UE Projet informatique - L1 S2 Option

Projet : le jeu du pendu

Printemps 2023

Table des matières

1 Consignes et organisation du projet

Le projet est à faire en binôme (ou seul) et comprend trois volets techniques :

1. un programme à réaliser en Python, contenant toutes les fonctions demandées dans la Partie guidée, et une quantité satisfaisante d'extensions (voire Partie Extensions pour plus de détails). Vous inclurez obligatoirement un programme principal dont le lancement fera la démonstration des fonctionnalités que vous avez implémentées. Vous pouvez mettre tout votre code dans le même fichier `.py`, ou bien apprendre à diviser votre code source en plusieurs fichiers [1]. La notation de la partie Python prendra en compte les éléments suivants :
   — Votre programme contient toutes les fonctions demandées (hors Partie Extensions)
   — Vos fonctions respectent les spécifications demandées de manière exacte, pour la Partie guidée
   — Les algorithmes et l'implémentation sont bien choisis (pas de solutions inutilement compliquées, même si le résultat est le bon).
   — La quantité et la qualité des extensions réalisées.
   — Votre code est propre, bien lisible, élégant, correctement indenté (présentation, noms de variable explicites, commentaires utiles, ...)
   — Votre programme principal permet, au lancement, d'avoir un bon aperçu des fonctionnalités implémentées et est agréable à utiliser (faites des choix pertinents).
2. un rapport à rédiger en Latex, qui devra :
   — contenir les noms des membres du groupe ainsi que le numéro du binôme choisi sur Moodle
   — contenir un lien hypertexte vers le dépôt gitlab du groupe (cf instructions au point 3)
   — expliquer votre organisation du travail tout-au-long du semestre

1. Ce n'était pas au programme du Semestre 1, mais il est assez facile de trouver un petit tutoriel sur la création de *modules* locaux.

- expliquer le/les choix que vous avez dû faire pour implémenter les fonctions `demander_proposition`, `dico_frequence` et `fabrique_liste_frequence` (et éventuellement d'autres choix de la Partie guidée que vous trouvez pertinents)
- expliquer les extensions que vous avez choisies de réaliser : en quoi elles consistent, comment l'on s'en sert, quels choix d'implémentation vous avez dû faire
- expliquer ce que contient votre programme principal et comment s'en servir
- être correctement structuré
- comporter une bibliographie si vous avez consulté des ressources externes pendant l'élaboration de votre projet
- utiliser les commandes Latex adéquates (tables des matières, sections, référence et label, listes, ...)

3. un dépôt sur `gitlab.isima.fr` [2] qui contiendra les différentes versions de votre projet (programmation en Python et rapport en Latex) tout au long du semestre. Veillez à :
   - ce que les différents membres du binômes contribuent significativement à alimenter le dépôt
   - faire des `commit` de manière régulière, dès lors qu'une fonctionnalité est ajoutée, avec des messages de commit pertinents
   - ce que votre chargé de TP ait le rôle de *Reporter* sur votre dépôt pour pouvoir cloner votre dépôt lorsqu'il/elle vous évaluera.
   - ce que votre dépôt ne contienne pas de fichier auxiliaire inutile (cf utilisation d'un fichier `.gitignore` vue au Semestre 1)

En plus des volets techniques, il y aura une petite soutenance orale **le 4 ou le 5 mai 2023**. Cela nous permettra notamment de vous poser des questions sur certaines parties de votre projet, pour vérifier si vous les maîtrisez bien, et d'estimer si le travail a été équitablement réparti entre les deux membres du binôme. Plus de détails sur cet oral vous seront donnés ultérieurement.

Dans un réel projet informatique commandé par un client et qui s'étale sur plusieurs semaines, il est presque systématique que le client change l'expression de son besoin alors même que vous avez déjà commencé à réaliser la solution. Avoir cela en tête influe sur la façon de concevoir la solution, pour qu'elle soit raisonnablement flexible aux changements. Pour reproduire une situation proche du réel, quelques consignes supplémentaires [3] vous seront données au fur et à mesure du semestre sur le fil de discussion *Consignes complémentaires Projet* dans les Annonces Moodle de l'UE Projet. Le sujet final du projet sera donc constitué du présent document **et** des consignes qui figureront sur la page web ci-dessus.

Mise en binôme Allez vous inscrire en binôme sur l'activité Choix binôme pour le projet sur l'espace Moodle de l'UE Projet. Les deux membres d'un binôme doivent être dans le même groupe de TP.

Rendu Le rendu final de votre programme est attendu pour le **Dimanche 16 avril 2022 à 23h55 au plus tard**. À cette date-là, vous devez :
- avoir déposé sur Moodle votre rapport en PDF
- avoir donné les accès à votre chargé de TP à votre dépôt gitlab, dépôt qui devra contenir, comme indiqué plus haut, les fichiers sources de votre rapport ainsi que le(s) fichier(s) source(s) de votre programme Python, et les fichiers auxiliaires nécessaires à sa bonne exécution.

Attention, ce que vous rendez doit refléter votre propre travail : d'une part, les deux membres du binôme doivent travailler sur le projet ; d'autre part, il est hors de question de copier les réponses d'un autre binôme. Le projet que vous rendez va être long, parsemé de commentaires et d'extensions personnalisées : l'équipe pédagogique qui encadre ce cours a des années d'expérience pour détecter la fraude sur les projets rendus ! Par ailleurs, la soutenance orale nous permettra de nous rendre compte si aucun des deux membres du binôme ne maîtrise ce qui est écrit dans le rendu.

Chaque membre doit contribuer de manière significative sur les trois aspects du projets, à savoir le code Python, le rapport LaTeX et les commits. Par exemple, il est hors de question qu'un membre d'un binôme travaille uniquement sur le code pendant que l'autre rédige le rapport. Chaque membre du binôme doit toucher à tout.

