TP 23
Aléatoire

Dans tout ce TP nous importons le module `random` avec la commande, à placer bien sûr une fois pour toute au début

```
import random as rd
```

et nous allons apprendre à l'utiliser.

Remarque importante 1 Nous avons fait le **choix** ici de l'appeler `rd`, mais on pourrait aussi importer uniquement les fonctions que l'on souhaite. Les fonctions sont donc à précéder de `rd`, par exemple `rd.randint(a, b)`. Un sujet écrit **doit** préciser comment les fonctions ont été importées.

Remarque importante 2 Nos fonctions étant aléatoires, il est important pour faire des tests de les relancer plusieurs fois. La plupart contiennent déjà des boucles qui simulent une expérience répétée n fois, où le nombre n est passé comme argument à la fonction.

Remarque importante 3 Une bonne pratique est de **toujours récupérer le contenu d'une fonction aléatoire dans une variable**. Par exemple, on n'écrit **pas**

```
if rd.randint(1, 6) == 6:
    ...
```

mais

```
x = rd.randint(1, 6)
if x == 6:
    ...
```

En effet, dans la première syntaxe, après l'exécution la valeur aléatoire qui a été utilisée est perdue...

I Présentation du module `random`

Le module `random` possède les fonctions suivantes, auxquelles on accède dans ce TP avec le préfixe `rd` :

1. `randint(a, b)` : donne un nombre entier au hasard entre les deux bornes `a` et `b`, de façon équiprobable. Contrairement à ce qui se passe avec `range`, les deux bornes sont bien incluses.
2. `random()` : donne un nombre réel aléatoire dans l'intervalle $[0, 1[$.

Ce sont les seules à connaitre en première année ; les lois de probabilités dites *continues* (choix d'un nombre réel au hasard, et non pas d'un nombre entier parmi un nombre fini de possibilités) seront étudiées nettement plus en détail en deuxième année. Mentionnons tout de même les fonctions suivantes qui sont faciles à utiliser et peuvent servir :

3. `uniform(a, b)` : donne un nombre réel au hasard entre les deux bornes `a` et `b`. On parle de **loi uniforme** car « chaque partie du segment a la même probabilité d'être choisie ». Mathématiquement cela signifie que la probabilité d'avoir un nombre qui tombe dans un morceaux $[x, y]$ de l'intervalle $[a, b]$ est proportionnelle à la longueur de cet intervalle ; c'est donc $\frac{y-x}{b-a}$, car $[a, b]$ a probabilité 1.
4. `randrange(a, b)` : similaire à `randint` mais où la borne `b` est **exclue**. On a aussi `randrange(n)` qui est équivalent à `randrange(0, n)`, c'est à dire `randint(0, n-1)`. Contrairement aux apparence cela est parfois **moins** casse-tête que `randint` : pour travailler avec des indices au hasard dans une liste `L` de taille `n` alors on peut utiliser `i = randrange(n)`, ce qui est « bien compatible » avec la syntaxe `for i in range(n)`, sans avoir à jongler entre des `n` et des `n-1`.
5. `choice(L)` : permet de choisir un élément au hasard dans une séquence `L` (liste, tuple, ...)

   ```
   >>> rd.choice(["oui", "non", "peut-être"])
   ```

6. `shuffle(L)` : mélange une liste `L` au hasard.
7. `gauss(mu, sigma)` : donne un nombre réel au hasard selon la loi normale (gaussienne) d'espérance `mu` et d'écart-type `sigma` (cela sera nettement approfondi en deuxième année).

II Cas de base

Exercice 1. L'exercice le plus basique est de simuler un lancement de pile ou face. Pour cela on tire un nombre entier au hasard par exemple entre 1 et 2 et on considère que 1 est pile et que 2 est face. On améliorera par étapes successives la fonction.

1. Écrire une fonction `pileface()` qui tire ainsi un nombre au hasard puis affiche `"pile"` ou `"face"` selon le résultat du tirage.
2. Améliorer la fonction précédente en une fonction `pilefaces(n)` qui effectue n tirages successifs, en affichant le résultat de chaque tirage.
3. Écrire une fonction `compte_pileface(n)` qui effectue n lancer de pile ou face et compte le nombre de piles et de faces obtenus, dans des variables `p` (nombre de piles) et `f` (nombre de faces). La fonction renvoie le tuple `(p, f)`.
4. Enfin, reprendre la même fonction mais qui à la fin renvoie non pas le nombre mais la fréquence (éventuellement en pourcentage) de piles et de faces. On l'appellera `simule_pileface(n)`.

