ANNEXE

A

RAPPELS D'INFORMATIQUE

Sommaire

A - Suites et séries

■ Comment manipuler une suite ?

SF1 Définir explicitement une suite

Le cas d'une suite définie explicitement est en tout point semblable à celui d'une fonction.

Exemple

Définissons la suite u donnée, pour tout $n \in \mathbb{N}^*$, par $u_n = \frac{1}{n^2+1}$.

```
def u(n):
    return 1/(n**2+1)
```

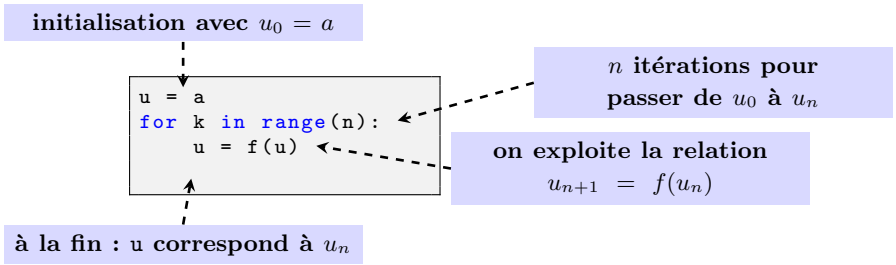
En effet :

```
>>> u(2)
0.2
>>> u(5)
0.038461538461538464
```

SF2 Définir une suite par récurrence

Considérons le cas d'une suite définie par une valeur initiale $u_0 = a$ et une relation : $\forall n \in \mathbb{N},\ u_{n+1} = f(u_n)$.

On définit une variable initialisée à a puis, au sein d'une boucle `for`, si l'on connaît à l'avance le nombre d'itérations (ou d'une boucle `while` si le nombre d'itérations est conditionné par un test), on remplace successivement les valeurs de la variable par les termes de la suite.



Exemples

1 ▹ Considérons la suite u définie par $u_0 = 2$ et : $\forall n \in \mathbb{N},\ u_{n+1} = \sqrt{1+u_n}$.

Écrivons une fonction d'en-tête `u(n)` qui renvoie la valeur de u_n.

```
def u(n):
    v = 2
    for k in range(n):
        v = sqrt(1 + v)
    return v
```

Ici, u est le nom de la fonction alors que v n'est qu'une variable locale à l'intérieur de la fonction.

```
>>> u(2)
1.6528916502810695
>>> v
Traceback (most recent call last):
  File "<console>", line 1, in <module>
NameError: name 'v' is not defined
```

2 ▹ Considérons la suite de Fibonacci définie par $F_0 = 0$, $F_1 = 1$ et : $\forall n \in \mathbb{N}$, $F_{n+2} = F_{n+1} + F_n$.

On vérifie que cette suite diverge vers $+\infty$.

Déterminons le premier terme de cette suite qui soit supérieur à 1000.

```
F = [0, 1]  # il est plus efficace de travailler avec deux termes successifs
while F[1] < 1000:
    s = F[1]
    t = F[0] + F[1]
    F = [s, t]
```

On trouve :

```
>>> F[1]
1597
```

SF3 Donner les premiers termes d'une suite

◇ Dans le cas d'une suite $\left(f(n)\right)_{n\in\mathbb{N}}$ donnée explicitement à l'aide d'une fonction f, on peut directement calculer l'image du vecteur correspondant aux entiers de 0 à n.

Exemple

Définissons une liste contenant les 10 premiers termes de la suite $\left(\frac{1}{n+1}\right)_{n\in\mathbb{N}}$.

```
>>> [1/(n+1) for n in range(10)]
[1.0, 0.5, 0.3333333333333333, 0.25, 0.2, 0.1666666666666666, 0.14285714285714285,
    0.125, 0.1111111111111111, 0.1]
```

On peut également utiliser une boucle en modifiant une liste existante :

```
L = [0]*10
for n in range(10):
    L[n] = 1/(n+1)
```

◇ Dans le cas d'une suite donnée par une relation de récurrence, il suffit de construire pas à pas une liste en exploitant les derniers termes ajoutés ou bien de l'initialiser à la bonne taille puis de modifier ses termes.

Exemple

Définissons une fonction d'en-tête `F(n)` renvoyant la liste des termes d'indice 0 à n de la suite de Fibonacci.

```
def F(n):
    L = [0, 1]
    for k in range(2, n):
        L.append(L[-1] + L[-2])
    return L
```

```
def Fbis(n):
    L = [0]*n
    L[1] = 1
    for k in range(2,n):
        L[k] = L[k-1] + L[k-2]
    return L
```