2. Voir Annexe A pour les instructions de configuration.
3. Rassurez-vous, les changements de consigne resteront raisonnables.

Vous pouvez bien sûr poser des questions à votre enseignant durant l'élaboration de votre projet. Nous aurons une séance hebdomadaire d'1h30 de TP pour que vous puissiez avancer sur votre projet et poser des questions. Il sera nécessaire de compléter par du travail personnel en-dehors de ces séances.

2 Le thème du projet : le jeu du pendu

Dans ce projet, nous souhaitons programmer un jeu bien connu des enfants : le pendu. On commence ici par donner les règles de ce jeu.

Généralement, ce jeu se joue à deux joueurs (que nous nommerons A et B), avec un crayon et une feuille de papier. L'un des deux joueurs doit deviner un mot choisi par l'autre en proposant des lettres. Il n'a qu'un nombre limité d'erreurs possibles. Si l'on suppose que le joueur A devine le mot choisi par B, la partie se déroule ainsi :

- B pense à un mot et dessine une rangée de tirets, chacun correspondant à une lettre de ce mot, en laissant apparentes la première et la dernière lettre du mot
- A propose une lettre.
  - Si la lettre fait partie du mot, B écrit la lettre à sa place, au-dessus du tiret correspondant, à tous les endroits où la lettre est présente dans le mot.
  - Sinon B dessine le premier trait du pendu.
- Le jeu continue ainsi avec A qui propose des lettres jusqu'à arriver à une des deux situations suivantes :
  - A a trouvé toutes les lettres du mot. Dans ce cas, A gagne.
  - Le pendu est entièrement dessiné (c'est-à-dire si on a atteint le nombre maximum d'erreurs autorisé). Dans ce cas, B gagne.

3 Représentation informatique de ce jeu

Dans ce projet, on va programmer le jeu du pendu. L'ordinateur jouera le rôle du joueur B, c'est donc lui qui fera deviner le mot. On programmera deux types de parties :

- une partie standard où ce sera un joueur (humain) qui joue contre l'ordinateur et propose des lettres. Dans ce cas on fixera un nombre d'erreurs à partir duquel on considérera que le joueur a perdu (correspond au nombre de morceaux à dessiner pour dessiner entièrement le pendu)
- une partie automatique où ce sera l'ordinateur qui tentera de deviner lui-même le mot (en jouant contre lui-même et en faisant comme s'il ne connaissait pas le mot). Dans ce cas, l'ordinateur jouera toujours jusqu'à gagner, et on regardera combien d'erreurs ont été faites avant de gagner

Le projet consistera en l'implémentation :

- des fonctions élémentaires nécessaires pour jouer une partie (initialisation et affichage textuel - section 4.1)
- des fonctions nécessaires pour permettre à l'utilisateur humain de jouer contre l'ordinateur (section 4.2)
- des fonctions nécessaires pour permettre à l'ordinateur de jouer contre lui-même (section 4.3)
- de fonctionnalités complémentaires pour aller plus loin (section 5)

Un des objectifs de ce projet va être de comparer différentes stratégies de l'ordinateur dans son choix de proposition de lettres, en estimant les probabilités de victoire avec trois stratégies distinctes sur de nombreuses simulations de parties.

Dans un premier temps, l'interface sera uniquement textuelle. Une interface graphique pourra être programmée la fin du projet si vous avez tout fini à l'aide du module `turtle` (facultatif).

Pour modéliser le jeu de pendu, à tout moment dans la partie, l'ordinateur aura en mémoire le mot à deviner `mot_myst` et le mot partiellement découvert. `mot_myst` n'est pas amené à être modifié en cours de partie, on utilisera donc une chaîne de caractères. Par contre, pour pouvoir modifier au fur et à mesure le mot partiellement découvert, on le stockera sous la forme d'une liste de caractères qu'on appelera `lmot_decouv`.

Par exemple, si le mot à deviner, `mot_myst`, est "BONJOUR" et qu'on choisit le "-" pour matérialiser une lettre non encore trouvée, `lmot_decouv` sera initialisée à la valeur ["B","-","-","-","-","-","R"]. Si ensuite le joueur propose la lettre "O", `lmot_decouv` sera mise à jour pour valoir ["B","O","-","-","O","-","R"].

On va maintenant pouvoir écrire les fonctions permettant de jouer au jeu du pendu.

4 Partie guidée

Le projet a été découpé en de nombreuses fonctions, on demande de respecter cette structure. Vous avez le droit d'écrire des fonctions supplémentaires non explicitement demandées dans cet énoncé. Autant que possible, **testez toutes vos fonctions au fur et à mesure que vous les écrivez**. Pensez notamment à tester les fonctions avec des arguments pathologiques, qui sont souvent source de bugs : liste vide, liste contenant plusieurs fois le même élement,.... Avant de coder une fonction, vous devez être convaincus que toutes les fonctions précédentes fonctionnent.

Votre programme devra être clair et commenté. **Il est obligatoire de documenter toutes les fonctions** (y compris celles non demandées), en indiquant, *a minima*, les arguments pris par la fonction, ses effets de bord et ce qu'elle renvoie. Cela vous aidera à vous y retrouver dans votre projet, et même à structurer vos idées.

Votre programme doit être régulièrement mis à jour dans votre repository GitLab. **Il est obligatoire de faire au moins un `commit` par fonction**. Rappel : les deux membres d'un binôme doivent chacun "commiter" de manière significative.

Attention : pour toutes les fonctions demandées dans ce projet, quand il y a des listes ou des dictionnaires en argument, ces listes et dictionnaires ne doivent pas être modifiées par la fonction, sauf si c'est explicitement demandé. N'hésitez pas à faire une copie de la liste si nécessaire.

