Python, numpy et matplotlib en BCPST

Ressources : Python Standard Library donnera accès à toute la documentation et à l'aide en ligne. On pourra aussi visiter : Python-prepa.github.io.

Opérations arithmétiques

Assignation : `a=1`
Addition, soustr., mult. : `a+b,a-b,a*b`
Division $a, b \in \mathbb{R}$: `a/b` résultat dans $\mathbb{R}$
Division $a, b \in \mathbb{Z}$: quotient `q=a//b`, reste $r = a\%b$
Incrémentation : `a+=b` equivaut à `a=a+b`
(Idem : `a-=b`, `a*=b` et `a/=b`)
Exponentiation : `a**b`

Opérateurs de comparaison

Égalité : `a==b`
Inégalités strictes : `a<b`, `a>b`
Inégalités larges : `a<=b`, `a>=b`
Différence : `a!=b`
Logique a et b : `a and b`
Logique a ou b : `a or b`

Listes

Une liste est définie entre crochets, les éléments étant séparés par une virgule. Elles peuvent contenir différents types d'éléments. *Exemple* : `L=[1,2,3.5,'a',-2.4,'test']`
initialisation possible avec `L = [None]*N` où $N \in \mathbb{N}$.
Accès à l'élément $i \in [\![0, N-1]\!]$: `L[i]`
✎ : `L[-1]` désigne le dernier élément de la liste `L[-2]` l'avant dernier, etc.
`L.append(x)` : Ajout d'un élément x à la fin
`L.extend(L2)` : Ajoute la liste L_2 à la fin de la liste L (on peut aussi faire `L3=L1+L2` mais on créé une nouvelle liste).
`L.insert(i,x)` : Insertion d'un élément x à la position i (on peut aussi faire `L[i:i]=[x]`.
`L.index(x)` : Renvoie l'indice de la première occurence de x.
`L.count(x)` : Nombre d'occurrences de x.
`L.remove(x)` : supprime la 1ère occurrence de x.
`L.pop([i])` : supprime l'élément en place i et le renvoie (Si l'indice n'est pas précisé, c'est le dernier élément de la liste qui est visé).
`L.replace(x,y)` : Échange de deux caractères.
`L.reverse()` : Inversion de la liste (on peut aussi faire `L[::-1]`).
`L.sort()` : Tri croissant de la liste.
`L.sort(reverse=True)` : Tri décroissant.
✎ Toutes ces *méthodes* font des listes des objets **mutables**. On ne créé pas grâce à elles de nouveaux objets, on se contente de les modifier (Pour `L.sort()` on perd la liste initiale... Si on veut garder la liste originale, il faut commencer par la copier ou écrire `L1=sorted(L)`.)
`L1=list(L)` ou `L1=L[:]` : Copie de la liste.
Listes d'entiers :
`range(N)` : Listes des entiers de 0 à $N-1$.
`range(M,N)` : Liste des entiers de M à $N-1$.
`range(M,N,p)` : Liste des entiers de M à $N-1$ avec un pas de p.
`[0]*n` : liste de n « zeros »(utile pour initialiser)
✎ *Attention !* Si `L=[1,2]`, alors `2*L=[1,2,1,2]`.

Création de listes à partir de boucles :
`[i**2 for i in range(1,5)]` : Liste des 4 premiers carrés.
`[i**2 for i in range(1,5) if i%2==0]` retourne `[4,16]`.

Liste de listes :
Pour `L=[[1,2,3],[4,5,6]]` on obtiendra : `L[0]=[1,2,3]` et `L[1][1]=[5]`
ou encore :
`[[i**2+j for i in range(3)] for j in range(2)]` retournera `[[0,1,4],[1,2,5]]`

`map()` : Applique une fonction à l'ensemble des éléments d'une liste.
Pour `L=["0.2","3","1.2e3"]` :
`map(float,L) = [0.2,3,1200]`

Ensembles

Un ensemble est entendu au sens mathématique comme collection d'objets non ordonnée, sans répétition possible... Par exemple : `E={3,2,5,'pomme'}`
Les ensembles sont pour Python des objets immuables. Ils sont optimisés pour les recherches d'appartenance et toute opération classique sur les ensembles.

