Autour des anagrammes

Mickaël Péchaud (mickaelpechaud@protonmail.com)

Mars 2021

- Documentez vos fonctions.
- Testez au fur et à mesure les fonctions que vous écrivez !

Un fichier `anagrammes.py` à compléter est fourni.

1 Introduction

Une *anagramme* d'un mot m est un mot obtenu par permutation des lettres de m. Dans une anagramme, on ignore la casse (majuscule/minuscule), les symboles diactritiques (accents et cédilles), tirets et autres apostrophes. Ainsi, les mots suivants sont des anagrammes :

- «nectar» et «carnet»
- «sortie» et «rôties»
- «niche» et «Chine»

L'objectif de ce TP est d'écrire différentes fonctions permettant de trouver rapidement des anagrammes de mots (ou de listes de mots à la fin du TP).

Le fichier `listemotsfrancais.txt` fourni est un fichier texte contenant 629 569 mots français [1].

Vous trouverez en début du fichier `anagrammes.py`

- une fonction `enleve_diacritiques_majuscules`, qui renvoie la chaîne de caractères en minuscule et sans symboles diacritiques ni apostrophes ni tiret correspondant à la chaîne passée en argument,
- une fonction `charger_liste_mots`, qui permet de charger le fichier `listemotsfrancais.txt` sous forme d'une liste de chaînes de caractères.

Ces fonctions seront utiles à partir de la partie 4.

Dans la suite, sauf précision contraire, les mots seront considérés comme étant en minuscule et sans symbole diacritique.

2 Conversions

1. Quelle est en pratique la principale différence entre une chaîne de caractères (i.e. un objet de type `str`) et une liste de caractères ? Compléter les fonctions de conversions entre ces deux types.

> Les caractères d'une `str` ne sont pas modifiables contrairement à ceux d'une liste de caractères.
> La liste de caractères est donc plus souple d'utilisation, mais ne pourra par contre pas être utilisée comme clé d'un dictionnaire.

Dans la suite, prêter attention au type des paramètres et valeur de retour des fonctions ! **Sauf mention contraire, «mot» désignera une chaîne de caratères.**

3 Tester si deux mots sont anagrammes l'un de l'autre

Dans toute cette partie, on supposera que les deux mots testés sont de même longueur.

Une bien mauvaise méthode

On commence par étudier une très «mauvaise» méthode pour savoir si deux mots m_1 et m_2 sont anagrammes l'un de l'autre. On va engendrer toutes les permutations des lettres de m_1, et vérifier si l'une d'elle est égale à m_2.

2. Écrire une fonction *récursive* `permutation_aux` qui prend en argument une liste l de longueur n et un indice $i \in [\![0, n]\!]$ et renvoie la liste des permutations de l laissant fixes les éléments d'indices strictement inférieurs à i. Par exemple :

1. Fichier engendré par à partir du dictionnaire français d'*aspell* (`http://aspell.net/`)

```
>>> permutations_aux(['a','b','c','d'], 1)

[['a', 'b', 'c', 'd'],
 ['a', 'b', 'd', 'c'],
 ['a', 'c', 'b', 'd'],
 ['a', 'c', 'd', 'b'],
 ['a', 'd', 'c', 'b'],
 ['a', 'd', 'b', 'c']]
```

On ne se souciera pas des éventuels doublons dans le résultat renvoyé.

3. En déduire une fonction `permutation` qui prend en argument une liste l et renvoie la liste de ses permutations.
4. En déduire une fonction `sont_anagrammes1`, qui prend en argument deux mots et teste s'ils sont anagrammes l'un de l'autre.
5. En quoi cette fonction est-elle «mauvaise» ?

> Dans le pire des cas, toutes les permutations du premier mot vont être engendrées. On effectuera donc au moins $n!$ comparaisons de listes, ce qui va devenir prohibitif pour des mots de plus de quelques lettres...

Nous allons donc chercher d'autres façons de procéder.