4.1 Initialisation du jeu et affichage

4.1.1 Fonction `importer_mots`

Ecrire une fonction `importer_mots` qui prend en argument un nom de fichier `nom_fichier` et qui renvoie la liste des mots contenus dans ce fichier, **en se limitant aux mots d'au moins trois lettres.** Dans le fichier passé en argument, il y aura un mot par ligne, pas de caractères accentués, pas de cédille, pas de traits d'union. Les mots pourront être écrits en minuscules ou majuscules (ou un mélange des deux). Dans la liste retournée, vous devrez avoir converti tous les mots en majuscules et retiré les éventuels caractères blancs (espaces, retours à la ligne, ...) inutiles en début de fin de ligne. Un fichier de mots `mots2.txt` est mis à votre disposition sur Moodle pour tester la fonction. Pour manipuler les chaînes de caractères (notamment le passage minuscules/majuscules et pour l'utilisation de `strip`), il peut être utile de se référer au cours d'introduction à Python au S1 ou à la documentation en ligne de Python : https://docs.python.org/3/library/stdtypes.html#textseq

4.1.2 Fonction `choisir_mot_alea`

Ecrire une fonction `choisir_mot_alea` qui renvoie un mot choisi aléatoirement dans une liste de mots passée en argument. Pour cela, vous importerez le module `random` en écrivant `import random` au début de votre fichier (avant les fonctions), puis vous utiliserez la fonction `randint`. Vous irez lire la documentation correspondant à cette fonction pour savoir l'utiliser : https://docs.python.org/fr/3/library/random.html. N'hésitez pas à utiliser la fonction Rechercher (Ctrl+F) de votre navigateur pour tomber directement sur la partie de la page concernant `randint`.

4.1.3 Fonction `initialiser_mot_part_decouv`

Ecrire une fonction `initialiser_mot_part_decouv` qui prend en argument le mot à deviner `mot_myst` (au moins 3 lettres) et un caractère optionnel `car_subst`, et qui renvoie la liste des lettres du mot mystère dans laquelle on a remplacé toutes les lettres sauf la première et la dernière par le caractère `car_subst`. L'argument `car_subst` est optionnel, sa valeur par défaut est `"-"`.

Exemples :
`initialiser_mot_part_decouv("BONJOUR")` renvoie ["B","-","-","-","-","-","R"]
`initialiser_mot_part_decouv("BONJOUR",car_subst="*")` renvoie ["B","*","*","*","*","*","R"]

4.1.4 Fonction `afficher_potence_texte`

Pour indiquer au joueur l'avancement de la partie, nous allons afficher un message pour indiquer au joueur s'il s'approche du nombre limite d'erreurs à ne pas dépasser pour pouvoir gagner. Le message se dévoilera progressivement, au fur et à mesure que le nombre d'erreurs augmentera. Ainsi, au lieu de perdre quand le pendu est entièrement dessiné, on perdra quand le mot "PERDU" sera entièrement écrit. Comme ce mot ne contient que 5 lettres, et qu'on peut souhaiter autoriser davantage d'erreurs, on le complètera avec autant de points d'exclamation "!" que nécessaire plus un "!" final supplémentaire qui s'affiche toujours (pour mieux visualiser combien d'erreurs sont encore autorisées). La fonction `afficher_potence_texte` prendra en argument l'entier `nb_err` correspondant au nombre d'erreurs faites et l'entier `nb_err_max` correspondant au nombre d'erreurs à atteindre pour perdre la partie (on suppose que `nb_err_max` ≥ 5).

Par exemple, si on considère qu'on perd au bout de 8 erreurs, on obtiendra les affichages suivants selon le nombre d'erreurs :

`nb_err`	Affichage obtenu
0	`--------!`
1	`P-------!`
2	`PE------!`
3	`PER-----!`
4	`PERD----!`

`nb_err`	Affichage obtenu
5	`PERDU---!`
6	`PERDU!--!`
7	`PERDU!!-!`
8	`PERDU!!!!`

Ecrire la fonction `afficher_potence_texte`, prenant en argument `nb_err` et `nb_err_max`, et affichant la chaines de caractères (de longueur `nb_err_max`) correspondante indiquée par le tableau ci-dessus.

4.2 Partie pour un joueur humain

On va maintenant écrire les fonctions qui vont permettre de jouer une partie de pendu.

4.2.1 Fonction `demander_proposition`

Ecrire une fonction `demander_proposition` qui prend en argument la liste de toutes les lettres déjà proposées dans les tours précédents de la partie en cours `deja_dit`. Cette fonction :

- demande à l'utilisateur de saisir une lettre
- vérifie que l'utilisateur a bien saisi une lettre (un seul caractère, entre "A" et "Z") et que la lettre n'est pas dans la liste `deja_dit`
- redemande une lettre tant que la condition précédente n'est pas vérifiée
- renvoie la lettre majuscule saisie par l'utilisateur (une fois que celle-ci vérifie la condition)

Attention : l'utilisateur doit pouvoir saisir des lettres majuscules ou minuscules sans que cela pose un problème. Ainsi, si le joueur propose "m" et que cette lettre ("m" ou "M") n'a pas déjà été proposée, la fonction renverra "M". Par contre, les caractères accentués ne doivent pas être acceptés. Une fois que vous avez vérifié que votre chaîne de caractères contenait un unique caractère, vous pouvez tester s'il s'agit bien d'une lettre majuscule en comparant `s >= "A"` et `s <= "Z"` (et de manière similaire pour les minuscules).