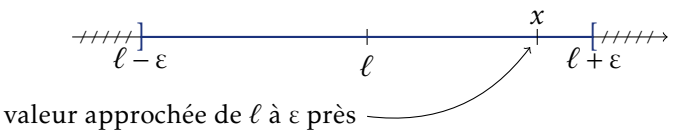
Ce qui donne :

```
>>> F(10)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
>>> Fbis(10)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

■ Comment obtenir une valeur approchée d'une limite de suite ?

SF4 Comprendre le principe d'une approximation

Le principe de toute approximation réside sur le fait que, pour tout réel $\varepsilon > 0$, tout réel x dans l'intervalle $]\ell-\varepsilon, \ell+\varepsilon[$ vérifie $|\ell - x| < \varepsilon$ donc fournit une approximation de ℓ à ε près (c'est-à-dire avec une erreur d'au plus ε).



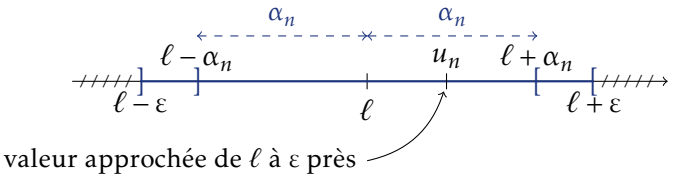
SF5 Exploiter une majoration de l'erreur

Considérons une suite $(u_n)_{n\in\mathbb{N}}$ et un réel ℓ tels qu'il existe une suite $(\alpha_n)_{n\in\mathbb{N}}$ vérifiant :

$$\forall n \in \mathbb{N},\ |u_n - \ell| \leqslant \alpha_n \quad \text{et} \quad \alpha_n \xrightarrow[n\to+\infty]{} 0.$$

Il résulte tout d'abord du théorème d'existence de limite par encadrement que la suite $(u_n)_{n\in\mathbb{N}}$ est convergente de limite ℓ.

Par ailleurs, pour tout réel $\varepsilon > 0$, **il suffit** que $\alpha_n < \varepsilon$ pour que u_n fournisse une valeur approchée de ℓ à ε près.



Exemple

Considérons la suite $(x_n)_{n\in\mathbb{N}}$ définie par $x_0 = 7$ et : $\forall n \in \mathbb{N},\ x_{n+1} = \dfrac{x_n - 1}{\ln(x_n) - 1}$.

On montre que la suite $(x_n)_{n\in\mathbb{N}}$ est convergente et, en notant ℓ sa limite et à l'aide de l'inégalité des accroissements finis, on obtient :

$$\forall n \in \mathbb{N},\ |x_n - \ell| \leqslant \left(0,17\right)^n.$$

Déduisons-en une fonction d'en-tête `approx(eps)` donnant une valeur approchée de ℓ à `eps` près.

```
def f(x):
    return (x-1)/(log(x)-1)

def approx(eps):
    x = 7
    n = 0
    while (0.17)**n > eps:
        x = f(x)
        n = n+1
    return x
```

```
def approxbis(eps): # alternative
    x = 7
    maj = 1
    n = 0
    while maj > eps:
        x = f(x)
        n = n+1
        maj = maj * 0.17
    return x
```

On trouve par exemple :

```
>>> approx(1e-5)
6.305395279271691
```

En fonction de l'expression de α_n, il arrive que l'on puisse explicitement déterminer un entier n tel que $\alpha_n < \varepsilon$ auquel cas on peut ensuite utiliser une boucle `for`.

Exemple

Reprenons l'exemple précédent pour illustrer cette idée.

On a :

$$\begin{aligned}(0,17)^n \leqslant \varepsilon &\iff n\ln(0,17) \leqslant \ln(\varepsilon)\\ &\iff n \geqslant \frac{\ln(\varepsilon)}{\ln(0,17)}.\end{aligned}$$

On en déduit le programme suivant :

```
def approxter(eps):
    n = ceil(log(eps) / log(0.17))
    x = 7
    for k in range(n):
        x = f(x)
    return x
```

SF6 Utiliser des suites adjacentes

On rappelle que deux suites sont dites *adjacentes* lorsque l'une est croissante, l'autre décroissante et l'écart entre les deux converge vers 0.

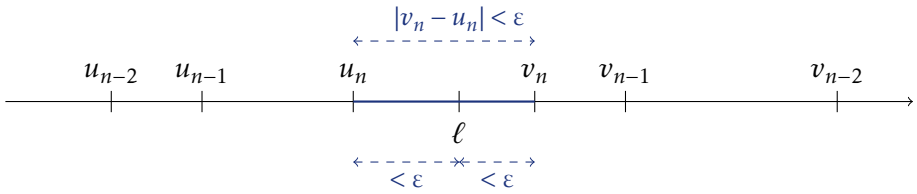
Pour fixer les idées, considérons le cas d'une suite $(u_n)_{n\in\mathbb{N}}$ croissante et d'une suite $(v_n)_{n\in\mathbb{N}}$ décroissante telles que :

$$|v_n - u_n| \xrightarrow[n\to+\infty]{} 0.$$

Dans ces conditions, les deux suites convergent et leur limite est la même. De plus, en notant ℓ cette limite, on a :

$$\forall n \in \mathbb{N},\ u_n \leqslant \ell \leqslant v_n.$$

Il s'ensuit que, pour tout réel $\varepsilon > 0$, il suffit que $|v_n - u_n| < \varepsilon$ pour que tout réel entre u_n et v_n fournisse une valeur approchée de ℓ à ε près.



Exemple

On définit les deux suites u et v par $u_0 = 1$, $v_0 = \sqrt{2}$ et :

$$\forall n \in \mathbb{N},\ u_{n+1} = \frac{u_n + v_n}{2} \quad \text{et} \quad v_{n+1} = \sqrt{u_n v_n}.$$

On admet que ces deux suites sont adjacentes.

Déduisons-en une fonction d'en-tête `approx(eps)` renvoyant une valeur approchée de la limite commune de ces deux suites à eps près.

