

TD Dictionnaires

Dans les exercices, chacune des fonctions demandées devra être testée.

Exercice 1 : Fonction mini_maxi

Ecrire une **fonction mini_maxi** qui prend en entrée une liste non vide de nombres et qui renvoie un dictionnaire comportant 2 éléments :

- La clé est soit la chaîne "maximum", soit "minimum"
- La valeur correspond respectivement au maximum ou au minimum des nombres de la liste en entrée.

Exemple d'exécution :

```
L = [1, 2, 15, -3, 25, 10, -6, -5]
```

```
d = mini_maxi(L)
```

```
print( d ) #Affiche {'maximum': 25, 'minimum': -6}
```

Exercice 2 : Dérivation de polynômes

On souhaite gérer des polynômes définis suivant des puissances croissantes.

Exemple : $P_1(x) = 2 + 3x - 5x^4$

1. Dans une 1^{ère} approche, on choisit de représenter ce polynôme par une liste de nombres pour les coefficients, classés par ordre de puissances croissantes.

Exemples :

Pour $P_1(x) = 2 + 3x - 5x^4$, la liste sera constituée de : [2,3,0,0,-5]

Pour $P_2(x) = 4$ (polynôme constant), la liste sera [4]

Pour $P_3(x) = 0$ (polynôme nul), la liste sera [0]

Ecrire une **fonction derivePolynome** qui prend en paramètre une liste représentant le polynôme (suivant le format précédent) et renvoie une liste comportant les coefficients du polynôme dérivé.

Exemples :

Pour $P_1(x) = 2 + 3x - 5x^4$, la dérivée vaut : $3 - 20x^3$

```
derivePolynome( [2,3,0,0,-5] ) renvoie [3,0,0,-20]
```

Déterminer en fonction du degré n du polynôme les **complexités en temps et en espace** de **derivePolynome**.

2. Dans une 2^{ème} approche, on choisit de représenter ce polynôme par un dictionnaire dont les clés sont les degrés des monômes non nuls et les valeurs les coefficients associés.

Exemples :

Pour $P_1(x) = 2 + 3x - 5x^4$, le dictionnaire sera composé de : { 0 : 2, 1 : 3, 4 : -5 }

Pour $P_2(x) = 4$ (polynôme constant), le dictionnaire sera { 0 : 4 }

Pour $P_3(x) = 0$ (polynôme nul), le dictionnaire sera { 0 : 0 }

Ecrire une **fonction derivePolynome_Dict** qui prend en paramètre un dictionnaire représentant le polynôme (suivant le format précédent) et renvoie un dictionnaire des coefficients du polynôme dérivé.

```
derivePolynome_Dict ({0 : 2, 1 : 3, 4 : -5}) renvoie {0 : 3, 3 : -20}
```

Déterminer en fonction du degré n du polynôme les **complexités en temps et en espace** de **derivePolynome_Dict**.

Exercice 3 : Implémentation d'une table de hachage

L'objectif de cet exercice est de construire une table de hachage «à la main», afin de définir un équivalent des dictionnaires Python(dict).

Dans ce but, il faut d'abord disposer d'une fonction de hachage adaptée. C'est l'objet de la première partie.

Définitions de fonctions de hachage

Pour être adaptée à l'usage par une table de hachage, une fonction de hachage doit être : rapide,

- cohérente : pour une même clef, on obtient un même code,
- injective : pour des clefs différentes, on obtient des codes différents. Pour des codes identiques, les clefs sont nécessairement identiques. Dans le cas contraire, on obtient une collision qu'on cherche à minimiser. Comme la table de hachage est de dimension finie, les collisions sont inévitables. Donc l'injectivité est sacrifiée.
- uniformément répartie : pour des clefs qui se ressemblent, les codes obtenus doivent être très différents, ceci pour limiter les collisions.

En pré-traitement, on réalise l'encodage de la chaîne de caractère *ch* en nombre entier de la manière suivante :

$$\sum_{i=0}^{n-1} \text{ord}(\text{chaîne}[i]) * 256^i$$

Pour rappel :

`ord(c)`: renvoie le nombre entier représentant le code Unicode du caractère passé en paramètre.

Exemples : `ord('a')` renvoie le nombre entier 97
`ord('€')` (symbole euro) renvoie 8364.

