Plan du cours

Création d'un objet de type dictionnaire

- Les dictionnaires en Python sont des objets permettant de stocker des données sous la forme :

  "clef" : valeur associé

- On créé un objet dictionnaire en Python par la syntaxe :

```
>>> dico = {"maths" : 9.5 , "physique" : 8, 11 : "essai"}
```

  ce qui créé un objet de type `dict` contenant les couples `"maths": 9.5` ...

- Les premiers éléments de ces couples sont appelés "clés" du dictionnaire créé

  - les clefs ne doivent pas être des objets mutables.
  - les clefs peuvent néanmoins être de différents type : entier, flottant, chaine de caractères ... qui sont des objets non mutables.
  - on ne peut pas utiliser de liste comme clef pour la même raison : elles sont mutables.

```
In [6]: dic = {[3,4] : "essai"}
TypeError: unhashable type: 'list'
```

  - on peut par contre utiliser des tuples (non mutables) mais à condition qu'ils ne contiennent pas d'objets mutables.

Accès clefs d'un dictionnaire I

- Accès aux clefs d'un dictionnaire :
  - On peut récupérer les clefs d'un dictionnaire par la méthode `keys` :

```
>>>  dico.keys()
dict_keys(['maths', 'physique', 11])
```

  - on obtient alors un objet itérable c'est à dire pouvant être parcouru dans un structure de boucle :

```
In[12] for clef in dico.keys() :
           print(clef)
maths
physique
11
```

  - On peut également transformer cet objet en liste :

```
In [17]: list(dico.keys())
Out[17]: ['maths', 'physique ']
```

Accès valeurs d'un dictionnaire I

1. Les clés sont utilisables comme des index dans les listes

```
>>>  dico["maths"]
9.5
```

Questions 1.

1. Écrire une fonction python qui prend en entré un dictionnaire et renvoie sous forme de liste toutes les valeurs qu'il contient.

Accès aux valeurs

- cette fonctionnalité est bien sûr déjà présente en Python de manière native :

```
In [4]: dico.values()
Out[4]: dict_values([9.5, 8, 'essai'])

In [5]: list(dico.values())
Out[5]: [9.5, 8, 'essai']
```

- L'objet `dico.values()` étant également itérable.

Modification des dictionnaires

Les dictionnaires sont mutables, on peut modifier les élements, ajouter des éléments, supprimer des éléments.

- Modifier la valeur associée à une clef :

```
In [8]: dico['maths'] = 9
In [9]: dico
Out[9]: {'maths': 9, 'physique': 8, 11: 'essai'}
```

- Ajouter un nouvel élement. Il suffit d'affecter à une nouvelle clef du dictionnaire une valeur :

```
In [11]: dico['anglais'] = 2 #ajout d'une clef anglais/valeur 2
In [12]: dico
Out[12]: {'maths': 9, 'physique': 8, 11: 'essai', 'anglais': 2}
```

- Retirer un élément.

```
In [13]: del dico[11] #suppression de l'élément de clef 11
In [14]: dico
Out[14]: {'maths': 9, 'physique': 8, 'anglais': 2}
```

Recherche dans un dictionnaire

1. On peut tester si une clef est présente dans un dictionnaire avec l'opérateur `in`

```
In [1]: notes = {"Henri" : 9.5, "Laurent" : 11}
In [2]: 'Henri' in notes
Out[2]: True
```

2. Et de même on peut tester la présence d'une valeur mais pas avec la syntaxe précédente ::

```
In [3]: 11 in notes.values()
Out[3]: True
```

Questions 2. Écrire une fonction python qui prend en entrée une chaine de caractères et renvoie un dictionnaire dont les clefs sont les lettres trouvées et les valeurs le nombre d'occurence des lettres dans la chaîne.

