MODÉLISATION D'EXPÉRIENCES ALÉATOIRES

Nous proposons ici de modéliser des expériences aléatoires simples avec des listes et le module `random`. Ce type de modélisation se trouve au cœur des oraux de mathématiques de type agro-veto (même si les oraux ne s'y résument pas !)

Dans le chapitre sur `numpy`, nous diposerons d'autres outils pour le faire.

Avertissement : il existe de multiples façons de modéliser les expériences. Nous proposons souvent plusieurs méthodes possibles. Le but n'est pas de faire un catalogue, mais de s'approprier celles qui nous parlent le plus.

1 TIRAGES DANS UNE URNE MODÉLISÉE PAR UNE LISTE

A Modélisation d'une urne

Un moyen simple de modéliser une urne est d'utiliser une liste que l'on remplit comme l'urne. Par exemple, pour une urne qui contient p boules blanches et q boules noires, on peut créer une liste avec p chiffres 1, et q chiffres 0.

On adaptera sa modélisation en fonction du contexte. Si on veut compter le nombre de boules blanches pour différents tirages, ce seront elles que l'on codera par 1. Si, au contraire, on compte plutôt les boules noires, alors on échangera le codage. Le plus important ici est de *commenter* son code pour expliquer à l'utilisateur la convention choisie.

```
def modelUrne(p,q):
    urne = p*[1]+q*[0]
    return urne
```

Bien sûr, on peut généraliser à davantage de couleurs ou de valeurs.

B Tirage avec remise

Méthode de base : Pour tirer un objet avec remise, il suffit de le désigner dans l'urne (sans y toucher). On pourra utiliser pour cela la commande **choice**.
On peut ensuite réiterer l'expérience aléatoire de nombreuses fois de façon indépendantes. Dans certains cas, on voudra conserver la liste des résultats successifs, ou simplement le nombre de succès au cours des n tirages (par exemple le nombre de fois que l'on tire le chiffre 1).

```
# tirage avec remise
def tirage(urne):
    return choice(urne)

# tirages successifs avec remise.
# on enregistre les résultats dans une liste.
def tirageSuccessif(urne, nb):
    res = []
    for i in range(nb):
        res.append(choice(urne))
    return res

# tirages successifs avec remise.
# on compte le nombre de fois que l'on a obtenu la valeur k.
def tirageNbSucces(urne, k, nb):
    succes = 0
    for i in range(nb):
        succes += (choice(urne) == k)
    return succes
```

Méthode alternative : On peut également modéliser ce tirage en choisissant une position aléatoire dans l'urne. On utilise alors la commande **randrange**(`i,j,k`) qui tire un nombre aléatoire dans l'intervalle donné par **range**(`i,j,k`). Comme pour la commande **range**, on peut diminuer le nombre d'arguments.

```
# tirage avec remise
# utilisation de randrange
```

```
def tirage(urne):
    indice = randrange(len(urne))
    return (urne[indice])

# tirages successifs avec remise.
# on enregistre les résultats dans une liste.
def tirageSuccessif(urne, nb):
    res = []
    for i in range(nb):
        res.append(tirage(urne))
    return res

# tirages successifs avec remise.
# on compte le nombre de fois que l'on a obtenu la valeur k.
def tirageNbSucces(urne, k, nb):
    succes = 0
    for i in range(nb):
        succes += (tirage(urne) == k)
    return succes
```

C Tirage sans remise :

Cette fois-ci, il faut modifier l'urne à chaque tirage. Pour cela nous proposons de fonctionner avec effets de bords. Cela signifie que la liste (variable supposée globale) qui représente l'urne est modifiée à chaque tirage.

Attention :
Si on veut effectuer plusieurs fois l'expérience de façon indépendante, alors il faudra recréer une nouvelle urne à chaque fois.
(ou travailler avec des copies d'une urne *originale*, en veillant à ce que les copies ne pointent pas toutes vers le même objet mémoire)

Méthode de base : La méthode la plus simple est sans doute de mélanger l'urne avec **shuffle** puis de tirer le dernier élément.

Lorsque l'on veut réaliser plusieurs tirages, il suffit d'un seul `shuffle`, puis on tire les éléments les uns à la suite des autres avec `pop()`.

```
# tirage sans remise
def tirage(urne):
    shuffle(urne)
    return (urne.pop())

# n tirages sans remise
# mélange l'urne, mais ne la modifie pas.
def tirageSuccessif(urne, nb):
    shuffle(urne)
    return (urne[:nb])

# tirage sans remise,
# on compte le nombre de fois que l'on obtient la valeur k au cours de nb tirages
# l'urne est modifiée à la fin
def tirageNbSucces(urne, nb):
    succes = 0
    shuffle(urne)
    for i in range(nb):
        succes += (urne.pop() == 1)
    return succes

# pour ne pas modifier l'urne initiale
def tirageNbSucces(urne, nb):
    succes = 0
    urneLocale = urne[:]
    shuffle(urneLocale)
```

```
    for i in range(nb):
        succes += (urneLocale.pop() == 1)
    return succes
```

Remarque : Ici, dans l'algorithme du tirage successif, on prend les premiers éléments de la liste (sans les enlever), alors que dans les autres fonctions, on prend les derniers éléments (et on les enlève avec `pop()`). Cela n'a pas d'influence sur l'aléatoire, mais nous avons simplement écrit le plus facile à programmer dans chaque situation.