1. Ecrire une **fonction `transforme_chaine_entier(chaîne)`** qui prend en paramètre une chaîne de caractères *chaîne* et retourne la valeur :

$$\sum_{i=0}^{n-1} \text{ord}(\text{chaîne}[i]) * 256^i$$

Exemples d'exécution :

```
print ( "test devient ", transforme_chaine_entier( "test" ) )
affiche test devient 1953719668
print ( "dico devient ", transforme_chaine_entier( "dico" ) )
affiche dico devient 1868786020
```

2. Dans un second temps, on cherche à compresser la valeur encodée dans l'intervalle des index possibles $[0, m - 1]$. Si *m* est la taille de la table de hachage, on peut choisir :

- d'utiliser simplement une division :
`hachage_d(nb, m)` → `nb % m`
- d'utiliser une multiplication et une division :
`hachage_mult(nb, m)` → `partieEntiere(m × (c * nb % 1)`

c ∈ $]0, 1[$ étant une constante réelle.

Le choix d'une fonction de hachage est délicat et il n'existe pas de méthode pour atteindre l'optimal.

Ecrire puis tester les fonctions `hachage_d` et `hachage_mult` en Python.

Pour faire vos tests de fonction, vous pouvez utiliser : ***m=4500*** pour la taille de la table de hachage
c=(√5-1)/2 pour la constante *c* de la fonction de hachage `hachage_mult`

Tester la fonction sur la liste de tuples générée par l'appel de la fonction précédente pour les capitales des pays, avec la fonction `hachage_d`.

[[, [,], [,], [,], [,], [,], [,], [,],

- Tester cette fonction pour obtenir la capitale d'un pays présent dans le fichier.

- Astuce : Vous pouvez utiliser la fonction Enumate:

Exercice 4 : Résultat de QCM

Dans le cadre d'un examen en ligne, un QCM a été réalisé.

Les réponses correctes au QCM sont stockées dans un dictionnaire nommé *reponses_valides*. Les clés sont des chaînes de caractères de la forme 'Q1'. Les valeurs possibles sont des chaînes de caractères correspondant aux quatre réponses 'a', 'b', 'c', 'd', 'e'.

Exemple :

```
reponses_valides = {'Q1': 'a', 'Q2': 'c', 'Q3': 'd', 'Q4': 'e', 'Q5': 'b', 'Q6': 'c'}
```

Les réponses données par les élèves sont stockées dans le dictionnaire *reponses_Eleves* dont voici un exemple possible :

```
{
  '110': {'Nom': 'Abadie', 'Prenom': 'Juline', 'Q1': 'b', 'Q2': 'a', 'Q3': 'd', 'Q4': 'a', 'Q5': 'b', 'Q6': 'b'},
  '111': {'Nom': 'Baron', 'Prenom': 'Lila', 'Q1': 'a', 'Q2': 'c', 'Q3': 'd', 'Q4': 'e', 'Q5': 'b', 'Q6': 'c'},
  '112': {'Nom': 'Charrez', 'Prenom': 'Ines', 'Q1': '', 'Q2': 'd', 'Q3': 'd', 'Q4': 'e', 'Q5': 'a', 'Q6': 'c'},
  '113': {'Nom': 'Clapotas', 'Prenom': 'Marc', 'Q1': 'a', 'Q2': 'c', 'Q3': 'd', 'Q4': 'e', 'Q5': 'e', 'Q6': 'b'}
}
```

Les clés de ce dictionnaire correspondent aux numéros des étudiants (ex. 110, 111, ...)

La valeur est un dictionnaire comportant Nom, Prenom et réponses aux questions. Lorsqu'une élève n'a pas répondu à une question, la clé (Q1, Q2, ...) sera présente mais sa valeur sera la chaîne vide.

La notation du QCM est la suivante : 3 points par réponse correcte, -1 point par réponse incorrecte et 0 s'il n'y a pas eu de réponse.

1. Ecrire la **fonction correction_QCM (reponses_Eleves, reponses_valides)** qui, à partir des dictionnaires *reponses_Eleves* et *reponses_valides* passés en paramètres, calcule le score de chaque élève et le stocke dans le dictionnaire *reponses_Eleves* (ajout d'une clé 'Score' et la valeur du score)

Exemple de dictionnaire résultant :

```
{
  '110': {'Nom': 'Abadie', 'Prenom': 'Juline', 'Q1': 'b', 'Q2': 'a', 'Q3': 'd', 'Q4': 'a', 'Q5': 'b', 'Q6': 'b', 'Score': 2},
  '111': {'Nom': 'Baron', 'Prenom': 'Lila', 'Q1': 'a', 'Q2': 'c', 'Q3': 'd', 'Q4': 'e', 'Q5': 'b', 'Q6': 'c', 'Score': 18},
  '112': {'Nom': 'Charrez', 'Prenom': 'Ines', 'Q1': '', 'Q2': 'd', 'Q3': 'd', 'Q4': 'e', 'Q5': 'a', 'Q6': 'c', 'Score': 10},
  '113': {'Nom': 'Clapotas', 'Prenom': 'Marc', 'Q1': 'a', 'Q2': 'c', 'Q3': 'd', 'Q4': 'e', 'Q5': 'e', 'Q6': 'b', 'Score': 10}
}
```