Remarque pour les curieux : la comparaison `s >= "A"` vérifie si l'encodage du caractère dans `s` est un numéro supérieur à l'encodage de la lettre "A", qui est le code ASCII numéro 65. Pour afficher le code ASCII d'un caractère `s`, on peut faire `ord(s)` ou consulter une table ASCII sur Internet, par exemple : http://www.table-ascii.com

4.2.2 Fonction `decouvrir_lettre`

Ecrire une fonction `decouvrir_lettre` qui prend en arguments la lettre proposée par le joueur à ce moment de la partie, que l'on appellera `lettre`, le mot à deviner `mot_myst` et le mot partiellement découvert `lmot_decouv` (liste de caractères). Cette fonction modifie par effet de bord la liste `lmot_decouv` qui lui est donnée en argument, en révélant la lettre `lettre` partout où elle est présente. Par ailleurs, la fonction renvoie le booléen vrai si au moins une nouvelle lettre a été découverte et faux sinon. Ici, on travaille forcément avec des majuscules.

Exemples :

- `decouvrir_lettre("O","BONJOUR",["B","-","-","-","-","-","R"])` renvoie vrai et transforme la liste passée en argument en `["B","O","-","-","O","-","R"]`
- `decouvrir_lettre("R","BONJOUR",["B","O","-","-","O","-","R"])` renvoie faux et laisse la liste en argument inchangée
- `decouvrir_lettre("R","ARROSOIR",["A","-","-","-","S","-","I","R"])` renvoie vrai et transforme la liste passée en argument en `["A","R","R","-","S","-","I","R"]`

4.2.3 Fonction `partie_humain`

La fonction `partie_humain` permet à une personne de jouer une partie de pendu. La fonction prend deux arguments obligatoires : le mot à deviner `mot_myst` et le nombre d'erreurs à partir duquel la partie est perdue `nb_err_max`. On suppose que `nb_err_max` est toujours au moins égal à 5. On ajoute également un argument optionnel, `car_subst`, le caractère par lequel on veut substituer les lettres non encore découvertes dans l'affichage (valeur par défaut : `"-"`). La fonction renvoie le booléen vrai si la partie a été gagnée et faux sinon. Cette fonction doit :

- afficher le mot partiellement découvert,
- afficher la liste des lettres déjà proposées pour cette partie (seulement s'il y a déjà eu au moins une proposition),
- demander au joueur sa proposition de lettre,
- si la lettre proposée est présente, afficher "Lettre presente" sinon afficher le nombre d'erreurs déjà commises,
- afficher enfin la potence au format texte, mais seulement si au moins une erreur a déjà été commise,
- recommencer les étapes ci-dessus jusqu'à ce que la partie soit terminée,
- afficher un message clair pour dire si la partie a été gagnée ou pas et afficher le mot qui était à deviner si le joueur ne l'a pas trouvé,
- retourner le résultat.

Pour cette fonction, il faudra bien veiller à utiliser le plus possible les fonctions déjà écrites. Il faudra aussi penser à gérer une liste `deja_dit` des lettres proposées (penser à l'initialiser et à la mettre à jour après chaque proposition).

Attention :

— Sentez vous libre de coder des fonctions auxiliaires si vous en avez besoin. Cela peut diminuer le nombre de lignes de code de grosses fonctions comme `partie_humain`.

— On souhaite un affichage esthétique du mot partiellement découvert. Si par exemple la liste des lettres du mot partiellement découvert est `["A","R","R","-","S","-","I","R"]`, on affichera `ARR-S-IR`.

— Plus généralement, l'affichage lors d'une partie de pendu doit être agréable à lire, n'hésitez pas à rajouter des éléments supplémentaires (sauts de lignes,...) dans un but purement esthétique.

Ecrire la fonction `partie_humain`.

4.2.4 Fonction `partie_humain_alea`

La fonction `partie_humain_alea` va permettre d'appeler la fonction précédente avec un mot choisi au hasard parmi une liste de mots. Cette fonction aura à peu près les mêmes arguments que la fonction précédente, mais l'argument `mot_myst` sera remplacé par l'argument `nom_fichier` contenant le nom du fichier dans lequel choisir

aléatoirement le mot à faire deviner. Cette fonction consiste simplement en des appels de plusieurs fonctions écrites précédemment. Comme pour la fonction `partie_humain`, on renverra vrai si la partie a été gagné et faux sinon.

Ecrire la fonction `partie_humain_alea`.

4.3 Partie en mode automatique

Nous allons maintenant nous intéresser à des parties jouées automatiquement. L'objectif de cette partie est de créer des fonctions qui permettent de deviner un mot en utilisant des stratégies automatiques. Une stratégie consiste à définir un ordre de proposition pour les lettres. On évalue la performance d'une stratégie en considérant combien d'erreurs ont été faites avant de trouver le mot. On va dans un premier temps écrire des fonctions qui permettent de préparer ces différentes stratégies. Pour chaque stratégie, il faut fabriquer les listes de lettres, ordonnées dans l'ordre dans lequel les lettres seront proposées.

4.3.1 Fonction `fabrique_liste_alphabet`

La première stratégie sera la stratégie "alphabétique", elle consiste à proposer les lettres dans l'ordre des lettres de l'alphabet : d'abord "A", puis "B", puis "C" ...

Ecrire une fonction sans argument `fabrique_liste_alphabet` qui renvoie la liste des lettres majuscules dans l'ordre alphabétique.

Remarque : Une manière élégante de faire, sans avoir besoin de lister à la main tout l'alphabet, est de faire une boucle `for` sur les codes ASCII (= encodage des caractères) entre celui de `"A"` et celui de `"Z"`, et de générer le caractère correspondant grâce à `chr(code)`. *Exemple :* `chr(65)` vaut `"A"` (`chr` est l'opération inverse de `ord`. Ainsi, `chr(ord("B"))` vaut `"B"`).

