

cours_2

March 10, 2021

Cours 2

Le cours du jour traite des points suivants:

- Fonctionnement des listes en Python
- Pourquoi parle t'on de liste au lieu de tableau
- Comment fonctionnent append et insert ?
- Quelle est leur efficacité respective ?

On va parler de technique...

0.1 Rappels liste

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

Extrait de la documentation

- **list.append(x)** - *Add an item to the end of the list. Equivalent to $a[len(a):] = [x]$.*
- **list.extend(iterable)** - *Extend the list by appending all the items from the iterable. Equivalent to $a[len(a):] = iterable$.*
- **list.insert(i, x)** - *Insert an item at a given position. The first argument is the index of the element before which to insert, so $a.insert(0, x)$ inserts at the front of the list, and $a.insert(len(a), x)$ is equivalent to $a.append(x)$.*
- **list.remove(x)** - *Remove the first item from the list whose value is equal to x. It raises a `ValueError` if there is no such item.*
- **list.pop([i])** - *Remove the item at the given position in the list, and return it. If no index is specified, $a.pop()$ removes and returns the last item in the list.*

0.2 Généralités sur 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 (Ex.: mot de 8 octets)

- Certaines valeurs peuvent être de type “adresse” (8 octets ?). On les représente symboliquement par une flèche.

0.2.1 Tableau en mémoire (vers. classique)

- 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:

$$@T[i] = @T[0] + i * \text{taille}(\text{element})$$

- 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.

0.2.2 Liste simplement (ou doublement) chaînée (vers. classique)

- Usuellement un ensemble de cases **non contigues** en mémoire.
- Chaque élément connaît son successeur (et prédécesseur si doublement chaînée).
- 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)

0.3 Listes (= Tableaux) en Python

0.3.1 Tests préliminaires

```
[1]: from tqdm.notebook import tqdm_notebook
import random
from time import time
n = int(10**5.5)
```

Sondage:

Après n appels à *append* ou *insert*, que prévoyez vous ?

- Append est plus rapide
- Insert est plus rapide
- Tous les deux prennent le même temps

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

```
fin = time()
tps_append.append(fin - debut)
```

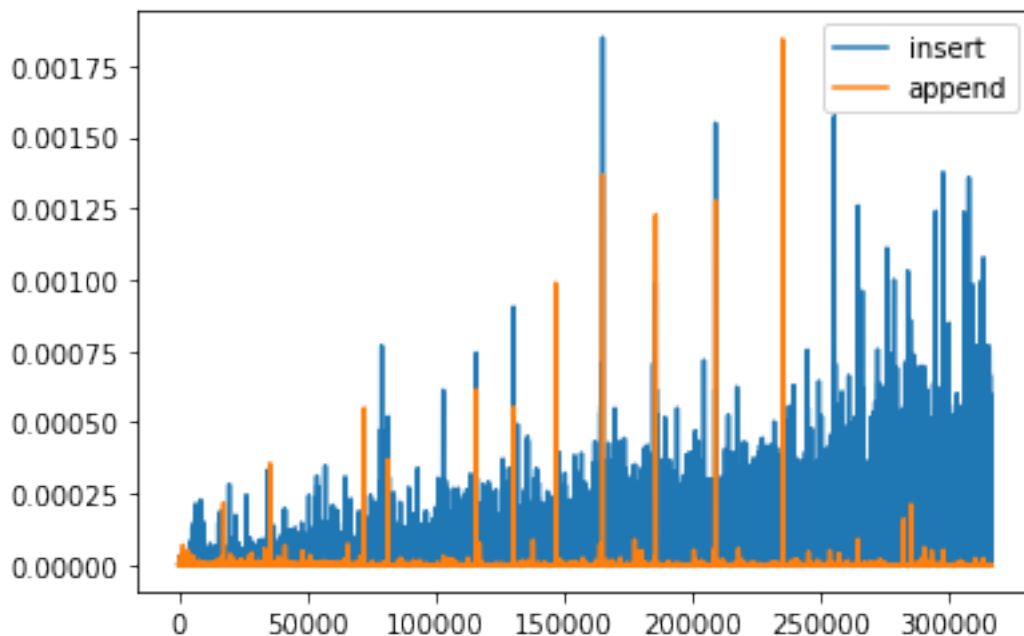
```
0%|          | 0/316227 [00:00<?, ?it/s]
```

