Université de Bordeaux
L2 Informatique/Maths-Informatique

Probabilités, Statistiques, Combinatoire : TM3

Simulations

L'objectif de cette feuille est de *simuler* des expériences aléatoires d'une part, et d'*observer* le résultat de ces expériences, en les répétant un grand nombre de fois, d'autre part.

Pour la génération aléatoire, on utilise le module `random` chargé avec la commande `import random` placée au début de votre fichier.

On utilisera deux fonctions du module `random` :

- `random.randint(a,b)`, un générateur aléatoire qui renvoie un nombre entier choisi au hasard (uniformément) dans $[[a, b]]$.
- `random.random()`, qui retourne un flottant compris entre 0 et 1, selon la "loi uniforme sur $[0, 1]$" : la probabilité de retourner une valeur située dans n'importe quel intervalle $[x, y]$ (avec $0 \leq x \leq y \leq 1$) est $y - x$, la longueur de l'intervalle.

On appellera *simulateur* d'une expérience, une **fonction** (sans arguments) qui simule un tirage selon l'expérience demandée, et retourne le résultat. Des appels successifs à la fonction retourneront des résultats indépendants. L'un des objectifs de cette feuille est d'écrire des simulateurs pour diverses expériences ; autant que possible, on cherchera à réutiliser ces simulateurs pour en construire d'autres.

Afin de **tester** empiriquement les simulateurs, on procèdera de la manière suivante : on appellera le simulateur un grand nombre n de fois ($n = 100$, 1000, un million : c'est la machine qui fait le travail), et on résumera les résultats en retournant une structure de données indiquant, pour chaque "valeur", le *nombre de fois* qu'elle a été obtenue. En divisant ces nombres par n, on obtiendra des *fréquences empiriques*, dont on espère qu'elles seront proches des probabilités qui définissent le modèle.

Dans cette séance, vous serez amené à écrire ces deux types de fonctions. Les fonctions dont le nom commence par `simule` sont supposées être des simulateurs ; les fonctions dont le nom commence par `teste`, sont des testeurs.

3.1 Les dictionnaires de `Python`

Pour manipuler les fréquences empiriques d'apparition d'événements, nous allons utiliser une structure de données de *dictionnaire*, fournie par le langage `python`.

Le dictionnaire est un conteneur – il n'y a pas d'ordre sur les données. Celles-ci sont organisées en couples de forme `clé : valeur`, par exemple `{ (1,4): 3, 6: 'badPassword'}` est un dictionnaire avec deux clés, le tuple `(1,4)` (auquel est associée la valeur 3) et l'entier `6` (auquel est associée comme valeur, une chaîne de caractères). Les clés peuvent être des entiers (comme dans un tableau), mais aussi des tuples, ou des chaînes de caractères (on ne peut pas utiliser de liste, ni aucun objet modifiable, comme clé d'un dictionnaire).

Pour initialiser un dictionnaire vide, on peut utiliser `monDict = {}`.

Pour ajouter un couple clé-valeur à un dictionnaire, on utilise la clé comme indice :

```
monDict[(1,2)] = 4
```

Si la clé n'existe pas encore dans le dictionnaire, elle est ajoutée au dictionnaire avec la valeur spécifiée après le signe `=` ; si la clé est déjà présente, l'ancienne valeur est remplacée par la nouvelle.

Pour accéder à une valeur précise, on utilise la même syntaxe qu'avec les listes :
`a = monDict[c]`
affectera la valeur associée à la clé `c` à la variable `a` – ou provoquera une erreur, si la clé n'existe pas dans le dictionnaire.

Pour tester si une clé appartient au dictionnaire, et aussi pour parcourir toutes les clés du dictionnaire, on utilise le mot-clé `in` comme pour les listes :
`(1,3) in monDict` vaut `False`, `(1,2) in monDict` vaut `True`.

```
for c in monDict:
    print(monDict[c])
```

affiche chaque clé présente dans le dictionnaire.

Simulation de lois de probabilités

1. Écrire une fonction `simuleDe()`, qui retourne le résultat du lancer virtuel d'un dé à 6 faces équilibré.

   Attention : ne pas oublier qu'en `Python`, un appel de fonction se fait avec des parenthèses, même s'il n'y a pas de paramètres : `simuleDe` est la fonction (le simulateur), pour obtenir un tirage il faut utiliser `simuleDe()`.

2. Écrire une fonction `testeDe(n)` qui retourne un dictionnaire contenant les *fréquences empiriques* d'apparition des valeurs de 1 à 6 après n lancers d'un dé, calculées avec l'algorithme suivant :
   - Initialiser un dictionnaire vide.
   - Effectuer n lancers ; pour chaque lancer, si le résultat est déjà une clé présente dans le dictionnaire, incrémenter de 1 la valeur associée ; sinon, ajouter la clé et lui associer la valeur 1.
   - Avant de retourner, normaliser, c'est-à-dire diviser toutes les valeurs (nombres d'occurrences des différentes clés) par n pour obtenir des valeurs dont la somme soit 1.

   La valeur retournée de `testeDe(5)` pourrait donc ressembler à { `1 : 0.2, 2 : 0.4, 3 : 0.2, 6 : 0.2`} (ici les clés sont les résultats possibles de l'expérience, et les valeurs, les fréquences observées, sous formes de flottants)

3. Tester votre fonction `testeDe(n)` avec des valeurs de n comme $10, 100, 1000, 1000000$, chacune à plusieurs reprises. Commenter les résultats obtenus, notamment en ayant en tête la vision "fréquentiste" des probabilités.

