entière du résultat : ainsi `3/4` renverra 0 et non `0.75`. Mais nous utiliserons Python 3, où « tout se passe bien » .

```
>>> L[:4]
[4, 2, 7, -2]

>>> L[6:]
[10, 5, 0]
```

2.1.2 `range`

On peut facilement créer des listes dont les éléments suivent une progression arithmétique. Cette commande `range` aura une grande importance dans les boucles `for`.

L'opérateur `range` permet de créer une liste d'entiers successifs.

```
L1 = list(range(10))
```

crée la liste `L1=[0,1,2,3,4,5,6,7,8,9]`. Attention, il n'y a pas 10 !!

```
L = list(range(1,10))
```

crée la liste `L=[1,2,3,4,5,6,7,8,9]`.
(on peut retenir que les arguments spécifiés sont « le premier qui y est, et le premier qui n'y est pas »).

On peut spécifier un pas :

```
L = list(range(1,10,2))
```

renverra `[1,3,5,7,9]`.

2.1.3 Définition en compréhension

Python comprend aussi une définition de liste dont la syntaxe se rapproche de l'écriture mathématique d'un ensemble. Si on veut créer la liste des k^2 pour $k = 1,2,...,10$, on cherche donc à créer l'ensemble[2]

$$\{k^2 \mid k \in [\![1,10]\!]\}$$

On écrit donc

```
L = [k**2 for k in range(1,11)]          # et non pas (1,10) !!!
```

et on obtient bien `[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]`.
On peut également spécifier une condition : par exemple pour lister les carrés strictement inférieurs à 1000, on peut observer que $50^2 = 2500$, donc considérer les carrés des entiers de 1 à 50 suffira ; et on filtre ensuite ces valeurs avec la condition < 1000.

On écrit alors :

```
L = [k**2 for k in range(1,51) if k**2 < 1000]
```

et on obtient `L=[1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256, 289, 324, 361, 400, 441, 484, 529, 576, 625, 676, 729, 784, 841, 900, 961]`.

2.2 Modification

Une liste est un objet *mutable* : on peut changer ses éléments. Si `L=[3,4,-1]`, la commande

```
L[1]=1000
```

[2] pas exactement, car on veut tenir compte de l'ordre... mais bon on voit quand même l'idée.

modifiera `L` ; on aura après exécution `L=[3,1000,-1]`.

On dispose aussi d'opérateurs qui agissent sur une liste, nommés *méthodes* : la syntaxe de base est

```
L.methode(arguments)
```

Citons :

- Méthode `append` : ajouter un élément `a` à une liste `L` :

  ```
  L.append(a)
  ```

- Méthode `count` : compter le nombre d'éléments ayant une valeur donnée dans une liste `L`.
  Si `L=[2,1,2,-3,2,2,-7]`,

  ```
  L.count(2)
  ```

  renvoie 4 (le nombre de « 2 » dans `L`).

- Méthode `remove` : retire de la liste le premier élément égal à la valeur passée en argument.
  Si `L=[2,1,2,-3,2,2,-7]`,

  ```
  L.remove(2)
  ```

  modifie L en `[1,2,-3,2,2,-7]` ;

  ```
  L.remove(-3)
  ```

  modifie L en `[1,2,2,2,-7]` ;

On a enfin quelques fonctions à connaître :

- `len(L)` renvoie la longueur d'une liste `L` ;
- `del(L[i])` enlève l'élément d'indice `i` de la liste (la liste obtenue a donc un élément en moins).
  Avec `L=[2,1,2,-3,2,2,-7]`, `del(L[3])` modifie la liste L, qui devient `[2,1,2,2,2,-7]`.
- Un test logique pour savoir si un élément est dans une liste :

  ```
  2 in L
  ```

  vaudra `True` si 2 appartient à `L`, et `False` sinon.

2.3 Opérations sur les listes

À retenir : une liste Python n'est pas un vecteur !!
Même si ça y ressemble beaucoup, il manque ce qui constitue la structure d'un espace vectoriel : les opérations.

On a bien en Python une addition de deux listes, et la multiplication d'une liste par un entier positif...

```
>>> L=[1,2,6]
>>> M=[5,3,-1]

>>> L+M
[1, 2, 6, 5, 3, -1]

>>> 3*L
[1, 2, 6, 1, 2, 6, 1, 2, 6]
```

mais il s'agit là de concaténations. On ne peut pas multiplier par autre chose qu'un entier positif : `-L`, `L/2`, ... renvoient des erreurs signifiant qu'on cherche à faire une opérations sur des objets inadaptés.

