Python en ECG1, parcours mathématiques approfondies

1 Démarrage

Lors du lancement, deux fenêtres s'ouvrent.
La première fenêtre, appelée interpréteur ou console, permet de faire des calculs ou de définir des petites fonctions etc. Vous verrez dans cette fenêtre une invite de commande `>>>` ou `In [1] :` (selon l'éditeur utilisé), qui signifie que Python attend une instruction.

La deuxième fenêtre, appelée éditeur, permet d'écrire des scripts (ou programmes) un peu plus longs, de les enregistrer pour les utiliser lors d'une nouvelle session et/ou de les exécuter : attention, lors de l'exécution, les messages et affichages éventuels apparaissent dans la console.

Calculs dans la console : cf TP 1

Opérations usuelles : + - * / et ** (puissance)

Reste et quotient de la division euclidienne de m par b : m%b et m//b

Attention, les fonctions usuelles ainsi que les constantes usuelles π, e, ne sont pas directement accessibles. Il faudra importer une bibliothèque (appelée aussi parfois librairie) pour les utiliser. Voir Section 3.

Variables :
Une variable est une case mémoire de l'ordinateur à laquelle on a donné un nom. On peut affecter des valeurs à des variables en utilisant le signe =
Par exemple la syntaxe `x=2*3+5` crée une variable appelée `x` et lui affecte la valeur 11.

Un nom de variable est un mot composé de lettres et de chiffres. En mathématiques, une variable est presque toujours désignée par une seule lettre. En informatique, on préfèrera utiliser des noms de variables plus parlants : par exemple, si l'on calcule une moyenne, au lieu de l'appeler `x` ou `m`, on l'appellera `moyenne`. A la relecture, le programme informatique n'en sera que plus clair !

Une **variable** peut contenir n'importe quel **type** d'objet :

- entier `int` pour `integer`, réel `float`
- matrice ou tableau `array`
- chaîne de caractère `str` pour `string`. Une chaîne de caractères est une succession de lettres ou caractères délimitée par des guillemets ' ... ' ou " ... ". Par exemple `"Bonjour"`.
- booléen `bool` : type de variables à deux états `True` "vrai" ou `False` "faux".
- liste `list` : ce type n'apparait pas explicitement dans votre programme donc ne sera pas étudié cette année.

Pour connaître le type d'une variable `x` il suffira de taper l'instruction `type(x)`.

Affichage :
Dans la console, un calcul est automatiquement affiché, et pour afficher une variable nommée par exemple `a`, il suffit de taper l'instruction `a`
Mais dans l'éditeur, il faut utiliser la fonction `print( ....)` : par exemple, il faudra marquer `print(5*7)`, pour obtenir le résultat du calcul 5×7, ou `print(a)`, pour afficher le contenu de la variable `a`.
Pour un affichage plus précis, on combinera des affichages successifs grâce à la syntaxe `print(.. , .. , ..)`.
Par exemple : `print('a=',a)` affiche la chaîne de caractère `a=` puis la valeur de la variable `a`.

Commentaires :
Pour augmenter la lisibilité d'un script (surtout si le programme est utilisé lors d'une séance ultérieure), il est conseillé d'insérer des commentaires pour expliquer, ou rappeler le contenu d'une variable : il suffit de mettre `#` et ce qui suit ne sera pas lu (donc ni exécuté ni affiché).

Raccourci : pour obtenir directement une instruction tapée plus haut sans la réécrire, utiliser les touches ↑ ↓.

2 Programmation en python

2.1 Indentation

Au cours des syntaxes présentées dans cette section, il faudra être vigilant à l'indentation.
En effet, dans une boucle en python (for ou while), il n'y a pas de `end` pour signifier que l'on 'sort' de la boucle. C'est l'indentation qui joue ce rôle. Toutefois l'éditeur vous facilitera la tâche car dès que vous marquerez les `:` à la fin de la première ligne de la commande (que ce soit après `if`, `else`, `for`, `while` ...), l'indentation se fera automatiquement. Il faudra juste penser à revenir à gauche lors de la fin de la commande.

