Bilan du travail à la maison

Exercice 1
Avez-vous des question concernant les exercices traités sur codingame ?

Les deux prochains exercices sont a traiter sur une feuille à part, et à rendre.

Exercice 2

1. Décrire ce que va afficher le programme suivant :

```
for k in range(3):
    print(k)
```

2. Quelle est la valeur contenue dans la variable `somme` à la sortie de la boucle ?

```
somme = 0
for n in range(4):
    somme += n*n
```

3. Que va afficher le programme suivant ?

```
for lettre in "Ab cd" :
    print(lettre)
```

Exercice 3
Écrire un programme qui calcule dans une variable `somme` la somme de tous les entiers de 1 à 175.

Exercice 4
Pour le prochain TD, traiter chez vous les feuilles n°8 et 9 (de la page https://www.codingame.com/playgrounds/53303/) qui reprennent des éléments de ce TD.

I Quelques outils sur les chaînes de caractères

Une chaîne de caractères est un objet informatique qui représente du texte. Techniquement, en python, on les distingue par des guillemets simples ou doubles.

Voici deux exemples de chaînes de caractères.

```
'Guillemets simples, touche 4'
''Guillemets doubles, touche 3''
```

Les guillemets encadrent la chaîne.

1.1 Structure

Un chaîne est un itérable contenant uniquement des caractères (un caractère est l'équivalent d'une chaîne ne contenant qu'une seule lettre)

```
s = ''Une chaîne d'exemple''
len(s)  # le nombre de caractères, espaces compris
s[0]  # le premier caractère
```

À la différence des listes, on ne peut pas modifier les caractères contenus dans une chaîne.

```
s[0] = ''u''  # ceci doit renvoyer une erreur, à bien lire.
```

1.2 Itérer sur une chaîne

Comme pour les listes, il y a deux manières d'accéder séquentiellement à tous les caractères d'une chaîne

```python
for i in range(len(s)):
    # i est un entier, représentant un indice dans la chaîne.
    # i va prendre les valeurs 0, 1, 2...
    s[i]  # il s'agit du caractère d'indice i
```

ou alors

```python
for c in s:
    # c est directement un caractère
    # Avec notre exemple de chaîne s, c va prendre comme valeurs
    # successives ''U'', ''n'', ''e'', '' ''...
```

II Anagrammes

Cadre de l'étude

- On appellera "mot" une chaîne de caractères uniquement composée des 26 lettres de l'alphabet en minuscules. Ainsi on interdit les espaces, accents, cédilles....
- Un **anagramme** d'un mot m un mot obtenu par permutation des lettres de m (on change l'ordre). Par exemple "carte", "tracé", "écart" sont des anagrammes. Rappelons que nous ne considérons pas les accents. Dans le code (sauf mention explicite du contraire), on considère que tous les mots sont sous la forme normalisée du point précédent.

Le but de ce TP est de pouvoir déterminer si deux mots donnés sont anagrammes l'un de l'autre, puis de trouver la liste de tous les anagrammes d'un mot donné.

2.1 Quelques outils

Nous allons tout d'abord créer deux fonctions qui nous aideront dans la suite

Dans la suite, nous allons compléter le fichier python **td3.py** fourni avec cet énoncé. La modification de ce fichier se fera dans l'éditeur puis on utilisera au choix la touche F5 ou le bouton "Run file" pour envoyer le contenu de ce fichier vers la console au moment de tester nos fonctions.

Exercice 5
Tester dans la console

```python
type(''une chaine'')
type(42)
```

La fonction `type` permet de connaître le type d'un objet python. Ainsi les chaînes de caractères sont de type `str` et les entier de type `int`.

Exercice 6
Compléter la fonction `nb_occurences(mot: str, lettre: str) -> int` qui prend en argument un mot et une lettre (une chaîne de 1 caractère) et retourne le nombre de fois où la lettre donnée apparaît dans le mot.

Remarquez l'écriture de notre fonction. Nous avons noté après chaque argument ":" puis le type attendu. Il s'agit d'une aide à la compréhension de cette fonction qui indique ici que les deux arguments doivent être des chaînes de caractères.
Après la parenthèse fermante de la liste d'argument on a noté "->" puis le type de la valeur retournée.