La méthode précédente constitue la base de la simulation d'une expérience aléatoire, répétée plusieurs fois, pour laquelle on s'intéresse à la fréquence d'apparition d'un évènement. Plus le nombre de répétitions est grand, plus la fréquence se rapproche de la probabilité. Pour tous les programmes suivants il est intéressants de travailler en améliorant sa fonction par étapes successives, pour faire d'abord **un** tirage aléatoire **puis** l'imbriquer dans une boucle à répéter n fois **puis** calculer les nombres de résultats obtenus et enfin les fréquences. Cela permet aussi de tester son programme au fur et à mesure...
Reprenons la même méthode pour cette fois le lancer de dé. On ne va pas créer 6 variables mais utiliser une liste de longueur 6.

Exercice 2. On lance un dé équilibré à 6 faces. Écrire une fonction `simule_de(n)` qui tire des dés n fois et renvoie la liste de longueur 6 donnant la fréquence d'apparition de chacune des faces.
Cela nécessite au départ d'initialiser une liste de longueur 6 *pour compter les lancers. À cause des problèmes de numérotation on considère que* `L[x]` *correspond au nombre de fois qu'a été obtenu le nombre* $x+1$ *comme face du dé. À la fin on peut créer une nouvelle listes donnant les fréquences, à partir de celle donnant les nombres.*

Remarque 1. Ici par exemple, il est peut-être plus clair de considérer que les faces du dé sont numérotées entre 0 et 5 et d'utiliser `randrange(6)`... Disons peu importe pour l'instant du moment qu'on ne se trompe pas.

On souhaite étudier des pièces biaisées avec une probabilité de 0,7 d'obtenir pile. Pour cela la méthode est de choisir un nombre **réel** x au hasard entre 0 et 1, puis de considérer qu'il s'agit d'un pile si $x < 0{,}7$ et d'un face sinon.

Exercice 3. Écrire une fonction `simule_piece_biaisee(n)` qui simule n fois un lancer d'une telle pièce biaisée, et renvoie le tuple formé de la fréquence des piles et faces obtenues.

La généralisation naturelle des deux cas précédents est de lancer un dé truqué. On suppose que les probabilités d'obtenir chaque face sont les suivantes.

1	2	3	4	5	6
$\frac{1}{12}$	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{24}$	$\frac{1}{8}$	$\frac{1}{4}$

(1)

L'idée naturelle est de tout mettre au même dénominateur puis de diviser l'intervalle $[\![1, 24]\!]$ en 6 parties, et de trouver quelle partie va correspondre à quelle face... Par exemple on va considérer que 1 et 2 correspondent à la face 1, puis que les nombres de 3 à 8 à la face 2 et ainsi de suite ; on a donc besoin de remplir à la main, avant d'écrire le programme, un tableau des cumuls :

1	2	3	4	5	6
$\frac{2}{24}$	$\frac{6}{24}$	...	...	...	...
2	8	...	...	...	...

(2)

Exercice 4. Écrire une fonction `simule_de_biaise(n)` qui répète n fois la simulation du lancer d'un tel dé biaisé, et renvoie la liste des fréquences obtenues pour chaque face.

III Diverses situations

Exercice 5. On admet la proposition suivante : si on tire au hasard un point du plan à l'intérieur d'une zone rectangulaire, la probabilité de tomber dans un domaine donné est proportionnelle à son aire.
Écrire une fonction `simule_pi(n)` qui répète n fois l'expérience de tirer au hasard un point, représenté par ses deux coordonnées, du carré $[0,1]\times[0,1]$, et donne la fréquence des points qui tombent dans le quart de cercle de centre 0 et de rayon 1. En fait, il est plus intéressant d'avoir la fréquence multipliée par 4...

Exercice 6. On souhaite choisir un nombre au hasard entre 1 et 10, mais pas 5.
On idée naïve serait de tirer un nombre au hasard, puis de recommencer si jamais le résultat est égal à 5. Il faudrait avoir vraiment beaucoup de malchance pour tirer plus de quelques fois de suites le nombre 5 et donc on va bien y arriver.
Mais on propose plutôt l'algorithme suivant :

1. On tire un nombre au hasard `x` entre 1 et 9,
2. Si `x` est entre 1 et 4 : c'est bon, on renvoie `x`,
3. Sinon, c'est qu'il est entre 5 et 9, et alors on « décale de 1 »pour obtenir un nombre entre 6 et 10 et c'est celui-ci qu'on renvoie.