4.3.2 Fonction `fabrique_liste_alea`

La deuxième stratégie sera celle de l'aléa. Mais plutôt que de choisir une lettre aléatoirement à chaque tour de jeu (et de vérifier si elle a déjà été donnée ou pas), l'ordinateur va créer en début de partie une liste, mélange des lettres de l'alphabet, qui va lui donner l'ordre de proposition des lettres pour l'ensemble de la partie. Cette liste aléatoire sera recréée à chaque nouvelle partie. Ecrire la fonction `fabrique_liste_alea` qui ne prend aucun argument et renvoie la liste des lettres majuscules dans un ordre aléatoire. On pourra utiliser la fonction `fabrique_liste_alphabet` et la fonction `shuffle` du module `random` qui prend en argument une liste, a pour effet de bord de la mélanger, et ne renvoie rien.

4.3.3 Fonction `fabrique_liste_frequence`

La troisième stratégie que nous allons étudier est plus complexe que les deux précédentes. L'ordinateur va utiliser une liste de mots pour comparer les fréquences des différentes lettres. Il proposera en premier la lettre qui sera la plus présente dans la liste de mots dans laquelle il a fait son apprentissage.

1. Ecrire la fonction `dico_frequence` qui prend en argument un nom de fichier `nom_fichier` et qui crée et renvoie un dictionnaire dont les clés sont les lettres majuscules et les valeurs correspondantes sont le nombre de fois où la clé apparaît dans le fichier.

   Exemple : si le fichier "mes_mots.txt" contient :

   ```
   BONJOUR
   PENDU
   INFORMATIQUE
   PYTHON
   ```

   `dico_frequence("mes_mots.txt")` renverra : `{'I': 2, 'E': 2, 'O': 4, 'N': 4, 'J': 1, 'A': 1, 'U': 3, 'F': 1, 'B': 1, 'Q': 1, 'M': 1, 'P': 2, 'R': 2, 'D': 1, 'T': 2, 'H': 1, 'Y': 1}`

(en pratique le fichier utilisé contiendra beaucoup plus de mots, et on pourra donc supposer qu'il contient au moins une fois chaque lettre de l'alphabet)

2. Ecrire la fonction `lettre_la_plus_frequente` qui prend en argument un dictionnaire `dico` dont les clés sont des lettres majuscules et les valeurs des entiers (comme celui obtenu avec la fonction précédente). La fonction renvoie la lettre (c'est-à-dire, la clé) qui est associée au plus grand entier dans le dictionnaire (correspond ici à la lettre la plus fréquente). En cas d'égalité, on renvoie une des lettres de valeur maximum.
   Exemple :
   si `dico = {'I': 2, 'E': 2, 'O': 4, 'N': 4, 'J': 1, 'A': 1, 'U': 3, 'F': 1, 'B': 1, 'Q': 1, 'M': 1, 'P': 2, 'R': 2, 'D': 1, 'T': 2, 'H': 1, 'Y': 1}` alors `lettre_la_plus_frequente(dico)` renverra 'O' ou 'N'.
3. Ecrire la fonction `fabrique_liste_freq` qui prend en argument un nom de fichier `nom_fichier` et qui renvoie la liste des lettres majuscules, de la plus fréquente à la moins fréquente dans le fichier `nom_fichier`. On utilisera les fonctions `dico_frequence` et `lettre_la_plus_frequente`.

Pour que cette stratégie soit efficace, il faut apprendre sur beaucoup de mots, donc on veut éviter de relire le fichier de mots à chaque fois. On ne générera qu'une fois la liste des lettres par ordre de fréquence et on la passera en paramètre de la partie.

4.3.4 Fonction `partie_auto`

Ecrire une fonction `partie_auto` qui a deux arguments obligatoires et deux arguments optionnels :

- `mot_myst` : le mot à deviner (obligatoire)
- `liste_lettres` : la liste ordonnée des lettres qui correspond à la stratégie choisie (la liste contient les 26 lettres de l'alphabet, en majuscules),
- `affichage` : un booléen spécifiant si on souhaite ou pas avoir des affichages textuels pour être en mesure de suivre le déroulement de la partie. Si `affichage` vaut vrai alors on fera à peu près les mêmes affichages que pour la fonction `partie_humain` au détail près qu'on ne demande pas les lettres et qu'on n'affiche pas la potence texte (puisqu'il n'y a pas de nombre maximum d'erreurs), mais qu'on affiche quand même à chaque étape la lettre proposée. On prendra un caractère au choix pour remplacer l'affichage des lettres non découvertes. Si `affichage` vaut faux, la fonction ne doit absolument rien afficher (argument optionnel, valeur par défaut `True`).
- `pause` : un booléen (faux par défaut) spécifiant si on souhaite ou pas avoir des pauses dans l'affichage textuel, lorsque l'affichage est activé (du type : "Appuyez sur n'importe quelle touche pour continuer" après chaque affichage du mot à découvrir).

Cette fonction fait jouer à l'ordinateur une partie en mode automatique en utilisant une liste de lettres correspondant à une stratégie. Ici, il n'est pas question de gagner ou de perdre, l'ordinateur joue jusqu'à avoir trouvé le mot et la fonction renvoie le nombre d'erreurs faites avant de découvrir complètement le mot.

Ecrire la fonction `partie_auto` et la tester avec des listes de lettres produites selon les trois stratégies.

Attention : Avant de recoder ce qui a déjà été fait, veillez à bien regarder quelles fonctions des Sections 4.1 et 4.2 peuvent être réutilisées.

Avant de passer à la partie Extensions, testez méticuleusement les fonctions `partie_humain` et `partie_auto`, par exemple avec plusieurs mots mystère, pour vous assurer qu'elles fonctionnent correctement. Commencez également à rédiger votre le rapport Latex sur cette partie guidée.

