Exercice 2

March 15, 2018

1 Analyse reproductible en Python Exercice 2 - Séance 1

Auteur : Philippe Tu BDTN - Promotion 2019 Date de création : 10 / 03 / 2018 Références utilisées pour écrire ce notebook : http://bigocheatsheet.com/ (Un site qui décrit la complexité de temps et d'espace pour les structures de données connues) https://www.ics.uci.edu/~pattis/ICS-33/lectures/complexitypython.txt (Une page qui explique la complexité des fonctions utilisées pour les structures de données de Python)

2 Les Lists :sont des tableaux extensibles, on peut y mettre tout type de variable de Python.

Complexité de temps :

Les + : O(1) pour l'accès

Les - : O(n) pour la recherche, l'insertion, suppression.

Complexité d'espace (mémoire) : O(n) Choix de structure de donnée : C'est une structure de donnée qui ne souffre pas de gros défauts, cependant ce n'est pas la structure de donnée la plus intéréssante, d'autres structures de données ont une complexité O(1) pour des opérations d'insertion, de suppression (ex : Stack, Queue, Single-linked List)

Un exemple pour illustrer la linéarité O(n) de l'insertion avec le code qui suit :

```
In [36]: import random as rd
         from timeit import default_timer as timer
         #Dans cet exemple, nous allons mettre en évidence la complexité O(n) de l'insertion
         from timeit import default_timer as timer #timeit est une librarie qui inclut des fon
         def testElements(nbrElement):
             start = timer()  #Début de mesure de temps
             liste=[]
             for number in range(0,nbrElement):
                 liste.append(rd.randint(0,9))
             end = timer()    #Fin de mesure de temps
             return end-start
         print("Pour 100 élements")
         testElements(100)   #Le temps de processing
```

```
Pour 100 élements
```

```
Out[36]: 0.00015104006433830364
```

```
In [37]: print("Pour 1000 élements")
         testElements(1000)#Le temps de processing
```

```
Pour 1000 élements
```

```
Out[37]: 0.0022762676378533797
```

```
In [88]: print("Pour 100000 élements")
         testElements(1000000)#Le temps de processing
```

```
Pour 100000 élements
```

```
Out[88]: 0.1314526427529472
```

On constate bien une linéarité du temps de calcul pour l'opération d'insertion, O(n) en insertion est donc vrai.

3 Les tuples

Ce sont des lists dont les élèments indexés sont immutables (que l'on ne peut plus modifier). La déclaration d'un tuple se fait via des crochets [] et une liste se déclare via des accolades {}. Les tuples n'ont pas de fonction append ni extend, on ne peut pas retirer d'élements (via pop ou remove). -Les tuples sont un peu plus rapide que les lists car ils contiennent des valeurs immutables. -Les tuples sont dont utilisés lorsque les variables à indexer seront immutables.

En terme de complexité de temps et d'espace, les tuples et les lists sont très similaires : Les + : O(1) pour l'accès

Les - : O(n) pour la recherche, l'insertion, suppression

Faisons un essai sur l'acces des elements d'un tuple :

```
In [135]: def tupleTestAccess(nbrElementDivDix):
              start = timer()
              tuple = (1,2,3,4,5,6,7,8,9,10)*nbrElementDivDix
              print(tuple)
              end = timer()
              return end-start
          tupleTestAccess(10) # Pour 100 elements
```

```
(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
```

```
Out[135]: 0.00011392004853405524
```

```
In [134]: tupleTestAccess(100) # Avec 10 fois plus d'élèments, le temps de calcul est juste 4
```

```
(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
```

```
Out[134]: 0.0004800002043339191
```

4 Les dictionnaires

Un dictionnaire est un objet python qui permet d'associer des clés (keys) et des valeurs (values). En terme de complexité : l'accession , l'insertion et la suppression sont en O(1) en moyenne. Au pire des cas (valeurs non arrangées par exemple), ces opérations sont tous les trois en complexité O(n).

Exemple dans l'insertion de données :

```
In [38]: def testDictionaryInsertion(nbrInsert):
             start = timer()
             dictionary ={}
             for k in range(1,nbrInsert):
                 dictionary.update({rd.randint(0,9):rd.randint(0,9)})
             end = timer()
             return end-start
         testDictionaryInsertion(1000)

Out[38]: 0.0048017087153766624

In [39]: testDictionaryInsertion(100000)

Out[39]: 0.29499020586240476
```

Les timings sont linéaires comme le montrent les résultats

5 Les Sets

Les sets sont des listes de données non ordonnées (non indexées), les sets ne peuvent pas contenir de doublons (donc on ne trouve un élèment 0 ou 1 fois seulement), les élèments d'un set sont hachables et pas dans les lists et dictionnaires. Les sets utilisent le hash des élèments pour l'accession par exemple.

En terme de complexité, les sets bénéficient d'O(1) sur de nombreuses opérations dufait de ces caractéristiques : Les + : O(1) pour l'insertion, suppression Les - : Union, Intersection, Difference, Difference symétrique sont à O(n+t) tel que n : nombre d'élèment du premier set et t : nombre d'élèment du deuxième set.

Faisons un test sur l'insertion :

```
In [57]: def testInsertionSet(nbrIteration):
             creationList=[4,6]
             randomSet = set(creationList)
             start = timer()
             for value in range(1,nbrIteration):
                 set.update(set([rd.randint(0,9)]))
             end = timer()
             return end-start
         testInsertionSet(1000)

Out[57]: 0.0017689607548163622
```

```
In [58]: testInsertionSet(100000)
```

```
Out[58]: 0.18870664051496533
```

L'insertion est linéaire.