Concours Blanc : corrigé du TP d'Informatique

PTSI A et B Lycée Eiffel

20 mai 2016

Simulations de lancers de dés.

Précision préliminaire : beaucoup de programmes demandés dans ce TP faisaient intervenir des simulations aléatoires, les résultats indiqués dans ce corrigé peuvent évidemment différer de ce qu'on obtiendrait lord d'une autre simulation de la même expérience, mais les commentaires restent globalement valables.

1. Il suffit de faire la commande suivante :

```
def de() :
    return randint(1,6)
```

2. Faisons les choses en créant la liste à la main, puis en augmentant d'un à chaque tirage le terme de la liste dont l'indice correspondant au tirage obtenu (en décalant les valeurs du tirage aléatoire pour les faire coller à la numérotation des listes) :

```
def lancerde(n) :
    liste=[0,0,0,0,0,0]
    for i in range(n) :
        liste[randint(0,5)]+=1
    return liste
```

3. On peut réutiliser le programme précédent pour sortir la moyenne :

```
def moyennede(n) :
    liste=lancerde(n)
    return sum((i+1)*liste[i] for i in range(6))/float(n)
```

On peut également reprendre la première fonction écrite :

```
def moyennede(n) :
    s=0
    for i in range(n) :
        s+=de()
    return s/float(n)
```

J'obtiens pour mes simulations une moyenne de 2.6 pour $n = 10$, puis 3.43 pour $n = 100$ et 3.532 pour $n = 1000$. La moyenne théorique d'un lancer de dé est de 3.5, plus on répète de simulation, plus on va se rapprocher de cette valeur. Par exempl,e pour $n = 1\ 000\ 000$, j'obtiens une moyenne de 3.49903.

4. On va bien sûr utiliser une boucle while pour cette question :

```
def premiersix() :
    n=1
    while de() != 6 :
        n+=1
    return n
```

5. J'écris directement un programme qui va renvoyer le nombre de 1 obtenues et le maximum (la liste intégrale ne présentant que peu d'intérêt) :

```
def geometrique(n) :
    liste=[]
    for i in range(n) :
        liste.append(premiersix())
    return liste.count(1),max(liste)
```

Ma simulation me donne 164 fois 1 sur 1000 simulations, ce qui semble normal, puisqu'on devrait avoir un 1 avec probabilité $\frac{1}{6}$ (on a une chance sur six d'obtenir un six dès le premier lancer), et $\frac{164}{1000}$ est très proche de $\frac{1}{6}$. La valeur maximale que j'ai obtenue est 34. En refaisant une simulation sur 100 000 essais, j'obtiens 16 974 fois 1 et un maximum de 62.

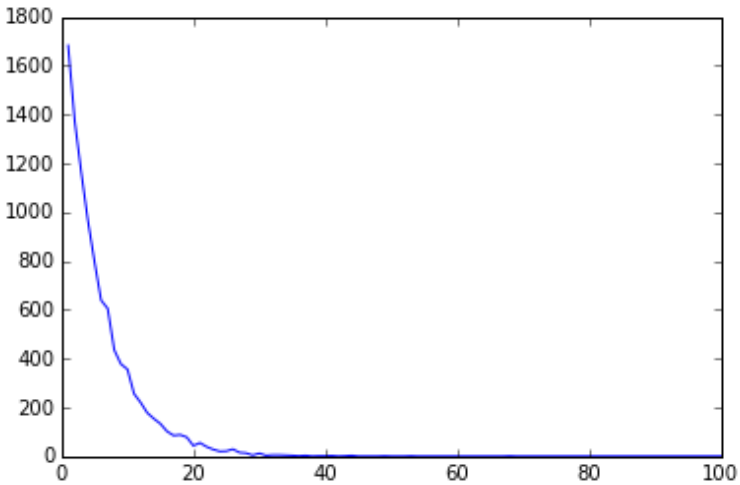
6. Je vais donner un nom différent à cette nouvelle fonction (dans mon programme, la première valeur dans la liste finale sera toujours égale à 0 puisqu'il faut au moins un essai pour obtenir un six, je n'ai pas décalé les valeurs dans la liste) :

```
def geometriquebis(n) :
    liste=[0 for i in range(101)]
    for i in range(n) :
        liste[premiersix()]+=1
    return liste
```

