Programmation I
Projet - Jeu de lettres (v2)

Présentation

On s'intéresse à un jeu de lettres décrit par les règles suivantes :

- 1 - On fixe un dictionnaire contenant un ensemble de mots "acceptés".
- 2 - On joue avec un ensemble de lettres, qui est vide au début du jeu.
- 3 - Chaque joueur doit choisir une lettre à tour de rôle, qui est ajoutée à l'ensemble. Le choix d'une lettre compte comme un tour.
- 3A - A chaque tour, s'il existe un mot du dictionnaire contenant exactement les mêmes lettres que celles de l'ensemble, le joueur qui a introduit la dernière lettre a perdu (c'est lui qui a "complété" l'anagramme).
- 3B - A chaque tour, s'il n'existe aucun mot dans le dictionnaire contenant (au moins) toutes les lettres de l'ensemble, le joueur qui a introduit la dernière lettre a perdu (c'est lui qui a rendu la combinaison de lettres "impossible").

Le but du jeu est évidemment de ne pas être le joueur perdant. L'ordre des lettres ne compte absolument pas, il faut donc choisir sa lettre pour former un mot plus long dont toutes les lettres sont contenues dans un mot du dictionnaire sans pour autant former l'anagramme d'un mot complet, comme illustré dans l'exemple ci-dessous :

Mise en place : On fixe un dictionnaire minimaliste contenant juste les mots "test" et "set". Deux joueurs participent à la partie.

Déroulement : Tous les mots du dictionnaire étant formés à partir des lettres 't', 'e' et 's', choisir une autre lettre que ces trois là fait automatiquement perdre la partie (règle 3B ; par exemple, il n'existe aucun mot du dictionnaire qui contienne la lettre 'a').
On suppose que le premier joueur choisit alors la lettre 't'. Le second choisit la lettre 's'. L'ensemble des lettres du jeu est alors "ts".

C'est à nouveau le tour du 1er joueur. S'il choisit la lettre 'e', l'ensemble devient "tse", ce qui permet de former le mot "set", auquel cas il perd la partie (3A). S'il choisit la lettre 's', l'ensemble devient "tss", or aucun mot du dictionnaire ne contient deux fois la lettre 's'. Il perd donc également la partie (3B).
Son seul choix intelligent est donc la lettre 't', formant la séquence "tst". Le second joueur doit alors jouer, et se retrouve coincé : il peut jouer, mais perdra automatiquement la partie à son tour. (S'il choisit 'e', il perd à cause de la règle 3A ; dans tous les autres cas il perd à cause de la règle 3B.)

Le jeu est évidemment peu intéressant dans le cas d'un dictionnaire aussi petit (le premier joueur gagnera toujours s'il ne fait pas d'erreur), mais en choisissant par exemple un dictionnaire complet dans une langue quelconque, le jeu devient beaucoup plus difficile et intéressant.

Ce jeu présente quelques similarités avec le jeu de lettres conçu en Licence SDL, à ceci près que l'ordre des lettres ne compte désormais plus, ce qui change complètement la combinatoire correspondante (et donc l'algorithme associé).

Objectifs

L'objectif du projet est de concevoir un programme qui permet de jouer à ce jeu. Idéalement, le programme devra fonctionner en ligne de commande, être interactif (proposer au joueur de lui expliquer les règles, lancer une partie, lire les lettres choisies par les joueurs et afficher le résultat à la fin), et bien évidemment s'assurer du respect des règles (le programme devra vérifier à chaque tour si le joueur a perdu, et arrêter la partie si c'est le cas).

Au delà de cette partie essentielle, le but final est d'essayer de concevoir un programme avec quelques caractéristiques bonus, par exemple :

- Aussi rapide que possible - le délai entre chaque tour ne devrait pas dépasser la minute même pour de très grands dictionnaires (p.e. `bdlex`).
- Qui soit clair et lisible - chaque fonction/variable doit avoir un rôle clair et compréhensible, ou au moins un commentaire décrivant son utilité.
- Agréable à utiliser - capable de gérer de façon gracieuse une éventuelle faute de frappe (si, par exemple, le joueur entre plusieurs lettres ou bien aucune) sans provoquer de bug ni mettre fin brutalement à la partie.
- Capable de gérer des règles alternatives - par exemple, si l'on lance une partie en précisant un nombre initial de joueurs, éliminer le perdant et repartir de zéro jusqu'à ce qu'il n'y aie plus qu'un seul joueur en lice ; ou bien encore de préciser en début de jeu si on veut ou non tenir compte des accents.