création d'ensembles :

`E={3,2,5,'pomme'}`
ou d'abord une liste :
`L=[2,3,2,5,3,'pomme',5,5]`
puis `E=set(L)`
`E.add('oeuf')` : ajout d'un élément dans `E`
`E.remove(2)` : suppression.
(retourne `E={3,'pomme',5,'oeuf'}`)
`E.update([7,8])` : ajoute 7 et 8 à `E`.
`E.clear()` : vide l'ensemble E. `E=set()`

Les tests d'appartenance :

`5 in E` : `True`
`8 in E` : `False`

Opérations classiques :

`E|E2` : $E \cup E2$ - Union de E et de $E2$
`E & E2` : $E \cap E2$ - Intersection de E et de $E2$.
`E-E2` : $E \setminus E2$ - Différence ensembliste.
`E^E2` : Différence symétrique : $E \Delta E2$.
`E <= E2` : $E \subset E2$.
`E.isdisjoint(E2)` : Vrai ou Faux selon que $E \cap E2 = \oslash$

Tuple :

Un tuple est très semblable à une liste sauf qu'il n'est pas modifiable. Pour le créer, on remplace les crochets par des parenthèses.
`t = ()` : tuple vide
`t = (3,)` : tuple réduit à un élément (✎ Attention : si on écrit `t=(3)` alors Python néglige les parenthèses et considère que `t` est un entier...)
`t = (2,'essai',True)` : tuple formé de trois éléments qui peuvent être de type distinct (✎ On peut omettre si on le souhaite les parenthèse et écrire `t = 2,'essai',True`.
Pour modifier un tuple il faut le transformer en liste en écrivant : `L=list(t)`
✎ On les utilise souvent pour des boucles « Pour » sur deux indices... Ainsi `zip` associe deux listes entre elles sous forme de tupple.
Exemple : `villes=[Paris,Nice,Lyon]; population=[2*10**6,4*10**5,10**6]` Alors `zip(villes,population)=[(Paris,2*10**6), ...(Nice,4*10**5),(Lyon,10**6)]`
Application :

```
for v,p in zip(villes,population):
   print(p,'habitants à ',v)
```

retourne :
« 2 00 0000 habitants à Paris »
« 400 000 habitants à Nice »
« 1 000 000 habitants à Lyon »

`enumerate(L)` : Fonction très utile pour indexer une liste...permet de retourner une liste de tupple de la forme `(i,L[i])`.
Exemple :

```
villes=['Nantes','Orvault','Rezé']
for i,ville in enumerate(villes)
   print(str(i),ville)
```

retourne :
0 Nantes
1 Orvault
2 Rezé

Chaînes de caractères

Elles se manipulent comme une séquence.
Exemple : `s='python'` est une séquence.
`s[0]` : Retourne la première lettre 'p'.
`s[-1]` : Retourne la dernière lettre 'n'
`s[:3]` : Retourne 'Pyth', etc.
✎ Si la chaîne contient des apostrophes, utiliser les guillemets : `s="python, c'est super !"`
Les méthodes propres aux chaînes de caractères :
`s.upper()` : 'PYTHON, C'EST SUPER!'
`s.replace('python','la bio')` : `'la bio, c'est super !'`
✎ Application classique :
`s.replace(" ","")` : supprime les espaces de la chaîne.
`s.strip()` : supprime en début ou fin de chaîne toute tabulation ou retour chariot (codés par un '\n' sous Python).

`s.lstrip(s1)` : supprime tout caractère de s1 situé en **début** de chaîne (idem avec `s.rstrip()` pour la fin de la chaîne...).
`L=s.split()` : Transforme la chaîne en liste en considérant par défaut que le séparateur est l'espace.
Exemple :
`L=['python,', "c'est", 'super', '!']`
Permet des modifications de la liste. On peut ainsi se passer de la fonction `replace`.
Exemple :
`L[2]='pénible'` permet d'obtenir :
`L=['python,', "c'est", 'pénible', '!']`
✎ Pour revenir à une chaîne de caractères, on utilise `join` :
`' '.join(L)` : `"python, c'est pénible !"`
`'*'.join(L)` : `"python,*c'est*pénible*!"`
`s.find('sup')` : Premier indice où le mot 'sup' est présent (ici 14).