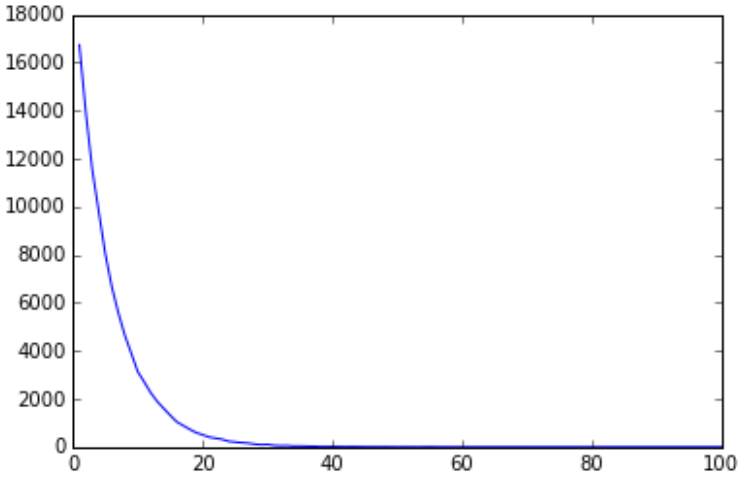
7. On peut écrire un programme ressemblant à ceci :

```
import matplotlib.pyplot as plt
abscisse=range(1,101)
ordonnee=geometriquebis(10000)[1 :]
plt.plot(abscisse,ordonnee)
```

Voici une allure de courbe :



On doit normalement obtenir une courbe qui s'approche d'une exponentielle décroissante. Bien sûr, plus on fait de simulations, plus la courbe sera lisse. Un exemple avec 100 000 simulations :



Lancers de pièces.

1. Il suffit en fait d'utiliser la fonction random et de regarder si le résultat obtenu est plus grand que p. En effet, un random donnera une valeur inférieure à p avec probabilité p. Un programme minimaliste :

   ```
   def piece(p) :
       return int(random()<p)
   ```

2. Allons-y :

   ```
   def suitelancers(p,n) :
       if p<0 or p>1 :
           return 'Espèce de crétin, p est une probabilité !'
       pile=0
       for i in range(n) :
           pile+=piece(p)
       return pile
   ```

   Sans surprise, la proportion de Piles obtenus sur 100 lancers se rapproche de p. Sur un essai avec $p = 0.1$, j'ai obtenu 13 Piles ; avec $p = 0.2$, 17 Piles ; avec $p = 0.5$, 47 Piles ; avec $p = 0.8$, 84 Piles. En augmentant pour tester avec $n = 10\ 000$, j'obtiens 5 028 Piles pour $p = 0.5$ et 7990 Piles pour $p = 0.8$.

3. On peut taper le code suivant :

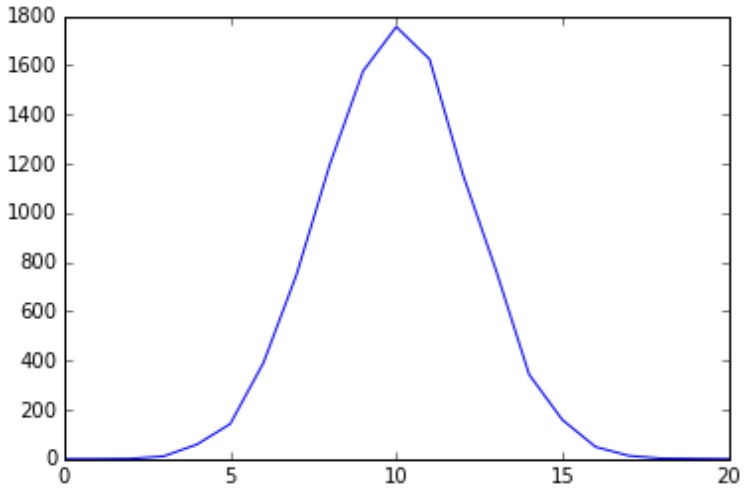
   ```
   liste=[0,0,0,0,0,0]
   for i in range(1000) :
   liste[suitelancers(0.5,5))]+=1
   print(liste)
   ```

   Ma simulation me donne la liste suivante : $[36, 144, 325, 286, 166, 43]$. On devrait théoriquement obtenir une liste symétrique (on a autant de chance d'obtenir cinq Piles que pas de Piles, un Pile que quatre Piles etc) et des valeurs nettement plus grandes au milieu de la liste. Plus précisément, si on s'y connait un peu en lois binômiales, on sait que les probabilités théoriques des six possibilités sont de $\frac{1}{32}$, $\frac{5}{32}$, $\frac{10}{32}$, $\frac{10}{32}$, $\frac{5}{32}$ et $\frac{1}{32}$, ce qui correspond plus ou moins à ce qu'on obtient (comme d'habitude, sur un plus grand nombre de simulations, ça marche mieux).