6. On propose l'algorithme suivant pour tester si deux mots sont anagrammes : étant donnés deux mots m_1 et m_2, on cherche si chaque lettre de m_1 apparaît dans m_2, et si réciproquement chaque lettre de m_2 apparaît dans m_1. Cet algorithme est-il correct ?

> Non. Il renvoie par exemple vrai pour `'aba'` et `'aab'`, qui ne sont pas anagrammes l'un de l'autre. L'algorithme teste en fait si les deux mots ont le même ensemble de lettres, alors qu'il faudrait tester s'ils ont le même *multiensemble* de lettres (un multiensemble étant un ensemble dans lequel le nombre d'apparition des éléments compte).

7. Avant de passer à la suite, réfléchir à une autre stratégie pour résoudre le problème posé.

Une seconde méthode

8. Écrire une fonction `mot_to_dico`, qui prend en argument un mot m et renvoie un dictionnaire
   - dont les clés sont les lettres de m
   - et la valeur correspondant à une clé k est le nombre d'occurrences de k dans m.
9. En déduire une fonction `sont_anagrammes2` (on pourra utiliser `==` pour comparer l'égalité de deux dictionnaires, avec une complexité temporelle $O(n)$ où n est la taille commune des dictionnaires, et $O(1)$ s'ils sont de tailles différentes). Quelle est sa complexité ?

> L'insertion ou la modification dans un dictionnaire s'effectuant en temps constant, `mot_to_dico` est de complexité $O(n)$, ainsi que `sont_anagrammes2`.

Une troisième méthode

10. Écrire une fonction `tri_liste` qui trie la liste de caractères l passée en argument, en utilisant votre algorithme de tri préféré. On indique que `<` permet de comparer des caractères.
11. En déduire la fonction `tri_mot`, qui prend en argument un mot, et renvoie la version triée de ce mot.
12. En déduire une fonction `sont_anagrammes3`. Quelle est sa complexité ?

> La comparaisons de chaînes et les conversions chaînes $\leftrightarrow$ listes s'effectue en temps $O(n)$. Le facteur limitant est donc l'algorithme de tri – qui doit être de complexité $O(n\log(n))$ ou $O(n^2)$ selon votre choix d'il y a deux questions.

13. Effectuer une rapide comparaison empirique de temps d'exécution des trois versions de `sont_anagrammes`. Vous pourrez utiliser la fonction `process_time()` du module `time`.

> Sur le corrigé du TP – qui utilise traitreusement `sort` pour trier une liste – les deux dernières méthodes donnent des temps d'exécution du même ordre de grandeur.

4 Recherche de toutes les anagrammes d'un mot donné

Dans cette partie, les mots peuvent tous avoir des majuscules et des symboles diacritiques. La notion d'anagramme s'entend cependant toujours comme dans l'introduction.

14. En vous appuyant sur l'une des fonctions `sont_anagrammes` ci-dessus, écrire une fonction `liste_anagrammes1`, qui prend en argument un mot et une liste de mots, et renvoie la liste des anagrammes du mot dans la liste. Par exemple

```
>>> liste_anagrammes1('préparation', liste_mots)
['apparieront', 'préparation', 'appaireront']
```

Quelle est sa complexité ? Constater empiriquement son temps d'exécution sur quelques exemples.

> Dans la version du corrigé, `sont_anagramme2`, de complexité $O(n)$ est utilisé.
> Si l'on note N la taille de la liste de mots, on obtient donc une complexité $O(nN)$.

On cherche maintenant à *pré-traiter* la liste de mots pour obtenir un algorithme plus efficace.
Pour cela, on va créer un dictionnaire à partir de la liste de mots

- dont les clés k sont des chaînes de caractères triées (sans diacritiques ou majuscules)
- dont la valeur correspondant à k est la liste des mots qui sont anagrammes de k.