4. Écrire une fonction `simulePremierSix()` qui simule le lancer répété d'un dé et retourne le nombre total de lancers nécessaires pour obtenir le résultat 6 pour la première fois.

5. Écrire une fonction `testePremierSix(n)` comparable à la fonction `testeDe()`, mais pour l'expérience consistant à attendre le premier six ; comme pour la fonction `testeDe()`, expérimenter avec des valeurs de `n` de 10, 100, 1000 et 1000000 et commenter les résultats obtenus.

6. Faire la même chose avec l'expérience correspondant à lancer deux dés, et à prendre la somme des deux dés : écrire une fonction `simuleDeuxDes()` qui simule cette opération, et une fonction `testeDeuxDes(n)` qui retourne un dictionnaire contenant les fréquences observées des différents résultats possibles pour n tirages. Quelles sont les fréquences que vous vous attendez à observer ?

7. On va maintenant utiliser une autre primitive, `random.random()`, qui retourne un résultat flottant entre 0 et 1 ; de manière approchée, pour $0 \leq a < b \leq 1$, la probabilité que `random.random()` retourne un résultat compris entre a et b est $b-a$ (oublions que l'intervalle $[0, 1]$ est un ensemble non dénombrable).

   On se propose de tirer un nombre entier aléatoire selon la méthode suivante : pour un nombre flottant p, compris entre 0 et 1, on tire de manière répétée des nombres avec `random.random()`, et on calcule leur *produit* ; et on retourne le nombre de tirages nécessaires pour atteindre un produit inférieur à p.

   Écrire un simulateur `simuleMystere(p)` pour cette expérience, et une fonction de test `testeMystere(n,p)` qui réalise n fois cette expérience et retourne un dictionnaire des fréquences observées. Expérimenter avec votre fonction `testeMystere()`, et proposer, pour $p = e^{-1} \simeq 0.3679$, des valeurs approchées pour les probabilités que `simuleMystere(p)` retourne $1, 2, 3, \ldots$. **Indication :** pour identifier expérimentalement ces probabilités, on pourra essayer de multiplier les fréquences observées par $e \simeq 2.718$, et de "reconnaître" les fréquences.

3.2 Pour les curieux : le Monty Hall

On va s'efforcer de simuler le jeu télévisé du Monty Hall. Rappelons-en le principe : le candidat se trouve face à trois portes, dont l'une, choisie aléatoirement, cache un cadeau précieux - les deux autres cachent un "cadeau" sans valeur. Le candidat peut désigner une porte de son choix, ce sur quoi l'animateur du jeu (qui sait quelle porte est la bonne) ouvre une porte et révèle ce qu'elle cache ; cette porte est toujours une porte non désignée par le candidat, et ce n'est jamais la porte cachant le cadeau précieux (si l'animateur a le choix, on suppose qu'il choisit aléatoirement la porte à ouvrir). Le candidat doit alors faire un choix de quelle porte ouvrir – et de gagner le cadeau qu'elle cache.

La question, qui défraya la chronique, est de savoir quelle stratégie le joueur devrait adopter. Citons trois possibilités :

— il peut ouvrir la porte initialement désignée ;
— il peut choisir systématiquement d'ouvrir la porte qu'il n'a pas désignée (parmi celles qui n'ont pas été ouvertes)
— il peut choisir aléatoirement entre la porte initialement désignée, et l'autre porte non encore ouverte.

On ne va pas ici chercher à modéliser l'expérience et à analyser la probabilité de gagner le cadeau précieux ; on va plutôt programmer les différentes stratégies, et les simuler.

On va donc écrire plusieurs fonctions :

— `initJeu()`, qui "choisit" aléatoirement le numéro de la "bonne" porte (un entier de 1 à 3) ;
— `choixJoueur()`, qui représente le choix initial du joueur : elle doit retourner un entier entre 1 et 3 ;

— `animateur(bon,choix)` : représente la porte qu'ouvre l'animateur quand `bon` est le numéro de la "bonne" porte, et `choix` est le numéro de la porte désignée par le joueur ;
— `strategieJoueur(choixInitial,choixAnimateur)`, qui simule le choix du joueur quand il a initialement désigné la porte `choixInitial` et que l'animateur a ouver la porte `choixAnimateur` ;
— `simuleJeu(iJ,cJ,a,sJ)`, qui prend en entrée les quatre fonctions précédentes (sous forme de fonctions), les utilise pour simuler le déroulement du jeu, et retourne le résultat : 1 si le joueur repart avec le cadeau de valeur, 0 sinon.

Votre travail consiste à écrire chacune des fonctions (et au moins trois versions de la fonction de stratégie du joueur), et à simuler le jeu suffisamment de fois pour trancher la polémique ! Notez qu'il est également possible de faire varier la stratégie initiale du joueur (peut-être qu'il préfère la porte 1 ?), mais aussi celle de l'animateur (à ceci près qu'il n'a jamais le droit d'ouvrir la "bonne" porte, ni celle désignée par le joueur : il n'a donc de choix que si le joueur a désigné la "bonne" porte ; mais par exemple on peut imaginer, et simuler, un animateur qui, quand il a le choix, ouvrirait toujours la porte au plus petit numéro...).