Feuille de TP Python 03

Simulations probabilistes/modèles finis

L'objet de ce TP est essentiellement de mettre en place une méthodologie pour traiter effectuer des simulations probabilistes.

On se donne un espace probabilisé[1] $(\Omega, \mathscr{T}, \mathbb{P})$. Les fonctions Python de tirage au sort que nous allons rencontrer (par exemple `numpy.random.rand`) sont des modèles de variables aléatoires au sens où chaque appel à l'une de ces fonctions équivaut à tirer au sort, indépendamment des tirages précédents, un nombre flottant, suivant une loi donnée. En ce qui concerne `numpy.random.rand`, la fonction de tirage au sort fondamentale, il s'agit de la loi uniforme sur l'intervalle $]0,1[$, que nous verrons en cours d'ici début octobre.

Pour information à retenir, une v.a. U est distribuée uniformément sur $]0,1[$ si pour tout intervalle I contenu dans $]0,1[$,

$$\mathbb{P}(U \in I) = \text{longueur}(I)$$

Imaginons que dans une fonction Python `f` donnée, nous ayons besoin de 10000 valeurs aléatoires tirées indépendamment suivant une loi uniforme sur $]0,1[$ pour calculer le résultat de la fonction `f`. Mathématiquement, on se donne 10000 variables aléatoires *indépendantes* $U_1, \dots U_{10000}$, suivant toutes la loi uniforme sur $]0,1[$ (*i.e.* un 10000-échantillon de la v.a. modèle U, distribuée uniformément sur $]0,1[$) et le résultat calculé de la fonction peut-être vu comme fonction des 10000 valeurs $\texttt{f}(U_1, \dots, U_{10000})$. Le résultat est en ce sens une variable aléatoire définie sur Ω.

Chaque éxécution de la fonction `f` retourne une valeur tirée au sort suivant les règles imposées (par la définition de la fonction `f`). Une suite (dans une même session de l'interpréteur) d'exécutions de `f` retourne une suite de valeurs d'un échantillon de `f`. Différentes exécutions au sein d'une même session Python sont réputées indépendantes au sens probabiliste.

1 Tirer un nombre au hasard-Tirer un objet au hasard

1.1 Les modules `random` et `numpy.random`

Pour tirer un nombre au hasard (quel hasard ?), Python dispose de dizaines de fonctions. Ces fonctions diffèrent essentiellement par la *loi* du nombre aléatoire retourné. Elles relèvent essentiellement de deux modules `random` et `numpy.random`, on trouve des listes des fonctions définies par ces modules aux adresses

`https://docs.python.org/3.6/library/random.html`

et

`http://docs.scipy.org/doc/numpy/reference/routines.random.html`

Afin de ne pas démultiplier le nombre de bibliothèques utilisées, nous nous cantonnerons à l'utilisation de `numpy.random`, et, plus restrictif encore, à l'utilisation des seules fonctions `numpy.random.randint()`/`numpy.random.choice()` (pour leur côté pratique) et `numpy.random.rand()` (car c'est en fait la seule nécessaire, toutes les autres en découlent).

Voici un exemple typique de ce que nous allons faire et ne pas faire : Le but du script est de composer une liste de `NS = 10000` nombres tirés au sort suivant une loi binomiale $\mathscr{B}(\texttt{nbre\_tirages}, \texttt{proba\_succes})$ de paramètres imposés puis l'afficher l'histogramme.

— (Ce que l'on ne fera pas sauf contre-ordre express). On voit dans la documentation que la fonction `binomial(n, p)` retourne un nombre aléatoire tiré au sort suivant une loi binomiale de paramètres imposés. Le script demandé peut s'écrire en utilisant cette fonction :

```
#Binomiale builtin
import numpy as np # on importe la bibliothèque numpy avec son surnom standard
import matplotlib.pyplot as plt
NS = 10000
proba_succes = 0.3
nbre_tirages = 11
s = []
for k in range(NS) :
    s.append(np.random.binomial(nbre_tirages,proba_succes))
#Affichage histogramme des valeurs de s
plt.hist(s, bins = 10*nbre_tirages)
```