Voici à titre d'exemple un extrait du dictionnaire que l'on souhaite obtenir :

```
{
 ...
 'acehoppr' : ['achopper', 'approche', 'approché', 'choppera'],
 'acehopps' : ['achoppes', 'échoppas', 'achoppés'],
 'aacehinoppt' : ['achoppaient'],
 ...
}
```

15. Écrire la fonction `liste_mots_to_dict` correspondante, et générer le dictionnaire, qui servira dans toute la fin du TP.

16. En déduire une fonction `liste_anagrammes2`, qui prend en argument un mot et un dictionnaire, et renvoie *en temps constant en la taille du dictionnaire* la liste des anagrammes du mot.

17. Trouver l'ensemble de mots tous anagrammes les uns des autres le plus grand possible.

5 Anagrammes avec plusieurs mots

On veut maintenant faire des anagrammes de listes de mots : étant donnés un ou plusieurs mots, on cherche toutes les listes de deux, trois, quatre... mots tels que la concaténation de ces mots forme une anagramme de la concaténation des mots de départ.
Par exemple, parmi les ensemble de deux ou trois mots possibles pour «informatique», on trouve «quota+infirme», «if+romantique», «requin+à+motif» ou encore «tif+maroquiné» (et bien d'autres).

Fonctions préliminaires

On dit que m est un *sous-mot* de m' si toute lettre de m apparait également dans m' avec un nombre d'occurrences au moins égal (indépendamment de l'ordre des lettres).

18. En utilisant `mot_to_dico`, écrire une fonction `sous_mot1` qui prend en argument deux mots `str1` et `str2`, et teste si le premier est un sous-mot du second.

19. On considère maintenant le cas où `str1` et `str2` sont des mots *triés*. Notons a la première lettre de `str1`, et $p(a)$ la position de a dans `str2` si elle y apparaît. En remarquant que `str1` est un sous-mot de `str2` si et seulement si la a apparaît dans `str2` et que `str1[1 :]` est un sous-mot de `str2[p(a)+1 :]`, écrire une fonction `sous_mot2` utilisant une fonction auxiliaire *récursive* pour répondre à cette question.

20. Écrire une fonction `soustrait` qui prend en argument deux mots triés `str1` et `str2` tels que le premier soit un sous-mot du second, et renvoie le mot trié correspondant à `str2` auquel on a enlevé les lettres de `str1` (autant de fois que leur nombre d'occurrence dans `str1`). On pourra utiliser au choix une stratégie itérative ou récursive.

21. Écrire une fonction `tous_anagrammes_liste` qui prend en argument un dictionnaire et une liste de mots triés $[m_1, \dots m_n]$, et renvoie la liste de toutes les listes de la forme $[m'_1, \dots m'_n]$, où pour tout $i \in [\![1, n]\!]$, m'_i est une anagramme de m_1 appartenant à la liste de mots de départ. Par exemple

```
>>> tous_anagrammes_liste(dictionnaire, ['aeiimnqstuu', 'fou', 'er'])
[['musiquaient', 'fou', 'Re'], ['musiquaient', 'fou', 'ré'],
 ['musiquaient', 'ouf', 'Re'], ['musiquaient', 'ouf', 'ré']]
```

Anagrammes avec plusieurs mots

Nous pouvons maintenant écrire une fonction *récursive* `anagramme_liste` qui prend en argument :
- un dictionnaire `dictionnaire`
- une liste `accumulateur` de mots triés
- un mot trié `reste`
- un entier `n` strictement positif

et qui *affiche* (au format de votre choix) toutes les listes constituées de `len(accumulateur)+n` mots français telles que
- les `len(accumulateur)` premiers éléments soient chacun une anagramme du mot trié correspondant d'`accumulateur`
- la concaténation des `n` derniers mots soit une anagramme de `reste`.

