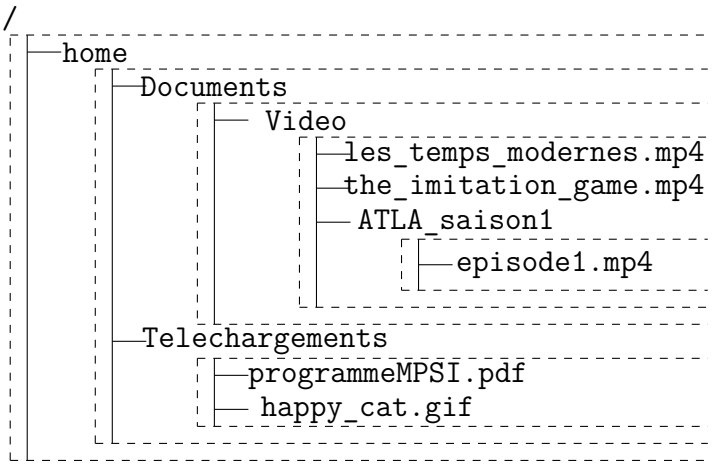
ProustGPT

Activité 06

Entre 1913 et 1927, Marcel Proust a publié *À la recherche du temps perdu*, roman tellement long qu'il a été édité en 7 tomes, s'étendant sur approximativement 3000 pages. L'objectif de ce TP est d'analyser ce roman d'un point de vue informatique, avant de construire un générateur de phrases « à la Proust ».

Partie 1 : lecture de fichiers

Les fichiers d'un ordinateur sont organisés sous forme d'arborescence : toutes les données de l'ordinateur sont stockées dans des dossiers qui sont inclus les uns dans les autres en suivant une structure d'arbre. Tout fichier a une adresse, appelée **chemin absolu** : il s'agit du chemin à prendre à partir du dossier racine (ici `/`) pour arriver jusqu'au fichier en question.



Dans l'arborescence ci-contre, décrite pour un système Linux, le chemin absolu de `episode1.mp4` est `/home/Documents/Video/ATLA_saison1/episode1.mp4`.

Dans ce TP, nous allons ouvrir à l'aide de Python des fichiers textes (sous format `txt`). On utilise pour cela les fonctions `open`, `read` (ou `readlines`), `split` et `close`. À gauche est décrite la syntaxe associée à la manipulation d'un fichier nommé « `file.txt` ».

`fichier = open(`*`chemin`*`, MODE)`	ouvre le fichier correspondant au chemin absolu, qui est désormais manipulable par le biais de la variable `fichier`. Le chemin absolu est écrit sous le format d'un string. `MODE` est un argument optionnel, et représente le mode de lecture : s'il vaut `"r"`, le fichier est en lecture seule ; `"w"` signifie qu'on écrase le fichier pour pouvoir écrire dessus ; `"a"` signifie que le fichier est préservé et que ce qui est écrit sera ajouté à la fin. Par défaut, `MODE` vaut `"r"`. Le type de `fichier` est très particulier, et n'est pas manipulable directement.
`texte = fichier.read(N)`	supprime les `N` premiers caractères de `fichier` pour les stocker dans `texte`. `texte` est désormais une chaîne de caractères. Attention : la commande ne modifie pas le fichier lui-même, seulement ce qui est contenu dans la variable `fichier`. `N` est optionnel : si `N` n'est pas précisé, l'intégralité du fichier est vidée et stockée dans `texte`.
`texte = fichier.readlines(N)`	`texte` contient désormais un tableau de strings : chaque string correspondant à une ligne du fichier initial. `N` est optionnel, et signifie qu'on a lu les `N` premières lignes de `fichier`.
`fichier.close()`	ferme l'accès au fichier. À mettre après avoir terminé la lecture d'un fichier.

Exercice 1

1. Télécharger le fichier `du_cote_de_chez_swann.txt`, et le placer dans le dossier `Documents`.
2. Le chemin de ce fichier est `"C:\\Users\\User\\Documents\\du_cote_de_chez_swann.txt"` : ouvrir, dans la fenêtre de gauche, le fichier dans une variable `fichier1`.
3. En utilisant `readlines`, créer une variable `tome1` contenant l'intégralité du premier tome. À quoi correspond une case de `tome1` ?
4. Observer `tome1[0]` : vous observez un problème d'encodage. Pour le régler, il faut modifier ce que vous avez écrit selon le format `fichier = open(`*`chemin`*`, MODE, encoding = "utf-8")`.
5. De même, créer des variables `tome2` pour le second tome. On fera bien attention à bien fermer le fichier entre deux lectures. **Vous pourrez ajouter les autres tomes lorsque vous aurez atteint la partie 3, mais pas maintenant.**
6. Créer une variable `oeuvre_integrale` contenant l'intégralité de l'œuvre.
7. Quelle est la longueur du plus long paragraphe de l'oeuvre ?
8. Combien y a-t-il de dialogues commençant par un tiret ? (pour obtenir le caractère tiret utilisé dans le document, on pourra ouvrir l'un des romans avec le bloc-notes et copier-coller le caractère-tiret correspondant, qui n'est pas le tiret classique).
9. Vérifier ce que vaut `oeuvre_integrale[7]`. Que cela représente-t-il ? S'agit-il d'un ou de plusieurs caractères ?
10. Vérifier à la fin de `oeuvre_integrale[8]` : le dernier caractère est-il utile ?
11. Modifier `oeuvre_integrale` pour qu'elle ne contienne plus ces informations inutiles.

