Dictionnaires

Rappels Python :
- ✗ `8 = 3` est interdit en python
- ✗ `s[2] = 'e'` (s étant une chaîne, par exemple 'abcde') est interdit en python
- ✓ `L[1] = 'e'` (L étant une liste, par exemple [0,0]) est valide : une liste est ***mutable*** (modifiable).

On distingue les objets ***mutables*** i.e. modifiables (liste, tableau numpy, ensemble, dictionnaire…) des objets ***non mutables*** (entier, flottant, booléen, chaîne, tuple …).

Dictionnaires – Partie 1 - TP

Objectifs ✔ :
- ✓ Définition d'un dictionnaire, clés, valeurs
- ✓ Création d'un dictionnaire, accès à un élément
- ✓ Parcours d'un dictionnaire
- ✓ Ajout, suppression dans un dictionnaire

TP (cf. site public) : construire un formulaire résumant les commandes à partir du notebook.

Un ***dictionnaire*** (ou encore ***table d'association***) est une structure de données associant un ensemble de ***clefs*** (keys) à un ensemble de ***valeurs*** (values).
Dans la suite, **C** désigne l'ensemble des clefs c et **V** l'ensemble des valeurs v.

Dictionnaires – Partie 2 – Implémentation

Objectif ✔ : implémentation (réalisation pratique du côté de la machine) des dictionnaires et problèmes posés par sa mise en œuvre.

Problématique

Une liste, un tuple, une chaîne de caractère, un tableau numpy, de longueur n, est indexé par les entiers *consécutifs* de 0 à n-1.
Associer un entier à une « case mémoire » semble naturel et simple.

Rappel de quelques complexités temporelles (simplifiées) :

	Liste python	Liste triée (coût du tri fusion : O($n$log(n))
Accès à un élément (L[i])	O(1)	O(1)
Recherche (e in L)	O(n)	O(log(n)) (recherche dichotomique)
Insertion	O(1)	O(1)
Suppression	O(1)	O(1)

Cependant, dans un dictionnaire les ***clés*** indexant les valeurs sont des objets non mutables ***quelconques***, il va donc falloir associer un entier à chaque objet-clé afin de réutiliser la structure de tableau/liste (indexé par cet entier) mais **le nombre de clés envisageable est *infini*…**

Les problèmes à résoudre pour implémenter la structure de dictionnaire sont donc :
- ✗ ***Associer à chaque clé un entier dans un intervalle donné de façon efficace*** ; on souhaite en réalité améliorer les performances du tableau précédent en réalisant les 4 opérations élémentaires listées ci-dessus avec une complexité temporelle en O(1).
- ✗ ***Gérer le fait que le nombre de clés est infini alors qu'un tableau est nécessairement de taille finie***.

Exercice 1 – Clés sous forme de chaînes de caractères – Un problème de taille…
Combien de chaînes contenant au plus 4 caractères peut-on construire avec les 26 lettres minuscules et les 26 lettres majuscules ?

Implémentation de la structure de dictionnaire

1 - Stratégie

Si n est la taille du dictionnaire, les entiers associés aux clés doivent être dans l'intervalle [0, n-1] pour que la structure de tableau puisse se ramener à une structure de type liste/tableau.

Il s'agit donc de trouver une fonction f : C → [0, n-1] associant un entier dans [0, n-1] aux différentes clés c ∈ C envisageables.

2 – Fonction de hachage – Associer un entier à tout objet

Il existe en python une fonction, appelée ***fonction de hachage*** notée ***hash***, associant à un objet quelconque o un entier ($-2^{63} \leq hash(o) \leq 2^{63}-1$ sur une machine 64 bits avec python).
La valeur entière $hash(o)$ est ***imprévisible*** (la fonction utilise une valeur « aléatoire »).

Exemples (si vous testez ces commandes, vous n'obtiendrez pas les mêmes valeurs) :

```
>>> hash('a')
7259379804136194288

>>> hash('dictionnaire python haché')
-7676901553335792521

>>> hash(3.14)              # La fonction hash peut être utilisée avec des nombres
322818021289917443

>>> hash((-1,8,0,5))       # La fonction hash peut être utilisée avec des tuples
-8870506748683291145

>>> hash([-1,8,0,5])
Traceback (most recent call last): TypeError: unhashable type: 'list'
```

La fonction hash de python renvoie un entier pour tous les objets non mutables (immuables) : int, float, bool, str, tuple… mais cet entier appartient à un intervalle très grand et sa valeur dans cet intervalle est imprévisible.