4. On fait quasiment la même chose que ci-dessus, avec un plot en plus :

```
liste=[0 for i in range(21)]
for i in range(1000) :
liste[suitelancers(0.5,20))]+=1
plt.plot(range(21),liste)
```

Un exemple de courbe :



Comme expliqué ci-dessus, la courbe doit être symétrique, et prendre des valeurs nettement plus grandes au centre de l'intervalle, avec une allure globale « en cloche » (même s'il ne s'agit pas ici d'une courbe de gaussienne).

Marches aléatoires.

1. La version du programme affichant uniquement le point d'arrivée :

```
def marche(n) :
    a=0
    for i in range(n) :
        if randint(1,2)==1 :
            a+=1
        else :
            a-=1
    return a
```

Une version où on affiche la liste des positions :

```
def marche2(n) :
    l=[0]
    for i in range(n) :
        if randint(1,2)==1 :
            l.append(l[-1]+1)
        else :
            l.append(l[-1]-1)
    return l
```

2. On peut évidemment reprogrammer les calculs correspondants, mais les plus flemmards (comme moi) ne se priveront pas pour utiliser les fonctions déjà existantes en Python :

```
def marche3(n) :
    l=marche2(n)
        return l.count(0),max(l)
```

Pour $n = 100$, ma simulation me donne 5 passages par 0 et un maximum de 11 (notons en passant qu'on affiche vraiment le maximum, qui ne correspond pas forcément à la plus grande distance atteinte par rapport au point de départ, puisque celle-ci peut être obtenue pour une valeur négative). Pour $n = 1000$, j'obtiens 40 passages par 0 et un maximum de 56. Pour $n = 10\ 000$, je trouve 3 passages par 0 et un maximum de 154. Après avoir réessayé, le nombre de passages par 0 est très aléatoire. Le maximum atteint est assez variable aussi, mais semble globalement relativement proportionnel à n (il faudrait vraiment des simulations plus pointues pour étudier ces variables précisément).

3. On peut écrire une petit fonction :

```
def moyennemarche(n) :
    l=marche2(n)
    return sum(l)/float(n)
```

Pour 1 000 étapes, j'obtiens une moyenne de -20.942. Pour 10 000 étapes, je trouve -95.2502, et pour 100 000 étapes, j'ai eu une moyenne de -89.13394. La moyenne devrait théoriquement éviluer relativement peu avec n (du moins nettement moins vite qu'une relation de proportionnalité). Il n'y a bien sûr aucune raison particulière au fait que j'ai obtenu plusieurs fois de suite des moyennes négatives.

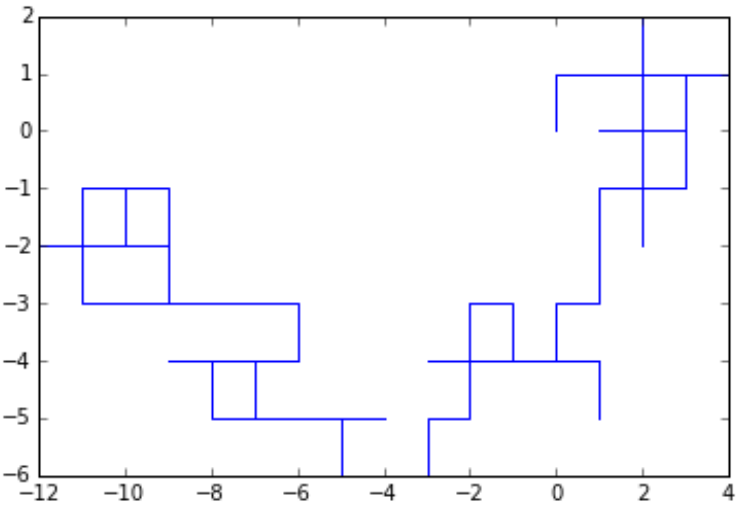
4. Je crée deux liste distinctes pour l'abscisse et l'ordonnée du point, ce sera plus pratique pour la suite. J'en profite d'ailleurs pour faire en même temps la question suivante, en traçant la trajectoire obtenue :

```
def marche2d(n) :
    absc=[0]
    ordo=[0]
    for i in range(n) :
        r=randint(1,4)
        if r==1 :
            absc.append(absc[-1]+1)
            ordo.append(ordo[-1])
        elif r==2 :
            absc.append(absc[-1])
            ordo.append(ordo[-1]+1)
        elif r==3 :
            absc.append(absc[-1]-1)
            ordo.append(ordo[-1])
        else :
            absc.append(absc[-1])
            ordo.append(ordo[-1]-1)
    plt.plot(absc,ordo)
    return
```

