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.
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 déjà été importées, ou alors c'est à vous de le faire.

Avertissement

Nos fonctions étant par nature 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.
De plus, une bonne pratique est de **toujours récupérer le contenu d'une fonction aléatoire dans une variable**. Par exemple, on n'écrit **jamais**

```
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... Et si on l'utilise plusieurs fois on n'obtient pas le même résultat !

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 uniformément 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 choisi uniformément dans l'intervalle $[0, 1]$. La documentation précise que l'intervalle est en fait $[0, 1[$, mais la probabilité d'obtenir exactement 1 est nulle donc cela fait peu de différence.

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**, comme avec `range`. On a aussi `randrange(n)` qui est équivalent à `randrange(0, n)`, c'est-à-dire `randint(0, n-1)`. Contrairement aux apparences cela est parfois *moins* casse-tête que `randint` : pour travailler avec des indices au hasard dans une liste `L` de longueur `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 uniformément au hasard un élément 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 ; on peut retenir qu'environ 2/3 des valeurs sont dans l'intervalle $[\mu - \sigma, \mu + \sigma]$ et environ 95 % des valeurs sont dans $[\mu - 2\sigma, \mu + 2\sigma]$.

Enfin toutes sont listées dans l'aide du module `help(rd)`, et chacune possède aussi sa propre aide.

II Pour démarrer

On s'intéresse d'abord aux cas les plus basiques possibles, et on raisonne en améliorant sa fonction par étapes successives.

Exercice 1. Pour s'échauffer, on simule 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.

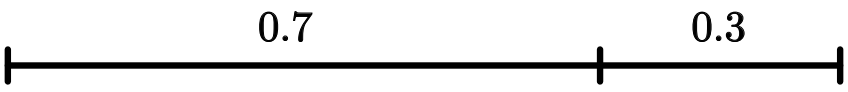
1. Écrire une fonction `pileface()` qui tire ainsi un nombre au hasard puis affiche `"pile"` ou `"face"` selon le résultat du tirage.
2. Améliorer la fonction précédente en une fonction `pilefaces(n)` qui effectue n tirages successifs, en affichant le résultat de chaque tirage.
3. Écrire une fonction `compte_pilefaces(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 la liste de deux éléments `[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_pilefaces(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...

Exercice 2. On s'intéresse maintenant au dé équilibré à 6 faces. Écrire une fonction `simule_dé(n)` qui tire des dés n fois et renvoie la liste de longueur 6 donnant la fréquence (ou bien, pour commencer, le nombre) d'apparition de chacune des faces. Pour compter on n'utilisera pas 6 variables, mais directement une liste de longueur 6 initialement remplie de zéros. *Attention car les indices de la liste sont numérotés à partir de* 0 *alors que les faces d'un dé sont numérotés à partir de* 1...

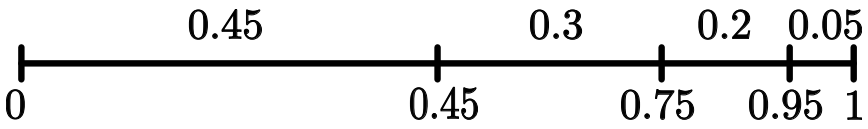
III Quand est-ce qu'on biaise

On souhaite maintenant 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. Cela correspond bien à l'idée que la probabilité est de 0,7 de choisir un nombre dans $[0; 0{,}7]$, et 0,3 de le choisir dans $[0{,}7; 1]$.



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

Cette méthode se généralise à plusieurs choix possibles, chacun avec sa probabilité ; il faut interpréter cela en terme de choix d'une partie du segment $[0, 1]$.



Exercice 4. Un client anonyme se présente à la boulangerie. Selon son humeur, il choisit au hasard entre le pain au chocolat avec probabilité 0,45, le croissant avec probabilité 0,3, le pain suisse avec probabilité 0,2 et une simple baguette avec probabilité 0,05.

