Mastermind

Le mastermind est un jeu où il faut déterminer, en le moins de coups possibles, un code formé de quatre (ou plus, mais ici on se cantonnera à des codes de 4 couleurs) pions de couleur, parmi 6 couleurs différentes. Un joueur décide du code à deviner, qu'il cache à l'autre joueur et à chaque proposition de l'autre joueur, indique avec des pions noirs et blancs le nombre de pions de la bonne couleur et à la bonne place, et ceux de bonne couleur mais mal placés.

On modélisera les couleurs par les chiffres de 1 à 6, le code et les propositions par des listes de chiffres. Ainsi, si le code à deviner est $[1, 3, 2, 3]$, alors avec $[2, 3, 1, 1]$ il y a un pion bon et bien placé, et deux bons mais mal placés...

Récupérer sur le site `https://cahier-de-prepa.fr/pcsi-bdb/docs?rep=100` le fichier `mastermind.py` (dans informatique->DM Mastermind) et l'ouvrir dans votre environnement python favori.

Pensez à éditer la ligne @author, et sauvegardez le fichier en ajoutant votre nom au nom du fichier. Vous devrez me renvoyer par mail à l'adresse `phjondot@gmail.com` le fichier python complété au plus tard le vendredi 3 janvier, tout comme vous pouvez à cette même adresse m'envoyer vos questions et demandes d'aide (votre code ne marche pas et vous ne comprenez pas pourquoi etc…)

La fonction `genereCode` a déjà été écrite. Son rôle est, comme son nom l'indique, de choisir le code à deviner (mais aucune valeur n'est renvoyée, si ce n'est `None` bien sûr).

Celui-ci est stocké dans une variable (globale) nommée `codeSecret`.

1. Compléter la fonction `testeCombinaison`, qui admet un argument obligatoire : `comb` (la proposition faite par le joueur qui cherche à deviner le code) et un argument optionnel `code` (le code à découvrir), et qui renvoie un couple `(bpp, bmp)` formé du nombre de pions bons et bien placés, et du nombre de pions bons mais mal placés.

   Si un seul argument est fourni, la combinaison proposée sera comparée au code contenu dans la variable globale `codeSecret`.

   On fera attention à ne modifier ni la combinaison proposée, ni le code secret... Attention, ce n'est pas si facile et s'y reprendre à plusieurs fois est tout à fait normal !

2. Ce faisant, vous pouvez déjà vous familiariser avec le jeu de mastermind, en jouant à découvrir le code secret choisi aléatoirement par la machine, avec les deux fonctions `genereCode` et `testeCombinaison`. Essayez !

3. On va programmer désormais un algorithme permettant à l'ordinateur de deviner le code secret. La méthode envisagée est la suivante : on ordonne toutes les combinaisons par ordre lexicographique. En pratique, c'est un peu comme si on comptait en base 6, mais avec des chiffres de 1 à 6 plutôt que les chiffres de 0 à 5...

   Ainsi, la première combinaison est alors (pour 4 pions) $[1, 1, 1, 1]$, la suivante $[1, 1, 1, 2]$, la 7-ième $[1, 1, 2, 1]$, la 1296-ième et dernière $[6, 6, 6, 6]$.

   On va tester dans l'ordre les combinaisons, mais en tenant bien sûr compte des réponses données, et ce bien sûr jusqu'à trouver le code secret.

   Par exemple, si dans $[1, 1, 1, 1]$ il y a 2 pions bons et bien placés, alors la prochaine combinaison qui sera testée est $[1, 1, 2, 2]$. En effet, $[1, 1, 1, 2]$ ne saurait convenir, ni $[1, 1, 1, 3]$ car on sait qu'il y a deux 1 dans le code secret, pas 3...

   Compléter déjà la fonction `suivant` qui admet une combinaison (de 4 chiffres a priori pour unique argument, et qui renvoie la suivante. (On fera suivre la combinaison $[6, 6, 6, 6]$ de la combinaison initiale $[1, 1, 1, 1]$.)

   Il est ici permis, si on le souhaite, de modifier la combinaison fournie en entrée, sous forme de liste.

4. Compléter alors la fonction `IA()` pour qu'elle soit en mesure, en faisant appel à `testeCombinaison`, de détecter le code secret.

   On pourra afficher au fur et à mesure les combinaisons testées et les réponses du test, et on renverra un couple formé du code secret et du nombre de propositions faites jusqu'à trouver le code secret.