```
def approx(eps):
    u = 1
    v = sqrt(2)
    while abs(v-u) > eps:
        u, v = (u+v)/2, sqrt(u*v)
    return (u+v)/2
```

On obtient par exemple :

```
>>> approx(1e-5)
1.1981402347355923
```

■ Comment manipuler une série ?

SF7 Calculer une somme partielle d'une série

Le principe du calcul d'une somme partielle $S_n = \sum_{k=0}^{n} u_k$ est celui du calcul d'une suite définie par une relation de récurrence puisque l'on a $S_0 = u_0$ et, pour tout $n \in \mathbb{N}$, $S_{n+1} = S_n + u_{n+1}$.

Il convient donc d'initialiser une variable S à 0 puis de calculer S_n à l'aide d'une boucle.

On peut également initialiser S avec u_0 puis n'effectuer que n itérations au lieu de $n+1$. Il convient bien entendu d'adapter également le nombre d'itérations si la somme ne débute pas à l'indice 0.

Exemple

Définissons une fonction d'en-tête `somme(n)` renvoyant $\sum_{k=1}^{n} \frac{1}{k^2}$.

Si l'on cherche à déterminer la première somme partielle inférieure ou supérieure à une valeur donnée alors on ne connaît pas à l'avance le nombre d'itérations donc on doit utiliser une boucle `while`.

Exemple

On a vu que : $\sum_{k=1}^{n} \frac{1}{k} \xrightarrow[n\to+\infty]{} +\infty$.

Déterminons le plus petit entier $n \geqslant 1$ tel que $\sum_{k=1}^{n} \frac{1}{k} > 10$.

SF8 Calculer un «produit partiel»

Le calcul d'un produit partiel $P_n = \prod_{k=0}^{n} u_k$ est semblable à celui d'une somme partielle si ce n'est que les sommes sont remplacées par des produits et que l'on initialise avec la valeur 1 au lieu de 0.

Exemple

Définissons une fonction d'en-tête `produit(n)` renvoyant $\prod_{k=1}^{n} \ln\left(1+\frac{1}{k^2}\right)$.

SF9 Obtenir une valeur approchée d'une somme de série

Considérons une série de terme général $(u_n)_{n\in\mathbb{N}}$ et notons pour tout $n \in \mathbb{N}$:

$$S_n = \sum_{k=0}^{n} u_k.$$

Dans le cas où la série est convergente, de somme $S = \sum_{k=0}^{+\infty} u_k$, on peut définir le reste d'ordre $n \in \mathbb{N}$:

$$R_n = \sum_{k=n+1}^{+\infty} u_k = S - S_n$$

et la suite $(R_n)_{n\in\mathbb{N}}$ converge donc vers 0.

Si l'on dispose d'une majoration du reste d'ordre n, alors on est dans une situation vue dans le cadre des suites.

Exemple

La série de terme général $\frac{1}{n!}$ converge et a pour somme $\sum_{k=0}^{+\infty} \frac{1}{k!} = e$.

Posons, pour tout entier $n \geqslant 1$: $S_n = \sum_{k=0}^{n} \frac{1}{k!}$.

On montre que l'on a : $\forall n \in \mathbb{N}^*,\ |e - S_n| \leqslant \frac{2}{n!}$.

Écrivons une fonction d'en-tête `approx(eps)` renvoyant une valeur approchée de e à eps près.

Considérons le cas particulier où u_n est de la forme $u_n = (-1)^n a_n$ avec $(a_n)_{n\in\mathbb{N}}$ une suite décroissante de limite nulle (donc une suite positive ou nulle). D'après le critère spécial de convergence des séries alternées (hors programme mais connu), la série de terme général u_n est dans ce cas convergente.

Pour démontrer ce résultat, on montre en fait que les suite $(S_{2n})_{n\in\mathbb{N}}$ et $(S_{2n+1})_{n\in\mathbb{N}}$ sont deux suites adjacentes. On se trouve donc dans une situation connue pour les suites.

Exemple

La série de terme général $\dfrac{(-1)^{n+1}}{n}$ converge et a pour somme $\displaystyle\sum_{k=1}^{+\infty} \frac{(-1)^{k+1}}{k} = \ln(2)$.

Posons, pour tout entier $n \in \mathbb{N}$: $S_n = \displaystyle\sum_{k=1}^{n} \frac{(-1)^{k+1}}{k}$.

Dans ces conditions, on vérifie que les suites $\left(\displaystyle\sum_{k=1}^{2n} \frac{(-1)^{k+1}}{k}\right)_{n\in\mathbb{N}}$ et $\left(\displaystyle\sum_{k=1}^{2n+1} \frac{(-1)^{k+1}}{k}\right)_{n\in\mathbb{N}}$ sont adjacentes.

Écrivons une fonction d'en-tête `approx(eps)` renvoyant une valeur approchée de $\ln(2)$ à eps près.

B - Approximations

■ Comment approcher les solutions d'une équation de la forme $f(x) = 0$?

En général, les méthodes utilisent des suites définies par récurrence et exploitent la continuité de la fonction f. Cependant, le seul algorithme à maîtriser de façon autonome est le suivant.

SF10 Programmer une recherche dichotomique

Considérons une fonction f *continue* sur un intervalle $[a,b]$ et telle que $f(a)$ et $f(b)$ soient de signes opposés.

D'après le théorème des valeurs intermédiaires, il existe au moins un réel en lequel f s'annule.

Pour déterminer une valeur approchée d'un tel réel (mais c'est également l'idée d'une démonstration possible du théorème des valeurs intermédiaires), on peut utiliser l'***algorithme de dichotomie*** qui consiste à définir deux suites $(a_n)_{n\in\mathbb{N}}$ et $(b_n)_{n\in\mathbb{N}}$ de la façon suivante :

- on pose $a_0 = a$ et $b_0 = b$;
- une fois $a_0,\dots,a_k$ et $b_0,\dots,b_k$ construits, on considère $f\left(\frac{a_k+b_k}{2}\right)$:
  - si $f\left(\frac{a_k+b_k}{2}\right)$ est du même signe que $f(b_k)$ alors on pose $b_{k+1} = \frac{a_k+b_k}{2}$ et $a_{k+1} = a_k$;
  - sinon, on pose $a_{k+1} = \frac{a_k+b_k}{2}$ et $b_{k+1} = b_k$.

Dans ces conditions, $(a_n)_{n\in\mathbb{N}}$ et $(b_n)_{n\in\mathbb{N}}$ sont deux suites adjacentes.

La continuité de f sur $[a,b]$ permet d'en déduire que ces suites convergent vers un réel x_0 vérifiant $f(x_0) = 0$.