Il faut en fait voir une liste Python comme rien de plus qu'une collection d'objets délimitée par des crochets.
Pour manipuler des vecteurs au sens de l'algèbre linéaire, voir `numpy` plus bas.

3 if

Attention aux tabulations

Les tabulations sont codantes dans Python !! Cela veut dire qu'*elles font partie intégrante du code*, alors que dans d'autres langages c'est plutôt une convention pour produire du code lisible. Leur absence provoquera une erreur !
Il est impératif que le contenu d'une boucle `for` / `while`, d'une instruction conditionnelle `if`, ou le corps d'une fonction définie par un `def` soit décalé d'une tabulation ou de 4 « Espace » . En contrepartie, le corps des boucles et du `if` ne se terminent pas par un « `end` » : la tabulation délimite le bloc de manière non ambiguë.
Des lignes de code décalées vers la droite par une tabulation seront dites *indentées* (sûrement un anglicisme).

L'instruction conditionnelle `if` fonctionne comme dans les autres langages de programmation... mais ici comme dit ci-dessus il faut indenter (Pyzo le fait automatiquement[3], heureusement).
Les lignes « `if [condition]` » ou « `elif [condition]` » se finissent par « : » .
Il n'y a pas de `then`, ni de `end` à la fin. On peut enchaîner plusieurs conditions avec `elif`, ou conclure avec `else`.

```
if a>2:
    b=1
elif a>1:
    b=2
else:
    b=30000
```

Si on n'indente pas :

```
if a>2:
b=1
```

on obtient l'erreur suivante :

```
IndentationError: expected an indented block
```

4 for

Ici non plus, pas de `end` et on indente.
Quand on commence par « `for k ...` » , k va prendre successivement un ensemble de valeurs. En Python cet ensemble de valeurs est donné par un objet *itérable*.

L'itérable le plus simple est `range` :

```
for k in range(n):
    ...
```

itère sur `range(n)` qui est l'ensemble des valeurs $0,1,2,\ldots,n-1$. Ici il suffit d'utiliser `range(...)` ; `for k in list(range(...))` ne renvoie pas d'erreur mais c'est inutile et personne ne fait comme ça[4].
On peut dès lors utiliser toutes les variations de `range` vues ci-dessus.

On peut aussi itérer sur n'importe quelle liste `L`. Par exemple, si `L=[1,2,-3,1,4,1]`

```
for k in L:
    print(k)
```

affichera successivement tous les éléments de L.

On définira plus bas un « `array` » Numpy : c'est aussi un objet itérable.

[3] Encore mieux : si vous constatez qu'il ne le fait pas, c'est parce que vous avez oublié les « : » .
[4] Argument très important en informatique !

5 `while`

Mêmes conventions syntaxiques : le corps de la boucle est défini par l'indentation, il n'y a pas de `end`, et on met deux points à `while [condition]` :

```
a=1
while a**2 < 20:
    a=a+1

print(a)
```

renvoie le premier entier `a` tel que $a^2 \geqslant 20$ (donc 5...).
On voit donc ici que la ligne `a=a+1` est dans la boucle `while` (car elle est indentée), alors que la ligne `print(a)` est hors de cette boucle (car elle démarre « au même niveau » que le `while`).

6 Définition d'une fonction

Les fonctions sont une pièce essentielle de toute tentative de programmation. C'est un procédé qui reçoit des informations au départ (des *arguments*) et qui délivre un résultat (l'*image* de ces arguments par la fonction).
En Python, la syntaxe est

```
def nom_fonction(arguments):
    ...
    bloc
    d'instructions
    indenté
    ...
    return resultat
```

Par exemple, avec $x \mapsto x^3 - 2x$

```
def f(x):
    return x**3-2*x
```

mais on peut faire mieux qu'une formule et mettre un algorithme. Par exemple, la fonction suivante prend une liste `L` et un réel `a` et renvoie le nombre d'éléments de `L` supérieurs ou égaux à `a` :

```
def superieur(L,a):
    c=0 # compteur
    for l in L:
        if l>=a:
            c=c+1
    return c
```

