TP 1 : Aléatoire en Python

Consignes pour tous les TPs.

1. Créer un fichier Python par exercice (par exemple `TP1exo1.py`, `TP1exo2.py`, ...) dans lequel vous écrivez les fonctions.
2. Pour écrire les fichiers Python et les exécuter, il y a plusieurs possibilités. Vous pouvez utiliser celle que vous préférez. Quelques choix possibles sur les machines de la FdS :
   - éditer les fichiers Python dans votre éditeur de texte préféré (vim, ...), et les exécuter avec IPython (lancer `ipython3` dans un terminal, et `%run xxx.py` pour exécuter le fichier `xxx.py`) ;
   - utiliser un IDE spécialisé pour Python comme Spyder3 ou Pyzo, très similaire en plus épuré (pour utiliser IPython dans Pyzo, aller dans le menu « Shell > Edit shell configurations... » et cocher « Use IPython shell if available ») ;
   - utiliser un IDE généraliste comme VSCodium (à configurer).
3. Enregistrez bien vos fichiers et gardez les tout au cours du semestre !
4. ***Attention !*** *Dans ce cours d'algorithmique, les TPs sont faits en Python 3. Il faut prendre garde à ne pas utiliser de la syntaxe et des fonctionnalités de SageMath, qui sont vues dans le cours de calcul formel et scientifique.*

Exercice 1. *Découverte des bibliothèques*

L'objectif de cet exercice est de découvrir les fonctionnalités de base de la bibliothèque `random` qui permet de tirer des entiers, flottants, etc. de manière (pseudo-)aléatoire. Pour illustrer les tirages effectuées, on utilise la bibliothèque `matplotlib` qui permet de tracer des graphiques. On peut charger ces deux bibliothèques de la manière suivante :

```
from random import *                     # chargement de toute la bibliothèque
from matplotlib import pyplot as plt     # chargement d'un des modules de matplotlib
```

1. La bibliothèque `random` permet de tirer des entiers aléatoires, avec les fonctions `randint` et `randrange`.
   - **i.** Écrire une fonction `entiersAleatoires(n,a,b)` qui renvoie une liste de `n` entiers aléatoires tirés avec `randint(a,b)`.
   - **ii.** Écrire une fonction `entiersAleatoires2(n,a,b)` qui renvoie une liste de `n` entiers aléatoires tirés avec `randrange(a,b)`.
   - **iii.** *Test.* Générer deux listes L1 et L2 de taille 1000 avec les deux fonctions précédentes, avec `a = 1` et `b = 100`. Tracer l'histogramme de chacune avec `matplotlib` :

     ```
     plt.hist(L1, bins=100) # histogramme avec 100 colonnes
     plt.show()             # Affichage du résultat
     ```

     Les distributions semblent-elles uniformes ? Quelle est la différence entre `randint` et `randrange` ?
2. La bibliothèque `random` permet également de tirer des flottants aléatoires, avec `random()` et `uniform(a,b)`.
   - **i.** Écrire une fonction `flottantsAleatoires(n)` qui renvoie une liste de `n` flottants aléatoires tirés avec `random()`.
   - **ii.** Écrire une fonction `flottantsAleatoires2(n,a,b)` qui renvoie une liste de `n` flottants aléatoires tirés avec `uniform(a,b)`.
   - **iii.** *Test.* Déterminer le comportement de `random` et `uniforme` en générant deux listes de taille 1000 avec les fonctions précédentes, avec par exemple `a = -3` et `b = 10` pour la seconde. Tracer les valeurs obtenues avec `matplotlib` :

     ```
     plt.plot(L) # Ligne brisée
     plt.show()
     ```
3. On cherche à générer des points dans un disque D de centre $(0,0)$ et de rayon 1. Pour cela, on tire deux réels x et y aléatoires entre -1 et 1, et on teste s'ils vérifient l'équation du disque $x^2+y^2 \leq 1$. Si ce n'est pas le cas, on *rejette* le point (x,y) et on recommence.

i. Écrire une fonction `pointsDisque(n)` qui tire n points dans D, avec la méthode décrite. *Attention : il ne suffit pas de tirer n couples, on veut n points dans le disque.*

ii. Afin d'effectuer moins de tirages aléatoires, on utilise la méthode suivante : pour tirer un point (x, y), on commence par tirer x, qu'on garde jusqu'à la fin ; ensuite, on tire y et on teste si $(x, y) \in D$; si ce n'est pas le cas, on *rejette* uniquement y et on tire une nouvelle valeur de y. Écrire une fonction `pointsDisque2(n)` qui tire n points dans D avec cette deuxième méthode.

