cours_2

January 14, 2020

1 Tableaux et Listes

Une **liste** est une structure de données qui contient une séquence de valeurs.
Syntaxe:

```
[<valeur_1>, <valeur_2>, ..., <valeur_n>]
```

- Les valeurs ne sont pas nécessairement de même type.
- Une liste est une séquence

1.1 Généralités

1.1.1 Organisation de la mémoire

- La mémoire peut être vue comme un long ruban avec des zones protégées
- Chaque case mémoire (un octet) dispose d'une adresse propre.
- Le stockage d'une valeur peut prendre plusieurs octets.
- Certaines valeurs peuvent être de type "adresse" (8 octets ?). On les représente symboliquement par une flèche.

1.1.2 Tableau en mémoire

- Usuellement un ensemble de cases **contigues** en mémoire de **même taille**.
- On fait de l'arithmétique avec la taille d'un objet pour trouver la position d'un élément:

$$@T[i] = @T[0] + i * taille(element)$$

- Avantage: Accès direct (rapide) à un élément.
- Inconvénient: Pas idéal pour des mises à jour:
  - Ajout: décalage si pas à la fin
  - Insertion/ suppression: décalage/suppression.

1.1.3 Liste simplement (ou doublement) chainée

- Chaque élément connait son successeur (et prédecesseur si doublement chainée).
- Dispersion des éléments en mémoire.
- Une valeur spéciale indique la fin (et le début) de liste.

- Avantage: idéal pour des mises à jour (ajout, insertion, suppression)

- Inconvénient: Accès séquentiel à un élément (très lent si longue liste)

Tableau ou Liste:

- On doit faire un compromis entre efficacité de l'accès et des mises à jour.
- Dépend des langages de programmation (Python vs C)

1.2 Listes (= Tableaux) en Python

1.2.1 Tests

[1]:
```
from tqdm.notebook import tqdm_notebook
import random
from time import time
n = 10**5
```

[2]:
```
L = []
tps_append = []
for _ in tqdm_notebook(range(n)):
    val = random.randint(0, 100)
    debut = time()
    L.append(val)
    fin = time()
    tps_append.append(fin - debut)
```

```
HBox(children=(FloatProgress(value=0.0, max=100000.0), HTML(value='')))
```

[3]:
```
L = []
tps_insert = []
for _ in tqdm_notebook(range(n)):
    pos = random.randint(0, len(L))  ## insertion aléatoire
    val = random.randint(0, 100)
    debut = time()
    L.insert(pos, val)
    fin = time()
    tps_insert.append(fin - debut)
```

```
HBox(children=(FloatProgress(value=0.0, max=100000.0), HTML(value='')))
```

[4]:
```
import matplotlib.pyplot as plt
%matplotlib notebook
```

```
x = list(range(n))

plt.plot(x, tps_insert, label = "insert")
plt.plot(x, tps_append, label = "append")
plt.legend()
plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

1.2.2 Liste et mémoire en Python

Une liste étant un objet, même une liste vide doit occupé de la place:

[1]:
```
from sys import getsizeof as sof    ## Nombre d'octets en mémoire
sof([])
```

[1]:
```
72
```

[6]:
```
# 8 octets par valeur de plus
sof([1]), sof([1, 2]), sof([1, 2, 3])
```

[6]:
```
(80, 88, 96)
```

[7]:
```
# Et ce quel soit le type et ses valeurs
sof(["lepetitchat"]), sof(["le", "petit"]), sof(["le", "petit", "chat"])
```

[7]:
```
(80, 88, 96)
```

[8]:
```
# Même en mélangeant les types
sof(["lepetitchat"]), sof(["le", ["petit"]]), sof(["le", ["petit"], ["chat",␣
↪False]])
```

[8]:
```
(80, 88, 96)
```

Python ne stocke donc pas directement les valeurs dans la liste mais des **références** vers les données:

- Liste = Tableau d'adresses vers les élements de la liste
- Le type des éléments de la liste n'est pas connue de la liste
- Cela optimise la place mémoire

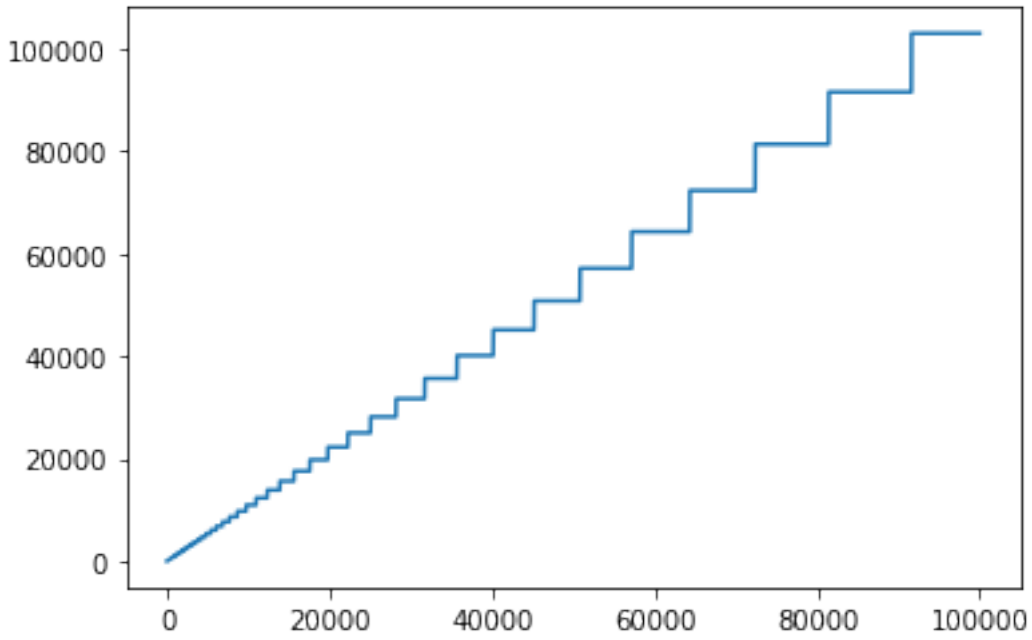
La croissance de la taille d'une liste est gérée par la règle:

```
new_allocated = newsize + newsize // 8 + 3 si newsize < 9 (+6 sinon)
new_allocated ~ 1.125 * newsize
```

Cela donne 0, 4=1+3, 8=5+3, 16=9+1+6, 25=17+2+9, ...

[2]:
```
import matplotlib.pyplot as plt
allocated = [0]
```

```
for size in range(1, 10**5):
    if size <= allocated[-1]:
        allocated.append(allocated[-1])
    else:
        newsize  = size + size // 8
        if size < 9:
            allocated.append(newsize+3)
        else:
            allocated.append(newsize+6)
plt.figure()
plt.plot(allocated)
plt.show()
```



Aparté: Principe identique pour les autres types

Octets	type	scaling notes
28	int	+4 bytes about every 30 powers of 2
49	str	+1-4 per additional character (depending on max width)
48	tuple	+8 per additional item
64	list	+8 per additional item
240	dict	6th increases to 368; 22nd, 1184; 43rd, 2280; 86th, 4704; 171st, 9320

1.2.3 Ajout en tête

Quel est la meilleure solution pour le “prepend” (ajout en tête) ?

- Insertion systématique en position 0.
- Append pour chaque élement puis reverse à la fin.

1.2.4 Conclusion

Classification (parcours, reallocation, cout)

- L[index]: rapide
- L.append: rapide (ev. réallocation)
- insert(index, objet) : lent
- remove(valeur): parcours jusqu'a trouve (lent) + décalage (lent) -> lent
- pop(index) ou del L[index]: décalage (lent)
- pop() (suppression dernier): rapide
- index(valeur): parcours jusqu'a trouve (lent)

1.3 Compléments sur les listes

1.3.1 Listes en compréhension

```
L = [expression(x) for x in iterable if condition(x)]
```

où *iterable* est une séquence (liste, range, chaine, . . .)

- La condition est optionnelle.
- Le résultat est une liste.
- En général sur une seule ligne.

Quelques usages:

- Construire une liste à partir de *range*.
- Créer une liste à partir d'une autre (filtrage)
- Se substituer à un *for* simple.
- Une liste de listes.

1.3.2 Parcours simultané de plusieurs listes avec *zip*

```
zip(*iterables)
```

où *iterables désigne 0, 1, 2, . . . objets iterables (liste, chaine, dict, . . .)

- le type retourné est *zip* qui est iterable ;)

[10]:
```
type(zip())
```

[10]:
```
zip
```

[3]:
```
L1 = [2, 3, 5]
L2 = ['m', 'e', 't']

for (e1, e2) in zip(L1, L2):
    print(e1, '---', e2)
```

```
2 --- m
3 --- e
5 --- t
```

[4]:
```
for i, e1, e2 in enumerate(zip(L1, L2)):
    print(i,':', e1, '---', e2)
```

```
␣
↪---------------------------------------------------------------------------

ValueError                                Traceback (most recent call␣
↪last)

<ipython-input-4-3876be83ae22> in <module>
----> 1 for i, e1, e2 in enumerate(zip(L1, L2)):
      2     print(i,':', e1, '---', e2)


ValueError: not enough values to unpack (expected 3, got 2)
```

[5]:
```
for i, (e1, e2) in enumerate(zip(L1, L2)):
    print(i,':', e1, '---', e2)
```

```
0 : 2 --- m
1 : 3 --- e
2 : 5 --- t
```

On s'arrête sur la plus courte séquence:

[ ]:
```
L1 = [2, 3, 5, 6, 1, 4]
L2 = ['m', 'e', 't']

for e1, e2 in zip(L1, L2):
    print(e1, '---', e2)
```

Autre approche possible (sans zip):

[ ]:
```
L1 = [2, 3, 5, 6, 1, 4]
L2 = ['m', 'e', 't']

for i in range(min(len(L1), len(L2))):
    print(L1[i], '---', L2[i])
```

Cela devient difficile de faire mieux que:

[6]:
```
L1 = [2, 3, 5, 6, 1, 4]
L2 = ['m', 'e', 't', 'i', 'e', 'r']
L3 = [2**i for i in range(10)]
```

```
L4 = [True, True, False]*4

for e1, e2, e3, e4 in zip(L1, L2, L3, L4):
    print(e1, '---', e2, '---', e3, '---', e4)
```

```
2 --- m --- 1 --- True
3 --- e --- 2 --- True
5 --- t --- 4 --- False
6 --- i --- 8 --- True
1 --- e --- 16 --- True
4 --- r --- 32 --- False
```