Lorsqu'il s'agit d'un **tirage sans remise exhaustif** : on tire tous les éléments de l'urne. Cela revient simplement à choisir un ordre aléatoire pour notre liste (c'est une permutation).

Il suffit alors de la seule commande

```
shuffle(urne)
```

Méthode alternative : Comme pour le tirage avec remise, on peut aussi pointer directement vers un indice avec **randrange** et le supprimer avec **pop**. Dans les algorithmes suivants, l'urne est modifiée à chaque fois. Comme précédemment, on peut empêcher ce résultat en créant une urne locale.

```
def tirage(urne):
    indice = randrange(len(urne))
    return(urne.pop(indice))

# tirages successifs sans remise.
# on enregistre les résultats dans une liste.
def tirageSuccessif(urne, nb):
    res = []
    for i in range(nb):
        res.append(tirage(urne))
    return res

# tirages successifs sans remise.
# on compte le nombre de fois que l'on a obtenu la valeur k.
def tirageNbSucces(urne, k, nb):
    succes = 0
    for i in range(nb):
        succes += (tirage(urne) == k)
    return succes
```

2 APPROXIMATION DES PROBABILITÉS ET DE L'ESPÉRANCE

A Approximation numérique de la probabilité

On admet que lorsque l'on réalise une expérience de nombreuses fois et de façon indépendante, la proportion de chaque événement tend vers sa probabilité. Dans le cas d'une variable aléatoire réelle, la moyenne des résultats tend vers l'espérance.

Ainsi, il suffit de calculer la proportion ou la moyenne obtenues lors de la répétition de l'expérience pour avoir une valeur approchée de la probabilité et de l'espérance. Ceci est très simple à mettre en œuvre.

Par exemple, on dispose d'une urne qui contient p boules blanches et q boules noires. On note X le nombre de boules blanches obtenus lors de `nb` tirages avec remise. On cherche une approximation numérique de $p(X = k)$.

```
urne = p*[1]+q*[0]

# nombre de boules blanches pour nb tirages avec remise dans l'urne
def tirage(urne, nb):
    res = 0
    for i in range(nb):
        res += choice(urne)           # 1 si blanc, 0 si noir
    return res

```

```
#p(X=k)
def proba(urne, nb, k, n = 1000):
    res = 0
    for i in range(n):
        res += (tirage(urne,nb) == k) # on ajoute 1 si le tirage donne k
    return res/n                      # proportion
```

Mais 1000 répétitions de l'expérience sont-elles suffisantes. Faut-il n'en faire que 50, ou au contraire 10 000 ?
Pour établir une conjecture, on peut appliquer cette fonction pour différentes valeurs de n, et lorsqu'il semble que cela converge, on s'arrête. Mais il y a beaucoup mieux en utilisant les outils graphiques.

B Visualisation graphique de la convergence

On peut calculer successivement les proportions en fonction du nombre de fois que l'on réalise l'expérience et les afficher sur un graphique. Ainsi, on verra graphiquement lorsque la courbe semble proche de sa limite.

```
#p(X=k)
def proba(urne, nb, k, n = 1000):
    res = 0
    for i in range(n):
        res += (tirage(urne,nb) == k) # on ajoute 1 si le tirage donne k
        plot(i+1,res/(i+1),'r,')
    return res/n                      # proportion
```

Remarques importantes :

- Lorsque l'on trace les valeurs successives, il est souvent préférable de tracer point par point, plutôt que d'enregistrer toute la liste. Cela évite d'utiliser trop de place en mémoire inutilement.
- Pour chaque point, il faut diviser par $i+1$. En effet, au premier "tour", on est à $i = 0$: il y a donc un décalage de 1, entre le nombre d'expériences que l'on a réalisées et la valeur du compteur i. C'est cohérent avec la valeur retournée en fin d'algorithme, car à la fin de la boucle, $i = n-1$.
- Ne pas oublier de renvoyer une valeur numérique avec `return` à la fin. La visualisation graphique n'est pas toujours suffisante (et n'est pas réutilisable par d'autres programmes).