5 Partie Extensions

En plus des fonctionnalités présentée dans la Partie guidée, vous devez réaliser d'autres fonctionnalités, peu ou pas guidées, que l'on appellera des extensions. Le but est de personnaliser votre projet pour le différencier

des autres binômes, et aussi de montrer que vous êtes capables de concevoir une solution même lorsque vous êtes moins guidés.

Pour cette partie, vous serez probablement amenés à écrire plusieurs fonctions auxiliaires qu'il vous faudra définir vous mêmes. Cette partie est beaucoup moins guidée que les précédentes, à vous de structurer votre travail avec les fonctions qui vous semblent adéquates.

L'idée générale de la Partie Extensions est : imaginez d'autres fonctionnalités (ou améliorations de fonctionnalités existantes) et implémentez-les. Faites preuve d'originalité. Néanmoins, nous vous proposons quelques idées (section 5.1) pour le cas où vous seriez en manque d'imagination. Si vous n'êtes pas très inspirés, vous pouvez par exemple combiner deux extensions issues des propositions, avec une que vous aurez inventé. Au contraire, si vous avez plein d'idées, vous pouvez proposer trois extensions différentes de celles présentées ci-dessous. L'important est de proposer un contenu cohérent et innovant.

Nous ne pouvons pas répondre à la question : *combien d'extension faut-il réaliser ?* car cela dépend de la complexité et de la qualité des extensions choisies/imaginées. Disons que 2 à 4 extensions de difficulté/qualité suffisantes semble être une fourchette raisonnable, pour vous donner un ordre d'idée.

Vous devrez, dans votre rapport, expliquez quelles extensions vous avez réalisé, comment s'en servir, et brièvement comment vous avez procédé. De plus, vous ajouterez dans le programme principal du code très bien commenté permettant d'expliquer et d'illustrer (avec des exemples à décommenter) au mieux ces nouvelles fonctionnalités.

5.1 Propositions

5.1.1 Menu

On souhaite être accueilli à l'exécution de votre programme par un menu proposant les différentes options (partie manuelle par un humain, ou partie automatique par l'ordinateur avec éventuellement choix de la stratégie d'apprentissage des fréquences de lettres, lancement des extensions que vous faites etc....). Par *menu* on entend quelque chose de similaire à ce qui a été vu dans l'exercice *Stock de fenêtres* ou *Carte et billet SNCF* du cours Introduction à Python au S1.

5.1.2 Interface graphique

On souhaite ajouter un argument optionnel `mode_graphique` à la fonction `partie_humain`. Quand cet argument vaut vrai, la fonction doit prendre en charge un affichage graphique pour la partie de pendu. Quand il vaut faux, rien ne change. La valeur par défaut sera faux. Modifier la fonction `partie_humain` pour prendre en compte cet argument et ajouter ainsi un affichage graphique plus visuel de la partie de pendu. On conservera toutefois l'affichage texte en plus. Pour cet affichage, vous pouvez utiliser un module graphique facile à utiliser appelé `turtle` (pour lequel vous trouverez des tutoriels en ligne, par exemple https://zestedesavoir.com/tutoriels/944/a-la-decouverte-de-turtle/). Vous pouvez vous contenter d'afficher la potence en la complétant au fur et à mesure, ou compléter aussi par l'affichage du mot en cours de découverte (utiliser pour cela la fonction `turtle.write`). D'autre options sont envisageables... à vous d'imaginer ! Sentez vous également libre d'utiliser un autre package que `turtle`, par exemple `tkinter`.

5.1.3 Générer la liste de mots

En vous documentant un peu sur le module `faker` ici : https://faker.readthedocs.io/en/latest/index.html, créez vous-même un fichier texte contenant une grande liste de mots, avec le même format que celui proposé dans mots.txt (qui vous est fourni). *Indice : dans la documentation, chercher la page du Provider appelé* `lorem`. On peut générer des mots en français ou en anglais. Attention si vous utilisez le français : les caractères non-ASCII (tels que é, à, û, ç, ...) risquent de vous poser des problèmes, à vous de choisir de rester en anglais ou de trouver une solution.

5.1.4 Comparaison statistique des stratégies

L'intérêt d'avoir programmé le pendu en mode automatique avec plusieurs stratégies possibles est de faire une comparaison statistique des trois stratégies sur un grand nombre de parties.

- Dans un premier temps, on pourra calculer le minimum, le maximum et la moyenne du nombre d'erreurs faites avant de trouver le mot sur un grand nombre de simulations pour les trois stratégies (testez d'abord votre programme sur une dizaine de simulations, avant de passer à 100, 1000, 10000 ...). Les trois stratégies devront être comparées sur les mêmes mots.
- Il est assez naturel de penser que le nombre d'erreurs dépend de la longueur du mot à deviner. Une autre étude intéressante consiste donc à comparer graphiquement les résultats obtenus avec les trois stratégies selon la longueur du mot à deviner. On peut par exemple tirer au sort un grand nombre de mots, récupérer le résultat obtenu (c'est-à-dire le nombre d'erreurs) pour chacune des stratégies et le stocker dans un dictionnaire qui a pour clé les longueurs des mots à deviner et pour valeur correspondante la liste du nombre d'erreurs faites pour cette longueur de mot. Cela vous permettra d'obtenir un dictionnaire par stratégie. Ensuite, pour visualiser ces données, on fera un graphique en utilisant le module `pyplot` de la librairie `matplotlib`. Pour éviter d'avoir à écrire le nom `matplotlib.pyplot` devant chaque fonction de ce module que vous utiliserez, vous pouvez l'importer ainsi : `import matplotlib.pyplot as plt` et ainsi il suffira d'écrire `plt` devant chaque fonction utilisée. Pour visualiser la fenêtre graphique, après avoir donné à l'ordinateur toutes les instructions pour créer le graphique, il ne faudra pas oublier de faire `plt.show()`. Vous trouverez la documentation de `pyplot` ici :
  http://matplotlib.org/api/pyplot_api.html *ou tapez pyplot tutorial sans un moteur de recherche*
  Sur le graphique, on fera figurer en abscisse la longueur du mot et en ordonnée le nombre moyen d'erreurs pour trouver un mot de cette longueur. Il y aura une courbe par stratégie.