Fonctions :

En analyse : utiliser le mot clef `lambda`

```
f=lambda x:x**2+1
```

ou encore, si plusieurs variables d'entrée et de sortie :`f=lambda x,y:x**2+y**2`

Le reste du temps : mot clef `def`
Exemple :

```
def f1(x):
    if x >= 0:
        return 1+x
    else:
        return 1-x
```

ou encore :

```
def f(x,y):
 s=x+y
 d=x-y
 return s,d
```

Exple d'appel : `sum,diff=f(3,5)`

Paramètres avec valeurs par défaut :
`def mafonction(p1,p2,p3=valeurParDefaut)`
✎ On placera ces paramètres en dernier !

Description des paramètres (clarification du code) :

```
from typing import List
proportion = int
Mot = str
def mafonction(p1,p2:proportion,test=Mot)
```

ou

```
def mafct(Lind:list[int],nb_ind:int,
noms:list[str])->set[int]
```

appelée par :

```
> > > mafct([1,2,3],5,["joe","max"])
```

qui retourne :

```
{2,5}
```

et en fin de fonctions :

```
if __name__ == "main":
```

Boucles et instruc. conditionnelles :

1. Boucle de 0 à $N-1$ (N entiers) :
   ```
   for i in range(N):
       action(i)
   ```

2. Boucles de N à $M-1$ par pas de p (entiers) :
   ```
   for i in range(N,M,p):
       action(i)
   ```

3. Boucles sur une liste quelconque :
   Par exemple : `L=[1,1.5,3,2]`. Alors :
   ```
   for i in L:
       action(i)
   ```
   ou encore : `print([i**2 for i in L])`
   produira l'écriture : `[1,2.25,9,4]`

4. Instructions conditionnelles :
   ```
   def f(x):
       if x>=1 :
         return 0.
       elif x<0:
         return -1.
       else:
         return 1.
   ```

✎ Si `L=[1,1.5,3,2]`, alors
`[i**2 for i in L if i**2>3]` retourne `[9,4]`

Fonctions mathématiques usuelles :

```
from math import *
```

Toutes les fonctions seront directement utilisables, comme *sin*, *cos*, *exp*, *log*, etc.

Travailler avec des fractions :

```
from fractions import *
u=Fraction(a,b) # expression réduite
```

Interagir avec l'utilisateur :

```
T = input('entrer un texte : ')
a = int(input('entrer un entier : '))
x = float(input('entrer un réel : '))
```

Fonctions usuelles (numpy) :

On retrouve ici aussi toutes les fonctions usuelles au programme (*exp*, *ln*, *cos*, *sin*, *tan*, *Arctan*...).

intérêt : permet le calcul de `np.sin(X)` où `X = np.arange(1,11)`

Matrices (numpy) :

`np.array(L)` : transformation d'une liste en matrice. Exple : Si `L1=[1,2,3]` ou `L2=[[1,2,3],[4,5,6]]` alors `M1=np.array(L1)` et `M2=np.array(L2)` permettent d'obtenir respectivement une matrice d'une ligne trois colonnes et deux lignes, trois colonnes.
On aura par exple : `M1[2]=3` et `M2[0,1]=2`

création de matrices standards :

