comparatifStructures

January 1, 2017

Les structures de données en Python

1 Introduction

Il existe différents types de structures de données en Python.

Nous allons étudier ici : - les listes - les tuples - les dictionnaires - les sets

2 Présentation de 4 structures de données

2.1 Les listes

La liste est une structure de donnée permettant de stocker des éléments. Ceux-ci sont indéxés, le premier élément se trouve à l'indice 0, le deuxième à l'indice 1 etc. Les éléments d'une liste ne sont pas forcément du même type. Les listes pevent être tronquées et concaténées.

Avantages :

- on peut y stocker n'importe quel type d'élements - de nombreuses fonctions permettent de manipuler les listes

Inconvénients : - il faut connaitre la position d'un élément pour y accéder - la suppression d'un élément prend beaucoups de temps car tous les éléments suivants on besoin d'être décalés

Exemple :

Un exemple d'utilisation des listes est le "stack" il consiste à stocker des objets dans une liste au fur et à mesure et à traiter le dernier objet de la liste lors des phases de traitement, c'est le modèle "last-in, first-out".

```
In [1]: stack = [3, 4, 5]
        stack.append(6)
        stack.append(7)
        print(stack)

        print(stack.pop())
        print(stack)

        print(stack.pop())
        print(stack.pop())
        print(stack)
```

```
[3, 4, 5, 6, 7]
7
[3, 4, 5, 6]
6
5
[3, 4]
```

2.2 Les tuples

Un tuple est une liste de valeurs immuable. Sa valeur ne peut pas être changée une fois instanciée. Les tuples peuvent être imbriqués ou combinés pour créer d'autres tuples avec les opérateurs (','), ('+') et ('*').

Avantages :
- on peut y stocker n'importe quel type d'élements - un tuple est immuable, ce qui garanti l'intégrité des données

Inconvénients : - un tuple ne peut pas être modifié

Exemple :

Un exemple d'utilisation du tuple pourrait être de stocker une collection d'éléments invariants pendant tout le programme afin de s'assurer de leur intégrité. On peut par exemple stocker des variables caractérisant la machine sur laquelle le programme est éxécuté ainsi que le date d'execution :

```
In [2]: import datetime
        import platform

        detailsMachine = (platform.platform(), platform.machine(), datetime.date.to
        print(detailsMachine)
```

```
('Windows-10-10.0.14393-SP0', 'AMD64', datetime.date(2017, 1, 1))
```

2.3 Les dictionnaires

Un dictionnaire est une collection d'objet fonctionnant sur un système clé-valeur. Une clé est unique et peut être n'importe quel type de variable immuable : un entier, une string, un tuple...

Avantages :
- on peut y stocker n'importe quel type d'élements - la suppression d'un élément est plus rapide que pour une liste puisque les données ne possèdent pas d'index

Inconvénients : - il est nécessaire de créer une clé unique à l'ajout d'une nouvelle entrée - les éléments ne sont pas indéxés

Exemple :

Un exemple d'utilisation du dictionnaire serait par exemple de stocker l'age relatif à une personne.

```
In [3]: ages = {'bob': 22, 'tim':34}
        print(ages)
        ages['tom'] = 20
        print(ages)
```

```
        del ages['bob']
        print(ages)
```

```
{'tim': 34, 'bob': 22}
{'tom': 20, 'tim': 34, 'bob': 22}
{'tom': 20, 'tim': 34}
```

2.4 Les sets

Un set est une collection désordonnée et ne contenant chaque élément qu'une seule fois. On ne peut pas modifier un élément d'un set, seulement en ajouter ou en retirer.

Les sets peuvent être comparés entre eux : on peut ainsi connaitre les éléments dans un set a qui ne sont pas dans un autre set b, ou bien les éléments en commun entre a et b.

Les sets ne sont donc pas appropriés pour le simple stockage de valeurs, mais plus pour les manipuler.

Avantages :
- on peut y stocker n'importe quel type d'élements - on peut faire des opération sur des sets pour les comparer

Inconvénients : - convient peu pour le simple stockage de données

Exemple :

Un exemple d'utilisation des sets serait par exemple de comparer deux sets contenant des caractères et de récupérer seulement les caractères communs aux deux sets.

```
In [4]: set1 = set('azertyuiopmlkjhgfdsq')
        set2 = set('poiuytreza')
        set3 = set1 & set2
        print(set1)
        print(set2)
        print(set3)
```

```
{'y', 'i', 'z', 'k', 'r', 'l', 'g', 'q', 't', 'o', 'u', 'p', 'm', 'f', 'h', 'd', 's
{'a', 'y', 'i', 'z', 'r', 'o', 't', 'u', 'p', 'e'}
{'y', 'i', 'z', 'r', 'o', 't', 'u', 'p', 'a', 'e'}
```