   Un exemple possible d'exécution (ici la fonction affiche au fur et à mesure toutes les combinaisons testées et le résultat de chaque test) :

   ```
   >>> genereCode()
   >>> IA()
   [1, 1, 1, 1] (2, 0)
   [1, 1, 2, 2] (0, 3)
   ```

`[2, 3, 1, 1] (2, 1)`

`[4, 2, 1, 1] (3, 0)`

`[5, 2, 1, 1] (4, 0)`

`([5, 2, 1, 1], 5)`

Indications : on créera deux listes pour garder trace de toutes les combinaisons proposées et des réponses apportées. Dans l'exemple ci-dessus, au bout de deux itérations, la première liste sera `[[1, 1, 1, 1], [1, 1, 2, 2]]` et la seconde `[(2, 0), (0, 3)]`.

Au moment de tester une nouvelle combinaison `C`, non encore testée, il faut qu'avec testeCombinaison et les arguments `[1, 1, 1, 1]` et `C`, la réponse soit $(2, 0)$, et qu'avec les arguments `[1, 1, 2, 2]` et `C`, la réponse soit $(0, 3)$. Si ce n'est pas le cas (pour l'un ou pour l'autre) alors inutile de proposer `C` et on passe à la combinaison suivante dans l'ordre lexicographique.

5. En combien de coups l'algorithme trouve-t-il en moyenne le code secret ? (On fera des statistiques sur un nombre conséquent de tirages aléatoires, de l'ordre du millier, voire on testera tous les codes secrets possibles)

6. Ce nombre moyen est-il amélioré si on décide de partir d'une autre combinaison de départ que $[1, 1, 1, 1]$? (Essayer, bien sûr...)

1 Annexe : quelques rappels sur les listes

Souvenez-vous que si, `L` est une liste, l'instruction `K = L` attribue un nouveau nom de variable à la même liste, et qu'alors toute modification de la liste que référence `K` conduira à modifier également la liste que référence `L` (c'est la même !) :

```
>>> L = [1, 2]
>>> K = L
>>> K.append(3)
>>> L
    [1, 2, 3]
>>> L[1] = 0
>>> K
    [1, 0, 3]
```

De ce fait, pour copier le contenu d'une liste et créer une nouvelle liste, dont les changements ne seront pas répercutés sur la liste initiale, il faut faire une copie de la première dans la seconde. Pour ce faire, on peut faire appel à la fonction `copy` du module `copy`, ou utiliser une tranche (slice en anglais) :

```
>>> L = [1, 2]
>>> K = L[:] # ceci crée une nouvelle liste formée des mêmes éléments que L
>>> K.append(3); L[0] = 0
>>> L, K
    ([0, 2], [1, 2, 3])
```

Autre possibilité : `K = copy(L)` si au préalable on a importé cette fonction par `from copy import copy`.

2 Explication détaillée de l'algorithme envisagé sur un autre exemple

Je détaille l'algorithme décrit dans l'énoncé sur un exemple. On suppose ici que le code secret est `[4, 3, 2, 1]` et que la première combinaison testée est `[1, 1, 1, 1]`. Alors la valeur de retour de testeCombinaison pour cette combinaison est `(1, 0)` et on garde trace des deux dans deux listes, l'une est ainsi `[[1, 1, 1, 1]]` et l'autre `[(1, 0)]`.

La combinaison suivante dans l'ordre lexicographique est `[1, 1, 1, 2]`, mais la valeur de retour de `testeCombinaison([1, 1, 1, 1], [1, 1, 1, 2])` est `(3, 0)` et non `(1, 0)` si bien qu'on sait que `[1, 1, 1, 2]` ne saurait être le code secret recherché. On passe donc à la combinaison suivante, et ainsi de suite, jusqu'à parvenir à `[1, 2, 2, 2]` qui elle semble possible.

On la confronte alors au code secret, on obtient `(1, 1)`, et on rajoute cette nouvelle combinaison et sa réponse aux listes qui gardent trace de toutes les tentatives.

(Les deux listes en question sont à ce moment `[[1, 1, 1, 1], [1, 2, 2, 2]]` et `[(1, 0), (1, 1)]`.)

On passe à `[1, 2, 2, 3]` qui est incompatible (avec la seconde tentative), jusqu'à la prochaine combinaison envisageable compte tenu des tentatives précédentes, laquelle est `[3, 1, 2, 3]`.

Puis seront testées, dans l'ordre : `[3, 2, 1, 4]`, `[4, 1, 3, 2]` et enfin `[4, 3, 2, 1]`.

Attention à une erreur classique : si vous avez fait le choix dans `suivant` de modifier la liste L fournie en argument, alors pensez, lorsque vous ajoutez la combinaison testée à la liste des combinaisons testées jusque-là, à en placer une copie sur la liste, sinon l'objet que vous venez d'ajouter à votre liste va changer de valeur et bien sûr votre recherche échouera...