Non seulement c'est juste une habitude à prendre, mais en plus, vous réaliserez vite qu'un programme indenté (et avec quelques commentaires) est bien plus lisible !

2.2 Fonctions et dialogue avec l'utilisateur

Souvent un programme a besoin d'une valeur pour pouvoir s'exécuter : imaginez que le programme calcule les termes successifs d'une suite. Il faut bien lui indiquer quel est l'entier n, pour lequel on cherche la valeur de u_n !

Il y a deux façons de demander cette valeur (ou ces valeurs) à l'utilisateur :

1. via `input`
2. via la syntaxe d'une fonction.

Syntaxe via input : elle vous semblera peut-être la plus simple à prime abord, mais vous verrez rapidement qu'elle est contraignante, et plus lourde à écrire. On ne la mettra que peu en oeuvre.

Exemple : la syntaxe `n=int(input('entrer un entier'))` commence par faire afficher dans la console le message "entrer un entier", puis attend une valeur tapée par l'utilisateur, avant de la transformer en entier et de l'affecter à la variable qui s'appelle `n`.
Si il vous faut la valeur de u_0, qui sera un réel (et non toujours un entier), la syntaxe devient :

```
u=float(input('entrer le premier terme de la suite'))
```

Remarque : que se passe-t-il si on met `n=input('entrer un entier n')` ?
Dans ce cas, la valeur entrée par l'utilisateur est vue comme une chaîne de caractère, et non comme un nombre. On ne pourra donc pas l'utiliser pour faire des opérations.
Voilà une des raisons pour lesquelles cette syntaxe n'est pas très agréable : il faut définir le type de l'entrée.

Syntaxe d'une fonction.
Pour comprendre les fonctions en python, on peut commencer par faire l'analogie avec les fonctions mathématiques : à un paramètre d'entrée, par exemple x, elle renvoie une valeur $f(x)$. Dans ce cas, la syntaxe correspondante est :

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

Pour l'appliquer dans la console : on tapera l'instruction `f(3)` (ou `a=f(3)` si on veut garder en mémoire la valeur) et la valeur affichée sera `6`.
Cette fonction en python, correspond à la fonction mathématiques $f : x \mapsto 2x$.

Mais les fonctions en python peuvent être bien plus générales : elles peuvent avoir plusieurs paramètres d'entrée, comme aucun, peuvent renvoyer plusieurs valeurs, comme aucunes, et faire alors plusieurs affichages ou actions variées.

Syntaxe générale pour des fonctions qui renvoient des valeurs :

```
def g(x,y,..):
    instructions indentées, diverses et variées
    return a,b,...
```

Dans le cas où il n'y a pas de paramètre d'entrée, il faut mettre comme en-tête : `def g():` et pour l'appeler on écrit `g()`, sans rien mettre dans les parenthèses.

Dans le cas où les fonctions ne renvoient aucune valeur spécifique mais ne font que des calculs et/ou des affichages, ne pas finir par `return` mais quitter l'indentation. Ces fonctions sont parfois appelées procédures.

Remarque 1 : mettre un print() dans une fonction n'est pas souhaitable, car cette valeur affichée ne pourra plus être utilisée (via une affectation) dans la suite du programme contrairement à une valeur obtenue via un return.

Remarque 2 : lorsque la fonction renvoie plusieurs valeurs, le type de la sortie est un `tuple` (tuple d'entiers ou tuple de réels ou ...). Ce n'est ni une liste, ni un tableau. C'est un type particulièrement basique : on ne peut rien en faire à part en extraire l'une des coordonnées.

Remarque 3 : Le paramètre d'une fonction n'a pas de type prédéfini. Selon les opérations effectuées au coeur de la fonction, on peut imaginer des paramètres d'entrée de différents types.
Par exemple, la fonction `f` ci-dessus (qui à x renvoie 2*x) peut être appliquée à un entier ou un réel, mais également à un tableau ou une matrice ...