Il peut y avoir plusieurs instructions `return` dans un code, qui correspondent essentiellement à divers cas de figure pour le calcul de l'image. Toutefois une fonction ne renvoie qu'un seul résultat, donc le processus se terminera dès qu'un `return` est rencontré.
Par exemple, on veut coder la fonction qui associe à x : x^2 si $x < 0$ ou $1 - x$ si $x \geqslant 0$. Une manière de programmer cela est la suivante :

```
def f2(x):
    if x<0:
        return x**2
    else:
        return 1-x
```

En fait, mettre un `else` est inutile et la version suivante fonctionne :

```
def f2(x):
    if x<0:
        return x**2
    return 1-x
```

En effet si $x < 0$ le programme exécute « `return x**2` » ce qui renvoie la valeur x^2 et termine la fonction ; le programme ne rencontrera pas l'instruction « `return 1-x` » . Par contre, si $x \geqslant 0$, le contenu du `if` n'est pas lu ; le programme va directement chercher « `return 1-x` » et renvoie donc $1 - x$.
C'est une manière plus élégante de procéder. Vous pourriez objecter qu'on n'est pas là pour faire de l'art, mais en fait cette méthode de coder est la plus répandue, et sera très certainement celle qui sera présente dans les sujets de concours. Il faut donc s'y habituer et la mettre en pratique.

7 Packages

En Python, un package est un catalogue de fonctions et de types d'objets spécifiques, qui ne sont pas chargés par défaut dans le programme. Il faut donc les *importer* si on en a besoin.

Numpy est par exemple un package dont nous nous servirons beaucoup. On peut l'importer avec

```
import numpy
```

On dispose alors, notamment, de fonctions mathématiques usuelles : la racine carrée se note `sqrt`, MAIS comme elle vient de `numpy`, elle se note en fait `numpy.sqrt`.

Ce préfixe rend la syntaxe un peu complexe, mais est bien utile car on peut avoir un conflit de nom entre plusieurs fonctions venant de packages différents.

Pour se défaire du préfixe, on peut importer les commandes une par une :

```
from numpy import sqrt
```

importe seulement la racine carrée, qui sera appelée cette fois par `sqrt` « tout court » .
On peut en fait importer toutes les commandes d'un package par :

```
from numpy import *
```

et ici toutes les commandes `numpy` seront accessibles sans préfixe. Comme dit c'est un peu dangereux et on évite généralement cette méthode.

On peut aussi raccourcir le préfixe en donnant un alias au package importé avec `as` :

```
import numpy as np
```

importe `numpy` et ses commandes, mais change le préfixe « `numpy.` » en « `np.` » .
Avec cet import, la racine carrée s'appelle par `np.sqrt`
Nous adopterons cette dernière solution : les packages sont importés avec un alias simple (imposé par le programme... et les traditions).

7.1 Numpy

`numpy` est le package essentiel pour faire des mathématiques avec Python.
Nous l'importerons toujours par

```
import numpy as np
```

Parmi les nouveaux objets accessibles après import, il y a un objet proche d'une liste dénommé `array` (et donc, en fait, `np.array`). C'est un objet itérable : on peut définir une boucle `for` avec (voir la section sur la boucle `for`).

On peut convertir une liste usuelle en `np.array` de la manière suivante :

```
>>> np.array([1,2,3])
array([1, 2, 3])
```

Pour des représentations graphiques de suites, on voudra souvent créer les listes `[0,1,...,n]` ou `[1,2,...,n]` (obtenues usuellement par `range`) sous forme d'`array` : pour cela on dispose de la commande `np.arange` qui fonctionne avec la même syntaxe.

```
np.arange(10)
```

crée l'array : `array([0,1,2,3,4,5,6,7,8,9])`.

Un `np.array` est donc essentiellement une liste, mais les opérations définies dessus permettent de s'en servir comme représentation d'un vecteur.