```
def dichotomie(f, a, b, epsilon):
    g, d = a, b  # bornes de gauche et de droite, initialement a et b
    while abs(g-d) > epsilon:
        m = (g+d)/2
        if f(g) * f(m) < 0:  # si f(g) et f(m) sont de signes opposés
            d = m
        else:
            g = m
    return (g+d)/2  # toute valeur entre g et d convient
```

Exemple

Déterminons des valeurs approchées de $\sqrt{10}$.

Il suffit de définir une fonction qui s'annule en $\sqrt{10}$ (sans utiliser cette valeur).

```
def rac10(x):
    return x*x - 10
```

On obtient par exemple :

```
>>> dichotomie(rac10, 3, 4, 1e-8)
3.1622776575386524
>>> sqrt(10)  # pour comparer
3.1622776601683795
```

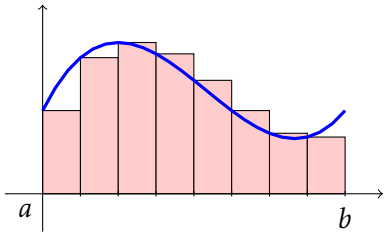
■ Comment déterminer une valeur approchée d'une intégrale ?

SF11 Connaître et programmer la méthode des rectangles

Soit $f : [a,b] \to \mathbb{R}$ continue par morceaux et $n \geqslant 1$ un entier.

Pour tout $k \in [\![0,n]\!]$, on pose $a_k = a + k\frac{b-a}{n}$.

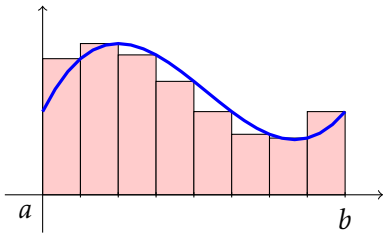
La *méthode des rectangles à gauche* consiste à remplacer f sur $[a_k, a_{k+1}]$ par la fonction constante qui coïncide avec f au point a_k :



Cela revient à remplacer $\displaystyle\int_a^b f(t)\mathrm{d}t$ par :

$$\mathrm{R}_n(f) = \frac{b-a}{n}\sum_{k=0}^{n-1} f(a_k).$$

La *méthode des rectangles à droite* consiste à remplacer f sur $[a_k, a_{k+1}]$ par la fonction constante qui coïncide avec f au point a_{k+1} :



Cela revient à remplacer $\displaystyle\int_a^b f(t)\mathrm{d}t$ par :

$$\mathrm{R}'_n(f) = \frac{b-a}{n}\sum_{k=1}^{n} f(a_k).$$

Voici une façon de programmer la méthode des rectangles à gauche :

```
def gauche(f, a, b, n):
    R = 0
    for k in range(n):
        R = R + f(a + k*(b-a)/n)
    return (b-a)/n * R
```

Exemple

Appliquons cette méthode au cas de $\ln(2) = \displaystyle\int_1^2 \frac{1}{t}\mathrm{d}t$.