2.3 Instructions conditionnelles

```
if condition:
    instructions indentées
else:
    instructions indentées
```

Cette instruction répond à l'attente :
si une condition est vraie (par exemple $x \neq 0$), alors on effectue une ou plusieurs instructions (par exemple, on divise par x) et sinon, on en effectue d'autres (par exemple on affiche un message d'erreur).

Attention :
Le `else` est facultatif : dans ce cas, rien n'est fait quand la condition est fausse.

Le `else` n'est pas suivi d'une condition, puisque le test a déjà été fait après le `if`. Le `else` signifie par lui-même que la condition est fausse dans ce cas.

Une condition est une phrase qui est vraie ou fausse (donc de type booléen) : par exemple `x>0`.
Opérateurs de comparaison possibles : `==` (égal à), `!=` (différent de)
`> ou <` (strictement supérieur/inférieur), `>= ou <=` (supérieur/inférieur ou égal),
On peut également avoir besoin d'opérateurs logiques entre deux conditions : `and` (et), `or` (ou), mais sachez que python comprend des conditions sous forme d'encadrement (double inégalités).

Attention : ne pas confondre $x = 3$ et $x == 3$ lorsque x est une variable.
$x = 3$ est une affectation : elle correspond à l'instruction "donner à x la valeur 3".
$x == 3$ est une condition (donc vraie au fausse) : elle correspond à la question "x est-il égal à 3 ?".

Remarque : dans le cas d'une instruction conditionnelle simple, on peut tout écrire sur une ligne
`instruction if condition else instruction`

Plus généralement avec des alternatives multiples, au lieu d'emboîter des `if`, on peut utiliser la syntaxe `elif` contraction de `else` et de `if`. Ce qui donne :

```
if condition:
    instructions indentées
elif condition:
    instructions indentées
else:
    instructions indentées
```

2.4 Boucle For

Cette instruction s'utilise pour répéter automatiquement une suite d'instructions n fois, lorsque n est connu à l'avance. La syntaxe est la suivante :

```
for i in .... :
    instructions indentées dépendant ou non de i
```

Par exemple, si l'en-tête est : `for i in range(n):`, le groupe d'instructions est effectué une première fois avec `i=0`, une deuxième fois avec `i=1`,...., puis une dernière fois avec `i=n-1`.

`for i in range(1,n+1):` le groupe d'instructions sera répété pour toutes les valeurs entières de i entre 1 et n.

La syntaxe générale de `range()` est : `range(depart, arrêt ,pas)` :
la valeur "départ" est facultative, et vaut 0 par défaut. Attention, la valeur "arrêt" n'est pas incluse !
Enfin, le "pas", facultatif également, est par défaut 1 ; il peut également être négatif.
On peut résumer en for i in `range(1ere valeur qui y est, 1ere valeur qui n'y est plus , le pas):`.

Plus généralement, on pourrait rencontrer des en-têtes du type `for i in T:` où T peut être un tableau ou une matrice, une liste ou une chaîne de caractère... Alors i prend successivement (et dans l'ordre) les "valeurs" de T.

2.5 Boucle While

L'instruction `while` s'utilise pour répéter un groupe d'instructions tant qu'une condition est vraie : on ne connaît donc plus a priori le nombre de répétitions nécessaires. La syntaxe est la suivante :

```
while condition:
    instructions indentées
```

Remarques :

1. Le groupe d'instructions peut ne pas être exécuté du tout (si dès le départ la condition est fausse).
2. Avant d'éxéctuer une boucle `while`, vérifier que la condition devient fausse au bout d'un certain temps. Sinon, le programme ne s'arrêtera jamais.
3. Souvent, on aura besoin de connaître le nombre d'exécutions de la boucle. Dans ce cas, il suffira de mettre en place une variable compteur, qui est initialisée avant la boucle `while` (par exemple `c=0`), puis qui est incrémentée de 1 à chaque passage, avec l'instruction `c=c+1` ou en raccourci `c+=1`, dans la boucle.