- Opérations d'espace vectoriel : addition / multiplication par un scalaire:

```
>>> L=np.array([1,2,6])

>>> M=np.array([5,3,-1])

>>> L+M
array([6, 5, 5])

>>> 3*L
array([ 3,  6, 18])

>>> -M/5
array([-1. , -0.6,  0.2])
```

- Matrices : une matrice est codée comme un `np.array` qui est la liste de ses lignes. Par exemple, $M = \begin{pmatrix} 1 & 2 & -1 \\ 2 & 3 & 2 \\ 0 & 0 & 1 \end{pmatrix}$ sera codée par

```
M=np.array([[1,2,-1],[2,3,2],[0,0,1]])
```

Un appel à M dans la console renvoie le résultat sous forme présentable :

```
>>> M
array([[ 1,  2, -1],
       [ 2,  3,  2],
       [ 0,  0,  1]])
```

On peut additionner deux matrices de même format et multiplier une matrice par un réel avec les mêmes commandes que précédemment. Le format d'une matrice `M` (nombre de lignes, nombre de colonnes) est renvoyé par `np.shape(M)`.

Le produit matriciel **n'est pas** obtenu avec `*` : celui-ci effectue un produit composante par composante. Il existe deux commandes : la commande `np.dot(A,B)` (au programme) et la syntaxe `A@B` (pour Python 3.5 et suivantes).
Il existe aussi une commande sous forme de méthode pour calculer le produit AB : `A.dot(B)`. Ceci n'est pas au programme.

```
>>> A=np.array([[0,1,-6,4],[2,1,1,1]])
array([[ 0,  1, -6,  4],
       [ 2,  1,  1,  1]])

>>> B=np.array([[0,1,4],[1,-1,1],[5,0,4],[3,3,-3]])
array([[ 0,  1,  4],
       [ 1, -1,  1],
       [ 5,  0,  4],
       [ 3,  3, -3]])

>>> np.dot(A,B)
array([[-17,  11, -35],
       [  9,   4,  10]])

>>> A@B
```

```
array([[-17,  11, -35],
       [  9,   4,  10]])

>>> A.dot(B)
array([[-17,  11, -35],
       [  9,   4,  10]])
```

(rappel : il existe des matrices non-carrées !)

L'inverse d'une matrice s'obtient avec le package `numpy.linalg` : voir ci-dessous.

- On dispose d'une opération mathématiquement discutable mais bien pratique : si `M` est une matrice `np.array()` et a un nombre, l'addition `M+a` ajoute a à toutes les composantes de M.

```
M=np.array([[1,2,-1],[2,3,2],[0,0,1]])

>>> M
array([[ 1,  2, -1],
       [ 2,  3,  2],
       [ 0,  0,  1]])

>>> M+2
array([[3, 4, 1],
       [4, 5, 4],
       [2, 2, 3]])
```

- La transposition se note `np.transpose(M)` (ou `M.transpose()`, hors-programme).
- On dispose de raccourcis pour créer des matrices particulières (toujours en format `np.array`) :
  `np.zeros((m,n))` renvoie une matrice $m \times n$ remplie de zéros.
  (NB : observer les deux parenthèses : la fonction `np.zeros` prend *un argument* qui est *le couple* `(m,n)` ; et non les deux arguments `m` et `n`).
  `np.ones((m,n))` renvoie une matrice $m \times n$ remplie de 1.
  `np.eye(n)` renvoie la matrice identité I_n.

On dispose enfin des fonctions mathématiques usuelles (qui *n'existent pas* dans Python avant import) :

- `np.exp`, `np.log` (c'est le logarithme népérien ln), `np.abs`, `np.sqrt`, `np.floor` (partie entière).
- On a les deux constantes mathématiques usuelles (`np.e`, `np.pi`).

7.2 numpy.linalg

C'est un sous-package de `numpy`. Si on a importé `numpy` avec l'alias `np`, les commandes de ce package sont accessibles avec le préfixe `np.linalg.[nom de fonction]`.
On peut toutefois décider de mettre un alias sur ce sous-package aussi (ce qui sera fait traditionnellement) :

```
import numpy.linalg as al
```

Ceci nous donne accès à des fonctions d'algèbre linaire plus poussées :

- inversion d'une matrice (`al.inv(M)`)
- mise d'une matrice à la puissance n (`al.matrix_power(M,n)`)
- calcul du rang d'une matrice (`al.matrix_rank(M)`)
- recherche des valeurs / vecteurs propres (`al.eig(M)`).
  Pour le nom : en anglais, valeur propre se dit eigenvalue et vecteur propre eigenvector (de l'allemand eigen : propre).