Remarque :

Si vous testez la fonction *hash*, vous ne trouverez pas les mêmes résultats !
La fonction de hachage utilise des variables « aléatoires » qui sont modifiées à chaque redémarrage du noyau python (les dictionnaires sont donc reconstruits à ce moment).

3 – Repliement des valeurs sur l'intervalle [0, *n*-1]

Il existe différentes fonctions de hachage (telles que la fonction *hash* de python), on note *hash* une fonction de hachage quelconque et *c* une clé quelconque.

La fonction $h : C \to [0, n\text{-}1]$ définie par $\boldsymbol{h(c) = hash(c) \bmod n}$ ($\boldsymbol{h(c) = hash(c) \,\%\, n}$ en python) est telle que $0 \leq h(c) \leq n\text{-}1$ (*hash*, fonction de hachage quelconque).
La fonction ***h*** est une ***fonction de hachage*** associant un entier **dans l'intervalle [0, *n*-1]** à une **clé quelconque *c***.
Un tableau *t* de taille *n* indexé par l'entier $h(c)$ (avec $0 \leq h(c) \leq n\text{-}1$) peut donc être utilisé pour stocker les associations (c, v) représentant un dictionnaire.
Un tel tableau est appelé ***table de hachage***.

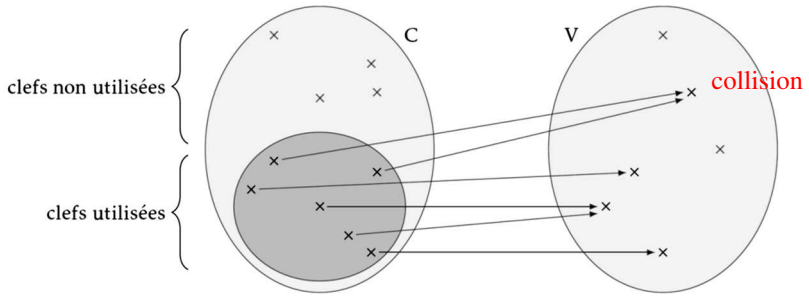
Rq : le résultat de $hash(c) \,\%\, n$ (reste dans la division Euclidienne de l'entier $hash(c)$ par n) est nécessairement dans l'intervalle [0, *n*-1].

4 – Collisions

Le problème réside dans le fait qu'on ne sait pas construire une fonction $hash(c)$ injective : il peut arriver que deux clés distinctes $c_1 \neq c_2$ soient associées à la même valeur $hash(c_1) = hasj(c_2)$. Même si la fonction *hash* était injective, la fonction $h(c) = hash(c) \,\%\, n$ ne le serait pas (par exemple, 26 % 5 et 16 % 5 renvoient 1).

Un couple de clés distinctes (c_1, c_2) avec $c_1 \neq c_2$ et $h(c_1) = h(c_2)$ est appelé une ***collision***.

Il s'agit d'un problème majeur, il faut donc mettre au point des solutions pour remédier à ces collisions.
Un dictionnaire est donc un sous-ensemble T de C×V tel que pour toute clef $c \in C$ il existe *au plus* un élément $v \in V$ tel que $(c, v) \in T$. Les éléments de T sont appelés des *associations*.



Une fonction de hachage doit :
- être facile à calculer (complexité temporelle en O(1)) ;
- avoir une distribution la plus uniforme possible afin d'éviter les collisions.

5 – Résolution des collisions

1. Résolution des collisions par chaînage

Une solution possible consiste à stocker les valeurs dont les clefs sont rentrées en collision dans un même « paquet ». Si la répartition entre les différents paquets est équilibrée, chaque paquet ne contiendra qu'un petit nombre de valeurs, et on retrouvera rapidement la valeur associée à une clef en la cherchant dans son paquet.

La figure ci-dessous illustre cette solution pour un tableau de taille $n = 7$ avec :
$h(c_1) = h(c_4) = 6$ (collision pour les clés c_1 et c_4)
$h(c_3) = h(c_5) = h(c_6) = 1$ (collision pour les clés c_3, c_5 et c_6)

Index h(c)	0	1	2	3	4	4	5	6	7
Associations (c, v)		(c_3,v_3) (c_5,v_5) (c_6,v_6)		(c_2,v_2)				(c_1,v_1) (c_4,v_4)	

Pour trouver la valeur associée à une clef *c* présente dans le dictionnaire, on procède alors en deux temps :
– on calcule $h(c) = hash(c) \bmod n$ pour déterminer l'emplacement du tableau où se trouve le paquet contenant l'association recherchée ;
– on procède à une recherche dans le paquet pour trouver cette association.