1. On verra la définition dans le cours de mathématiques

— (Ce qu'on fera habituellement car dans l'esprit de ce qui est attendu). Le seul hasard autorisé provient de la fonction `np.random.rand()`. On va donc construire une fonction `Bernoulli(p)` qui retourne un nombre valant soit 0 soit 1 calculé à partir de la valeur retournée par une utilisation de la fonction `np.random.rand()`, ceci avec probabilité `p`. On peut en déduire l'écriture d'une fonction (la nôtre) `Binomiale`(n,p) retournant la somme des résultats de `n` appels indépendants à la fonction `Bernoulli(p)`. Le script demandé peut s'écrire en utilisant ces fonctions :

```
#Binomiale manuelle
import numpy as np # on importe la bibliothèque numpy avec son surnom standard
import matplotlib.pyplot as plt
def Bernoulli(p):
    u = np.random.rand()
    .....
    return .....
def Binomiale(n,p):
    S = 0
    for k in range(n):
        S += ......
    return S
#Fin des fonctions de simulation maison

NS = 10000
proba_succes = 0.3
nbre_tirages = 11
s = []
for k in range(NS) :
    s.append(Binomiale(nbre_tirages,proba_succes))
#Affichage histogramme des valeurs de s
plt.hist(s, bins = 10*nbre_tirages)
```

Le programme du concours impose que vous sachiez simuler des v.a. suivant des lois classiques discrètes et continues (« à densité ») à partir d'une variable uniforme sur $[0,1[$. De toutes ces belles fonctions `numpy` qui simulent des lois classiques ou beaucoup plus exotiques, vous n'avez donc le droit, dans un premier temps, de n'en utiliser qu'une : celle qui calcule un nombre aléatoire avec distribution $\mathscr{U}_{[0,1[}$, *i.e.* `numpy.random.rand()`

Pour nous simplifier la vie, même si nous sommes en mesure d'écrire notre version de cette fonction, on utilisera aussi,

— pour simuler des problèmes d'urnes, `numpy.random.randint(n)`, qui retourne un entier tiré uniformément dans les entiers entre 0 et $n-1$,

— pour simuler des chaînes de MARKOV, `numpy.random.choice(L, p = [...])`, qui retourne un élément d'une liste `L` suivant une loi de probabilité `p` passée en paramètre.

1.2 Travail à effectuer

1. Compléter le script du texte introductif dans un script nommé `intro.py`, à savoir :
   (a) Ecrire une fonction `Bernoulli(p)` où `p` est un `float`, retournant 0 avec probabilité $1-p$ et 1 avec probabilité p. Le hasard utilisé dans cette fonction provient uniquement de `numpy.random.rand()`.
   (b) Ecrire une fonction `Binomiale(n,p)` où `n` est un `int`, `p` est un `float`, retournant un nombre entier entre 0 et n tiré suivant la loi binomiale $\mathscr{B}(n,p)$. Le hasard utilisé dans cette fonction provient uniquement de la fonction `Bernoulli`.
   (c) Utilisant les méthodes [2] `numpy`, `mean()` et `std()`, faire imprimer moyenne et variance de la liste `s` de nombres au hasard ainsi que les valeurs théoriques de l'espérance et de la variance de la v.a. simulée.
2. Dans un script nommé `uniforme.py`, Simuler la fonction `np.random.randint` : écrire une fonction `EntierUniforme(n)` où `n` est un `int` retournant un entier entre 0 et $n-1$ tiré au hasard uniforme. On utilisera le fait que si u est tiré uniformément sur l'intervalle $[0,1[$, $\lfloor n.u \rfloor$, la partie entière de $n.u$, est un entier tiré uniformément entre 0 et $n-1$. La fonction `np.floor` donne la partie entière d'un nombre réel et la fonction `int` transforme un `float` en `int`.
   Le hasard utilisé doit provenir de la fonction `np.random.rand`.
   Utiliser cette fonction pour tracer l'histogramme d'une suite de `NS = 1000` termes tirés uniformémént en 0 et 9.
3. Dans un script nommé `vadiscrete.py` :
   (a) Ecrire une fonction `VaDiscrete(p)` où `p` est une liste de `float` positifs dont la somme vaut 1 et retournant un nombre entier (un `int` donc) entre 0 et `len(p)-1` tiré suivant la loi décrite par `p`, *i.e.*

$$\mathbb{P}(\texttt{VaDiscrete(p)} = k) = \texttt{p[k]}$$

2. Si `tab` est un `ndarray` de nombres, `tab.mean()` et `tab.std()` retournent respectivement la moyennne et l'écart-type des nombres du tableau `tab`. Cette syntaxe `nom_variable.nom_fonction()` est une syntaxe typique de programmation orientée objet. Dans ce contexte, les fonctions associées à des types d'objets s'appelent des « méthodes ».