5.1.5 Stratégies plus évoluées

Vous pouvez imaginer des stratégies plus évoluées, par exemple, en étudiant la fréquence de chaque lettre non pas dans l'ensemble des mots présents dans votre fichier référence, mais en vous limitant à la fréquence des lettres dans les mots de votre fichier ayant le même nombre de lettres que le mot à deviner (puisque le nombre total de lettres du mot à deviner est connu). Pour ne pas avoir à construire le dictionnaire des fréquences pour chaque nouveau mot à deviner, cela signifie qu'il faut écrire une fonction qui à partir d'un nom de fichier fabrique le dictionnaire dont les clés sont les longueurs des mots dans le fichier et la valeur est le dictionnaire des fréquences correspondant. A partir de ce dictionnaire, il faudra construire un autre dictionnaire, dont les clés seront toujours les longueurs des mots, et la valeur associée la liste des lettres de la plus fréquente à la moins fréquente pour les mots de cette longueur.

Testez cette nouvelle stratégie avec `partie_auto`. Vous pouvez ajouter cette nouvelle stratégie à l'étude statistique précédente, si vous l'avez également codé.

Il existe des stratégies encore plus poussées que celle-là. Ne vous limitez pas !

5.2 Vos extensions

Ayez de l'imagination !

A Annexe – Configuration du dépôt GitLab

A.1 Bien démarrer

Dans les salles SCI 002 à SCI 006 : il faut se connecter en cliquant sur *Portail pédagogique VDI* et ensuite Linux. Si vous arrivez sur un écran de veille avec l'heure indiquée en grand au centre, réveillez le système (par la touche Échap ou un clic-glissé vers le haut) et ré-authentifiez-vous.

Dans les salles SCI 007, SCI 008, PHY 214 et ISIMA B011 : normalement, ces salles disposent d'un véritable système d'exploitation Linux installé. Si l'ordinateur est déjà démarré sous Windows, il est nécessaire de redémarrer en guettant attentivement le menu de *dual-boot* (double-démarrage). Ce menu vous permet de choisir entre plusieurs systèmes d'exploitation. En l'absence de choix explicite de l'utilisateur, le système par défaut (probablement Windows) est automatiquement pris au bout d'un certain temps (d'où l'attention requise). Vous devez choisir, à l'aide des flèches du clavier (haut/bas), une distribution (variante) de Linux : Ubuntu, Debian, Fedora, ... puis valider avec Entrée.

Depuis n'importe quelle machine et n'importe quel système d'exploitation : vous avez accès aux machines virtuelles depuis un navigateur web : vdiportail.dsi.uca.fr (ou en installant le logiciel VMWare Horizon). Pour la suite des instructions, voir le paragraphe sur les salles SCI 002 à SCI 006.
Pour optimiser les ressources, n'utilisez pas ces machines virtuelles si vous êtes déjà sur un ordinateur disposant de Linux avec tous les outils nécessaires.

A.2 Créer un projet GitLab

Choisissez un membre de l'équipe – que l'on nommera le *propriétaire* dans la suite – et faites-lui créez un nouveau projet **privé** sur gitlab.isima.fr comme ceci :

1. cliquez sur le bouton bleu Nouveau projet ;
2. choisissez la boîte Create blank project ;
3. choisissez un nom de projet, puis valider via le bouton bleu Create project.

C'est ce projet qui va contenir vos fichiers .py mais aussi votre rapport LaTeX pour le projet du pendu. Une fois créé de la sorte, le dépôt peut d'ores et déjà être cloné par le propriétaire en une copie de travail sur sa machine.

Dans un second temps, le propriétaire doit ajouter l'autre membre de l'équipe au dépôt gitlab. Ceci peut être fait comme suit, depuis la page principale du projet :

1. allez dans le menu *Project information* (panneau latéral de gauche), puis sélectionnez le sous-menu *Membres* ;
2. ajoutez l'autre membre de l'équipe, en précisant le rôle *Maintainer* (pour donner des droits en écriture et en lecture) mais sans préciser de date d'expiration de l'accès.

Désormais, l'autre membre peut également cloner le dépôt sur sa machine.

Enfin, par la même méthode, ajoutez votre chargé(e) de TP au projet, en lui octroyant le rôle ***Reporter***.

En cas de problème, contactez au plus vite votre chargé de TP.

Configuration avancée. Il est possible d'aller plus loin dans la configuration de gitlab, en utilisant les clés `ssh` (et l'accès `ssh`). Cette configuration n'est pas obligatoire mais très pratique si vous en avez marre de taper vos identifiants à chaque `pull` ou `push`. Elle ne concerne pas directement votre dépôt, mais votre compte personnel sur le gitlab de l'ISIMA. Cela se passe ici, où vous trouverez des liens de documentation pour créer votre clé `ssh`. Une fois votre compte gitlab configuré correctement, vos identifiants ne seront plus demandés à chaque échanges (`pull`, `push`) avec le dépôt gitlab.

A.3 Premier commit

On va commencer par cloner le projet que vous venez de créer sur GitLab, si ce n'est pas déjà fait.