Écrire la fonction `pas_cinq()` qui tire un nombre au hasard entre 1 et 10, mais pas 5 (ne pas oublier de la tester plusieurs fois de suite).

Exercice 7. En s'inspirant de la méthode précédente, écrire une fonction `paire_differents(a, b)` qui renvoie un tuple `(x, y)` garantis aléatoires et avec $x \neq y$, entre `a` et `b` inclus.

Exercice 8. Écrire une fonction `ADN(n)` qui génère une séquence ADN de longueur n au hasard, composée parmi les quatre lettres A, T, C, G.
Étant donnée une ***liste*** `L` *de* ***chaines de caractères****, la syntaxe* `"".join(L)` *permet de concaténer toutes les chaines de la liste en une seule.*

IV Processus aléatoire

Exercice 9. Implémenter les fonctions du problème du DS 7. On rappelle l'énoncé :

> Le climat des régions tempérées est caractérisé par une alternance de périodes de beau temps et de mauvais temps, entrecoupées par des périodes plus brèves et plus extrêmes de froid intense ou de forte chaleur. Ce problème propose d'étudier un modèle (très) simplifié de ce type de climat comportant quatre états : gel (noté G), pluie (noté P), soleil (noté S) et canicule (noté C). On suppose que ces états évoluent quotidiennement selon les règles suivantes :
> — s'il gèle un jour, alors, le lendemain, ou bien il continue de geler avec probabilité 1/7 ou bien il pleut avec probabilité 6/7 ;
> — s'il pleut un jour, alors, le lendemain, ou bien il gèle avec probabilité 1/7, ou bien il continue de pleuvoir avec probabilité 3/7, ou bien le soleil succède à la pluie avec probabilité 3/7 ;
> — s'il fait soleil un jour, alors, le lendemain, ou bien le temps se dégrade à la pluie avec probabilité 3/7, ou bien il reste ensoleillé avec probabilité 3/7, ou bien il devient caniculaire avec probabilité 1/7 ;
> — si un jour est caniculaire, alors, le lendemain, ou bien le temps redevient simplement ensoleillé avec probabilité 6/7, ou bien il reste caniculaire avec probabilité 1/7.

Pour cela, on représente les états par une chaine de caractères composée d'une seule lettre `"G"`, `"P"`, `"S"` ou `"C"`.

1. Écrire une fonction `lendemain(etat)` qui prend en argument en tel état et renvoie un état possible pour le lendemain, calculé aléatoirement selon ces règles.
2. Écrire une fonction `simule_climat(jours)` qui simule cette expérience en donnant l'état obtenu après le nombre de jours donnés, en partant de l'état initial S.
3. Enfin, écrire une fonction `frequence_climat(jours, n)` qui renvoie la fréquence, lorsque l'on répète n fois l'expérience, de chaque état où le climat évolue selon ce processus pendant ce nombre de jours consécutifs. On pourra utiliser, non pas une liste mais un dictionnaire, initialisé avec `{"G": 0, "P": 0, "S": 0, "C": 0}`.

Exercice 10. *Urne de Pólya*
On considère l'expérience suivante. Une urne contient initialement deux boules, une noire (N) et une blanche (B). On répète un nombre p fois de suite l'expérience suivante : on tire un boule au hasard dans l'urne, puis on la remet, et on remet **en plus, une nouvelle boule** de la même couleur. Ainsi au départ si on tire une boule noire, au tirage suivant on sera dans une urne avec deux boules noires et une blanche ; et si au contraire on tire une boule blanche au départ, alors pour le tirage suivant on aura une urne avec une boule noire et deux blanches. Au bout de p tirages, il y a donc $p+2$ boules en tout, et pour chaque couleur au moins 1 et au plus $n+1$.
Écrire une fonction `polya(p)` qui effectue cette expérience avec p tirages successifs dans une même urne et renvoie le tuple formé du nombre de boules blanches et noires après les p tirages ; puis une fonction `simule_polya(p, n)` qui répète n fois l'expérience et renvoie une liste de taille $p+2$ où l'élément d'indice i est la fréquence des expériences aboutissant à une composition de l'urne avec i boules blanches.
Indication : la fonction `polya` contient une boucle avec deux variables `N, B` qui comptent le nombre le boules de chaque couleur au fur et à mesure de l'expérience.
Reprendre la même expérience mais avec une urne initiale contenant 2 boules blanches et 3 boules rouges.