1. Écrire une fonction `choix_boulangerie()` qui choisit au hasard un élément parmi ceux-ci, et l'affiche. On considère qu'un nombre entre 0 et 0,45 correspond au pain au chocolat, entre 0,45 et 0,75 correspond au croissant, etc.
2. Écrire une fonction `simule_boulangerie(n)` qui répète l'expérience n fois, et renvoie un dictionnaire de 4 clés nommées `"pain au chocolat"`, `"croissant"`, `"pain suisse"`, `"baguette"`, indiquant le nombre de fois où chaque élément a été choisi.

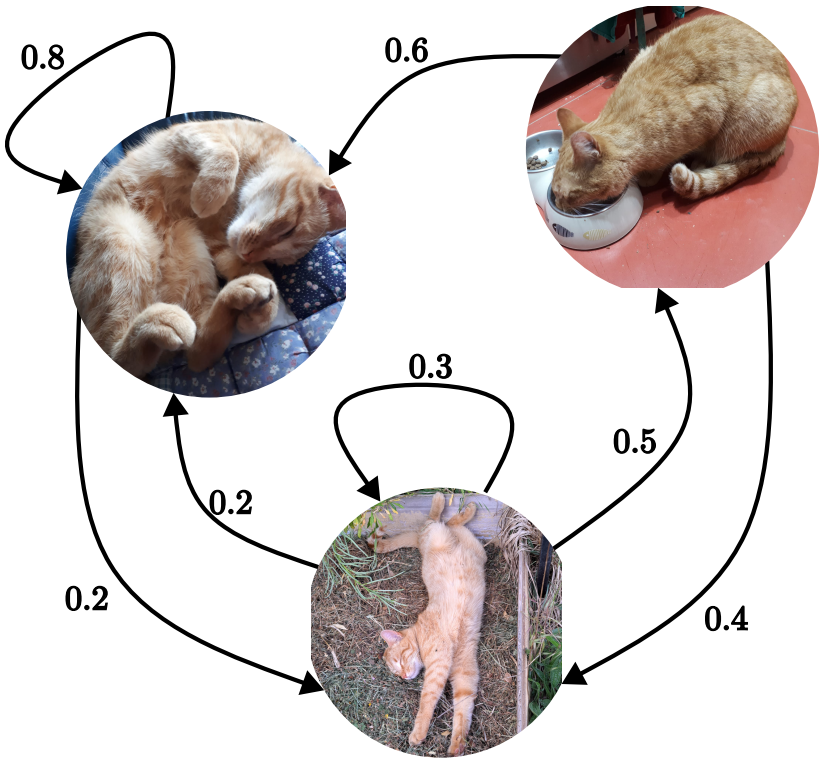
IV Processus aléatoires

Il s'agit maintenant d'étudier des situations un peu plus complexes.

Exercice 5. Le chat (le même que celui du TP sur les graphes !) passe sa vie entre trois activités : dormir (D), sortir dehors (S), et manger (M). À chaque heure, il passe d'un état à l'autre de façon aléatoire selon les règles suivantes :

1. S'il dort, alors il continue à dormir avec probabilité 0,8 ou sort dehors avec probabilité 0,2.
2. S'il est dehors, alors il reste dehors avec probabilité 0,3, va manger avec probabilité 0,5 et va dormir avec probabilité 0,2.
3. S'il mange, alors il va dormir avec probabilité 0,6 ou va dehors avec probabilité 0,4.

On peut représenter cette situation par un graphe pondéré et orienté (attention, il y a un poids dans chacun des sens), où les poids indiquent les probabilités de passer d'une activité à une autre. On note les activités simplement par les caractères `"D"`, `"S"` et `"M"`.



1. Écrire une fonction `suivant(x)` qui prend en argument l'activité actuelle du chat, et choisi au hasard l'activité suivante selon les règles décrites.

2. Écrire une fonction `simule_chat(n)` qui simule le changement d'activités du chat, n fois de suite, en démarrant par un chat qui dort.
3. Améliorer la fonction précédente pour qu'elle renvoie un dictionnaire de trois clés `"D"`, `"S"`, `"M"`, indiquant le pourcentage du temps que le chat a passé dans chaque activité. Tester avec différentes activités de départ.