Compléments pour aller plus loin :

— Pour sortir prématurément d'une boucle `while` (ou `for`), on peut avoir recours à la commande `break` : par exemple, si la syntaxe
```
if a==0:
    break
```
apparait dans la boucle, la boucle est interrompue dès que la variable `a` vaut 0.

— Parfois, on ne veut pas exécuter une partie des instructions de la boucle `while` (ou `for`) lorsqu'une variable prend une certaine valeur. Dans ce cas, on peut utiliser la commande `continue` comme suit :
```
if a==0:
   continue
```
Les instructions qui sont entre le `continue` et la fin de la boucle ne sont alors pas exécutées lorsque `a` vaut 0. La boucle se poursuit ensuite normalement.

3 Mathématiques et python : la bibliothèque `numpy`

3.1 Généralités

La bibliothèque `numpy` permet d'effectuer des calculs numériques avec python, et de manipuler des tableaux (=vecteurs=atrices lignes), ou plus généralement des matrices de toute taille.
Au moment de commencer votre session, il faudra importer cette bibliothèque via la commande :
`import numpy as np`

Variantes possibles pour l'importation :
(a) `import numpy` : dans ce cas, toutes vos commandes devront être précédées de numpy et non du préfixe np
(b) si l'on veut ne pas avoir de préfixe : `from numpy import *` (le * signifiant "tout" ; sinon remplacer * par le nom des fonctions que l'on veut introduire : par exemple `from numpy import exp, log`)

Dans toute la suite, (et conformément au programme) je considère que la bibliothèque numpy a été importée avec l'alias np via la syntaxe : `import numpy as np`

Variables prédéfinies : π et e via les commandes `np.py` et `np.e`

Fonctions usuelles prédéfinies : `np.exp` `np.log` pour le ln `np.sin` `np.cos` `np.tan` `np.sqrt` pour la racine carrée, `np.abs` pour la valeur absolue, `np.floor` pour la partie entière, etc.

Comme ce sont des fonctions python :
— ne pas oublier les ( ) au moment de les appliquer, par exemple : `np.exp(3)`
— le paramètre d'entrée n'a pas un type prédéfini. Il peut bien sûr être un entier ou un réel, mais également un tableau ou une matrice : dans ce cas, la fonction s'applique à tous les coefficients de la matrice.

3.2 Création et modification des tableaux (=vecteurs)

Les tableaux (= vecteurs = matrices lignes) peuvent être crées **"à la main"**, c'est-à-dire en tapant tous leurs coefficients, avec la commande np.array(). Par exemple,
la syntaxe `V=np.array([0,-1,1])` crée le tableau $(0,-1,1)$.
Variante : `V=np.array((0,-1,1))`

Le type de ces objets est `numpy.ndarray`.
Pour connaitre la taille du tableau V : `np.shape(V)`. Dans l'exemple précédent, l'affichage serait `(3,)`.

Tableaux ou vecteurs prédéfinis
— `np.zeros(n)` crée un vecteur de taille n ne contenant que des 0.
— `np.ones(n)` crée un tableau de taille n ne contenant que des 1.

— la syntaxe `np.linspace(deb,fin,nbre)` permet de créer un vecteur qui contiendra `nbre` éléments, en partant de la valeur `deb` et en s'arrêtant à la valeur `fin` tous également espacés. Par exemple
`np.linspace(2,5,7)` renvoie le vecteur $(2, 2.5, 3, 3.5, 4, 4.5, 5)$.
On verra l'intérêt de cette commande au moment des graphiques.