- Possédant une IA - vous pourriez même envisager d'ajouter dans votre programme une option pour permettre au joueur de jouer contre l'ordinateur, avec éventuellement un niveau de difficulté paramétrable (par exemple, l'IA ne connaitrait qu'une partie du dictionnaire, ou pourrait faire des erreurs de temps en temps).

Modalités

Le projet de programmation doit être réalisé en petit groupe (pas plus de 3 personnes).

Le travail effectué devra être envoyé par mail à l'adresse `jkirman@labri.fr`, avant le 16 décembre 2012 **dernier délai** (un accusé de réception sera envoyé systématiquement pour confirmation).

Le mail devra contenir en pièce jointe un fichier source Python (extension `.py`) ainsi qu'un fichier texte co-rédigé (si possible au format PDF) détaillant le contenu du programme et la contribution de chacun.

Le programme devra au minimum couvrir la partie centrale du projet (permettre à deux joueurs humains de s'affronter en utilisant les mots d'un fichier dictionnaire fourni). Un programme lisible, efficace et/ou traitant les questions les plus avancées, sera noté plus favorablement. (La clarté et l'exhaustivité du rapport joint pourront également rentrer en ligne de compte.)

Par ailleurs (est-il besoin de le préciser ?), le fait de poser des questions ne vous coûtera pas de points, n'hésitez donc pas à profiter du temps de TD ou à m'envoyer un mail si vous bloquez sur un point particulier.

Questions

Répondre aux questions suivantes ne fait pas directement partie du projet ; toutefois, ces questions sont conçues pour vous guider dans l'implémentation de plusieurs points, qui vont s'avérer utiles pour le développement du projet : ouverture et lecture de fichiers, interaction avec l'utilisateur, estimations de complexité pour un problème algorithmique, utilisation du module `Counter`, etc...

Les accès disque en Python

Plutôt que de construire un dictionnaire "à la main" au début de chaque partie en entrant les mots un par un, il serait plus simple de disposer d'un (ou plusieurs) fichiers réutilisables contenant tous les mots dont on veut se servir, et de demander au programme de puiser les mots d'un fichier.

Lire des fichiers sur le disque dur en Python se fait en plusieurs étapes : il faut d'abord "ouvrir" le fichier visé, puis on peut accéder à son contenu ; une fois obtenue l'information désirée, il n'y a plus qu'à refermer le fichier (pour éviter de consommer des ressources systèmes inutilement).

1. Créez un fichier dans le répertoire courant, nommé "`test.txt`". Entrez-y une série de mots quelconques, sans espaces, en revenant à la ligne entre chaque mot. Sauvegardez le fichier.

2. Lancez l'interpréteur Python, puis entrez l'instruction suivante :
   `file=open("test.txt")`
   Sauf faute de frappe (ou changement de répertoire), l'instruction ne doit pas provoquer d'erreur. Affichez le contenu de la variable `file`.

3. La variable `file` contient un objet d'un type particulier, qui possède de nombreuses sous-fonctions (appelées méthodes en programmation Orientée Objet). Entre autres, la méthode `file.readline()` permet de lire une ligne complète dans le fichier et de passer à la suivante. Le contenu de la ligne est retourné sous la forme d'une chaîne de caractères.
   Stockez en utilisant la méthode `file.readline()` le contenu de la première ligne du fichier dans une variable.

4. Par hypothèse, il est difficile de savoir combien de lignes contient un fichier sans l'avoir lu en entier. En revanche, si toutes les lignes contiennent chacune un mot jusqu'à la fin du fichier, chaque appel à `file.readline()` retournera un mot, sauf le dernier (une fois la dernière ligne atteinte) qui retournera une chaîne vide, de longueur 0.
   Ecrivez une boucle qui affiche un par un tous les mots contenus dans le fichier. La boucle doit répéter son bloc d'instructions tant que le mot retourné par `file.readline()` est différent de "".