Le hasard utilisé doit provenir de la fonction `np.random.rand`.

Noter que la fonction déjà définie `np.random.choice` fait directement un travail similaire.

(b) Ecrire un bloc de code calculant une liste de `NS = 10000` nombres tirés suivant la loi `p=[0.1,0.2,0.4,0.3]` et afficher l'histogramme de cette liste.

(c) Donner la moyenne et la variance de cette liste. Faire imprimer ces nombres ainsi que les valeurs théoriques de l'espérance et de la variance de la somme considérée.

(d) Ecrire, en utilisant la fonction `VaDiscrete(p)`, une fonction `EntierUniforme2(n)` qui retourne un entier tiré uniformément entre 0 et $n-1$.

4. Dans un script nommé `urnes.py` :

(a) On rappelle qu'une p-liste (avec $p \in \mathbb{N}^*$) d'éléments d'un ensemble E est un élément de E^p. Ecrire une fonction `UrneAvecRemise(n,p)` où `n` et `p` sont des entiers retournant une p-liste d'entiers compris entre 0 et $n-1$ tirée de façon uniforme parmi les p-listes.

(b) Dans la zone de script principal, faire le tirage (avec remise) et l'impression de six 5-listes d'entiers entre 0 et 6.

5. (a) On rappelle qu'une p-liste sans répétition (avec $p \in \mathbb{N}^*$) d'éléments d'un ensemble E est un élément de E^p dont aucune paire d'entrées ne sont égales. Ecrire une fonction `UrneSansRemise(n,p)` où `n` et `p` sont des entiers retournant une p-liste sans répétition d'entiers compris entre 0 et $n-1$ tirée de façon uniforme parmi les p-listes sans répétition.

On pourra utiliser la technique de tirage dans une liste avec élimination du terme juste tiré.

```
#Tirage dans urne en retirant
#L est une liste
indice = np.random.randint(len(L))
element = L.pop(indice) #retourne élément tiré au hasard, l'élimine de la liste L
```

(b) Dans la zone de script principal, faire le tirage et l'impression de 6 5-listes sans répétition d'entiers entre 0 et 6.

(c) Par la même technique, écrire une fonction `Battage(L)`, qui, étant donnée une liste quelconque `L` retourne une liste permutée (uniformément) ayant les mêmes éléments que `L` et tester cette fonction en battant 10 fois une même liste non numérique [3]

2 Le collectionneur de vignettes PANINI

2.1 Problématique

On cherche à faire des expérimentations autour des problèmes qui se posent à un collectionneur de vignettes PANINI.
Une collection de vignettes PANINI se compose de la façon suivante

1. On achète un album (le *collecteur*) vierge qui contient N emplacements [4] dans lesquels on peut coller une vignette autocollante destinée à cet emplacement.
2. On peut ensuite acheter des pochettes de n vignettes distinctes, prises au hasard parmi les N possibles, vignettes qu'on colle aux emplacements prévus.

Le collecteur est rempli (et la collection est finie) lorsqu'il n'y a plus d'emplacements vides.
Les questions qui se posent sont p.ex.

1. le nombre moyen de pochettes à acheter pour arriver à compléter le collecteur ?
2. une estimation du nombre maximal de pochettes à acheter pour être sûr à 75% d'avoir complété son collecteur ?
3. le nombre de pochettes à acheter pour que q collectionneurs procédant à des échanges aient tous un collecteur complet ?
4. ....

Certains de ces problèmes peuvent se régler théoriquement par la théorie des probabilités, ce n'est pas l'objet ici. Notre but est de pouvoir procéder à des simulations sur ordinateur.

2.2 Travail à faire

Ecrire des fonctions Python répondant aux questions suiavntes.