iii. Copier la fonction `affichagePoints` qui dessine une liste de couples (x, y).

```
def affichagePoints(L):
    X = [x for x,y in L] # on récupère les abscisses...
    Y = [y for x,y in L] # ... et les ordonnées
    plt.scatter(X, Y, s = 1) # taille des points minimale
    plt.axis('square')       # même échelle en abscisse et ordonnée
    plt.show()
```

iv. *Test.* Créer deux listes de points avec les méthodes précédentes, et les afficher. Les deux méthodes fournissent-elles une distribution uniforme sur le disque ? *Penser à utiliser suffisamment de points, et tenter de trouver une explication à ce que vous observez !*

4. La bibliothèque `random` propose la fonction `getrandbits(k)` qui renvoie un entier d'exactement k bits.

i. Pour générer un entier entre 0 et $N-1$ où N possède k bits, on peut tirer un entier x de k bits et renvoyer $x \bmod N$. Écrire une fonction `aleatoireModulo(N)` qui effectue ce tirage.

ii. Une autre méthode consiste à tirer un entier x de k bits : on renvoie x si $x < N$, sinon on recommence. Écrire une fonction `aleatoireRejet(N)` qui effectue ce tirage.

iii. Générer deux suites de 10 000 entiers aléatoires entre 0 et 100 avec `aleatoireModulo` et `aleatoireRejet`. Représenter graphiquement chacune des deux suites sous forme d'histogramme. Laquelle des deux méthodes produit des entiers uniformes entre 0 et $N-1$?

Exercice 2. *Élément majoritaire*

Un élément est *majoritaire* dans un tableau T de taille n s'il apparaît au moins $n/2$ fois, c'est-à-dire $\#\{i : T_{[i]} = x\} \geq n/2$.

1. On s'intéresse à la recherche de l'élément majoritaire dans un tableau qui en possède un.

i. Écrire une fonction de complexité quadratique `eltMajDet(T)` qui prend en entrée un tableau `T` *possédant un élément majoritaire* et renvoie cet élément majoritaire.

ii. Écrire une fonction `eltMajProba(T)`, probabiliste, qui effectue la même tâche en tirant aléatoirement des éléments dans `T` jusqu'à trouver un élément `x` qui est majoritaire. *On peut tirer un élément aléatoirement et uniformément dans une liste avec* `choice(T)`.

Pour tester les deux fonctions, on cherche à construire des tableaux T aléatoires mais qui contiennent un élément majoritaire. Pour construire T de taille n avec un élément majoritaire qui apparaît $k \geq n/2$ fois, on commence par tirer un élément majoritaire m entre deux bornes a et b. Ensuite, on tire $n-k$ entiers aléatoires entre a et b, différents de m. On peut alors construire T en mettant k copies de m et les $n-k$ entiers aléatoires différents de m.

iii. Écrire une fonction `tabAlea(n, a, b, k)` qui implante l'algorithme précédent. *On peut contruire le tableau en mettant toutes les copies de m au début ou à la fin, puis le mélanger grâce à* `shuffle(T)`.

iv. On souhaite maintenant créer des tableaux dans un ordre particulier. Écrire deux fonctions `tabDeb` et `tabFin` similaires à `tabAlea` :

— `tabDeb` renvoie un tableau dont le premier élément est l'élément majoritaire ;
— `tabFin` renvoie un tableau dans lequel l'élément majoritaire occupe les k dernières cases.

v. *Test.* Comparer les temps de calculs de `eltMajDet` et `eltMajProba` sur des tableaux de tailles 1 000 à 10 000, dans les situations suivantes (on peut calculer le temps de calcul avec `%time`) :

— tableau aléatoire, avec k qui augmente de $n/2$ à n ;
— tableau avec l'élément majoritaire en $1^{\text{ère}}$ case, avec k de $n/2$ à n ;
— tableau avec l'élément majoritaire à la fin, avec k de $n/2$ à n.

Déterminer les cas où l'algorithme déterministe est meilleur, ceux où l'algorithme probabiliste est meilleur, et ceux où les deux algorithmes ont une complexité comparable.

2. On cherche maintenant à détecter s'il existe un élément majoritaire dans un tableau quelconque. On utilise l'algorithme probabiliste suivant : on tire m éléments $x_1, \dots, x_m$ aléatoirement dans T, et on renvoie VRAI si l'un au moins de ces éléments est majoritaire, et FAUX sinon.
L'algorithme renvoie forcément FAUX s'il n'y a pas d'élément majoritaire, mais peut également renvoyer FAUX par erreur dans le cas où il existe un élément majoritaire.