```
def inverse(x):
    return 1/x
```

Ce qui donne :

```
>>> gauche(inverse, 1, 2, 1000)
0.6933972430599373
```

Il faut également savoir exploiter une majoration de l'erreur. Dans le cas de la méthode des rectangles à gauche ou à droite, si f est de classe $\mathscr{C}^1$ sur $[a,b]$ alors :

$$\left|\int_a^b f(t)\mathrm{d}t - \mathrm{R}_n(f)\right| \leqslant \frac{(b-a)^2}{2n}\sup_{[a,b]}\left|f'\right|.$$

Il s'ensuit que la méthode des rectangles à gauche ou à droite converge «à la vitesse» $\mathrm{O}\left(\frac{1}{n}\right)$.

SF12 Adapter les idées précédentes à d'autres méthodes d'approximation

D'autres méthodes sont plus rapides que la méthode des rectangles à droite ou à gauche. Par exemple, la *méthode du point milieu* consiste à remplacer f sur $[a_k, a_{k+1}]$ par la fonction constante qui coïncide avec f au point milieu $\frac{a_k+a_{k+1}}{2}$.

Cela revient à remplacer $\displaystyle\int_a^b f(t)\mathrm{d}t$ par :

$$\mathrm{M}_n(f) = \frac{b-a}{n}\sum_{k=0}^{n-1} f\left(\frac{a_k+a_{k+1}}{2}\right).$$

Si f est de classe $\mathscr{C}^2$ alors :

$$\left|\int_a^b f(t)\mathrm{d}t - \mathrm{M}_n(f)\right| \leqslant \frac{(b-a)^3}{24n^2}\sup_{[a,b]}\left|f''\right|.$$

```
def milieu(f, a, b, n):
    M = 0
    for k in range(n):
        M = M + f( ((a + k*(b-a)/n) + (a + (k+1)*(b-a)/n)) / 2)
    return (b-a)/n * M
```

Exemple

Considérons à nouveau l'exemple de $\ln(2) = \displaystyle\int_1^2 \frac{1}{t}\mathrm{d}t$.

```
>>> milieu(inverse, 1, 2, 1000)
0.6931471493099519
```

C - Autour du module Numpy

Le module Numpy et les vecteurs (tableaux à une dimension)

En général, on importe le module numpy avec l'alias np :

```
import numpy as np
```

Parmi ses nombreuses fonctionnalités, ce module apporte un type d'objet que l'on appelle *tableau* (*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'])
```

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.

Lé 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] = 'ecg'
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-28-5c6c611e4369> in <module>()
----> 1 a[3] = 'ecg'
ValueError: invalid literal for int() with base 10: 'ecg'
```

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 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])
```

Certaines instructions permettent 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. ])
```

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

Tableaux bidimensionnels

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]])
```

Pour désigner un élément d'un tableau bidimensionnel, on utilise une double 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
```

Ces tableaux 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`.

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]])
```

Comme pour les tableaux bimensionnels, 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]]])
```

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]])
```

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]])
```

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]])
```

Calculs d'algèbre linéaire

On va utiliser les deux modules `numpy` et `numpy.linalg`.

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

Considérons l'exemple de la matrice $M = \begin{pmatrix} 0 & 3 & -1 \\ -1 & 2 & 1 \\ -1 & 3 & 0 \end{pmatrix}$.

Le calcul du rang montre que cette matrice est inversible :

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

L'instruction `al.eig` fournit le tableau des valeurs propres et une matrice avec des vecteurs propres en colonnes :

```
>>> al.eig(M)
(array([ 1.,  2., -1.]), array([[-8.16496581e-01,  5.77350269e-01,  7.07106781e-01],
       [-4.08248290e-01,  5.77350269e-01, -5.55111512e-17],
       [-4.08248290e-01,  5.77350269e-01,  7.07106781e-01]]))
```

ce qui correspond à :

$$\ker(M - I_3) = \text{Vect}\Big((2,1,1)\Big),\ \ker(M - 2I_3) = \text{Vect}\Big((1,1,1)\Big) \text{ et } \ker(M + I_3) = \text{Vect}\Big((1,0,1)\Big).$$

Notons d'ailleurs que l'on a bien :

```
>>> al.matrix_rank(M - np.eye(3,3))
2
>>> al.matrix_rank(M - 2*np.eye(3,3))
2
>>> al.matrix_rank(M + np.eye(3,3))
2
```

La relation $M = PDP^{-1}$ est aisément vérifiable :

```
>>> P = al.eig(M)[1]
>>> D = np.diag(al.eig(M)[0])
>>> P
array([[-8.16496581e-01,  5.77350269e-01,  7.07106781e-01],
       [-4.08248290e-01,  5.77350269e-01, -5.55111512e-17],
       [-4.08248290e-01,  5.77350269e-01,  7.07106781e-01]])
>>> D
array([[ 1.,  0.,  0.],
       [ 0.,  2.,  0.],
       [ 0.,  0., -1.]])
>>> al.inv(P)
array([[-2.44948974e+00,  4.89506384e-15,  2.44948974e+00],
       [-1.73205081e+00,  1.73205081e+00,  1.73205081e+00],
       [-3.14018492e-16, -1.41421356e+00,  1.41421356e+00]])
>>> np.dot(np.dot(P, D), al.inv(P))
array([[ 6.66133815e-16,  3.00000000e+00, -1.00000000e+00],
       [-1.00000000e+00,  2.00000000e+00,  1.00000000e+00],
       [-1.00000000e+00,  3.00000000e+00, -8.88178420e-16]])
```