À partir d'une telle situation, on forme la matrice $M = \begin{pmatrix} 0{,}8 & 0{,}2 & 0{,}6 \\ 0{,}2 & 0{,}3 & 0{,}4 \\ 0 & 0{,}5 & 0 \end{pmatrix}$ ($M_{i,j}$ est la probabilité de passer de l'activité j vers l'activité i, où $i = 0, 1, 2$ représentant dans l'ordre les états D, S, M). La somme des valeurs sur chaque colonne est égale à 1. On peut démontrer que les fréquences de la question précédentes convergent vers un vers un vecteur $X = \begin{pmatrix} d \\ s \\ m \end{pmatrix}$ tel que $MX = X$ (un tel X est unique à multiplication par une constante près), avec tous ses coefficients positifs et la somme des coefficients valant 1.

4. Vérifier cette affirmation. On pourra former la matrice M, et le vecteur X, avec `numpy` (voire par exemple le Mémo Agro-Véto) et calculer le produit MX.

V Diverses situations

Exercice 6. 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 compte combien de fois le point tombe dans le quart de cercle de centre 0 et de rayon 1. En fait, on renverra la fréquence multipliée par 4...

Exercice 7. On souhaite écrire une fonction `couple_différents()` qui renvoie un couple de deux nombres aléatoires, disons entre 1 et 6 (lancers de dés), en garantissant que ces deux nombres sont différents.
Une idée naïve pourrait être de tirer les deux nombres au hasard, et éventuellement recommencer si les deux sont les mêmes. Il y a de fortes chances qu'au deuxième essai cela fonctionne, ou au pire quelques essais supplémentaires, mais pas des milliers...
Une meilleur idée est la suivante : on tire un premier nombre x entre 1 et 6, puis un deuxième nombre y entre 1 et 5. Si $y < x$ on renvoie bien le couple (x, y), mais sinon on augmente y de 1. Par exemple si on tire $x = 3$, cela revient à considérer que le tirage de $y = 1$ ou $y = 2$ correspond bien aux couples $(3, 1)$, $(3, 2)$, alors que les tirages $y = 3$, $y = 4$, $y = 5$ correspondent respectivement aux tirages $(3, 4)$, $(3, 5)$, $(3, 6)$.
Écrire cette fonction, et la tester pour suffisamment de répétitions pour que la probabilité d'avoir deux lancers de dés identiques soit très faible...

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`.
On pourra commencer par générer une liste de lettres individuelles. 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.

VI 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 (question philosophique !), il n'est pas capable de lancer un dé ou de tirer des boules, mais uniquement de faire des calculs. Pour produire des nombres aléatoires, les méthodes les plus simples sont basées sur certains types de suites récurrentes $u_{n+1} = f(u_n)$; elles peuvent prendre des grandes valeurs, mais on peut s'intéresser uniquement à leur réduction par exemple modulo 10 si on veut un nombre entre 0 et 9. La suite n'est donc pas du tout aléatoire, et est même nécessairement périodique *(pourquoi ?)*, 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, et on ne doit pas pouvoir facilement prédire le terme suivant en connaissant le ou les termes précédents,
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**.
Un exemple très simple est donné par

$$u_{n+1} = 137u_n + 187 \mod 256 \tag{1}$$

qui donne donc des nombres entre 0 et 255.
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. Par exemple si le nombre aléatoire sert à fournir un code de validation de paiement pour une carte bleue il est extrêmement important qu'on ne puisse pas prédire les nombres suivants, sinon un pirate pourrait valider des paiements à votre place ! C'est moins grave par exemple pour un jeu vidéo, où il s'agit de faire apparaitre un ennemi à un endroit au hasard, alors la qualité du hasard à l'échelle de millions de données importe bien moins que le temps de calcul et on veut un algorithme simple et rapide.
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. Il est donc bien adapté à des situations en sciences mais, malgré cela, il possède quelques failles et n'est pas considéré comme cryptographiquement sûr : ne pas l'utiliser pour générer des codes de carte bleue !