Le chaînage présente l'avantage de pouvoir utiliser un nombre *k* de clés supérieur au nombre *n* de cases du tableau.
Si on note $\alpha = k/n$ le taux de remplissage du dictionnaire, les paquets auront une taille moyenne de α (sous une hypothèse de hachage uniforme) et la recherche d'une association dans le dictionnaire utilisera un nombre de comparaison en moyenne de l'ordre de α qui doit rester faible sous peine de perdre en performance (temps de recherche).

Exercice 2 – Gestion des collisions par chaînage
On dispose d'un tableau de taille *n* (appelé table de hachage) dont les collisions sont résolues par chaînage.
Dans cet exemple, la table de hachage est de dimension $n = 9$.

a) Pour constituer la table de hachage (à compléter ci-dessous), on utilise la fonction $h : c \mapsto hash(c) \bmod n$ avec les valeurs $hash(c)$ ci-dessous pour chaque clé *c*.

Clés	c_0	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	c_9
hash(c)	5	28	19	15	20	33	12	17	10	18
h(c)										

b) Compléter la table ci-dessous associant un entier à chaque couple (c, v).

Index h(c)	0	1	2	3	4	5	6	7	8
Associations (c, v)									

2. Résolution des collisions par adressage ouvert

Une autre solution consiste, *en cas de collision* (si la case $hash(c)$ mod m est occupée) à chercher un emplacement libre dans lequel déposer la nouvelle association.
La recherche d'un emplacement libre porte le nom de ***sondage***.
– le sondage *linéaire* consiste à chercher une place libre en testant successivement les cases d'indices ($hash(c)$+1) mod m, ($hash(c)$+2) mod m, ($hash(c)$+3) mod m, … jusqu'à trouver un emplacement libre.
– le sondage *quadratique* procède de même mais en sondant les cases d'indices ($hash(c)$+1) mod m, ($hash(c)$+1+2) mod m, ($hash(c)$+1+2+3) mod m, …

Exercice 3 – Sondage linéaire, sondage quadratique

On considère la table associée à un dictionnaire de taille $n = 9$ (table de hachage à 9 cases) partiellement remplie :

Index	0	1	2	3	4	5	6	7	8
Associations (c, v)	(c_5,v_5)		(c_1,v_1)	(c_4,v_4)	(c_3,v_3)			(c_2,v_2)	

Dans quelle case de la table l'association (c_6, v_6) telle que $hash(c_6) = 3$ sera-t-elle stockée avec un adressage ouvert :
- par sondage linéaire ?
- par sondage quadratique ?

L'inconvénient du sondage linéaire est de former des « agrégats », autrement dit de longues successions de cases contigües occupées, qui nuisent à la répartition uniforme recherchée.

Un adressage ouvert exige que le nombre k de clefs soit inférieur à la taille n de la table puisque chaque case est associée à une unique clé.
Donc lorsque le taux de remplissage $\alpha = k/n$ se rapproche de 1, il devient de plus en plus difficile de trouver un emplacement vide : si $\alpha = 0{,}5$ un ajout nécessite en moyenne deux sondages et dix sondages lorsque $\alpha = 0{,}9$.
Lorsque le taux de remplissage α devient trop grand il est nécessaire de créer une table plus grande (la taille est en général doublée) pour préserver les performances (mais cet agrandissement de la table à un coût en temps et en espace mémoire).

Exercice 4 – Gestion des collisions par sondage linéaire

On dispose d'un tableau de taille n dont les collisions sont résolues par adressage ouvert à l'aide d'un sondage linéaire.
Exécuter manuellement l'algorithme d'insertion dans le tableau pour $n = 9$ et $h : c \mapsto hash(c) \mod n$ avec les valeurs $hash(c)$ indiquées dans le tableau ci-dessous.