Tester votre fonction avec plusieurs exemples qui couvrent les différentes réponses possiblement attendues.

Exercice 7
On souhaite maintenant, étant donné un mot, créer la liste de ses caractères. Pour créer une liste élément par élément on utilise la construction suivante :

```python
L = []  # une liste vide au départ
for ..... : # le for dépend de la situation
    elt =  # calcul d'un élément à ajouter à L
    L.append(elt)  # on ajout elt à la fin de la liste L
```

1. Calculer à la main la liste `L` suivante puis vérifier dans la console

```python
L = []
for i in range(5):
    L.append(i**2)
```

Vérifier le type de `L`

2. Compléter la fonction `liste_mot(mot: str) -> list` qui prend un mot en argument et retourne la liste contenant tous les caractères du mot.

2.2 À la recherche des anagrammes

Dans cette partie, nous cherchons à déterminer si `mot1` et `mot2` sont anagrammes l'un de l'autre.

Exercice 8
Voici un premier algorithme pour répondre à la question :

- Pour chaque caractère `c` de `mot1`, on compte le nombre d'occurrences de `c` dans `mot1` et dans `mot2`.
- Si ces nombres sont différents, on est sûr que la réponse à notre question est non. Sinon on poursuit le parcourt de `mot1`
- Une fois chaque caractère examiné et si tous les nombres coïncident, nos deux mots sont anagrammes l'un de l'autre.

1. Appliquer à la main cet algorithme sur les mots "niche" et "chien" puis sur "niche" et "chiens". Expliquer pourquoi la réponse de notre algorithme n'est pas correcte dans le deuxième cas. Que pouvons-nous vérifier en plus pour nous assurer que `mot1` et `mot2` sont bien anagrammes l'un de l'autre ?
2. Compléter la fonction `sont_anagrammes1(mot1: str, mot2: str) -> bool` qui retourne un booléen indiquant si `mot1` et `mot2` sont anagrammes l'un de l'autre. Les booléens en python se notent `True` et `False` (exactement comme ça, en particulier avec une majuscule)

```python
type(True)
```

3. Tester votre fonctions avec les exemples de la première question, puis avec "aaaabbbb" et "abababab". Quel reproche peut-on faire à notre algorithme ?

Exercice 9
Voici un deuxième algorithme, qui pourra être plus efficace :

- Commencer par convertir les deux mots en deux listes.
- Trier les listes (par ordre alphabétique)
- Vérifier si les listes triées sont égales ou non.

Pour la mise en place en python, nous aurons besoin de deux outils :

```python
L.sort()  # trie la liste L en modifiant l'ordre de ses éléments
# cette fonction n'a pas de valeur de retour

L1 == L2  # teste si les deux listes L1 et L2 ont exactement les mêmes
# éléments et dans le même ordre
```

Compléter la fonction `sont_anagrammes2` en implémentant cet algorithme (c'est à dire, écrire le code python qui correspond à l'algorithme proposé).

III Dictionnaires

3.1 Structure de données

Un **dictionnaire** est un autre type de données utilisable en python. Il permet de stocker des **valeurs** associées à des **clés**.
Plus précisément, un dictionnaire est un ensemble de (clé, valeur) où chaque valeur est associée à exactement une clé.

3.2 Utilisation en python

Pour créer un dictionnaire, on utilise des accolades de manière similaire aux crochets utilisés pour les listes.

```python
d = {}  # un dictionnaire vide
```

```
d = {''a'': 12}  # un dictionnaire contenant une paire de (clé, valeur)
# la clé est la chaîne ''a'' et la valeur le nombre 12
d[''a'']  # c'est la valeur associée à la clé ''a'', ici 12
d[0] = [2 ,4, 6]  # on crée un nouvelle paire (clé, valeur). La clé est un nombre
# et la valeur une liste.
```

