BLAISE PASCAL
PT 2022-2023

TP 1 – Informatique

C'est reparti !

Ce TP de redémarrage a pour but de vous faire réviser les bases de l'algorithmique et de la syntaxe Python : fonctions, boucles **for** et **while**, tests **if**, manipulation des listes et des dictionnaires, etc.

I - Fonctions simples sur les listes

Dans ce premier exercice, nous allons coder quelques fonctions simples s'appliquant à une liste. Bien sûr, il existe des fonctions toutes faites permettant de faire la même chose ... mais le but n'est pas de les utiliser ! Pour travailler dans un cadre unique, nous nous restreignons aux listes d'entiers, même si plusieurs des fonctions codées peuvent aussi s'appliquer aux listes de flottants, de chaînes de caractère, de booléens, etc.

Rappel de cours : vocabulaire des fonctions

- Une **fonction** informatique appelle (ou prend) des **arguments**, fait des opérations à partir de ces arguments, et retourne (ou renvoie) un **résultat**.
- Les variables définies uniquement à l'intérieur de la fonction sont appelées **variables locales**, par opposition aux **variables globales** qui sont définies pour l'ensemble du programme.
- Les variables locales peuvent remplir de multiples rôles : stockage temporaire du résultat, compteur, agrégation des données, etc.

Et enfin ... n'oubliez pas de tester vos fonctions sur des exemples simples !

1 - Écrire une fonction `somme(L)` prenant comme argument une liste `L` et renvoyant la somme de ses éléments.

2 - Adapter très légèrement la fonction précédente pour écrire une fonction `moyenne(L)`.

3 - Écrire une fonction `ecart_type(L)` renvoyant l'écart-type des éléments de `L`. On rappelle que l'écart-type d'une collection de nombres $\{x_n\}_{1\leq n\leq N}$ est défini par

$$\sigma = \sqrt{\frac{1}{N}\sum_{n=1}^{N}(x_n - \bar{x})^2} \qquad \text{avec } \bar{x} \text{ la moyenne des } \{x_n\}.$$

4 - Écrire une fonction `maxi(L)` renvoyant le plus grand entier présent dans `L`.

5 - Écrire une fonction `recherche_simple(a,L)` renvoyant un booléen qui indique si l'entier `a` est présent ou non dans la liste `L`.

6 - Écrire une fonction `nb_occurences(a,L)` renvoyant le nombre de fois où l'entier `a` apparaît dans la liste `L` ainsi qu'une liste contenant les différents indices auquel il apparaît. Si `a` n'apparaît pas dans `L`, la fonction renverra `0` et une liste vide.

II - Le jeu de la banquise

Le jeu *Little Cooperation*, édité par Djeco, est un jeu de société destiné aux enfants à partir de deux ans et demi. Il peut être joué seul ou de manière collaborative, tous les joueurs jouant à tour de rôle mais avec le même but. Ainsi, tout le monde gagne ou perd ensemble. L'objectif de cet exercice est de déterminer « expérimentalement » la probabilité de l'emporter en simulant un grand nombre de parties du jeu. Cette démarche est caractéristique des « méthodes de Monte-Carlo », ainsi nommées car elles étaient (sont ?) utilisées par des joueurs de casino pour estimer l'espérance de leurs gains.

Quatre animaux sont en excursion sur la banquise, et doivent regagner leur igloo en passant par un pont monté sur six piliers ... qui menace de s'écrouler, voir figure 1. À son tour, chaque joueur lance le dé, et agit en conséquence :

- face « pont » : un animal passe de la banquise vers le pont ;
- face « igloo » : un animal passe du pont vers l'igloo ;
- face « glaçon » : un pilier du pond fond, et est retiré du jeu.

Étienne Thibierge, 29 août 2022, www.etienne-thibierge.fr

Si l'action est impossible (p.ex. face igloo alors qu'aucun animal ne se trouve sur le pont), le tour est sans effet. L'ordre dans lequel les animaux sont joués est sans importance. La partie est gagnée si tous les animaux ont rejoint l'igloo avant que tous les piliers du pont ne se soient écroulés.

Figure 1 – Vue du plateau de jeu.

Un point très important avant de démarrer tout programme est de réfléchir à l'organisation des données. Ici, le nombre de piliers du pont encore présents sera stocké dans un entier `N` et la position des animaux sur le plateau de jeu dans un dictionnaire `animaux`, dont les clés seront les chaînes `'banquise'`, `'pont'` et `'igloo'`, et les valeurs le nombre d'animaux se trouvant à chaque endroit.

Rappel de cours : à propos des dictionnaires

En première approche, le type « dictionnaire » peut être vu comme une généralisation des listes, dont les indices sont remplacées par des **clés** pouvant être de type quelconque, et pas seulement entier. À chaque clé est associée une **valeur**, elle aussi de type quelconque. Les clés sont uniques, les valeurs pas nécessairement. Contrairement à une liste, un dictionnaire n'est pas ordonné.