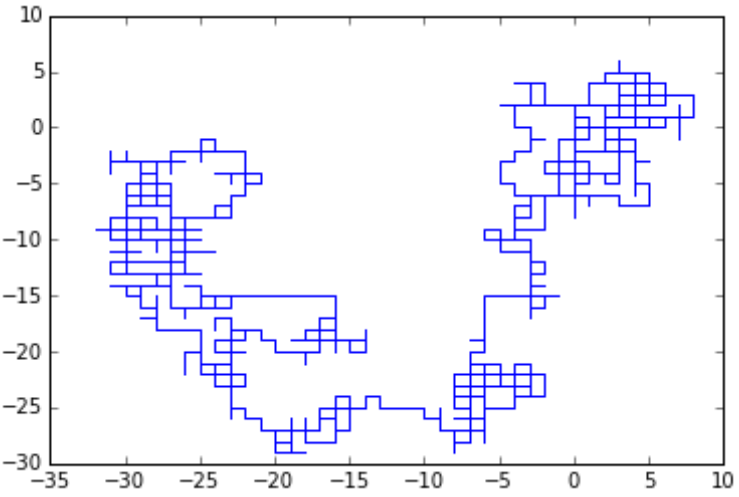
On cherche désormais à simuler une marche aléatoire en deux dimensions : l'objet se trouve initialement à la position $(0, 0)$ et peut avancer dans une des quatre directions (haut, bas,

gauche et droite) avec probabilité $\frac{1}{4}$ à chaque étape. Écrire un programme Python simulant une marche aléatoire de n étapes en deux dimensions.

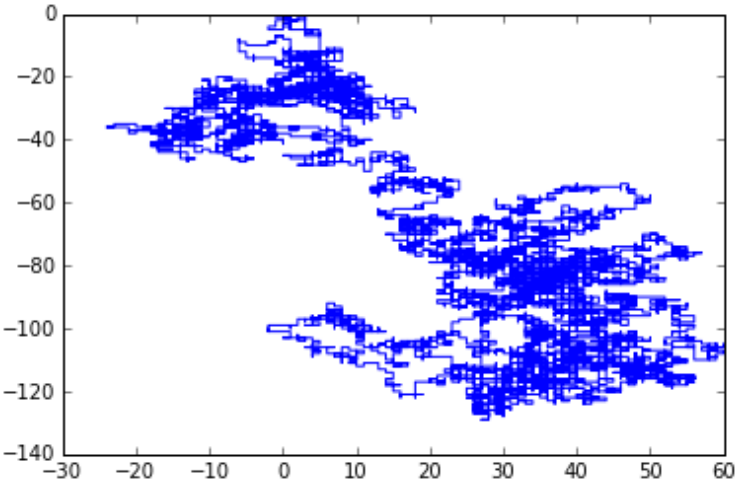
5. Le programme a déjà été écrit ci-dessus. Un exemple de trajectoire pour $n = 100$:



Puis un autre pour $n = 1000$:



Et un dernier pour $n = 10000$:



Ceux qui tiennent à en savoir plus sur les courbes obtenues iront se renseigner sur le mouvement brownien sur Internet.

6.
```
def marche2dbis() :
    absc=[0]
    ordo=[0]
    n=0
    while not(n>0 and absc[-1]!==0 and ordo[-1]==0) :
        n+=1
        r=randint(1,4)
        if r==1 :
            absc.append(absc[-1]+1)
            ordo.append(ordo[-1])
        elif r==2 :
            absc.append(absc[-1])
            ordo.append(ordo[-1]+1)
        elif r==3 :
            absc.append(absc[-1]-1)
            ordo.append(ordo[-1])
        else :
            absc.append(absc[-1])
            ordo.append(ordo[-1]-1)
    return n
```

Au premier essai, mon Python m'a donné l'impression de planter, mais il a fini par revenir à l'origine après 17 734 252 pas ! Au deuxième essai, j'ai eu droit à une « Memory error » de Python. Au troisième essai, il est revnue à l'origine au bout de deux pas (ce qui devrait se produire une fois sur quatre). Au quatrième aussi. Au cinquième, il a mis 1 840 980. Bref, c'est très très variable ! Pour les plus curieux, le mathématicien hongrois Pólya a démontré en 1921 que, dans le cas d'une marche aléatoire à deux dimensions, la probabilité de repasser par l'origine était égale à 1 (autrement dit, on finira toujours par repasser par 0). Par contre, pour le même problème en dimension 3 (ou plus), cette probabilité n'est plus égale à 1.