1. Comment représenter le collecteur en Python ? Comment vérifier qu'il est rempli ?
2. Comment tirer une pochette au hasard et l'ajouter dans le collecteur ?
3. Comment simuler une session d'achats de pochettes jusqu'à remplissage du collecteur ?

3. p.ex. Si `s='COUCOU'`, `list(s)` est la liste des lettres de la chaine (s), *i.e.* `['C','O','U','C','O','U']`.
4. $N = 681$ pour l'album de la coupe du monde de foot 2018, $N = 216$ pour l'album My Little Pony

4. Comment évaluer le nombre moyen de pochettes à acheter pour remplir son collecteur ?
5. Conclure en traçant, sur un même graphique, la courbe du nombre moyen de pochettes à acheter en fonction de N obtenue par simulation (faire un grand nombre de simulations) et la courbe de $N \mapsto N\ln N$ qui est une approximation théorique de cette valeur. Se limiter à $N \leq 50$ et $n = 1$.

On répondra à ces questions en écrivant un script `collectionneur.py`.

On pourra aussi, si cela fait sens, regrouper les fonction développées dans le TP en un module nommé `va.py`, module importé dans `collectionneur.py` afin de profiter de ces fonctionnalités personnnalisées.

3 Annexes

3.1 Listes

`L=[]` ou `L=list()`	fabrique une liste vide à remplir
`L=[1.0,'aa']`	fabrique une liste avec un préremplissage
`len(L)`	donne le nombre d'éléments de la liste, ils sont indicés de 0 à `len(L)-1`
`L[i]`	donne la valeur de l'élément de `L` d'indice i si $0 \leq i \leq$`len(L)-1`
`M=L[i:j]`	extrait une sous-liste M dont les éléments ceux de `L` sont indicés de i à $j-1$
`L[i:j]=M`	remplace la sous-liste des éléments de `L` indicés de i à $j-1$ par la liste `M`
`M=L`	Les deux étiquettes `M` et `L` sont sur la même liste en mémoire
`M=L[:]` ou `M = L.copy()`	créé une copie de `L` et lui affecte l'étiquette `M`
`L[-1]`	donne le dernier élément de la liste
`L.append(obj)`	ajoute l'objet `obj` en fin de liste
`L.extend(M)`	fusionne les listes `L` et `M` en plaçant `M` à la fin de `L`
`L.insert(index,obj)`	insère l'objet `obj` avant l'indice i, décale le reste de `L`
`L[i:i] = M`	insère la liste `M` à l'indice i de la liste `L`, décale le reste de `L`
`L.pop()`	efface le dernier élément de la liste `L` et le retourne
`L.pop(i)`	efface l'élément i de la liste `L` et le retourne
`L.reverse()`	inverse *en place* l'ordre des éléments de la liste
`L.sort()`	trie *en place* les éléments de la liste, pourvu que l'on puisse comparer les éléments
`M=sorted(L)`	`M` devient une copie triée de la liste `L`
`V=np.array(L)` ou `V=np.asarray(L)`	`V` devient un vecteur `numpy` composé avec les éléments de la liste de nombres `L`. Si `L` est une liste de listes de nombres, compose une matrice `numpy`.

`for obj in L :`
 	
Exemple :
`for k in L:`
`  print(k**2)`
	boucle sur les éléments de `L`, ne modifie pas `L`. Modifier `L` dans la boucle peut avoir des conséquences inattendues.
	Si on prend `L=[1,2,4]` affiche successivement `1`, `4`, `16`.

`M = [expression(obj) for obj in L]`
 	
Exemple :
`M=[k+1 for k in L]`
	fabrique une liste en bouclant sur les éléments de `L` et en leur appliquant `expression()`, ne modifie pas `L`.
	Si on prend `L=[1,2,4]` fabrique la nouvelle liste `[2,3,5]` et l'affecte à `M`.

`M = [expr(obj) for obj in L if cond(obj)]`
 	
Exemple :
`M=[k**2 for k in L if k>1]`
	fabrique une liste en bouclant sur les éléments de `L`, ne gardant que ceux vérifiant `cond()` et en leur appliquant `expr()`, ne modifie pas `L`.
	Si on prend `L=[1,2,4]` fabrique la nouvelle liste `[4,16]` et l'affecte à `M`.