Un dictionnaire peut être construit en donnant explicitement toutes ses entrées, par exemple :

```
France = {"capitale":"Paris", "population":63e6, 1789:"Révolution!!",
    "Republique?":True}
```

Il est également possible de partir du dictionnaire vide et d'y ajouter peu à peu des entrées avec des affectations de la forme `d[clé]=valeur`. Le nombre total d'entrées n'a pas besoin d'être défini au préalable, et l'ordre d'insertion est sans importance.

```
France = {}
France["capitale"] = "Paris"
France["population"] = 63e6
France[1789] = "Révolution!!"
France["Republique?"] = True
```

L'accès aux éléments du dictionnaire se fait par la syntaxe `d[clé]`, et il est possible de parcourir tous les éléments avec la syntaxe **for** `cle` **in** `d`.

Pour les lancers de dé aléatoire, on utilisera le module `random` de la bibilothèque `numpy`, importé (par exemple) avec l'alias **import** `numpy.random as rd`. La fonction `randint` permet alors de tirer aléatoirement un entier entre deux limites : `rd.randint(a,b)` renvoie un entier compris entre `a` inclus et `b` exclu.

7 - Écrire une fonction `un_tour` simulant un tour de jeu, c'est-à-dire un lancer de dé et la modification en conséquence du dictionnaire `animaux` ou du nombre de piliers `N`. Cette fonction prendra comme argument `animaux` et `N`, et retournera également `animaux` et `N`. Attention aux actions impossibles : le nombre d'animaux à un endroit du plateau ne peut pas être négatif !

> *Tester la fonction est indispensable, mais un peu plus compliqué que dans l'exercice précédent. Pour bien voir ce qu'elle fait, ne pas hésiter à inclure quelques* ***print*** *à des endroits bien choisis du code. Enfin, on pourra utiliser des dictionnaires écrits à la main, comme par exemple* `{"banquise":2, "pont":1, "igloo":1}` *(pour les cas qui marchent) ou* `{"banquise":0, "pont":0, "igloo":4}` *(pour les actions impossibles).*

8 - Écrire une fonction `une_partie` simulant une partie complète, c'est-à-dire autant de tours que nécessaire pour gagner ou perdre la partie. Cette fonction ne prendra aucun argument, et renverra 1 si la partie est gagnée et 0 sinon.

9 - Écrire une fonction `une_simu`, prenant comme argument un nombre `Np` de parties à jouer, et renvoyant le pourcentage de parties gagnées parmi les `Np`.

La fonction `une_simu` bien utilisée permet alors de répondre au problème de départ, à savoir la probabilité de l'emporter. Néanmoins, on comprend intuitivement que la fiabilité du résultat dépend du nombre `Np` de parties qui sont simulées pour estimer le pourcentage de victoires : un nombre trop petit ne pourra pas conduire à un résultat fiable ... mais qu'est-ce qu'un nombre de parties « suffisant » pour que le résultat soit « fiable » ?

Pour aborder cette question, nous allons de nouveau procéder de manière expérimentale en reproduisant un certain nombre de simulations identiques (donc chacune de `Np` parties), et en étudiant l'écart-type du pourcentage de succès, qui renseigne sur l'incertitude associée.

10 - Écrire une fonction `N_simu` prenant comme argument deux entiers `Ns` et `Np`, et réalisant `Ns` simulations de `Np` parties chacunes. Cette fonction renverra une liste de `Ns` éléments correspondant à la probabilité de succès renvoyé par chaque simulation.

11 - En faisant plusieurs essais avec `Ns=100`, estimer le nombre de parties à simuler pour que l'écart-type sur la probabilité de succès renvoyée par une simulation soit inférieur à 5 %.

III - Recherche dichotomique dans une liste triée

On dipose d'une liste d'entiers `L`, triée par ordre croissant. L'objectif est de déterminer si un entier `a` s'y trouve. L'algorithme utilisé est celui de la recherche dichotomique, plus efficace que celui de recherche simple du premier exercice, en particulier pour les listes de grande longueur :

- ▷ on regarde l'élément du milieu du tableau et on le compare à `a` ;
- ▷ s'ils sont égaux, c'est gagné ;
- ▷ s'il est plus grand, on poursuit la recherche dans la partie gauche de la liste ;
- ▷ s'il est plus petit, on utilise la partie droite.

12 - Coder une fonction `recherche_dicho(a,L)` renvoyant un booléen. Tester cette fonction sur les listes `[1,2,3,4]`, `[1,2,3,4,5]`, `[]` et `[1]`.

13 - En s'inspirant des idées utilisées dans le deuxième exercice, proposer et mettre en œuvre une méthode permettant de comparer « expérimentalement » l'efficacité des fonctions `recherche_dicho` et `recherche_simple` en fonction de la longueur de la liste (c'est-à-dire d'étudier expérimentalement la complexité des deux fonctions). N'hésitez pas à chercher sur internet ou à solliciter votre enseignant pour des fonctions ou modules permettant de répondre à des besoins particuliers ☺.