- **i.** Écrire une fonction `contientEltMaj(T, m)` qui implante l'algorithme décrit.
- **ii.** Pour tester la probabilité de succès de l'algorithme, on répète N fois l'expérience suivante : on tire un tableau aléatoire de taille n ayant un élément majoritaire apparaissant k fois, et on appelle `contientMajProba`. On compte le nombre de succès (VRAI) de l'algorithme. Écrire une fonction `testContient(n, a, b, k, m, N)` qui effectue ce test et renvoie la proportion de réussites.
- **iii.** On fixe $n = 1000$, $m = 1$ et $N = 1000$. Calculer la liste des proportions de réussites, pour $k = 500$, $550, \dots, 1000$ et tracer cette liste avec `plt.plot(...)`. Que concluez-vous ?
- **iv.** On fixe $n = 1000$, $k = 500$ et $N = 1000$. Calculer la liste des proportions de réussites, pour $m = 1$ à 10 et tracer cette liste avec `plt.plot(...)`. Que concluez-vous ?

Exercice 3. *Générateurs pseudo-aléatoires*

On s'intéresse aux générateurs congruentiels linéaires, qui sont des suites $(X_n)_n$ définies par une valeur initiale X_0 et une équation de récurrence $X_{n+1} = (aX_n + c) \bmod m$, où a, c et m sont des entiers définissant le générateur.

1.
- **i.** Écrire une fonction `suivant(xn, a, c, m)` qui calcule X_{n+1} à partir de X_n.
- **ii.** Écrire une fonction `valeurs(x0, a, c, m, N)` qui calcule les N premiers termes de la suite $(X_n)_n$, étant donné X_0.
- **iii.** *Tests.* Vérifier que les 20 premiers termes de la suite définie par $X_0 = 0$, $a = 3$, $c = 5$ et $m = 136$ sont `[0, 5, 20, 65, 64, 61, 52, 25, 80, 109, 60, 49, 16, 53, 28, 89, 0, 5, 20, 65]`. Que pouvez-vous prédire sur les termes suivants ?

On cherche à étudier les propriétés statistiques des générateurs congruentiels linéaires, en fonction des choix de a, c et m.

2. Le ZX81 contenait un générateur congruentiel linéaire avec paramètres $a = 75$, $c = 74$ et $m = 2^{16} + 1$. Calculer les 2^{20} premières valeurs à partir d'une graîne quelconque. À partir de quel moment les valeurs se répètent-elles ?

3. Dans (certaines versions de) la bibliothèque standard du langage C, le générateur utilisé a pour paramètre $a = 16807$, $c = 0$ et $m = 2^{31} - 1$.

- **i.** Vérifier graphiquement avec plusieurs graînes que le générateur semble bien aléatoire. *Utiliser* `plt.plot(V, 'r.')` *par exemple pour représenter les valeurs contenues dans* `V` *par des points rouges.*
- **ii.** On veut vérifier si deux graînes distinctes fournissent deux suites d'entiers aléatoires indépendantes. Calculer les 10000 premières valeurs obtenues à partir des graînes 123489 et 25743. Représenter graphiquement une suite par rapport à l'autre avec `plt.scatter(V1, V2, s=.2)`. Les deux suites semblent-elles indépendantes ?
- **iii.** Essayer de même avec deux graînes consécutives 123488 et 123489. Puis avec 5 et 6. Que concluez-vous ?

4. Plutôt que faire les calculs *à la main* avec `suivant`, il est plus propre de représenter un générateur congruentiel linéaire avec une classe Python. Définir une classe `Generateur` avec les caractéristiques suivantes :

- le constructeur `__init__` prend en entrée les trois paramètres a, b et m, et initialise des attributs `__a`, `__c` et `__m` à leur valeur, ainsi qu'un attribut `__x` à `None` ;
- une méthode `graine` prend en paramètre une graîne `g` et affecte la valeur `g` à `__x` ;
- une méthode `aleatoire` sans paramètre met à jour `__x` avec la valeur suivante et le renvoie (si `__x` n'est pas initialisé, une exception est soulevée) ;
- ajouter toutes les méthodes qui semblent intéressantes, telles que des méthodes d'affichage (`__repr__` et `__str__`).

La classe doit pouvoir s'utiliser de la manière suivante.

```
>>> g = Generateur(3, 5, 17)
>>> g.aleatoire()
...
ValueError: générateur non initialisé !
>>> g.graine(5)
>>> g.aleatoire()
3
```