Questions 3. Écrire une fonction python qui prend en entrée une chaine de caractères et renvoie un dictionnaire dont les clefs sont les lettres trouvées et les valeurs le nombre d'occurence des lettres dans la chaîne.

```
def comptage(texte) :
    resultat = {} #création d'un dictionnaire vide
    for caractere in texte :
        if caractere in resultat :
            resultat[caractere] += 1
        else :
            resultat[caractere] = 1
        return resultat
```

Itération sur des dictionnaires

- Comme pour les listes, les tuples et les chaines, les dictionnaires peuvent être parcourus dans un structure de boucle.

```
notes = {"Henri" : 9.5, "Laurent" : 11}
for nom in notes :
    print(nom)
```

va afficher les clefs présentes dans `notes`.

Itération sur des dictionnaires

- Comme pour les listes, les tuples et les chaines, les dictionnaires peuvent être parcourus dans un structure de boucle.

```
notes = {"Henri" : 9.5, "Laurent" : 11}
for nom in notes :
    print(nom)
```

va afficher les clefs présentes dans `notes`.

- Il en est de même avec pour les valeurs :

```
notes = {"Henri" : 9.5, "Laurent" : 11}
for val in notes.values :
    print(val)
```

Itération sur des dictionnaires

- Comme pour les listes, les tuples et les chaines, les dictionnaires peuvent être parcourus dans un structure de boucle.

```
notes = {"Henri" : 9.5, "Laurent" : 11}
for nom in notes :
    print(nom)
```

va afficher les clefs présentes dans `notes`.

- Il en est de même avec pour les valeurs :

```
notes = {"Henri" : 9.5, "Laurent" : 11}
for val in notes.values :
    print(val)
```

- On peut enfin parcourir les couples `clef,valeur` avec `notes.items`

```
for element in notes.items()  :
    print(element)
```

affichera :

```
('Henri', 9.5)
('Laurent', 11)
```

Questions 4. Écrire une fonction Python qui prend en entrée un dictionnaire et une valeur et renvoie la liste des clefs correspondant à cette valeur (cette liste étant éventuellement vide).

Questions 5. Écrire une fonction Python qui prend en entrée un dictionnaire et une valeur et renvoie la liste des clefs correspondant à cette valeur (cette liste étant éventuellement vide).

```
def fct(dico,valeur) :
    tmp = []
    for clef in dico :
        if dico[clef] == valeur :
            tmp.append(clef)
    return tmp
```

Vitesse de traitement/Complexité

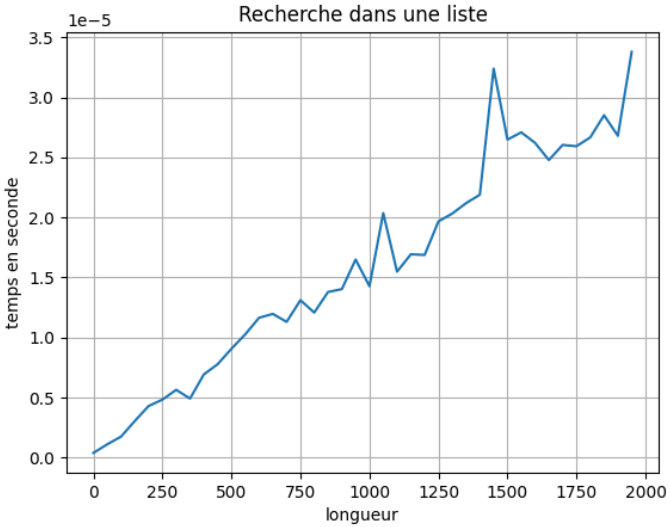
Questions 6. Effectuer des tests de temps pour la commande `in` avec des longueurs d'objets variant de 100 à 1000 dans les deux cas suivants :