- Les clés utilisables pour nous sont : des chaînes, des nombres. En toute généralité, tout objet non modifiable peut être utilisé comme clé.
- Les valeurs sont des objets python quelconques (nous connaissons : nombres entiers ou flottants, booléen, chaîne, liste, dictionnaire)

Pour tester si une clé est présente dans un dictionnaire (et avec le dictionnaire `d` du cadre précédent)

```
''a'' in d  # c'est le booléen True car la clé est présente dans d
2 in d # c'est le booléen False
```

3.3 Application aux anagrammes

Exercice 10
Compléter la fonction `dico_mot(mot: str) -> dict` qui prend un mot comme argument et renvoie le dictionnaire dont les clés sont les lettres distinctes du mot et les valeurs le nombre d'occurrences de chaque lettre.

Exercice 11
Si vous avez utilisé la fonction `nb_occurences` pour l'exercice précédent, créer une nouvelle version de la fonction `dico_mot(mot: str) -> dict` qui ne parcourt qu'une seule fois le mot donné. Indication : on peut, pour chaque lettre, vérifier si c'est déjà une clé et agir en conséquence.

Exercice 12
Appliquer la fonction `dico_mot(mot: str) -> dict` pour compléter la fonction `sont_anagrammes3`. On peut comparer les dictionnaires exactement comme les listes avec l'opérateur `==`

IV Liste d'anagrammes

Peut-être avez-vous remarqué en haut du fichier de script une fonction et la création d'une variable `liste_mots`. Cette liste contient une liste de mots français modifiés pour respecter nos conventions (aucun accent, cédille ou autre caractère spécial). Nous allons maintenant chercher, pour un mot donné, la liste de tous ses anagrammes en français.

Exercice 13
Compléter la fonction `liste_anagrammes1(mot: str, L: list) -> list` qui prend un argument un mot et une liste de mots et retourne la liste de tous les anagrammes de `mot` trouvé dans la liste `L`. On utilisera au choix une des trois fonctions précédentes pour vérifier si deux sont sont anagrammes l'un de l'autre.

Exercice 14
Pour améliorer le temps d'exécution de la recherche précédente, on se propose de pré-traiter la liste des mots. Nous allons créer un dictionnaire au format suivant :

- les clés sont des chaînes de caractères triées.
- la valeur associée à une clé donnée est la liste de tous les anagrammes de cette clé (c'est à dire la liste de tous les mots qui redonnent la clé lorsque l'on trie la chaîne par ordre alphabétique)

Nous allons procéder en deux temps

1. Compléter la fonction `trie_chaine(mot: str) -> str`. Pour cela nous aurons besoin de savoir créer une chaîne pas à pas. L'idée est la même que pour les listes mais on utilise la syntaxe

   ```
   s = ''une chaine''
   s = s + '' plus longue'' # le + entre chaînes est appelé concaténation :
   # on ajoute les caractères donnés à la fin de s
   ```

2. Compléter la fonction `dico_depuis_liste_mots(L: list) -> dict` qui prend une liste de mots en argument et retourne le dictionnaire décrit dans le préambule.
   Utiliser ensuite cette fonction pour créer dans la console le dictionnaire `d_mots` qui représente la liste `liste_mots`
3. Créer une fonction `liste_anagrammes2` dont on donnera la signature (ie. on précisera le type et la signification des arguments) qui utilise les concepts de cet exercice. Combien d'opérations sont maintenant nécessaires au calcul de la liste des anagrammes d'un mot donné ? (on comptera séparément les concaténation de chaînes et les accès aux valeurs d'un dictionnaire).
4. Trouver l'ensemble de mots anagrammes les uns des autres le plus grand possible.

```python
# d est un dictionnaire
d.keys() # retourne l'ensemble des clés de d, qui est in itérable.
for cle in d.keys():
    # ici on peut travailler avec d[cle] qui sera séquentiellement toutes les
    # valeurs contenues dans d
```

Exercice 15 (Pour aller plus loin)
On cherche maintenant à savoir si un mot est l'anagramme d'une concaténation de deux mots, et oui desquels. Compléter les fonctions restantes dans le fichier **td3.py**.
La fonction `est_sous_mot` est plutôt corsée :).