5. Lorsque vous avez fini de consulter un fichier (ou si vous voulez reprendre depuis le début), vous devez appeler la méthode `file.close()`, qui "ferme" le fichier. La variable `file` devient alors inutile, à moins que vous n'ouvriez à nouveau le fichier (avec la fonction `open`).

Vous pouvez maintenant utiliser un fichier texte simple (au format ASCII ou UTF-8) comme dictionnaire, en incluant dans votre programme une fonction qui ouvre un fichier et lit chacune de ses lignes en ajoutant les résultats dans une liste.

Interactivité

Vous savez déjà afficher des messages à l'écran dans le cadre d'un programme (grâce à la fonction `print`), mais le projet nécessitera également de recevoir des informations tapées au clavier par l'utilisateur. En Python, c'est l'objet de la fonction `input`.

1. Démarrez l'interpréteur Python, et entrez simplement l'instruction `input()`. Le programme se met alors en pause, et attend que l'utilisateur (vous) entre une séquence de caractères, terminée par la touche Enter .

2. Reprenez l'instruction précédente, en enregistrant cette fois la valeur retournée dans une variable `m`. Tapez une séquence de caractères, puis entrée, et examinez ensuite le contenu de la variable `m`.

3. La fonction `input` peut (optionellement) prendre en argument une chaîne de caractères, qui est considérée comme la question posée à l'utilisateur. Fermez l'interpréteur Python, et créez un nouveau programme, qui enregistre dans une variable `m` le résultat de l'appel à la fonction `input("Entrez un nombre au hasard :")`, puis affiche le contenu de cette variable.

Vous disposez maintenant d'une instruction qui vous permet de poser une question à l'utilisateur (comme : "Quelle lettre voulez-vous ajouter à la séquence ?") et d'obtenir sa réponse. Notez que rien n'oblige l'utilisateur à entrer exactement un caractère avant d'appuyer sur Enter . Il pourrait être judicieux de vérifier que l'utilisateur a entré une valeur correcte avant de continuer (et, le cas échéant, de reposer la question jusqu'à ce que ce soit bon).

Remarquez également que la variable `m` ne contient pas vraiment un nombre, mais une séquence de caractères (même si ces caractères sont des chiffres). Si vous essayez d'effectuer une addition avec la variable `m`, vous obtiendrez une erreur. Le projet nécessite surtout de manipuler des lettres et des mots, mais si vous avez besoin de passer de la chaîne de caractères "123" au nombre 123, vous pouvez utiliser la fonction `int(...)` qui prend en argument une chaîne de caractères contenant des chiffres et retourne le nombre correspondant. (Encore une fois, attention à ce que l'utilisateur n'entre pas de lettres quand vous attendez un nombre...)

Questions de combinatoire...

Pour détecter si un joueur a perdu, il faut vérifier qu'il n'existe pas d'anagramme de la séquence de lettres obtenue dans le dictionnaire (règle 3A). Imaginons un dictionnaire (très modeste !) de 10.000 mots, et une séquence obtenue après n tours de jeu sans perdant. La séquence contient alors n lettres.

1. On suppose que toutes les lettres données sont différentes. Si $n = 1$, il n'y a qu'un seul mot possible. Si $n = 2$, il y en a 2. A partir de $n = 3$, la question commence à devenir intéressante. Pouvez vous lister et compter le nombre de combinaisons possibles avec 3 lettres différentes ? Et 4 lettres différentes ?

2. Commencez à lister (au moins mentalement) les combinaisons pouvant être formées avec seulement 5 lettres différentes. Sans aller jusqu'au bout, sachez simplement qu'il y en a 5 fois plus que de combinaisons à 4 lettres. D'une manière générale, le nombre *permutations* d'un ensemble de lettres se calcule en mathématiques au moyen d'une formule très simple : si on dispose de n objets différents à ordonner, le nombre de combinaisons possibles est $1 \times 2 \times 3 \times \ldots \times (n-1) \times n$.[1] Calculez (ou faites calculer à l'ordinateur !) le nombre de permutations possibles pour des ensembles de 5 lettres, puis 6 lettres, 7 lettres, etc...