— la commande `np.arange()` a les mêmes arguments que la commande `range()` mais renvoie un objet de type matriciel `numpy.ndarray` (et non de type `range`) ce qui permet ensuite d'utiliser les opérations mathématiques de base. Par exemple
`np.arange(5)` renvoie le vecteur $(0, 1, 2, 3, 4)$
`np.arange(1,5)` renvoie le vecteur $(1, 2, 3, 4)$
`np.arange(5,15,2)` renvoie le vecteur $(5, 7, 9, 11, 13)$
Plus généralement, `np.arange(1ere valeur, 1ere valeur qui n'y est plus, pas)`

Numérotation :
Attention, la numérotation est particulière en python : le premier coefficient d'un tableau est le coefficient 0.

Soit un tableau `T` de taille n : ses coefficients sont donc numérotés de 0 à $n-1$.
En particulier `T[2]` permet d'atteindre le 3e coefficient du tableau `T`.

A savoir : pour aller chercher les derniers coefficients, on pourra utiliser des entiers négatifs.
`T[-1]` est le dernier coefficient du tableau `T` ; `T[-2]` est l'avant-dernier coefficient, `T[-3]` l'avant-avant dernier etc.

On peut modifier a posteriori un coefficient d'un vecteur via une affectation : `T[0]=1` modifiera le 1er coefficient du vecteur `T` pour lui affecter la valeur 1.

Plus généralement, pour extraire (et/ou modifier) une partie d'un vecteur, on pourra utiliser les syntaxes : `T[1:2]` (extrait les 2e et 3e coefficients de `T`), ou `T[2:]` (extrait tous les coefficients de `T` à partir du 3e).

Enfin, pour ajouter un élément `b` à un tableau `T`, on pourra utiliser la syntaxe `np.append(T, b)`.

Construction d'un tableau à l'aide d'une boucle for
On commence par prédéfinir le vecteur U (de bonne taille) : si il est de taille n

```
U=np.zeros(n)
U[0]=   # mettre alors le 1er coefficient de U
for i in range(1,n):
    U[i]=   # mettre alors le bon coefficient de U, qui peut dépendre de U[i-1]
```

Dans le cas où les coefficients de U se calculent au fur et à mesure et que vous préférez faire les calculs à part :

```
U=np.zeros(n)
u=
U[0]= u
for i in range(1,n):
    u=
    U[i]= u
```

Variante par construction du vecteur au fur et à mesure :

```
u=
U=np.array([u])
for i in range(n-1)
u=
U=np.append(U,u) # on "rajoute" u à notre vecteur U (pas au sens du + mais de la concaténation)
```

3.3 Opérations sur les tableaux =vecteurs

— Les opérations arithmétiques $+ * - / **$ se font coefficients par coefficients.

Pour le + (comme le −) entre des vecteurs, cela correspond donc à l'opération mathématique .
Cela n'est plus le cas pour les opérations $*$, / et $**$ (qui n'existent pas pour des matrices lignes ...)
Par exemple, si $L = (1, 2, 3, 4)$, la commande python `L**2` renverra le vecteur $(1, 4, 9, 16)$ (Python a appliqué le carré à chaque coefficient).

De plus, l'opération réel + vecteur a un sens : python rajoute le réel à chaque coefficient du vecteur.

— Rappel : on peut appliquer toutes les fonctions usuelles à des vecteurs.

— On pourra également utiliser les commandes suivantes : `np.sum(L)` (somme tous les coefficients du vecteur L), `np.prod(L)` pour le produit, `np.min(L)` et `np.max(L)` pour connaître le coefficient le plus petit (resp. grand) du vecteur etc.

— En probabilités, on verra l'intérêt de faire des tests sur les vecteurs : par exemple `L==N` ou `L>=1`... Les tests se font alors terme à terme.

3.4 Matrices rectangulaires (= tableaux bidimensionnels)

Pour créer une matrice à la main : `np.array()`

`np.array([[1,2,3],[4,5,6]])` crée la matrice $\begin{pmatrix}1&2&3\\4&5&6\end{pmatrix}$. (*attention aux doubles crochets !*)

Variante : `np.array(((1,2,3),(4,5,6)))` (*attention aux nombreuses parenthèses !*)

Matrices prédéfinies :
— `np.zeros([n,m])` (ou `np.zeros((n,m))`) crée une matrice de taille $n \times m$ remplie de 0.
— `np.ones([n,m])` (ou `np.ones((n,m))` permet de prédéfinir des matrices ne contenant que des 1.
— `np.eye(n)` renvoie la matrice identité de taille n.