Exercice 11. Le *problème de Monty-Hall* est un célèbre paradoxe de probabilités inspiré d'un jeu télévisé avec les règles suivantes :

1. Un candidat se trouve face à trois portes. Derrière l'une de ces portes se cache une voiture, derrière les deux autres se cachent une chèvre. Ces cadeaux ont été répartis au hasard derrière les portes.
2. Le candidat choisit une porte au hasard en espérant y découvrir la voiture, mais ne l'ouvre pas tout de suite.
3. Le présentateur, qui lui sait comment ont été répartis les cadeaux, montre alors au candidat ce qu'il y a derrière unes des portes : pas celle qu'il a choisi, ni celle contenant la voiture. Il y a en effet soit une soit deux chèvres parmi les deux portes qui le candidat n'a pas choisi.
4. Le candidat a alors possibilité de conserver son choix initial (ouvrir la porte qu'il avait choisi dès le départ) ou bien changer (pour la porte qui n'a pas été ouverte par le présentateur).

Le paradoxe est : calculer la probabilité de trouver la voiture dans chacun des deux cas… Y a-t-il une meilleur stratégie ?
Le but est d'écrire deux fonctions `simule_montyhall_conserve(n)` et `simule_montyhall_change(n)` qui répètent n fois cette expérience, où dans la première le candidat conserve son choix initial et dans la seconde il change de porte. Les fonctions renvoient la fréquence des répétitions où le candidat a bien trouvé la voiture.
On pourra numéroter les portes 1, 2, 3 *et choisir une porte revient simplement à choisir un nombre entre* 1 *et* 3. *En fait, on peut toujours supposer que la voiture est derrière la porte numéro* 1 *(quitte à renuméroter les portes). Il faut donc choisir au hasard une porte que le candidat choisit au premier tour. Puis éventuellement, dans le cas où le candidat a choisi la bonne porte dès le début, le présentateur choisit au hasard entre les deux portes à montrer contenant une chèvre. Ensuite ce ne sont que quelques `if` et `elif` correctement emboités.*

Annexe : quelques notions sur la génération des nombres aléatoires

Un ordinateur n'est pas capable de produire du *vrai* hasard. Quand bien même ce vrai hasard existerait, il n'est pas capable de lancer un dé ou de tirer des boules, mais uniquement de faire des calculs.
La méthode la plus simple se base en fait sur un certain type de suite de récurrence $u_{n+1} = f(u_n)$. La suite n'est donc pas du tout aléatoire, et est même nécessairement périodique, cependant :

1. La fonction f est choisie pour que la suite *ait l'air* le plus aléatoire possible,
2. La période de la suite doit être la plus grande possible,
3. Le nombre u_0 appelé *graine* (en anglais *seed*) est fixé en dépendant de divers paramètres de l'ordinateur comme par exemple le temps (date, heure, minutes, secondes). Des graines qui varient *un peu* vont produire des suites *très* différentes.

On parle de **nombres pseudo-aléatoires**.
Si la graine n'est pas aléatoire, alors la sortie sera toujours la même, ce qui peut provoquer divers bugs ou failles de sécurité : testez et vous devriez obtenir exactement le même résultat dans le même ordre.

```
>>> rd.seed(12) # choix d'une graine : 12
>>> rd.randint(1, 10)
8
>>> rd.randint(1, 10)
5
>>> rd.randint(1, 10)
9
>>> rd.randint(1, 10)
6
>>> rd.randint(1, 10)
3
```

Ces problématiques sont bien entendu cruciales dans de nombreux domaines dans lesquels il faut produire de l'aléatoire « de bonne qualité » (simulations en sciences, méthode de Monte-Carlo, mais aussi sécurité et cryptographie), et ont donc été largement étudiées à coup de statistiques et de raffinements algorithmiques.
Fort heureusement, Python utilise l'algorithme *Mersenne Twister 19937* qui est suffisamment élaboré, avec un aléatoire de bonne qualité selon de nombreux critères et une période de $2^{19937}-1$ soit environ $4{,}3 \cdot 10^{6001}$. Et c'est beaucoup. Beaucoup. Beaucoup.