Clés	c_0	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8
hash(c)	5	28	19	15	20	33	12	17	10
hash(c)%9									
(hash(c)+1)%9									
(hash(c)+2)%9									

Si nécessaire

Index	0	1	2	3	4	5	6	7	8
Associations (c, v)									

Quelques utilisations des tables de hachage

- ✓ Stockage des mots de passe : pour des raisons de sécurité, c'est le mot de passe haché qui doit être stocké par un programme (et non le mot de passe en clair !) car il est très difficile d'inverser la fonction de hachage (retrouver le mot de passe à partir de la valeur hachée).
- ✓ Vérification de l'intégrité d'un fichier après téléchargement : cf. MD_5 sur Wikipédia.
- ✓ Utilisation pour l'accélération des pages web via un « cache » (c'est ce cache qu'il faut effacer pour réinitialiser une page).
- ✓ Compression de textes via les algorithmes LZ_{77} et LZ_{78} : cf. Wikipédia.

Exercice 5 – Implémentation d'un dictionnaire

L'objectif est ici de simuler un dictionnaire de taille n grâce à une table de hachage sous forme de liste contenant des tuples de la forme (clé, valeur).
La taille n du dictionnaire est considérée comme une variable globale (définie au niveau du programme principal) donc accessible dans toutes les fonctions.

Savoir-faire – Documenter une fonction à l'aide d'un « docstring » (commentaire).

5.1. Ecrire une fonction h(c) renvoyant $hash(c)$ mod n.

```
def h(c):
    """
    Fonction de hachage dans l'intervalle [0, n-1].

    Paramètres :
        c : str
    Renvoie :
        int
    """
    # n = taille de la table (variable globale) – Déclaration facultative
    global n
    # Calcul de h(c)

```

5.2. Ecrire une fonction creer_dic(n) renvoyant une liste comportant n valeurs None.

```
def creer_dic(n):
    """
    Création d'un dictionnaire de taille n (valeurs None).

    Paramètres :
        n : int
    Renvoie :
        dict
    """

```

5.3. On suppose qu'un dictionnaire d a été créé grâce à l'instruction d = creer_dic(n) (techniquement, d est ici simulé par une liste).
Soit A une liste de tuples de la forme A = [(c1, v1), (c2, v2),...] de longueur strictement inférieure à n.
Chacun des tuples (c_i, v_i) correspond à une association à ajouter au dictionnaire d.
Ecrire une fonction ajout_dic0(d, A) ajoutant les associations listées dans A au dictionnaire d.
Cette fonction appellera la fonction h(c) afin de déterminer l'emplacement de l'association dans le dictionnaire (liste d) et ne cherchera pas à gérer les éventuelles collisions.

```
def ajout_dic0(d, A):
    """
    Ajoute les associations de la liste A au dictionnaire d sans vérification des collisions.

    Paramètres :
        d : dictionnaire (existant)
        A : liste de tuple où chaque tuple est de la forme (clé de type str, valeur)
    Renvoie :
        None (modifie le dictionnaire d, instruction return inutile)
    """
    global n            # Facultatif
    # Parcours des couples pour la constitution du pseudo-dictionnaire


```

5.4. Le programme principal est donné ci-dessous.

```
# _____Programme principal_____

# Taille de la table (variable globale)
n = 8

# Initialisation et remplissage du dictionnaire
dico1 = creer_dic(n)
ajout_dic0(dico1, [ ('i', 19), ('n', 14), ('f', 6), ('o', 15) ] )
print('Dictionnaire : ', dico1)

# Accès à une association via la clé
print("Test avec la clé 'i': ", dico1[h('i')])
```

Une exécution du code précédant fournit les résultats suivants :

Dictionnaire : [None, None, ('i', 19), None, ('f', 6), ('o', 15), None, None]
Test avec la clé 'i' : ('i', 19)

Comment constate-t-on qu'il y a eu une collision ?

5.5. Ecrire une fonction *hs*(cle, i) renvoyant la valeur hachée par sondage linéaire (cette fonction remplace *h*(cle) dans la suite).

5.6. Ecrire une fonction *ajout_dic*(d, A) analogue à ajout_*dic*(d, A) : la boucle for contiendra en particulier une boucle recherchant un emplacement disponible dans la table. On supposera que le dictionnaire n'est jamais complètement rempli.

Il est possible de vérifier que le problème de collision est résolu en exécutant le code (redémarrer le noyau jusqu'à ce qu'une collision se produise si nécessaire).

DM dictionnaires

Cf. lien notebook site public.