`np.zeros(N)` : matrice ligne de N zéros.
`np.zeros((N,M))` : matrice de N lignes, M colonnes de zéros.
`np.ones(N)` : matrice ligne de N « 1 ».
`np.ones([N,M],int)` : matrice N, M colonnes de 1 (entiers).
`np.zeros_like(a)` : matrice de zéros de même dimension et de même type qu'une matrice a (idem pour `np.ones_like(a)`...).
`np.arange(M,N,p)` : matrice ligne de réels dans l'intervalle $[M, N[$ par pas de p (si on écrit `np.arange(N)` alors par défaut $M = 0$ et $p = 1$.
`np.vstack([A, B])` : Concaténation verticale des matrices A et B (si possible).
`np.hstack([A, B])` : Concaténation horizontale des matrices A et B (si possible).
`np.linspace(a,b,N)` : matrice ligne allant de a à b avec N éléments uniformément répartis. Très utile pour tracer les fonctions. En particulier si `f1=lambda x:2*x+1` et `x=np.linspace(-1,2,100)`, alors `y=f1(x)` retournera une matrice ligne.
`b=a.copy()` : copie d'une matrice a dans une matrice b.
`np.diag(v)` : matrice diagonale avec le vecteur v sur la diagonale. Exple : `np.diag(np.arange(N))`, `np.diag(np.ones(N))`
`np.identity(N)` : matrice identité de taille N.
`np.diag(v,i)` : matrice avec le vecteur v sur la ième diagonale supérieure.
`np.diag(v,-i)` : matrice avec le vecteur v sur la ième diagonale inférieures.

Opérations sur les matrices (numpy) :

`A.shape` : Obtenir les dimensions de A.
`B=A.reshape(8,3)` permet d'obtenir une matrice `B` formée des valeurs de A mais de dimension différente.
`A.flatten()` : Transforme `A` en matrice ligne.
Extraction de sous-matrices d'une matrice `A` :
`A[:,0]` : Extraction de la première colonne.
`A[2:3,1]` : Extraction des troisièmes et quatrièmes lignes de la deuxième colonne.
`A[:2,:]` : Extraction des deux premières lignes.
`A[-2:,:]` : Extraction des deux dernières lignes.
`A = np.array([[2]])`. Alors `A.item()` retourne 2.
`np.concatenate((A,B),axis=0 ou 1)` : Concaténation de A et de B par les lignes ou par les colonnes sous réserve de compatibilité des tailles.
`A.argmax()` : Retourne l'indice du maximum du tableau A.
`A.argmin()` : Retourne l'indice du minimum du tableau A.
`A.item()` : Si `A = np.array([[a]])` retourne `a`.
`np.where(A == b)` : liste de listes d'indices / $a_{i,j} = b$.

Opérations mathématiques : toutes les opérations arithmétiques et fonctions classiques s'appliquent pour une matrice `a` mais **attention**, elles s'appliquent termes à termes... *Exemples :* `2*a+1`, `a**2`, `1+np.sin(a)`, `a*b`, etc.

`np.dot(a,b)` : Produit matriciel de a et b.
`A.transpose()` ou `A.T` : Transposée de A.
`np.linalg.inv(A)` : inverse de la matrice A.
`np.linalg.solve(A,b)` : solution x du système $A \cdot x = b$.
`np.linalg.matrix_rank(A)` : rang de A.
`np.linalg.eigvals(A)` : valeurs propres de A.
`np.linalg.eig(A)` : v. et vect. propres de A.
`np.linalg.norm(x)` : norme euclidienne de x.
`np.inner(x,y)` ou `x @ y` : produit scalaire des vecteurs x et y (matrices lignes).

Remarque : Parfois plus simple : utiliser la fonction `matrix` de la bibliothèque numpy.
Si `A = np.matrix(L1)` et `B = np.matrix(L2)` alors : `A+B`, `A-B` et `A*B` sont les opérations usuelles sur ces matrices.
`C = np.matrix(A)` : copie dans `C` de la matrice `A`.

Statistique descriptive : L'ensemble du programme peut être abordé à l'aide de numpy.
On considère X et Y deux séries statistiques univariées de même taille, entrées sous forme de liste `X=[x_1,...,x_n]` et `Y=[y_1,...,y_n]`.
On retiendra notamment :
`np.min(X)` : Minimum de la série X.
`np.max(X)` : Maximum de la série X.
`np.mean(X)` :Moyenne de la série X
`np.median(X)` : Médiane de la série X
`np.percentile(X,p)` : Expression du p-ième percentile où p est exprimé en pourcentage.
Exple : $\widehat{q}_{0.25}$ =`np.percentile(X,25)`
`np.var(X)` : Variance de la série X
`np.std(X)` : Ecart-type de la série X
`np.isnan(x)` : permet de savoir si x est un nombre ou pas (par exemple retourne `True` si `x=np.nan`.
`np.histogram(X,classes)` :
`np.histogram(X,classes,normed=True)` :
`np.cov(X,Y,bias=1)` : Matrice de covariance qui prend les `n=len(X)` valeurs de la série.
`np.corrcoef(X,Y)` : coefficient de corrélation.
`(a,b)=np.polyfit(X,Y,1)` : coefficient de la droite de régression $y = ax + b$.

