TP 21
Aléatoire

Le module Python `random` contient plusieurs fonctions permettant de produire de l'aléatoire. Nous allons l'étudier, en lien avec le chapitre mathématiques de probabilités. 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
```

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 plutôt

```
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` :

- `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.
- `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 et nous pouvons déjà faire de nombreuses choses avec ; 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 :

- `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.
- `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 de l'utiliser : pour travailler avec des indices au hasard dans une liste `L` de longueur `n` alors on peut utiliser `i = randrange(n)` puis accéder à `L[i]`, ce qui est bien compatible avec la syntaxe `for i in range(n)`, sans avoir à jongler entre des `n` et des `n-1`.
- `choice(L)` : permet de choisir uniformément au hasard un élément dans une séquence `L` (liste, tuple, …)

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

- `shuffle(L)` : mélange une liste `L` au hasard.
- `gauss(mu, sigma)` : donne un nombre réel au hasard selon la loi normale (gaussienne) d'espérance `mu` et d'écart-type `sigma`. Brièvement, la valeur de `mu` correspond à la moyenne des nombres tirés, et la valeur de `sigma` correspond à la largeur typique : plus `sigma` est grand plus les valeurs au hasard vont être étalées autour de `mu`.

Toutes les fonctions du module sont listées dans l'aide `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 *Échauffement*
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 lancers 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 numérotées à partir de* 1…

III Quand est-ce qu'on biaise

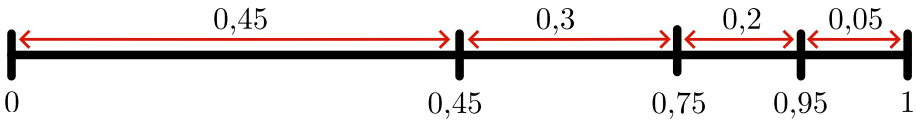
On souhaite maintenant étudier des pièces biaisées avec une probabilité de 0,7 d'obtenir pile. Pour cela la méthode est de choisir un nombre **réel** x au hasard entre 0 et 1, puis de considérer qu'il s'agit d'un pile si $x < 0{,}7$ et d'un face sinon. Cela correspond bien à l'idée que la probabilité est 0,7 de de choisir un nombre dans $[0; 0{,}7]$ et de 0,3 de le choisir dans $[0{,}7; 1]$ (les inégalités strictes ou larges n'ont pas d'importance ici). L'image à avoir en tête est qu'on découpe un intervalle en des morceaux de longueur proportionnelle à la probabilité à simuler.

Exercice 3
Écrire une fonction `simule_pièce_biaisée(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.

Pour simuler cette situation, on tire un nombre réel au hasard dans $[0,1]$. On considère qu'un nombre dans $[0;0{,}45[$ correspond au pain au chocolat, un nombre dans $[0{,}45;0{,}75[$ correspond au croissant, un nombre dans $[0{,}75;0{,}95[$ au pain suisse et un nombre dans $[0{,}95;1]$ à la baguette. Ce faisant, nous avons bien divisé l'intervalle $[0,1]$ en morceaux dont la longueur correspond à la probabilité :



1. Écrire une fonction `choix_boulangerie()` qui choisit au hasard une viennoiserie avec cette règle et renvoie une chaine de caractères `"pain au chocolat"`, `"croissant"`, `"pain suisse"`, ou `"baguette"`.
2. Écrire une fonction `simule_boulangerie(n)` qui répète l'expérience n fois, et renvoie un dictionnaire dont les quatre clés sont les quatre chaines de caractères précédentes, indiquant le nombre de fois où chaque viennoiserie a été choisie.

Le même client se présente maintenant dans une boulangerie allemande. Il choisit entre le Schokocroissant avec probabilité 0,32 (hummm), le Laugendreieck avec probabilité 0,19 (un délice), le Mohnbrötchen avec probabilité 0,06 (bon mais très simple), le Scharferkumpel avec probabilité 0,13 (épicé, pas pour tous les jours), le Rosinenschnecke avec probabilité 0,21 (délicieux mais très gras et sucré), et enfin le Erdbeerplunder avec probabilité 0,09 (bon, mais très sucré et disponible selon la saison).