Pour connaître la taille d'une matrice `np.shape()` :
si la matrice est de taille $n \times m$, cette commande renvoie (n, m).

Numérotation :
Comme pour les vecteurs, la numérotation est particulière en python : la première colonne s'appelle la colonne 0 et la première ligne est la ligne 0.

Soit M une matrice de taille $n \times m$: ses colonnes sont donc numérotées de 0 à $m-1$, et ses lignes sont numérotées de 0 à $n-1$.
`M[i,j]` est le coefficient situé à l'intersection de la colonne i et de la colonne j pour python : il correspond donc au coefficient $M_{i+1,j+1}$ de la matrice M. En particulier $M_{1,1}$ correspondra à l'élément `M[0,0]`.

Comme pour les vecteurs, on pourra utiliser des entiers négatifs type -1 ou -2.
Par exemple `M[0,-1]` représente le coefficient 1ere ligne et dernière colonne.

Modifications : on peut modifier a posteriori un coefficient d'une matrice via une affectation : `M[0,0]=1` modifiera le 1er coefficient de la matrice pour lui affecter la valeur 1.
Plus généralement, on pourra extraire et/ou modifier toute une ligne ou une colonne via
`M[i-1,:]` pour la i^e ligne et `M[:,j-1]` pour la j^e colonne.
Attention, `M[i-1,:]` comme `M[:,j-1]` seront considérées comme des matrices lignes par python.

Opérations sur les matrices rectangulaires
Comme pour les vecteurs, les opérations arithmétiques $+ \ * \ - \ / \ **$ se font coefficients par coefficients.

Le produit ne correspond donc pas au produit matriciel !! Par exemple, si $M = \begin{pmatrix}1&2\\3&4\end{pmatrix}$, la commande python `M**2` renverra la matrice $\begin{pmatrix}1&4\\9&16\end{pmatrix}$ (python a appliqué le carré à chaque coefficient). Ce n'est donc pas la matrice M^2.

Pour pouvoir effectuer le produit matriciel entre deux matrices N et M : `np.dot(N,M)` (raccourci :`M@N`).
Pour un produit matriciel de 3 matrices : `np.dot(M,np.dot(N,P))` (raccourci : `M@N@P`).

On peut également appliquer toutes les fonctions usuelles à des matrices.
Enfin, on pourra utiliser les commandes suivantes : `np.sum(M)`, `np.prod(M)`, `np.min(M)` et `np.max(M)` , `np.mean(M)`.

Bibliothèque supplémentaire :`numpy.linalg` `import numpy.linalg as al`

Cette bibliothèque permet d'obtenir l'inverse d'une matrice `M` via la commande `al.inv(M)`
ansi que les puissances de matrice via la commande `al.matrix_power(M,5)`.
Enfin, la commande `al.solve(A,B)` donne l'unique solution X du système $AX = B$ lorsque A est inversible.

4 Graphiques et python : la bibliothèque `matplotlib.pyplot`

La bibliothèque `matplotlib.pyplot` permet de créer des graphiques et de les afficher. Comme pour toute librairie, au moment de commencer votre session, il faut l'importer via la commande :
`import matplotlib.pyplot as plt`

Comme vos graphiques feront intervenir des vecteurs ou/et des fonctions usuelles, il vous faudra également importer la bibliothèque `numpy`.

Remarque importante : python construit une courbe point par point et interpole entre ces points (c'est-à-dire les relie). Pour représenter une courbe dans python, il faut donc préciser un vecteur d'abscisses `x`, le vecteur des ordonnées correspondantes `y`, puis utiliser les syntaxes

```
plt.plot(x,y) # crée le graphique
plt.show() # affiche le graphique (inutile avec certains éditeurs)
```

Pour représenter une suite de nombres, l'interpolation ne doit plus être faite entre les points. La syntaxe devient par exemple :
`plt.plot(x,y,"+")` # ou "x", ou "o" ... selon les symboles que l'on veut pour marquer les points !