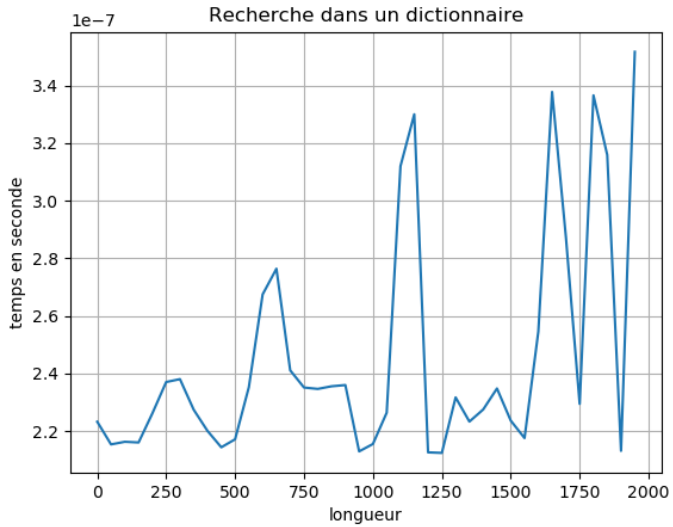
- dictionnaire du type `dico = 1:1, 2:2, ... , N:N` en recherchant si la clef `N+1` est présente.
- liste du type `L = [1,2,.. ,N]` en recherchant si la valeur `N+1` est présente.

Commenter.

Vitesse de traitement/Complexité

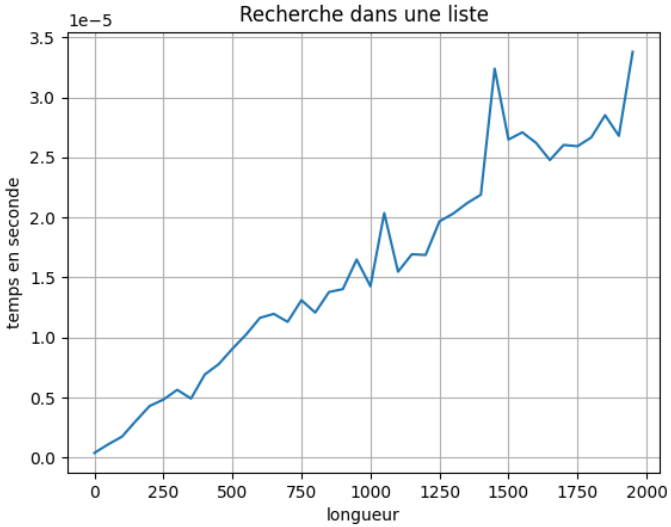
- Voici les deux graphiques obtenus en mesurant des temps de recherche d'un élément dans une liste alors que l'élément n'y est pas, puis le même test avec un dictionnaire.

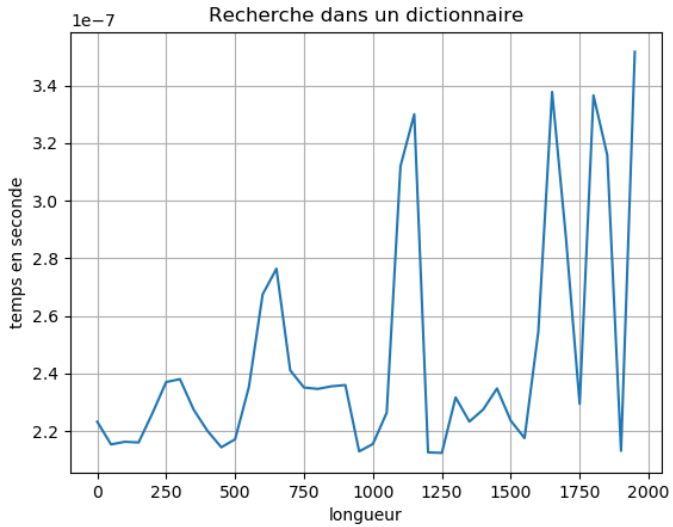




Vitesse de traitement/Complexité

- Voici les deux graphiques obtenus en mesurant des temps de recherche d'un élément dans une liste alors que l'élément n'y est pas, puis le même test avec un dictionnaire.