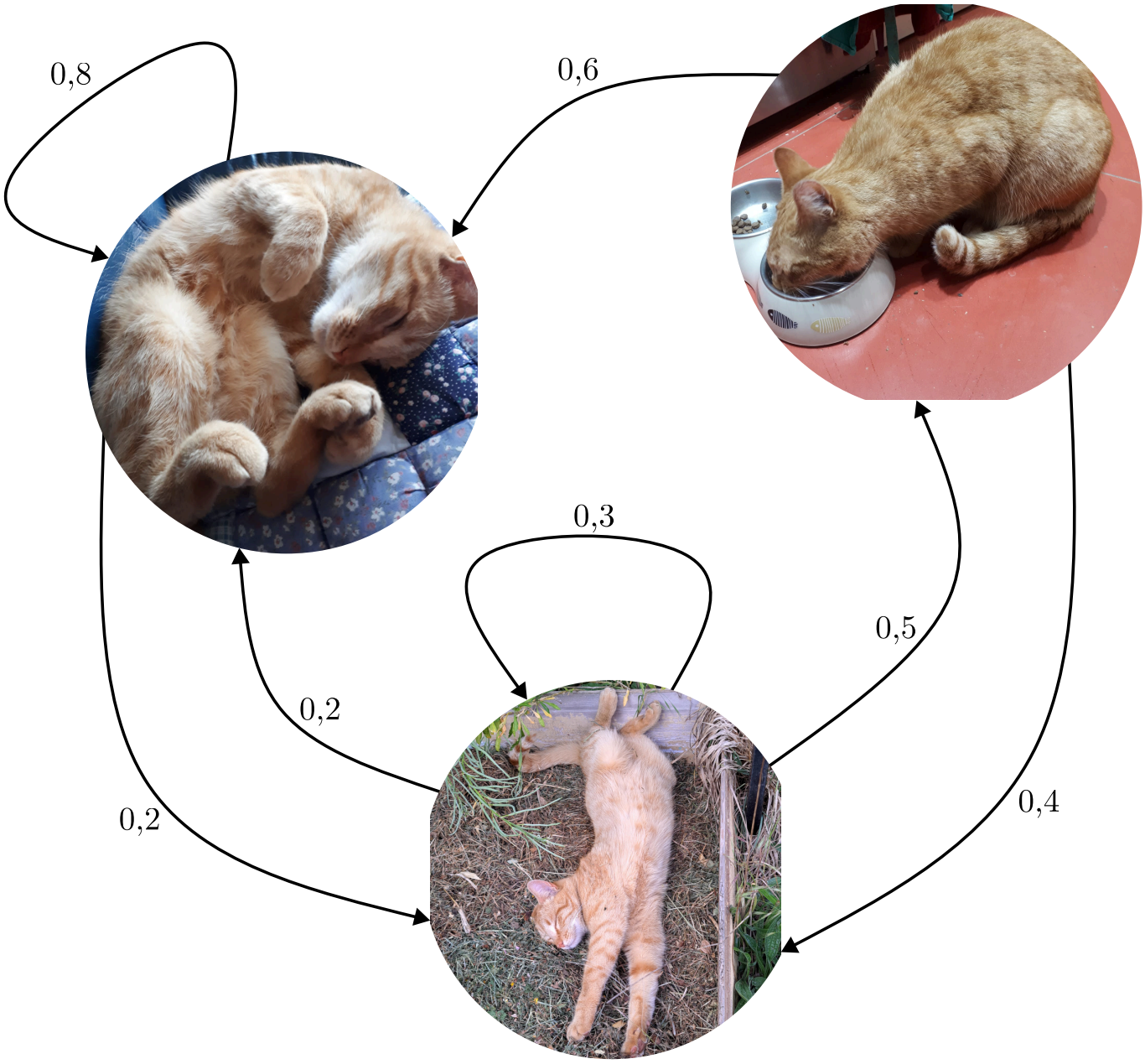
3. Reprendre les questions précédentes dans ce cas.

Figure 3. – Bäckerei Klingenstein à Essen
https://www.baeckerei-klingenstein.de/Filialen/Kurfuerstenstr-5-mit-Cafe

À retenir

Garder en tête l'interprétation géométrique : pour tirer au hasard entre plusieurs cas avec des probabilités données $p_1, ..., p_n$, à partir d'un nombre uniformément au hasard entre 0 et 1 (les seules fonctions auxquelles nous avons vraiment accès sont `randint()` et `random()` !), on fabrique une division de l'intervalle $[0, 1]$ en morceaux de longueurs proportionnelles aux probabilités $p_1, ..., p_n$. On est alors typiquement ramené à un test de « conditions en cascades » avec des `elif` successifs.

IV Diverses situations



Exercice 5

Le chat (le même que celui du TP sur les graphes !) passe sa vie entre trois activités : dormir, sortir dehors, et manger. À chaque heure, il passe d'un état à l'autre de façon aléatoire avec une certaine probabilité. Ces activités sont représentées comme les sommets d'un graphe orienté pondéré, et la probabilité de passer d'une activité à l'autre est indiquée sur l'arête. On note les activités simplement par les caractères `"D"`, `"S"` et `"M"` (ou bien par leur nom complet).