3 Performances

Afin d'évaluer les performances des différents types étudiés, je vais : - instancier chaque type de strucuture avec 1 000 000 d'entrées - accéder à 10 000 éléments de la structure - supprimer 10 000 éléments de la structure (sauf pour le tuple…) - insérer 10 000 éléments dans la structure (sauf pour le tuple…)

3.1 Création

```
In [5]: import time

        start_time = time.clock()
```

```
        myList = list(x for x in range(1000000))
        print("--- %s seconds to create the list ---" % (time.clock() - start_time)

        start_time = time.clock()
        myTuple = tuple(x for x in range(1000000))
        print("--- %s seconds to create the tuple ---" % (time.clock() - start_time

        start_time = time.clock()
        myDictionary = {x: x for x in range(1000000)}
        print("--- %s seconds to create the dictionary ---" % (time.clock() - start

        start_time = time.clock()
        mySet = set(x for x in range(1000000))
        print("--- %s seconds to create the set ---" % (time.clock() - start_time))
```

```
--- 0.09130350686056242 seconds to create the list ---
--- 0.0912691645745504 seconds to create the tuple ---
--- 0.09499234206495824 seconds to create the dictionary ---
--- 0.12328564887183613 seconds to create the set ---
```

3.2 Accès aux données

```
In [6]: import numpy

        # Creating a list of all the testing indexes
        indexes = list(map(lambda x: int(round(x)), numpy.linspace(0, 999999, 10000

        start_time = time.clock()
        for index in indexes:
            myList[index]
        print("--- %s seconds to acces elements of the list ---" % (time.clock() -

        start_time = time.clock()
        for index in indexes:
            myTuple[index]
        print("--- %s seconds to acces element of the tuple ---" % (time.clock() -

        start_time = time.clock()
        for index in indexes:
            myDictionary[index]
        print("--- %s seconds to access elements of the dictionary ---" % (time.clo

        start_time = time.clock()
        for index in indexes:
            index in mySet
        print("--- %s seconds to access element of the set ---" % (time.clock() - s
```

```
--- 0.001921983799914706 seconds to acces elements of the list ---
```

```
--- 0.0018256675035132064 seconds to acces element of the tuple ---
--- 0.00272133011226372 seconds to access elements of the dictionary ---
--- 0.0023096174190391228 seconds to access element of the set ---
```

L'accès au éléments des différentes structures prend un temps équivalent.

3.3 Suppression

```
In [7]: # I remove 10000 elements from indexes for the loop because i will delete e
        indexes2 = list(map(lambda x: int(round(x)), numpy.linspace(0, 999999-10000

        start_time = time.clock()
        for index in indexes2:
            del myList[index]
        print("--- %s seconds to delete elements from the list ---" % (time.clock()

        start_time = time.clock()
        for index in indexes2:
            del myDictionary[index]
        print("--- %s seconds to delete elements from the dictionary ---" % (time.c

        start_time = time.clock()
        for index in indexes2:
            mySet.remove(index)
        print("--- %s seconds to delete element from the set ---" % (time.clock() -

--- 3.489939683893073 seconds to delete elements from the list ---
--- 0.003850678161463783 seconds to delete elements from the dictionary ---
--- 0.0033110700582632546 seconds to delete element from the set ---
```

On peut voir dans cet exemple que les éléments de la liste prennent plus de temps à être supprimés. Cela est du a fait qu'a chaque suppression, tous les éléments suivant celui qui est supprimé sont décalés.

3.4 Insertion

```
In [8]: start_time = time.clock()
        for index in indexes2:
            myList.append(index)
        print("--- %s seconds to add elements to the list ---" % (time.clock() - st

        start_time = time.clock()
        for index in indexes2:
            myDictionary[index] = index
        print("--- %s seconds to add elements to the dictionary ---" % (time.clock

        start_time = time.clock()
```

```
        for index in indexes2:
            mySet.add(index)
        print("--- %s seconds to add element to the set ---" % (time.clock() - star
```

```
--- 0.002076326717508792 seconds to add elements to the list ---
--- 0.0034129126995408043 seconds to add elements to the dictionary ---
--- 0.005559502944752381 seconds to add element to the set ---
```

Le temps d'insertion de données est équivalent ici car les éléments sont ajoutés à la fin de la liste, ce qui n'entraine pas de décalage.

4 Sources

Général
https://docs.python.org/3/tutorial/datastructures.html
http://sthurlow.com/python/lesson06/
Listes
https://www.tutorialspoint.com/python/python_lists.htm
Tuples
https://www.tutorialspoint.com/python/python_tuples.htm
http://stackoverflow.com/questions/68630/are-tuples-more-efficient-than-lists-in-python
Sets https://www.programiz.com/python-programming/set
Dictionnaires https://www.tutorialspoint.com/python/python_dictionary.htm