On peut aussi calculer des puissances de matrices :

```
>>> np.dot(np.dot(P, al.matrix_power(D, 10)), al.inv(P))
array([[-1022.,  1023.,  1023.],
       [-1023.,  1024.,  1023.],
       [-1023.,  1023.,  1024.]])
>>> al.matrix_power(M, 10)
array([[-1022,  1023,  1023],
       [-1023,  1024,  1023],
       [-1023,  1023,  1024]])
```

Signalons également l'instruction `al.solve(A, b)` donnant la solution du système $Ax = b$ dans le cas où A est inversible :

```
>>> A
array([[ 0,  3, -1],
       [-1,  2,  1],
       [-1,  3,  0]])
>>> b = np.array([1, 2, 1])
>>> b
array([1, 2, 1])
>>> al.solve(A, b)
array([2., 1., 2.])
```

D - Simulation d'expériences aléatoires

Dans ce thème, nous allons utiliser le module `numpy.random` avec l'alias `rd` :

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

Il convient au préalable d'initialiser le générateur de nombres aléatoires :

```
>>> rd.seed()
```

■ Comment simuler une expérience aléatoire ?

SF13 Simuler à partir d'un tirage uniforme

La fonction `random` fournit un nombre « aléatoire » dans $[0,1[$ (selon une loi uniforme).

Plus simplement, `randint(a, b)` fournit un entier de $[\![a, b-1]\!]$ avec équiprobabilité.

Exemples

1 ▹ Écrivons une fonction simulant le lancer d'une pièce équilibrée donnant *pile* ou *face*.

```
def piece():
    resultat = rd.random()
    if resultat < 0.5:
        return "pile"
    else:
        return "face"
```

2 ▹ Écrivons une fonction simulant le lancer d'un dé équilibré à 6 faces.

```
def de():
    a = rd.random()
    if a < 1/6:
        return 1
    elif a < 2/6:
        return 2
    elif a < 3/6:
        return 3
    elif a < 4/6:
        return 4
    elif a < 5/6:
        return 5
    else:
        return 6
```

ou de façon plus concise :

```
from math import floor

def de_bis():
    return floor(6 * rd.random() + 1)
```

Mais pour cet exemple, il est naturel de plutôt utiliser directement :

```
>>> rd.randint(1, 7)
3
>>> rd.randint(1, 7)
6
```

3 ▹ Écrivons une fonction simulant le lancer d'un dé pipé à 6 faces donnant 6 une fois sur deux et donnant les faces 1 à 5 de façon équiprobable.

Cela revient à considérer un dé équilibré à 10 faces avec 6 sur la moitié des faces :

```
def de_ter():
    a = rd.randint(1, 11)
    if a > 6:
        return 6
    else:
        return a
```

SF14 Simuler une expérience complexe

Pour simuler une expérience fondée sur des lancers de pièces ou dés, tirages de boules, etc. on se contente souvent d'écrire des boucles :

- `for` dans le cas de la répétition déterminée à l'avance d'une certaine expérience ;
- `while` dans le cas où l'arrêt de l'expérience est conditionné à un certain résultat.

Exemples

1 ▹ On considère l'expérience aléatoire suivante.

On dispose de deux dés A et B ; le dé A a 4 faces rouges et 2 faces blanches alors que le dé B a 2 faces rouges et 4 faces blanches. On lance une pièce de monnaie truquée telle que la probabilité d'obtenir *pile* soit $\frac{1}{3}$:

- si l'on obtient *pile* alors on décide de jouer uniquement avec le dé A ;
- si l'on obtient *face* alors on décide de jouer uniquement avec le dé B.

Écrivons une fonction de paramètre n simulant cette expérience avec n lancers (du dé déterminé par le résultat du lancer de la pièce) et renvoyant le nombre fois où l'on a obtenu une face rouge.

On commence par le lancer de la pièce truquée.

```
def piece():
    """ sortie: pile (1) une fois sur 3 et face (2) deux fois sur 3 """
    a = rd.randint(1, 4)
    if a == 3:
        return 2
    else:
        return a
```

On définit maintenant deux fonctions simulant les deux dés.

```
def deA():
    a = rd.randint(1, 7)
    if a <= 4:
        return "rouge"
    else:
        return "blanc"

def deB():
    a = rd.randint(1, 7)
    if a <= 2:
        return "rouge"
    else:
        return "blanc"
```

Notons que l'on aurait pu se contenter d'une seule fonction donnant un résultat dans un tiers des cas et un autre dans les autres cas et l'employer dans trois contextes différents.

On peut alors simuler l'expérience.

```
def simulation(n):
    c = 0  # compteur
    p = piece()
    if p == 1:  # si on tombe sur pile
        for k in range(n):
            lancer = deA()
            if lancer == "rouge":
                c = c + 1
    else:
        for k in range(n):
            lancer = deB()
            if lancer == "rouge":
                c = c + 1
    return c
```

2 ▹ Le principe du jeu du craps est le suivant : le joueur lance deux dés ordinaires et non pipés et l'on compte, pour chaque lancer le nombre de points obtenus.

- Le joueur gagne au premier lancer lorsque la somme vaut 7 ou 11.
- Le joueur perd au premier lancer lorsque la somme vaut 2, 3 ou 12.
- Dans les autres cas, le joueur doit relancer les deux dés jusqu'à ce qu'il obtienne la somme initiale ou un 7 :
  - s'il obtient la somme initiale alors il gagne ;
  - s'il obtient un 7 alors il perd.