−− > Beaucoup d'arguments sont possibles dans la syntaxe `plt.plot` : la couleur de la courbe, l'épaisseur, la légende. Aucun de ces arguments n'est à connaitre par coeur. Certains seront vus dans les exercices. Pour la plupart, il est assez facile de deviner leur signification !

−− > Pour définir le domaine des axes : on pourra utiliser les commandes
`plt.xlim(xmin, xmax)` et `plt.ylim(ymin, ymax)`.
Pour que le repère soit orthonormé, on pourra rajouter la commande `plt.axis("equal")`.

Comment définir le vecteur des abscisses `x` ?
- via la syntaxe `np.arange(1ere valeur, 1ere valeur qui n'y est plus, pas)`
- via la syntaxe `np.linspace(deb,fin,nbre)` (rappel : crée un vecteur qui contiendra `nbre` éléments, en partant de la valeur `deb` et en s'arrêtant à la valeur `fin` tous également espacés).

En effet comme python interpole les points, il faut suffisamment de points pour que la courbe interpolée par python ressemble à la courbe mathématique !

Par exemple, si l'on veut représenter une courbe sur [0,1] :
`x= np.arange(0,1,0.01)` # pas de 0.01 pour avoir une centaine de points (maintenant si la courbe est très régulière, un pas de 0.1 peut suffire)
ou
`x=np.linspace(0,1,100)` # pour avoir 100 points dans le vecteur `x`.

Comment définir le vecteur des ordonnées correspondantes ?
- via les fonctions usuelles : en effet toutes les fonctions usuelles peuvent être appliquées à un vecteur (donc en particulier au vecteur d'abscisses `x`) et renvoient alors le vecteur des images correspondantes.
- via une fonction que vous avez créée au préalable, et qui peut renvoyer un vecteur.

5 Probabilités et python : la bibliothèque `numpy.random`

Commencer par importer la bibliothèque correspondante : `import numpy.random as rd`

5.1 Simulations d'expériences

Pour simuler la majorité des expériences aléatoires, il suffit de connaitre deux commandes :
- `rd.randint(a,b+1)` avec a et b des entiers : choisit au hasard un entier entre a et b.
  Comme pour la syntaxe `range` ou `np.arange`, *l'entier b+1 est exclu.*
- `rd.random()` (*ne rien mettre entre les parenthèses !*) : choisit au hasard un réel entre 0 et 1.

5.2 Echantillons de lois usuelles

Pour obtenir des simulations de lois usuelles discrètes, voici les commandes à connaitre :
- N simulations de $\mathcal{B}(n,p)$: `rd.binomial(n,p,N)`
- N simulations de $\mathcal{U}([\![a,b]\!])$: `rd.randint(a,b+1,N)`
- N simulations de $\mathcal{G}(p)$: `rd.geometric(p,N)`
- N simulations de $\mathcal{P}(\lambda)$: `rd.poisson(lambda,N)`
- N simulations de la loi uniforme sur $[0,1[$ càd N réels tirés au hasard entre 0 et 1 : `rd.random(N)`

Quelque soit la loi utilisée, la sortie est un vecteur(=tableau) formé de ces N simulations, donc de type `array`.

Pour obtenir la moyenne ou la variance d'un échantillon ℓ, on utilisera les syntaxes `np.mean(ℓ)` et `np.var(ℓ)`.

Complément :
La moyenne d'un échantillon "représente" l'espérance de la loi associée. En effet, en deuxième année vous verrez le résultat suivant :
Si X_1, X_2, ..., X_n sont des variables aléatoires indépendantes, suivant toutes la même loi et admettant une même espérance $E(X)$ alors $\frac{X_1+X_2+...+X_n}{n} \underset{n\to+\infty}{\longrightarrow} E(X)$ (la limite ayant un sens qui sera défini l'an prochain).
Dans le TP 7, vous pourrez constater que la moyenne empirique obtenue via la syntaxe `np.mean()` donne bien une valeur approchée de l'espérance de la loi commune aux simulations.