Cette fonction agit directement sur le contenu du dictionnaire *reponses_Eleves* (action avec effet de bord)

2. Proposer une **fonction correction_QCM_sansEB (reponses_Eleves, reponses_valides)** qui génère un dictionnaire résultant (donc sans effet de bord)
3. Les réponses des élèves ont été stockées de manière persistante dans le fichier *reponses.csv* ainsi structuré :


```

Numero, Nom, Prenom, Q1, Q2, Q3, Q4, Q5, Q6
0, reponse, reponse, a, c, d, e, b, c
110, Abadie, Juline, b, a, d, a, b, b
111, Baron, Lila, a, c, d, e, b, c
112, Charrez, Ines, , d, d, e, a, c
113, Clapotas, Marc, a, c, d, e, e, b
114, Darvis, John, a, b, c, b, b, c
115, Farton, Thibault, d, c, d, e, c, c
116, Galin, Ambre, a, c, d, e, b, c
117, Meuner, Chloé, a, c, d, e, b, c
118, Prante, Nina, b, a, b, e, b, a
119, Sarthe, Vanina, a, c, d, d, a, c
120, Zalu, Mehdi, b, a, c, e, b, d

```

Les 2 1^{ères} lignes du fichier sont particulières :

- La 1^{ère} ligne correspond au nom des clés du dictionnaire de chaque élève.
- La 2^{ème} ligne correspond aux réponses valides (les 3 1^{ères} colonnes sont inutiles pour nous)

Créer une **fonction `generation_dictionnaires(fichier)`** qui génère un dictionnaire au format de celui de `reponses_Eleves` et un dictionnaire au format de `reponses_valides`. Tester la génération sur le fichier `reponses.csv`.

Proposer 2 versions à cette fonction :

- Une qui traite le fichier comme un fichier texte quelconque
- Une qui traite le fichier comme un fichier csv en utilisant par exemple le module csv

Exemples d'utilisation :

<https://stackoverflow.com/questions/14091387/creating-a-dictionary-from-a-csv-file>

Exercice 5 : Anagrammes

Le but de cet exercice est de tester si deux mots sont des anagrammes. Cet algorithme classique se résout très bien avec les dictionnaires.

- 1- Une 1^{ère} approche est de parcourir le 1^{er} mot, et de compter le nombre d'occurrences de chaque lettre du mot. Puis, on réalise cette même opération sur le 2^{ème} mot et on regarde si les deux dictionnaires résultants sont égaux.

Écrire la fonction **`sont_anagrammes_v1(mot1, mot2)`** qui prend deux mots en paramètres et teste s'ils sont des anagrammes l'un de l'autre.

- 2- L'égalité entre deux dictionnaires s'écrit facilement mais est coûteuse. Une autre idée est pour le 2^{ème} mot, de le parcourir et d'enlever une occurrence de chaque lettre rencontrée dans l'alphabet du 1^{er}. On doit pouvoir détecter rapidement s'ils sont ou pas deux anagrammes.

Écrire la fonction **`sont_anagrammes_v2(mot1, mot2)`** suivant la 2^{ème} approche.

- 3- Créer une fonction récursive **`genere_Anagrammes`** qui génère la liste de tous les anagrammes (sans doublon) d'un mot passé en paramètre en utilisant un dictionnaire. La fonction prend d'autres paramètres que `mot`, à déterminer.

Exercice 6 : Algorithme de compression LZ78

Mettre en œuvre l'algorithme de compression LZ78.

1- Compression

Créer la fonction *compressionLZ78(texte)* qui effectue la compression du texte passé en paramètre et retourne le code, ainsi que le dictionnaire.

2- Décompression

L'algorithme de décompression fonctionne en sens inverse.

À partir de la liste alternée, appelée code, il faut reconstruire le dictionnaire au fur et à mesure.

Créer la fonction *decompressionLZ78(code, dico)* qui effectue la décompression.

Astuce : Pour faciliter la programmation, on peut inverser le dictionnaire utilisé pour la compression : les valeurs deviennent clés et les clés deviennent valeurs.