Calcul polynomial :

Soit $P = a_0 + a_1 X + \cdots a_n X^n = \sum_{k=0}^{n} a_k X^k$.
Ce polynôme s'exprime dans la base canonique sous la forme $P = (a_0, \cdots, a_n)$.

✎ Python choisit de l'exprimer par coefficients décroissants.
`P1=np.poly1d([1,2,3])` : $P_1(X) = X^2 + 2X + 3$.
`P1=np.poly1d([1,2,3],variable='Z')` :
$P_1 = Z^2 + 2Z + 3$
`P1(a)` : Évalue le polynôme P_1 en a.
`yP=np.polyval(P1,xx)` : Évalue $P1$ sur **l'ensemble** des valeurs de la liste `xx`.
`P_prim1=np.poly1d.deriv(P1,m=1)` : Dérivée première de P_1 soit `[2,2]`
`P_prim2=np.poly1d.deriv(P1,m=2)` : Dérivée seconde de P_1.
✎ On peut aussi construire un polynôme (normalisé) à partir de ses racines.
`P2=np.poly1d([1,2],True)` : $P_2(X) = (X-1)(X-2) = X^2 - 3X + 2$.
`P1.r` : Retourne les racines de P_1.
`P1.order` : retourne le degré de P_1
`P2.c` : retourne les **coefficients** de $P2$ sous forme de tableau.

Equa. diff. et Intégration

Dans les deux cas, on importera le module `integrate` de la bibliothèque `scipy`.

Pour intégrer :
```
from scipy import integrate
f = lambda t:... # description de f
I,abserr = integrate.quad(f,a,b)
```
: Approximation numérique de f sur l'intervalle $[a, b]$ ainsi que de l'erreur absolue commise sur le résultat.

Pour résoudre une équation différentielle :
Soit à résoudre : $Y' = f(Y, t)$ avec $y_0 = Y(0)$.
Alors :
```
sol = integrate.odeint(f,y0,t)
```

Tracé des courbes (matplotlib.pyplot)

`import matplotlib.pyplot as plt`
Permet de tracer des courbes définies par des valeurs numériques, à savoir des tableaux de points.
Exemple :
`plt.plot(Liste,drawstyle = 'steps')` : relie les points de la liste en « escalier ».
`X=np.linspace(-np.pi,np.pi,100)` : matrice ligne de 100 points entre $-\pi$ et π.
`plt.plot(np.sin(X))` : crée le graphe de sinus en les points de x en fonction de x.
`plt.plot(np.sin(X),linewidth=2)` : augmente l'épaisseur du trait (par défaut 1).
`plt.plot(X,sin(X),'g-',X,cos(X),'r:')` : crée le graphe de sinus en trait continu vert et de cosinus en pointillés rouges.
Se rapporter à l'aide : `help(plot)` pour obtenir tous les types de tracé possible.
✎ Si f est définie comme la fonction `f1` de la page 3, alors le calcul de `f1(X)` sera impossible. On pourra faire dans de tels cas :
`vf = np.vectorize(f1)`
`plt.plot(X,vf(X))`

`plt.savefig('nomfigure.jpg')` : Enregistre l'image spécifiée dans le répertoire courant. Les sauvegardes peuvent également être faites aux formats 'png', 'pdf', 'ps', 'eps' et 'svg'.
`plt.figure(n)` : ouvre la figure n°n.
`plt.figure('graphe de machin(x)')` : ouvre une fenêtre et lui donne un nom.
`plt.clf()` : efface une figure déjà ouverte.
`plt.close()` : ferme une fenêtre déjà ouverte.
`plt.close('all')` : ferme toutes les fenêtres ouvertes.