1. Pour cela, retournez sur l'onglet principal de votre projet. Vous allez remarquer un bouton bleu Clone en haut à droite : cliquez dessus. Si vous êtes à l'aise avec SSH, configurez votre projet en accès SSH. Sinon, copiez l'URL de la case *Clone with HTTPS*. Si vous voulez vous familiariser plus tard avec SSH, ce sera évidemment possible.
2. Utilisez la commande `git clone` dans le terminal Linux, suivie de l'URL copié précédemment. Veillez à exécuter cette commande dans un dossier prêt à accueillir votre projet (pas dans Téléchargements par exemple). Le terminal vous demande de rentrer vos identifiants GitLab, puis clone votre projet. Vous pouvez remarquer qu'un dossier est apparu, contenant .git ainsi qu'un ReadMe.

Maintenant, nous allons ajouter un fichier .py à votre projet pour que vous commenciez à coder.

1. Ouvrez Idle. Créez un nouveau fichier et sauvegardez le dans le dossier de votre projet. Ajoutez y deux lignes de code : `import copy` et `import random`.
2. On va maintenant signifier à Git que l'on a effectué des modifications à notre fichier. Dans le terminal, exécutez `git add` suivi du nom de votre fichier.
3. Lancez `git status`. N'oubliez pas que cette commande vous permet de visualiser ce que vous venez d'ajouter à l'index.
4. Faites un commit ! On vous laisse retrouver la syntaxe (cf cours de S1 ou mémo ci-dessous). Vous pourrez ajouter en commentaire "Importation packages random et copy" par exemple.
5. Pour l'instant, vos modifications ont été seulement prises en compte sur votre version locale du projet (c'est-à-dire sur l'ordinateur sur lequel vous êtes). On veut déplacer ses modifications sur le dépôt GitLab de votre projet, afin que votre binôme (et votre encadrant !) puisse les voir. Lancez `git push`. Lorsque c'est terminé, actualisez la page GitLab de votre projet. Le commit doit apparaître.

A.4 Pour la suite

Diverses indications pour la suite du projet :

1. A chaque fois que vous travaillez sur le projet, pensez à `git pull` pour récupérer la dernière version du projet (au cas où votre binôme ait fait des modifications).
2. Testez également lors des premières séances un premier commit pour le second membre du binôme, pour vérifier que tout fonctionne.
3. Le dépôt GitLab contiendra également le rapport LaTeX de votre projet. Vous pouvez configurer un `.gitignore` pour éviter de prendre en compte les fichiers de compilations (`.aux`, `.toc`, etc.).

> De manière générale, vous devez vous référer aux cours du S1 (Python ou Git+LaTeX) avant d'appeler votre encadrant à l'aide : ils contiennent tout ce dont vous avez besoin, au moins pour la partie guidée du Projet.

A.5 Mémo git

Bien sûr, ce mémo n'est pas exhaustif. D'autres commandes et d'autres options sont disponibles, n'hésitez pas à consultez la documentation (en ligne https://git-scm.com/doc, tapez par exemple git−checkout dans la barre de recherche pour voir les options possibles de git checkout ; ou le man, tapez par exemple man git−checkout dans le Terminal ; voir aussi man giteveryday pour un rappel des commandes les plus courantes, man gitglossary, man gittutorial, man gitworkflows, ...)

Commande et exemple	Description
git init	Initialiser un dépot vide
git status	Voir l'état de l'index
git add fichier git add tp.py	Ajouter un fichier à l'index
git commit −m message git commit −m "Ajout de telle fonctionnalite"	Faire un commit avec le contenu actuel de l'index
git diff	Visualiser les différences entre votre copie de travail et votre index (qui seraient donc laissées de côté si l'on faisait un commit immédiatement)
git diff −−cached ou git diff −−staged	Visualiser les modifications qui sont actuellement dans l'index, prêtes pour le prochain commit
git add −−patch fichier git add −−patch tp.py	Ajouter une partie des modifications d'un fichier à l'index (accès à l'éditeur pour indiquer les modifications à prendre en compte)
git clone url git clone https://gitlab.isima.fr/aulagout/tp−projet	Cloner un dépôt git distant
git pull	Récupérer la dernière version d'un dépôt git distant déjà configuré
git push	"Pousser" les commits faits localement vers un dépôt distant déjà configuré. Seuls les commits de la branche contenant la HEAD seront recopiés sur le dépôt distant.
git branch maBranche	Crée une nouvelle branche à l'endroit où se trouve la HEAD (référence sur le commit où se trouve la HEAD). La HEAD ne se met **pas** automatiquement sur cette nouvelle branche
git checkout maBranche	Passer dans la branche maBranche : la HEAD se met dans cette branche, et le répertoire de travail prend l'état du dernier commit de cette branche.
git branch maBranche puis git checkout maBranche ou, 2-en-1 sur les versions récentes : git switch −−create maBranche	Débuter une nouvelle branche à l'endroit où l'on se trouve, et continuer sur cette branche.
git checkout refDunCommit git checkout HEAD^ (reculer d'un cran) git checkout HEAD~2 (reculer de 2 crans) git checkout 316e0cd5d99084bf6a516a58f6a4eb92db9d5e5d	Le répertoire de travail prend l'état enregistré dans ce commit, la HEAD se détache de la branche en cours pour se placer sur ce commit. Si l'on souhaite repartir de ce commit pour enchaîner sur un autre commit, il faut créer une branche ici.
git merge autreBranche	Fusionne la branche autreBranche dans la branche actuelle
git reset −−soft refDunCommit	Place la HEAD sur le commit indiqué. Le répertoire de travail n'est pas modifié. L'index n'est pas remis à zéro.
git reset	Remet l'index à zéro. Le répertoire de travail n'est pas modifié.
git reset −−hard refDunCommit	Remettre **tout** à l'état du commit indiqué : la HEAD (et la branche courante, si l'on est sur une branche) est déplacée sur ce commit, l'index est remis à zéro, et le **répertoire de travail est mis à l'état enregistré dans le commit**. *Perte de données si l'état du répertoire de travail précédent le reset n'a pas été commité*