Cette dernière renvoie un array contenant les valeurs propres et une matrice de passage de la base canonique vers une base de vecteurs propres en cas de diagonalisabilité.

```
>>> M=np.array([[1,2],[0,2]])
array([[1, 2],
       [0, 2]])

>>> (vap,vep)=al.eig(M)

>>> vap
array([1., 2.])

>>> vep
array([[1.        , 0.70710678],
       [0.        , 0.70710678]])
>>> al.inv(vep)@M@vep
array([[1., 0.],
       [0., 2.]])
```

7.3 numpy.random

C'est un package permettant de simuler des tirages de variables aléatoires suivant des lois usuelles. On l'importera par :

```
import numpy.random as rd
```

De manière générale, la syntaxe permettant de renvoyer un ensemble de tirages aléatoires indépendants suivant une loi usuelle est

```
rd.[nom de la loi](paramètres, format)
```

Le paramètre « format » peut être un nombre `k`, auquel cas le résultat sera une liste à `k` éléments, ou un couple de deux entiers non nuls `(n,p)`, auquel cas le résultat sera une matrice de $\mathcal{M}_{n,p}(\mathbb{R})$. Il est optionnel : par défaut on renvoie une seule valeur.
Attention, passer un format égal à 10 ou un format égal à `(1,10)` ne renvoie pas le même résultat !! (essayez, et comptez les crochets).

- `rd.random()` renvoie un tirage d'une variable aléatoire suivant la loi uniforme sur $[0,1]$.
  Si on passe un format, on obtient une matrice de tirages indépendants : par exemple, `rd.random(4)` renvoie une ligne de 4 tirages uniformes sur $[0,1]$; et `rd.random((2,4))` (noter les deux parenthèses) une matrice 2×4 de tirages uniformes sur $[0,1]$.
- `rd.binomial(n,p,format)` renvoie des tirages de $X \hookrightarrow \mathcal{B}(n,p)$.

  ```
  >>> rd.binomial(10,0.3,(2,3))
  array([[2, 1, 4],
         [6, 4, 4]])
  ```

- `rd.randint(a,b,format)` : idem avec la loi $\mathcal{U}([\![a, b-1]\!])$.
- `rd.geometric(p,format)` : idem avec la loi $\mathcal{G}(p)$.
- `rd.poisson(lambda,format)` : idem avec la loi $\mathcal{P}(\lambda)$.
- `rd.uniform(a,b,format)` : idem avec la loi à densité $\mathcal{U}([a,b])$.
- `rd.exponential(beta,format)` : idem avec la loi $\mathcal{E}(\lambda)$ avec $\lambda = \dfrac{1}{\beta}$ (le paramètre à fournir est donc $1/\lambda$, c'est l'espérance de $\mathcal{E}(\lambda)$).
- `rd.normal(m,s,format)` : idem avec la loi $\mathcal{N}(m,s^2)$. Les paramètres à passer sont l'espérance et l'écart-type.

7.4 Matplotlib

Matplotlib est un package graphique de Python. Il permet d'afficher des nuages de points, des représentations graphiques de fonctions, des diagrammes en bâtons, des histogrammes, etc.

On importe en fait traditionnellement le sous-package `pyplot` :

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

7.4.1 Tracé d'une ligne brisée

La commande de base est `plt.plot` : elle reçoit une liste d'abscisses $[x_1,\dots,x_n]$ et une liste d'ordonnées $[y_1,\dots,y_n]$ et renvoie la ligne brisée joignant les points $(x_1,y_1),(x_2,y_2),\dots,(x_n,y_n)$ (dans cet ordre).

7.4.2 Tracé d'une courbe représentative

Pour tracer la courbe représentative d'une fonction sur un intervalle $[a,b]$, on a besoin d'une liste d'abscisses qui commence par a, finit par b, et comprend un grand nombre de points régulièrement espacés. La commande

```
np.linspace(a,b,n)
```

renvoie un `array` de n réels régulièrement espacés, dont le premier vaut a et le dernier vaut b.

```
>>> np.linspace(0,1,11)
array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1. ])
```

Ensuite, `X` étant un `array` d'abscisses, toute composée de fonctions `numpy` usuelles agit composante par composante sur X, ce qui rend aisé les représentations graphiques. Par exemple, la courbe de $x\mapsto e^{-x^2/2}$ sur $[-3,3]$ peut être obtenue par :

```
X = np.linspace(-3, 3, 1000)
plt.plot(X,np.exp(-X**2/2))
plt.show()
```