Les figures en statistiques :

`plt.hist(X)` : Trace et retourne un histogramme des valeurs de X (par défaut 10 classes).
`plt.hist(X,nc)` : Permet de modifier le nombre de classes par défaut.
`plt.hist(X,classes)` : Permet de spécifier des classes, d'amplitudes éventuellement distinctes
`plt.hist(X,classes,density='true')` : Permet de tracer un histogramme des fréquences relatives.
`plt.hist(X,classes,density='true',...`
`...histtype='stepfilled',cumulative='true')` : permet de tracer les fréquences cumulées réparties dans les classes spécifiées
`plt.hist(X,classes,normed=True,log = True` : Avec échelle logarithmique en ordonnée.
`plt.boxplot(X)` : Tracé d'une boîte à moustache.
`bp = plt.boxplot(V, whis=[2,98],...`
`...showfliers=True)` : Pour modifier les intervalles considérés.

Organisation des figures :

`plt.title('Titre du graphe')` : ajoute un titre sur la figure
Si on a tracé `plt.plot(x,f1(x),label='Graphe de f1')` il suffit d'écrire `plt.legend(loc='upper right')` pour afficher *Graphe de f1* assorti à la couleur de la courbe en haut et à droite du graphe.
Note 1 : `plt.legend(loc='best', frameon=False)` adapte la place de la légende en fonction du tracé.
Note 2 : Si on veut insérer des formules mathématiques, on peut le faire en *Tex* en faisant précéder le texte d'un « r ». Exemple : `plt.title(r'Graphe de $f(x)=e^{\alpha x}$')` affiche en titre : $f(x) = e^{\alpha x}$
`plt.scatter(i,j)` : Trace un cercle plein (taille et couleur au choix) à la position (i, j).
`plt.xlim(left=a,right=b)` : Permet de définir l'affichage des abscisses entre les bornes a et b (idem pour l'axe des ordonnées avec `bottom=a` et `top=b`).
`plt.grid()` : Ajout d'une grille.
`plt.xticks(np.arange(a,b,p))` ajoute des lignes verticales en pointillé entre a et b, séparées d'un pas de p.
`plt.axis('off')` : Élimine la présence des axes.
`plt.text(i,j,'Texte')` : Affiche 'Texte' à la position (i, j) sur l'image.
Des options sont possibles telles que : `rotation=90`, `alpha=0.5`, `color='red'`
`plt.axvline()` et `plt.axhline()` : Tracé des axes de coordonnées
`plt.axhline(color = 'r')` : gestion de la couleur des axes.
`plt.axvline(i, linestyle='-', color='red', alpha=0.3)` : Tracé d'une ligne verticale en pointillés en $x = i$.

`plt.vlines(i,a,b,colors='blue',...,label='...')` Tracé d'un segment reliant les points (i, a) à (i, b).
`plt.annotate('Texte',xy=(a,b),xytext=(c,d)...arrowprops=dict(facecolor='black',...shrink=0.05))` : Insertion d'un texte associé à une flèche qui pointe en xy.
`plt.arrow(x,y,dx,dy,head_width=0.0005,...head_length=0.0005, fc='r', ec='r',... width = 1e-5)` : Tracé d'un vecteur.
`plt.tight_layout(rect=[0,0,1.6,2])` : modifie la taille de la fenêtre.
`plt.subplot(325)` : Crée des sous-figures (3 lignes et 2 colonnes) dans un graphique et rend active la cinquième numérotée dans l'ordre naturel de lecture. A utiliser avec le même nombre de lignes et de colonnes avant chaque sous-figure d'une même fenêtre graphique.
`plt.tight_layout()` sans paramètre, évite le chevauchement des labels

Graphe 2D de matrices :
`im1=plt.matshow(M)` : représentation imagée de la matrice M, une couleur étant affectée à chaque coefficient distinct de M.
`plt.colorbar(im1)` : permet d'afficher la légende des couleur employée par Python.