- Recherche d'un élément (dans le pire des cas):
  - pour les listes : complexité $O(N)$ (linéaire) où N est la longueur de la liste.
  - pour les dictionnaire : en temps constant par rapport à la taille de celui-ci.

Inconvénients

1. Les dictionnaires ne sont pas des structures de données ordonnées on ne peut donc pas faire de slicing, ni de tri par ordre croissant par exemple.
2. Les dictionnaire consomme plus de mémoire.

Questions 7. Utiliser la structure de dictionnaire pour écrire une fonction qui prend en entrée une liste de nombres ou de chaînes de caractères et qui renvoie une liste formée des mêmes éléments mais sans doublons.

Questions 8. On suppose qu'on dispose d'un dictionnaire du type :

```
dico = {(nom,prénom) : {'age':a ,'taille':t, 'poids':p }, ....}
```

dressé à partir d'une liste de personnes. `a,t,p` étant des entiers et `nom,prénom` des chaînes de caracttères.
Écrire un script Python qui permet d'extraire de ce dictionnaire les noms de toutes les personnes dont la taille est supérieur à une valeur `x` qui pourra être fixée dans le script.

Questions 9. Utiliser la structure de dictionnaire pour programmer une fonction `taxicab(n)` prenant en entrée un entier naturel n et renvoyant le premier entier naturel qui s'écrit de `n` manières comme une somme de deux cubes d'entier naturels non nul

1. Par exemple le nombre $9 = 1^3 + 2^3$ ne s'écrit que d'une seule façon comme somme de deux cubes d' entiers naturels (à l'ordre près).

Questions 10. Utiliser la structure de dictionnaire pour programmer une fonction `taxicab(n)` prenant en entrée un entier naturel n et renvoyant le premier entier naturel qui s'écrit de `n` manières comme une somme de deux cubes d'entier naturels non nul

1. Par exemple le nombre $9 = 1^3 + 2^3$ ne s'écrit que d'une seule façon comme somme de deux cubes d' entiers naturels (à l'ordre près).
2. De même pour $16 = 2^3 + 2^3$
3. ... et ainsi de suite jusqu'au fameux nombre `taxicab(2)` qui sera le premier à avoir deux décompositions en somme de cubes d'entier naturels non nuls.

Retour sur les suites récurrentes

Questions 11.

1. - Écrire une fonction Python récursive `suite` qui prend en entrée les paramètres `a,b,u0,u1,n` et qui calcule le terme de rang n de la suite récurrente $u_{n+2} = au_{n+1} + bu_n$ à partir des valeurs initiales u_0, u_1.
   - Rappeler pourquoi cette fonction n'est pas efficace.

Retour sur les suites récurrentes

Questions 12.

1.
   - Écrire une fonction Python récursive `suite` qui prend en entrée les paramètres `a,b,u0,u1,n` et qui calcule le terme de rang n de la suite récurrente $u_{n+2} = au_{n+1} + bu_n$ à partir des valeurs initiales u_0, u_1.
   - Rappeler pourquoi cette fonction n'est pas efficace.
2. Écrire une fonction récursive `suite_memo` effectuant la même tache mais en utilisant un dictionnaire pour stocker les résultats déjà calculés et éviter leur recalcul. Consignes :
   - La fonction prend en entrée les coefficients a, b
   - les valeurs initiables sous forme d'un dictionnaire $dico = \{0 : u0, 1 : u1\}$
   - l'indice n du terme qu'on veut calculer.
3. Comparer les temps d'exécution.

Questions 13. Écrire une fonction `reverse` qui prend en entrée un dictionnaire formée de clefs et de valeurs et qui renvoie le dictionnaire dont les clefs sont les valeurs du premier dictionnaire et les valeurs la liste des clefs correspondantes dans le premier dictionnaire. Exemple :

```
{ 2:'a', 3:'a', 7:'x'}
```

devient :

```
{'a':[2,3] , 'x':[7]}
```