Partie 2 : analyse du texte

Exercice 2

Pour mieux analyser le travail de Proust, il faut découper chaque paragraphe en mots.

1. Tester la commande : `"oui bonjour coucou".split(sep = " ")`. En déduire l'objectif et la syntaxe de la méthode `split`.
2. Afin de mieux comprendre comment fonctionne `split`, écrire une fonction `split_a_la_main(texte,sep)` qui renvoie ce que fait `split`, mais en le codant à la main.

   Désormais, vous pouvez librement utiliser `split`.
3. Créer une variable `oeuvre_mots` qui contient les données stockées dans `oeuvre_integrale`, mais découpées mot à mot : on ignorera pour le moment les problèmes dûs aux signes de ponctuation, ainsi que la casse (majuscule vs. minuscule). Combien de mots obtenez-vous au total dans le roman ?

Exercice 3

On cherche à obtenir les nombres d'occurrences de chaque mot apparaissant dans le roman.
Pour cela, nous utiliserons la structure de données de dictionnaires : si vous ne l'avez pas déjà fait, c'est le moment de lire la feuille qui les présente !

1. Tester dans l'invite de commandes la commande `"Qu'ouis-je?".lower()`.
2. Créer une variable `occurrences`, qui est un dictionnaire vide.
3. Écrire un script permettant de remplir `occurrences` avec les occurrences des différents mots apparaissant dans le texte. Par exemple, à la fin de votre script, `occurrences["combray"]` vaudra 75.
4. Combien de mots **différents** apparaissent dans le roman ? Quel est le mot le plus long apparaissant dans le roman ?

Exercice 4

La loi de Zipf est une règle empirique qui établit que la distribution des occurrences des mots suit une loi inverse : considérons le mot le plus courant de l'œuvre (il s'agit de « de »). Alors le 2ème mot le plus courant sera 2 fois plus rare ; le 3ème 3 fois plus rare ; etc. Nous allons observer sur notre œuvre si la loi de Zipf est vérifiée.
Écrire une fonction `zipf()`, qui :

— trie, dans l'ordre décroissant, les occurrences des mots ;

— trace la courbe correspondant aux occurrences des 1000 mots les plus courants (au-delà, le graphe est trop aplati).

Selon l'algorithme de tri utilisé, l'exécution risque d'être longue (~ 3 minutes). On pourra rajouter quelque chose dans la boucle `for` de la forme `if i%100==0: print(i)` *afin de s'assurer que le programme tourne toujours comme prévu.*

Partie 3 : génération aléatoire de texte

Vous pouvez rajouter les 5 autres tomes. Si votre Pyzo crashe, retirez quelques tomes.

Exercice 5

1. Créer un dictionnaire `suivant`, de telle sorte que `suivant[m]` renvoie la liste (avec répétitions) des mots suivants `m` dans le roman. Par exemple, `suivant["survivait"]` renverra `['pendant', 'quelque', 'en', 'plus', 'à']` : ce sont les mots qui succèdent à « survivait » dans le roman, à un moment ou un autre.
2. Importer au début du fichier la bibliothèque `random` sous l'alias `rd`, et tester plusieurs fois dans l'invite de commandes `rd.randint(2,4)`.
3. Écrire une fonction `proust_gpt(début,longueur)` qui renvoie une phrase commençant par le mot `début`, et qui contient au total `longueur` mots, à la manière de Proust : on tirera le mot à utiliser aléatoirement parmi les successions possibles du mot précédent.
4. Écrire une fonction `proust_phrase(début)` qui renvoie une phrase commençant par le mot `début` et terminant par un point.
5. Tester le texte généré : est-il de bonne qualité ?

Exercice 6

Pour améliorer la qualité du texte généré, une idée consiste à générer le mot suivant non pas seulement à partir du mot précédent, mais à partir des deux mots précédents.

1. Créer une variable `suivant2`, dont les clés sont des tuples `(mot1,mot2)`, de sorte que `suivante2[mot1,mot2]` contiennent les mots apparaissant après `mot1` puis `mot2`.
2. Écrire une fonction `proust_gpt_2(début)` qui renvoie une phrase commençant par le mot `début` et terminant par un point.