Certaines répétitions peuvent en théorie être évitées (si la séquence avec laquelle on joue contient plusieurs fois la même lettre), mais sans entrer dans les détails, le gain est mineur. Un rapide calcul montre qu'en étant TRES optimiste (et en utilisant un ordinateur extrêmement puissant), calculer si un joueur a perdu au terme d'une partie un peu longue peut prendre trois jours.[2]

Clairement, construire tous les anagrammes possibles pour les comparer avec les mots du dictionnaire n'est pas la bonne solution. Une méthode plus efficace serait de rayer du dictionnaire tous les mots qu'il est devenu impossible de former avec les lettres mises en jeu, et de réduire ainsi la taille de celui-ci. Pour savoir si un joueur a perdu, il suffirait ensuite d'examiner les mots du dictionnaire un par un, et de compter le nombre de fois où chaque lettre apparait. Si le résultat de ce comptage est exactement le même pour un des mots du dictionnaire et pour la séquence de lettre en jeu, alors ce sont des anagrammes (et le joueur précédent à donc perdu, par la règle 3A). Par ailleurs, si on raye le dernier mot du dictionnaire, alors le joueur précédent a également perdu (règle 3B).

Pour effectuer le comptage, vous pouvez soit écrire vos propres fonctions (ce qui nécessite d'écrire des fonctions correctes pour comparer le nombre de lettres dans deux mots), soit utiliser le module Python `Counter` (ce qui nécessite

[1] Vous pouvez consulter rapidement l'article Wikipédia consacré aux permutations en mathématiques pour plus d'infos.

[2] Un calcul un peu moins optimiste montre que le dernier tour d'une partie tendue prendra au bas mot près de 200.000 ans sur un processeur moderne...

d'apprendre à se servir des fonctions Python correspondantes). Vous êtes libre de procéder comme vous l'entendez (ou même de trouver une solution radicalement différente), mais si vous optez pour la deuxième solution, la documentation est disponible à cette URL (section 8.3.2) :
`http://docs.python.org/3.3/library/collections.html`

Chronométrage (optionnel)

Supposons que vous ayez écrit une fonction `fonc1` qui effectue une tâche, et une autre fonction `fonc2` qui effectue le même travail, mais d'une manière différente. Vous aimeriez savoir laquelle est la plus rapide. Deux fonctions peuvent être utilisées à cet effet, dont les définitions peuvent être obtenues avec l'instruction `from time import time, clock`

Après avec importé ces fonctions, vous pouvez utiliser d'une part la fonction `time()`. Celle-ci retourne l'heure exacte en temps UNIX, en secondes. Si vous l'appelez juste avant et juste après l'appel à `fonc1`, et idem pour `fonc2` et que vous comparez la différence entre les deux premiers et les deux derniers résultats, vous saurez combien de temps a mis chaque fonction à s'exécuter.

Cette fonction a l'avantage de mesurer exactement le temps qu'il s'est réellement écoulé. L'inconvénient est que si l'ordinateur doit travailler sur autre chose en même temps (ou attendre que l'utilisateur tape une lettre, par exemple), le résultat sera faussé.

D'autre part, vous disposez également la fonction `clock()`. Sans entrer dans les détails, celle-ci mesure (toujours en secondes) le temps que le processeur a passé à effectuer des calculs pour votre programme depuis le démarrage de celui-ci. Le temps chronométré avec `clock` exclut donc le temps passé à attendre l'utilisateur, mais a l'inconvénient d'ignorer du même coup le temps passé par le processeur à attendre des données lues sur le disque dur. (Ce qui pose également un problème, si l'une de vos fonction est plus lente parce qu'elle a besoin d'aller chercher plus d'information sur le disque dur.)

Il n'existe pas de solution miracle, mais une utilisation judicieuse de ces deux fonctions vous apportera des informations utiles sur le temps d'exécution des diverses fonctions de votre programme (et peut vous aider à améliorer sa rapidité).