Pour des fonctions « maison » qui ne se présentent pas comme des fonctions `numpy`, il est possible qu'on ne puisse pas prendre directement `f(X)` où `X` est l'array des abscisses. Il existe un moyen de contourner cela rapidement[5] avec la définition en compréhension :

```
def f(x):
    if x<0:
        return 0
    return np.exp(-x)


X = np.linspace(-3, 3, 1000)
# plt.plot(X,f(X)) ne fonctionne pas ici
# on crée la liste des ordonnées à la main
Y = np.array([f(x) for x in X]) # définition en compréhension
plt.plot(X,Y)
plt.show()
```

7.4.3 Nuage de points

La commande `plt.scatter` reçoit une liste d'abscisses $[x_1,\dots,x_n]$ et une liste d'ordonnées $[y_1,\dots,y_n]$ et renvoie le nuage des points $(x_1,y_1),(x_2,y_2),\dots,(x_n,y_n)$.

[5] mais on peut aussi, sans profiter des astuces pythonnesques, utiliser une simple boucle `for`...

```
X = rd.random((1,100))
Y = rd.random((1,100))
plt.scatter(X,Y)
plt.show()
```

7.4.4 Diagramme en bâtons

`plt.bar` prend en argument une liste d'abscisses $[x_1,\dots,x_n]$ et une liste de hauteurs $[y_1,\dots,y_n]$ et renvoie un diagramme en bâtons, où le bâton à l'abscisse x_i a pour hauteur y_i.

```
plt.bar([1,2,3],[1,3,-1])
plt.show()
```

7.4.5 Histogramme

`plt.hist` permet de tracer des histogrammes.
Il s'agit de représenter graphiquement une liste de valeurs en les répartissant dans des classes (exemple : nombre de valeurs dans l'intervalle $[1,2[$, dans l'intervalle $[2,3[$, ...) et de représenter graphiquement chaque classe par un rectangle dont la surface est proportionnelle au nombre de valeurs dans cette classe.
On passe donc en arguments :

- Une liste de valeurs (sous forme de liste ou de `np.array`).
- un nombre n de classes (égal à 10 par défaut) : les valeurs seront alors regroupées en n classes de même largeur ;
- OU une liste de valeurs délimitant les classes.

Par exemple,

```
plt.hist(values, [1,3,5,10])
```

définit 3 classes : $[1,3[$, $[3,5[$, $[5,10]$. Les valeurs hors de ces 3 classes ne sont pas prises en compte.

Une option assez courante dans un histogramme est d'exiger que la somme des aires des rectangles soit égale à 1 (pour des raisons qui apparaîtront plus tard dans l'année) : pour cela il fait passer l'option `density=True`.

7.5 Pandas

`pandas` est un package Python permettant de traiter des données ; essentiellement il traite des fichiers `.csv` (*comma-separated values*) ou des feuilles Excel.
Il permet l'import, l'export de ces données et fournit quelques outils rudimentaires de statistiques (moyenne, médiane, etc.)

La convention d'import est

```
import pandas as pd
```

7.5.1 Le DataFrame

L'objet principal introduit par `pandas` est le `DataFrame` : grosso modo c'est un tableau amélioré. Notons quelques caractéristiques :

- Dans un DataFrame, chaque colonne a un format de données propre (on peut avoir une colonne d'entiers, une colonne de texte, une colonne de nombres à virgule, etc...)
- La première colonne d'un DataFrame est une numérotation des lignes (qui commence à 0).
- La première ligne n'est pas numérotée et est faite pour contenir les « titres » des colonnes.

```
import pandas as pd

df = pd.DataFrame([['Guerre Et Paix',25,1657],\
['Les Misérables',20,1344],\
['Fahrenheit 451',12,290]]\
,columns=['Titre', 'Prix','Nombre de pages'])
```

(NB : le backslash \ permet d'aller à la ligne dans une ligne de code) renvoie le DataFrame :

```
>>> df
            Titre  Prix  Nombre de pages
0  Guerre Et Paix    25             1657
1  Les Misérables    20             1344
2  Fahrenheit 451    12              290
```

On voit ici que la première colonne a été ajoutée automatiquement, et numérote les éléments du tableau. Les colonnes ont un titre spécifié par l'argument `columns` ; la première (l'index) n'a pas de titre.
Si `df` est le nom du DataFrame créé, on peut ensuite appeler chaque colonne avec la syntaxe `df[nom_colonne]` (on traite le nom d'une colonne comme un indice):

```
>>> df['Prix']
0    25
1    20
2    12
Name: Prix, dtype: int64
```

La colonne d'index est rappelée ; l'attribut `dtype` est le type de données contenues dans cette colonne : ici des entiers (codés sur 64 bits).