Écrire une fonction simulant une partie de craps et retournant 1 lorsque la partie est gagnée et 0 sinon, ainsi que la liste de tous les tirages.

SF15 Simuler à l'aide de lois usuelles

Les lois usuelles peuvent être simulées à l'aide de :

- `rd.random()` pour la loi $\mathcal{U}([0,1[)$;
- `rd.binomial(n,p)` pour la loi $\mathcal{B}(n,p)$;
- `rd.randint(a,b)` pour la loi $\mathcal{U}([\![a,b-1]\!])$;
- `rd.geometric(p)` pour la loi $\mathcal{G}(p)$;
- `rd.poisson(l)` pour la loi $\mathcal{P}(l)$;
- `rd.exponential(l)` pour la loi $\mathcal{E}(l)$;
- `rd.normal(m, s)` pour la loi $\mathcal{N}(m,s^2)$;
- `rd.gamma(n)` pour la loi $\gamma(n)$.

On peut également ajouter un paramètre afin d'obtenir la simulation d'un tableau de variables aléatoires mutuellement indépentantes.

■ Comment étudier des variables aléatoires ?

SF16 Simuler une variable aléatoire

Il s'agit de définir une fonction donnant le résultat d'une expérience aléatoire.

Exemples

1 ▹ On considère la variable aléatoire X définie de la façon suivante : on effectue deux tirages avec remise dans une urne contenant n boules numérotées de 1 à n et on note X l'écart entre les deux numéros.

Écrivons une fonction `X(n)`, d'argument égal au nombre n de boules, et simulant le résultat d'une expérience.

```
def X(n):
    a = rd.randint(1, n+1)
    b = rd.randint(1, n+1)
    return abs(a-b)
```

2 ▹ On considère la variable aléatoire X définie de la façon suivante : on effectue des tirages avec remise dans une urne contenant n boules numérotées de 1 à n tant que la suite des numéros obtenus est strictement décroissante (on s'arrête donc lorsque le numéro tiré est supérieur ou égal au précédent) et on note X le nombre total de tirages effectués.

Écrivons une fonction `X(n)`, d'argument égal au nombre n de boules, et simulant le résultat d'une expérience.

```
def X(n):
    a = rd.randint(1, n+1)
    b = rd.randint(1, n+1)
    c = 2  # compteur
    while b < a:
        a = b
        b = rd.randint(1, n+1)
        c = c + 1
    return c
```

On peut aussi choisir de stocker tous les résultats des tirages dans une liste et, par exemple, renvoyer également la liste des tirages.

```
def X_bis(n):
    L = [rd.randint(1, n+1), rd.randint(1, n+1)]
    while L[-1] < L[-2]:
        L.append(rd.randint(1, n+1))
    return L, len(L)
```

SF17 Représenter la loi d'une variable aléatoire

On peut représenter la loi d'une variable aléatoire par une liste. On répète un grand nombre de fois l'expérience et, à chaque fois, on ajoute un dans la cas adéquate de la liste. Il reste, à la fin, à diviser chaque case par le nombre d'expériences afin d'avoir les fréquences.

Exemple

Reprenons l'exemple précédent de la variable aléatoire X comptant le nombre de tirages jusqu'à avoir pour la première fois un résultat supérieur ou égal au précédent.

```
def loiX(n, N):
    """ entrée : nombre de boules n et nombre de répétitions N
        sortie : liste de longueur n+2 où la case k contient
                 la fréquence d'obtention de k
    """
    t = [0 for k in range(n+2)]
    for i in range(N):
        res = X(n)
        t[res] = t[res] + 1
    for k in range(len(t)):
        t[k] = t[k] / N
    return t
```

On obtient par exemple :

```
>>> loiX(5, 10000)
[0.0, 0.0, 0.6061, 0.3213, 0.0664, 0.006, 0.0002]
>>> loiX(5, 10000)
[0.0, 0.0, 0.6034, 0.3206, 0.0686, 0.0074, 0.0]
>>> loiX(10, 10000)
[0.0, 0.0, 0.5428, 0.3398, 0.0974, 0.0177, 0.0023, 0.0, 0.0, 0.0, 0.0, 0.0]
>>> loiX(10, 10000)
[0.0, 0.0, 0.5438, 0.3331, 0.1006, 0.0192, 0.003, 0.0003, 0.0, 0.0, 0.0, 0.0]
```

Dans cet exemple, les deux premières cases contiennent toujours 0 puisque $X(\Omega) = [\![2, n+1]\!]$.

On peut aussi utiliser des diagrammes en bâtons.

Si $x = [x_0, \ldots, x_{n-1}]$ et $y = [y_0, \ldots, y_{n-1}]$ alors on peut construire un diagramme en bâtons avec les coordonnées de X en abscisse et celles de Y en ordonnées à l'aide des instructions présentées dans l'exemple suivant.

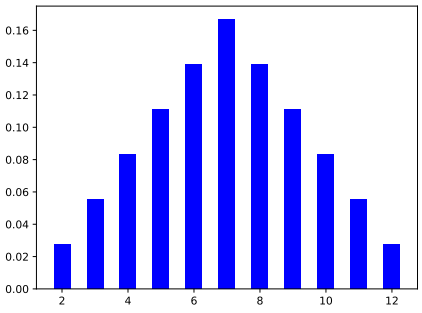
Exemple

On lance deux dés usuels équilibrés et on note X la somme des résultats obtenus.