5.3 Histogrammes :

`bibliothèque matplotlib.pyplot`

commande `plt.hist(vecteur, range=(---,---), bins=---)` où

`vecteur` contiendra toutes les simulations voulues
`range` définira la taille de la fenêtre c-est-à-dire les valeur min et max souhaitées dans l'histogramme (par défaut, `range =(min(vecteur), max(vecteur))`)
`bins` le nombre d'intervalles

Pour les représentations des lois usuelles discrètes, il sera important de bien corréler les valeurs dans `range` et dans `bins`, afin que les intervalles de l'histogramme soient centrés autour des valeurs prises.

Exemple : où `V` est le vecteur de simulations souhaité, `m=np.min(V)`, `M=np.max(V)`.
On veut donc qu'il y ait des intervalles centrés autour des entiers entre m et M. Le premier intervalle commencera à $m-0.5$ (et finira à $m+0.5$, pour ne contenir que la valeur m), et le dernier intervalle finira à $M+0.5$ (pour contenir là encore la valeur prise M). Comme $M+0.5=(M+1)-0.5$, on définira :
`b=np.arange(m-0.5,M+1.5)`
`plt.hist(V,range=(m,M),bins=b,rwidth=0.5)` (le dernier argument permet de réduire la largeur des barres)

Bien sûr, les options de `plt.hist` ne sont pas à connaitre par coeur !

Variante : on peut obtenir une syntaxe simplifiée via l'instruction `plt.hist( vecteur, range=(---,---), bins=100)`.
En effet, `bins=100` impose de faire 100 classes, ce qui permet bien de n'avoir qu'une seule valeur prise (au plus) dans chaque classe (puisqu'en pratique, il n'y a qu'un petit nombre de valeurs vraiment prises) : donc sur les 100 bâtons seulement quelques uns n'auront pas une hauteur nulle, et seront alors visibles. Mais les bâtons ne seront pas centrés autour des valeurs prises, donc la lecture de l'histogramme est un peu moins facile.

Dans certains éditeurs, après exécution de la commande `plt.hist`, sont affichés 3 choses :

1. le tableau qui contient le nombre d'occurrences de `V` dans chaque intervalle.
2. les bornes des intervalles utilisés.
3. l'histogramme lui-même.

Mais avec l'éditeur Spyder, sauf à taper `print(plt.hist(....))`, seul l'histogramme s'affichera.

Par défaut, la hauteur des barres de l'histogramme correspond au nombre d'occurrences (apparaissant dans `vecteur`) dans l'intervalle correspondant. Pour que les hauteurs correspondent aux fréquences d'apparition (on dit alors que l'histogramme est renormalisé), on rajoutera l'argument `density=True` dans `plt.hist()`

Remarque : pour faire afficher la loi théorique d'une variable aléatoire, on ne peut plus utiliser des simulations de cette variable. On ne fera donc plus un histogramme, mais un diagramme en bâtons, qui permet de mettre en hauteur du bâton la probabilité théorique correspondante.
La syntaxe n'est pas explicitement au programme, donc elle sera rappelée en exercice :
`plt.bar(abscisses, hauteurs, color='r', width=0.3)`, les abscisses et les hauteurs étant des vecteurs, r permet d'avoir la couleur rouge, et le 0.3 permet d'avoir une largeur de bâtons plus fine, pour que le 2e graphique ne recouvre pas le 1er.
Attention, pour comparer l'histogramme obtenus par des simulations avec le diagramme de la loi théorique, il faudra penser à renormaliser l'histogramme : car ce sont les fréquences d'apparition que l'on peut comparer aux probabilités théoriques d'apparitions (et non le nombre d'occurrences).