7.5.2 Outils statistiques

Des commandes permettent de récupérer des informations sur un DataFrame :

- `df.shape` renvoie la taille du DataFrame nommé `df` (nb de lignes, nb de colonnes). La colonne d'indices, et la ligne contenant les titres des colonnes ne sont pas décomptées. Ainsi, le DataFrame défini ci-dessus a pour taille `(3,3)`.
- `df.head()` renvoie les 5 premières lignes (donc la ligne de titre, + les 4 premières lignes de valeurs) ; on peut avoir les n premières lignes avec `df.head(n)`
- `df.describe()` renvoie des indicateurs statistiques sur les colonnes de nombres :

```
>>> df.describe()
             Prix  Nombre de pages
count    3.000000         3.000000
mean    19.000000      1097.000000
std      6.557439       716.190617
min     12.000000       290.000000
25%     16.000000       817.000000
50%     20.000000      1344.000000
75%     22.500000      1500.500000
max     25.000000      1657.000000
```

- `df.mean()` renvoie un tableau contenant la moyenne de chaque colonne de nombres :

```
>>> df.mean()
Prix                 19.0
Nombre de pages    1097.0
dtype: float64
```

(NB : ici on est passé en représentation de nombres à virgule `float64`).

Idem avec `df.std()` (écart-type) ; `df.median()` (médiane).

On dispose d'un outil de tri des lignes suivant une ou plusieurs[6] colonnes :

[6] ce qui signifie : trier selon les valeurs de la 1ère colonne, et départager les ex-aequo avec les valeurs de la seconde colonne.

```
>>> df.sort_values('Prix')
            Titre  Prix  Nombre de pages
2  Fahrenheit 451    12              290
1  Les Misérables    20             1344
0  Guerre Et Paix    25             1657
```

On peut passer une option `ascending=False` pour trier dans l'ordre décroissant.

7.5.3 Import de fichier

On se doute qu'il est plus intéressant de récupérer les données d'un fichier (un tableur par exemple) que de construire un DataFrame à la main.

On considérera ici des fichiers CSV. Outre le nom du fichier (exemple : `test.csv`), Python doit être capable de le localiser sur le disque. Il existe pour cela deux méthodes :

- Changer le *répertoire de travail* : c'est le répertoire où Python ira chercher vos fichiers. Pour cela, on utilise la commande `chdir` (*change directory*) du package `os` (*operating system*) :

  ```
  import os
  os.chdir("C:\\Documents") # mettre l'adresse du répertoire contenant votre fichier
  ```

  (NB : et non pas `"C:\Documents"` car le caractère \ est interprété de manière spécifique dans une chaîne de caractères Python) ;

- Utiliser la fonctionnalité « Démarrer le script » de Pyzo (Ctrl-Shift-E). Ceci relance une console Python et spécifie le répertoire de travail comme étant celui où se situe votre script. Pour que Python trouve votre fichier, il suffit alors de le placer dans le même répertoire que le script que vous écrivez.
  Il suffit de faire cela une fois : une fois que la console est calée sur le bon répertoire elle y reste.

La commande est

```
df = pd.read_csv('nom_du_fichier.csv')
```

Dans certains fichiers `csv`, les valeurs ne sont pas séparées par des virgules mais par un autre délimiteur (point-virgule notamment). On peut passer le délimiteur à considérer en argument :

```
df = pd.read_csv('nom_du_fichier.csv',delimiter=';')
```

Par défaut, `pandas` considérera que la première ligne du fichier `csv` contient le titre des colonnes, et il les rentrera comme telles dans le DataFrame. Si ce n'est pas le cas il faut mettre une option `header=None`. Dans ce cas les colonnes sont numérotées automatiquement 0,1,... et toutes les lignes sont prises comme des données.