Par exemple :

```
>>> anagramme_liste(dictionnaire, [], tri_mot('informatiquepourtous'), 2)
['mouftai', 'pronostiqueur']
['pronostiqueur', 'mouftai']

>>> anagramme_liste(dictionnaire, [], tri_mot('anagramme'), 3)
['an', 'agréa', 'mm']
['an', 'égara', 'mm']
['an', 'ragea', 'mm']
['na', 'agréa', 'mm']
['na', 'égara', 'mm']
['na', 'ragea', 'mm']
...

>>> anagramme_liste(dictionnaire, ['aaegr'], 'anmm', 2)
['agréa', 'an', 'mm']
['agréa', 'na', 'mm']
['égara', 'an', 'mm']
['égara', 'na', 'mm']
['ragea', 'an', 'mm']
...
```

22. Écrire la fonction `anagramme_liste`. Indications :
- Que faire dans le cas de base $n = 1$?
- Dans le cas $n \geqslant 2$, on parcourra les clés du dictionnaire à la recherche de sous-mots de `reste`. Pour chaque sous-mot ainsi trouvé, on fera un appel récursif de la fonction avec un dernier paramètre `n-1`, et d'autres paramètres bien choisis.

L'exécution de cette fonction peut-être vue comme une exploration des listes de mots par une méthode de *backtracking* (ou *retour sur trace*). Voir par exemple http://mpechaud.fr/scripts/parcours/index.html.

Pour aller plus loin

- L'exécution de `anagramme_liste` affiche les $n!$ permutations de chaque liste solution. Pour y remédier, modifier la fonction de façon à ce qu'elle n'affiche que les listes classées par ordre lexicographique croissant des mots triés (ce qui va au passage réduire la taille de l'espace exploré...).
- Afin d'accélérer les calculs on pourrait commencer par enlever du dictionnaire tous les mots qui ne sont pas sous-mots du mot dont on cherche une anagramme au départ. Coder et expérimenter !

Annexe

Voici les fonctions et méthodes sur les listes, dictionnaires et deque dont l'usage est autorisé – ainsi que les complexités que vous utiliserez pour les calculs de complexité (en fonction de la longueur de la liste n). Tout autre fonction dont vous auriez besoin doit être implémentée !

Fonctions et méthodes sur les listes

Opération	Exemple	Complexité
Création d'une liste vide	`l=[]`	$O(1)$
Accès direct	`l[0]`	$O(1)$
Longueur	`len(l)`	$O(1)$
Concaténation	`l1+l2`	$O(n1+n2)$
Ajout en fin de liste	`l.append(1)`	$O(1)$
Suppression en fin de liste	`l.pop()`	$O(1)$
Extraction de tranche	`l[1 :10]`	$O(n)$, où n est la longueur de la tranche.
Répétition	`[0]*k`	$O(n)$, où n est la longueur de la liste créée.
Création par compréhension	`[k**2 for k in range(n)]`	$O(n)$ si l'expression est évaluée en temps constant
Copie	`copy(l)`	$O(n)$
Test d'égalité	`l1 == l2`	$O(1)$ si les longueurs sont différentes, $O(n)$ sinon

Fonctions et méthodes sur les chaînes de caractères

Opération	Exemple	Complexité
Création	`s = 'Ma chaîne'`	$O(n)$
Accès direct	`s[0]`	$O(1)$
Longueur	`len(s)`	$O(1)$
Concaténation	`s1+s2`	$O(n1+n2)$
Extraction de tranche	`s[1 :10]`	$O(n)$, où n est la longueur de la tranche.

Fonctions et méthodes sur les dictionnaires

Opération	Exemple	Complexité
Création	`d = {cle : valeur}`	$O(1)$
Test d'appartenance d'une clé	`cle in d`	$O(1)$
Ajout d'un couple clé/valeur	`d[cle] = valeur`	$O(1)$
Valeur correspondant à une clé	`d[cle]`	$O(1)$
Égalité de deux dictionnaires	`d1 == d2`	$O(n)/O(1)$ si de tailles différentes
Ensemble des clés	`d.keys()`	$O(n)$