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, et pour un nombre de répétitions n assez grand.
4. (Mathématiques) On note, pour $n \in \mathbb{N}$, d_n la probabilité qu'après n heures le chat soit en train de dormir, s_n la probabilité qu'il soit sorti et m_n la probabilité qu'il soit en train de manger. Justifier que ces trois suites vérifient la relation de récurrence qu'on peut écrire sous forme matricielle

$$\forall n \in \mathbb{N}, \quad \underbrace{\begin{pmatrix} d_{n+1} \\ s_{n+1} \\ m_{n+1} \end{pmatrix}}_{X_{n+1}} = \underbrace{\begin{pmatrix} 0{,}8 & 0{,}2 & 0{,}6 \\ 0{,}2 & 0{,}3 & 0{,}4 \\ 0 & 0{,}5 & 0 \end{pmatrix}}_{A} \underbrace{\begin{pmatrix} d_n \\ s_n \\ m_n \end{pmatrix}}_{X_n}$$

puis que $\forall n \in \mathbb{N}$, $X_n = A^n X_0$; ici $X_0 = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$ si le chat est d'abord en train de dormir.

Remarque. La somme des probabilités sur les arêtes *sortantes* de chaque sommet du graphe est égale à 1, de même que la somme des nombres sur les *colonnes* de la matrice A, et on a $d_n + s_n + m_n = 1$ pour tout $n \in \mathbb{N}$. On a affaire à ce qui s'appelle une *chaine de Markov* : une évolution d'un processus en fonction du temps qui passe à chaque moment aléatoirement d'un état à un autre. On peut toujours la représenter par un tel graphe orienté pondéré et l'étudier avec une matrice.

Exercice 6 *Méthode de Monte-Carlo*

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 une partie du plan est proportionnelle à son aire.

Nous allons l'appliquer en choisissant au hasard un point du carré $[0,1] \times [0,1]$, représenté par ses deux coordonnées, et en comptant combien de fois le point est à l'intérieur du cercle de centre 0 et de rayon 1, pour en déduire une valeur approchée du nombre π.

Écrire une fonction `simule_pi(n)` qui répète n fois l'expérience cette expérience et renvoie la fréquence, multipliée par 4, des points qui tombent à l'intérieur du cercle.

Exercice 7 *Urne de Pólya*
On considère l'expérience aléatoire suivante. Au départ, on dispose d'une urne (opaque) contenant une boule rouge et une boule bleue (indistinguables au toucher). Nous allons répéter n fois le procédé suivant : on tire au hasard une boule, on regarde sa couleur, on la remet dans l'urne, et on ajoute en plus dans l'urne une nouvelle boule de la même couleur. Ainsi si on tire rouge au premier tirage, le deuxième tirage se fera dans une urne à deux boules rouges et une boule bleue ; alors que si on tire bleu au premier tirage, le deuxième se fera dans une urne avec une boule rouge et deux boules bleues ; et on continue.

Pour simuler cette expérience en Python, on représente la composition de l'urne par deux variables a et b, où a représente le nombre de boules rouges et b le nombre de boules bleues. Après n tirages on a toujours $a+b=n+2$; au départ $n=0$, $a=b=1$ et $a+b=2$.

1. Écrire une fonction `simule_urne(n)` qui répète l'expérience n fois, et renvoie la liste `[a, b]` représentant la composition de l'urne après n tirages.
2. Écrire une fonction `simule_composition_urne(n, N)` qui répète toute l'expérience précédente N fois (de tirer n fois consécutivement dans une urne contenant initialement une seule boule de chaque couleur), et renvoie une liste `C` de longueur $n+3$, où `C[i]` indique la fréquence en pourcentage des fois où, après n tirages, l'urne contient i boules rouges et $n+2-i$ boules bleues. Puis tester avec des valeurs de n modérées (moins de 10) et des grandes valeurs de N (plus de 1000). Qu'observez-vous ?

V 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 :

- La fonction f est choisie pour que la suite *ait l'air* le plus aléatoire possible,
- 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,
- 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

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

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
# essayez : tout le monde obtiendra le même résultat dans le même ordre
```

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 $4{,}3 \cdot 10^{6001}$: c'est beaucoup, beaucoup, beaucoup. Il est donc bien adapté à des situations en sciences. Malgré cela, il n'est pas considéré comme cryptographiquement sûr : en théorie, un pirate qui observe une séquence de 624 nombres aléatoires sortis consécutivement par la fonction `random()` pourrait prédire le suivant !