Représentons graphiquement la loi de X.

```
import matplotlib.pyplot as plt
import numpy as np
fig = plt.figure()
x = [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
y = [1/36, 2/36, 3/36, 4/36, 5/36, 6/36, 5/36, 4/36, 3/36, 2/36, 1/36]
width = 0.5
plt.bar(x, y, width, color='b')
plt.show()
```

Cela donne la représentation graphique suivante :



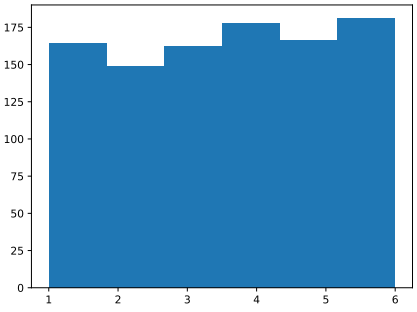
On peut également utiliser `plt.hist` pour obtenir un diagramme en bâtons.

Exemple

Représentons la répartition des résultats d'un grand nombre de lancers d'un dé usuel équilibré.

```
plt.figure()
N = 1000
t = []
for k in range(N):
    t.append(rd.randint(1, 7))
plt.hist(t, bins = 6)
plt.show()
```

Cela donne la représentation graphique suivante :



SF18 Estimer l'espérance d'une variable aléatoire

Soit X une variable aléatoire réelle, son espérance sous réserve d'existence est définie par :

$$\mathbb{E}(X) = \sum_{x \in X(\Omega)} x\mathbb{P}(X = x).$$

Une première approche pour estimer l'espérance d'une variable aléatoire consiste à simuler la loi de cette variable (donc à répéter un grand nombre de fois l'expérience et à stocker dans une liste les différentes fréquences d'obtention des valeurs) puis à calculer l'espérance à partir de ce tableau.

Bien entendu, le calcul de la variance s'obtient par la même méthode et la formule de König-Huygens :

$$\mathbb{V}(X) = \mathbb{E}(X^2) - \mathbb{E}(X)^2.$$

Exemple

Considérons à nouveau l'expérience suivante : on effectue deux tirages avec remise dans une urne contenant n boules numérotées de 1 à n et on note X l'écart entre les deux numéros.

On a vu que la fonction suivante permettait de simuler une expérience :

```
def X(n):
    a = rd.randint(1, n+1)
    b = rd.randint(1, n+1)
    return abs(a-b)
```

On simule la loi de X :

```
def loiX(n, N):
    t = [0 for k in range(n)]  # cf. X(Omega) = [0, n-1]
    for k in range(N):  # répétition de N expériences
        res = X(n)      # résultat d'une expérience
        t[res] = t[res] + 1  # on ajoute 1 dans la case adéquate
    return [x/N for x in t]  # on renvoie les fréquences
```

Pour estimer $\sum_{k=0}^{n-1} k\,\mathbb{P}(X = k)$, on multiplie respectivement chacune des cases précédentes par $0, 1, \ldots, (n-1)$ et on somme :

```
def EspX(n, N):
    tab = loiX(n, N)
    esp = 0
    for k in range(n):
        esp = esp + k*tab[k]
    return esp
```

On trouve par exemple :

```
>>> EspX(5, 1000)
1.625
>>> EspX(5, 1000)
1.595
```

La valeur théorique est $\mathbb{E}(X) = \dfrac{(n-1)(n+1)}{3n}$ donc, pour $n = 5$, $\mathbb{E}(X) = 1,6$.

Voici un autre exemple, avec $n = 10$, où l'on est censé obtenir $\mathbb{E}(X) = 3,3$:

```
>>> EspX(10, 10000)
3.3089
>>> EspX(10, 10000)
3.3116
```

Une autre approche repose sur le concept suivant. Considérons une suite de variables aléatoires $(X_n)_{n\in\mathbb{N}^*}$ indépendantes et de même loi, admettant une espérance, notée μ. On appelle *moyenne empirique* de $X_1,\dots,X_n$, la variable aléatoire :

$$\overline{X_n} = \frac{1}{n}(X_1 + \cdots + X_n).$$

La loi faible des grands nombre affirme alors :

$$\forall \varepsilon > 0,\ \mathbb{P}\left(\left|\overline{X_n} - \mu\right| > \varepsilon\right) \xrightarrow[n\to+\infty]{} 0.$$

Concrètement, on obtient donc une valeur approchée de l'espérance d'une variable aléatoire X en calculant la moyenne d'un grand nombre de réalisations de X.

Exemple

Reprenons l'exemple précédent.

```
def EspXbis(n,N):
    esp=0
    for k in range(N)
        esp = esp + X(n)
    return esp / N
```

On obtient par exemple :

```
>>> EspX(5, 1000)
1.614
>>> EspX(5, 1000)
1.559
>>> EspX(5, 1000)
1.542
>>> EspX(10, 1000)
3.323
>>> EspX(10, 1000)
3.303
```