Gestion des images :

Avec `matplotib.pyplot` :
`im=plt.imread('nom_image.png')` : lecture d'une image sous forme de tableau.
`im.shape` : Taille de l'image `plt.imshow(im)` : affichage d'un tableau.
Si image en couleur : `im.ndim = 3`
Si image en noir et blanc : `im.ndim = 2`
`plt.imshow(im,cmap=plt.cm.gray)` : Affichage en niveaux de gris
`plt.imsave(nomImage)` ou `plt.savefig()` : Sauvegarde au format de son choix.
Remarque : Pour un affichage « image par image » (faire un « film » de vos images) :
`image.set_data(tabImages)` : mise à jour des valeurs
`plt.draw()` : redessine l'image
`plt.pause(0.1)` : temps d'attente en secondes.

Probabilités :

✎ **Calcul de $n!$: import math** puis `math.factorial(n)`.
✎ **Coefficients binomiaux** : `scipy.special.binom(n,k)` ou `scipy.misc.comb(n,k)`

`import random as rdm`
`random.seed(100)` : initialisation de la « graine ». Permet la reproductibilité de l'expérience aléatoire.
`rdm.randint(a,b)` : loi uniforme sur $[\![a, b]\!]$.
`rdm.choice(seq)` : retourne un nombre au hasard au sein de la séquence `seq`.
`rdm.random()` : loi uniforme sur $[0, 1[$.
`rdm.uniform(a,b)` : loi uniforme sur $[a, b]$
`rdm.sample(seq,n)` : retourne une liste de n éléments **sans répétition** pris au hasard dans `seq`.
`rdm.sample(seq,len(seq))` : retourne une **permutation** des éléments de `seq`.

✎`import numpy.random as nrdm`
`nrdm.randint(a,b)` : loi uniforme sur $[\![a, b[\![$.
Intérêt : Créer une matrice formée d'entiers pris au hasard dans $[\![a, b[\![$.
`nrdm.randint(2,6,4)=array([2, 4, 5, 5])`
`nrdm.randint(2,6,(2,3))=`
`array([[2, 4, 2], [3, 5, 3]])`

densités et fonctions de répartitions des lois usuelles
`import scipy.stats`
On choisit dans la variable aléatoire de son choix dans la large liste proposée. Par exemple :
`from scipy.stats import poisson` : importe les principales fonctions liées à la loi de Poisson.
`poisson.pmf()` : Probability mass function (= Loi de probabilité)
`poisson.cdf()` : Cumulative density function (= Fonction de répartition)
`poisson.ppf()` : Percent point function (= fonction réciproque de la fonction de répartition)
`poisson.stats(lbda,moments='mv')` : Retourne la 'm'-oyenne et la 'v'-ariance et on peut ajouter de nombreuses autres caractéristiques...

Modélisation des lois usuelles :
Des algorithmes sont au programme. On peut sinon utiliser `Loi.ppf(rdm.random())` car X et $F_X^{-1} \circ U$ ont même loi **mais** la bibliothèque `numpy.random` permet de le faire simplement :

- `nrdm.binomial(10,0.3,4)= ...`
  `array([4, 2, 5, 2])` : 4 réalisations de la **loi binomiale** ($n = 10$, $p = 0.3$).
- `nrdm.binomial(1,0.6,4)=...`
  `array([1, 1, 0, 1])` : 4 réalisations de la **loi de bernoulli** ($p = 0.6$).
- `nrdm.binomial(5,0.5,size=(4,6))`
  retourne une matrice 4 lignes, 6 colonnes dont les coefficients sont issus d'une loi binomiale.
- `nrdm.hypergeometric(4,6,3,10)` : 10 réalisations de la **loi hypergéométrique** ($ns = 4$ (nb de succès), $ne = 6$ (nb d'échecs) et $n = 3$ (nb de tirages)). ✐ $N = ns + ne$.
- `nrdm.normal(0, 0.33, 1000)` : 1000 réalisation d'une loi normale centrée d'écart-type 0.33.
- `nrdm.geometric(0.6,4)=...` : 4 réalisations de la **loi géométrique** ($p = 0.6$).
- `nrdm.poisson(6,4)=...` : 4 réalisations de la **loi de poisson** ($\lambda = 6$).