C Estimation de l'espérance

On fait de même pour l'espérance, avec le calcul de la moyenne :

```
def esperance(urne, nb, n = 1000):
    res = 0
    for i in range(n):
        res += tirage(urne,nb)
        plot(i+1,res/(i+1),'r,')
    return res/n                      # moyenne
```

D Cas des tirages sans remise

Comme nous l'avons spécifié plus haut, lorsque le tirage est sans remise, il faut veiller à "remettre l'urne à zéro" entre chaque expérience. Par exemple, on peut utiliser des urnes "locales" pour cela (par contre, il faut faire un nouveau **shuffle** pour chaque expérience).

```
    urne = p*[1]+ q*[0]

# nombre de boules blanches pour nb tirages avec remise dans l'urne
# nb < p + q
def tirage(urne, nb):
    urneLocale = urne[:]
    shuffle(urneLocale)
    res = 0
```

```
    for i in range(nb):
        res += urneLocale.pop()            # 1 si blanc, 0 si noir
    return res

#p(X=k)
def proba(urne, nb, k, n = 1000):
    res = 0
    for i in range(n):
        res += (tirage(urne,nb) == k) # on ajoute 1 si le tirage donne k
        plot(i+1,res/(i+1),'r,')
    return res/n                       # proportion

def esperance(urne, nb, n = 1000):
    res = 0
    for i in range(n):
        res += tirage(urne,nb)
        plot(i+1,res/(i+1),'r,')
    return res/n                       # moyenne
```

3 Modéliser une loi de Bernoulli sans urne

Exemple type : on lance n fois de suite une pièce qui a la probabilité p de tomber sur pile et $1-p$ de tomber sur face.

On veut compter le nombre de fois que la pièce est tombée sur pile.

Programmation :

On tire un nombre aléatoire $x \in [0,1[$: si $x < p$, on renvoie 1 (pile), sinon, on renvoie 0 (face) :

```
def bernoulli(p):
    if random() < p:
        return 1
    return 0

def binomiale(p,n):
    res = 0
    for i in range(n):
        res += (random()<p)
    return res
```

4 Simuler une loi discrète non uniforme

On utilise pour cela la même idée que pour la loi de Bernoulli sans urne. Prenons un exemple :

On dispose d'un dé pipé dont la probabilité d'obtenir 6 est $\frac{1}{2}$, et la probabilité d'obtenir chacune des autres valeurs est $\frac{1}{10}$.

On modélise alors l'expérience aléatoire ainsi :

```
valeurs = [1,2,3,4,5,6]                    # valeur de la variables
p = 5*[1/10]+[1/2]        # probabilités associées

def tirage(valeurs, p):
    essai = random()
    f = 0                                  # fonction de répartition
    for i in range(len(p)):
        f += p[i]
        if essai < f:                      # p[i-1] < essai < p[i]
            return valeurs[i]
    return valeurs[-1]                     # inutile en théorie (en cas d'erreur d'
        approximation)
```

Ce programme se comprend mieux avec la fonction de répatition :

k	1	2	3	4	5	6
$p(X=k)$	0,1	0,1	0,1	0,1	0,1	0,5
$p(X\leq k)$	0,1	0,2	0,3	0,4	0,5	1

Par exemple, si `essai=0.25`, alors cela revient à obtenir un 3.

Pour tester dans quel intervalle de la fonction de répartition se trouve la variable `essai`, on la compare avec les valeurs successives de la fonction de répartition que l'on calculer tour à tour.

5 ÉVALUER DES PROBABILITÉS CONDITIONNELLES

Prenons à nouveau un exemple pour montrer comment on peut évaluer des probabilité conditionnelles avec Python.

On rappelle qu'évaluer des probabilités conditionnelles, revient simplement à changer de loi de probabilité. Il sera donc logique que la fonction ressemble à l'évaluation d'une probabilité *simple*.

On dispose de deux dés : un dé normal et un dé truqué (comme dans la section précédente). On tire un dé au hasard et on obtient un six. Quelle est la probabilité que le dé soit truqué.

On cherche donc la probabilité d'avoir un dé truqué, sachant que l'on a obtenu un six.

La modélisation est très simple. Comme nous l'avons vu en mathématiques, cela revient à restreindre notre univers. Ainsi, sur tous les essais, on ne retiendra que ceux qui donnent un six. Et parmi ceux-ci, on compte ceux pour lesquels le dé était truqué.

```
1
2  def tirage(listep):
3      essai = random()
4      for i in range(len(listep)):
5          if essai < listep[i]:
6              return i+1
7          essai -= listep[i]
8      return len(listp)
9
10 def probaCond(nb = 1000):
11     listeT = 5*[1/10]+[1/2]
12     listeN = 6*[1/6]
13     n = 0                                   # nombre de tirages avec un six
14     res = 0                                 # nb dés truqués sachant six
15     while n<nb:
16         de = random()
17         if de < .5:
18             lancer = tirage(listeT)
19         else:
20             lancer = tirage(listeN)
21         if lancer == 6:                     # sachant six
22             n += 1
23             if de < .5:                     # truqué sachant 6
24                 res += 1
25             plot(n, res/n, 'r,')
26     return res/n
```