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

```
0%|          | 0/316227 [00:00<?, ?it/s]
```

```
[4]: import matplotlib.pyplot as plt
x = list(range(n))

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



Conclusions partielles:

- **append** bien plus rapide qu'**insert**
- **append**: le temps semble indépendant de la taille de la liste
- **insert**: le temps croît la taille de la liste

0.3.2 Liste et mémoire en Python

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

```
[7]: from sys import getsizeof as sof    ## Nombre d'octets en mémoire
print(f"Entier: {sof(0)}")
print(f"Liste vide: {sof([])}")
```

Entier: 24

Liste vide: 56

Ne pas oublier que ce sont des objets avec des méthodes ...

```
[10]: #dir([])
```

```
[34]: # 8 octets par valeur de plus
sof([]), sof([1]), sof([1, 2]), sof([1, 2, 3])
```

```
[34]: (56, 64, 72, 80)
```

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

```
[35]: (56, 64, 72, 80)
```

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

```
[36]: (56, 64, 72, 80)
```

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

- Liste = Tableau d'adresses vers les éléments de la liste
- Le type des éléments de la liste n'est pas connue de la liste
- Cela optimise la place mémoire (intéressant pour la copie, pour **insert**)

Que faire si la taille initialement prévue est trop petite (insert ou append) ?

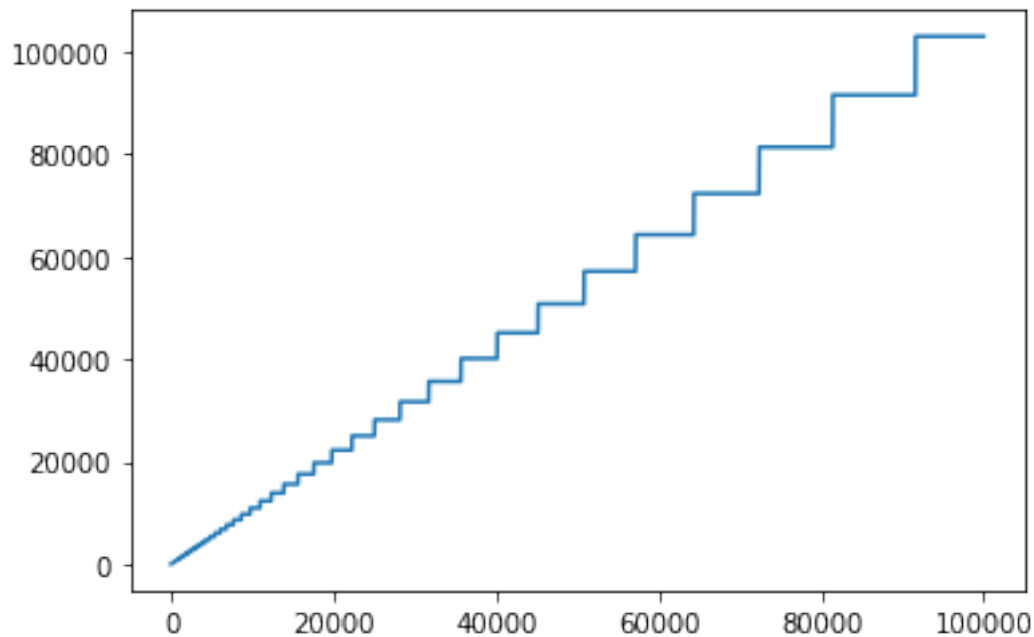
- Le système cherche un nouvel emplacement plus grand (peu coûteux)
- Il recopie l'ancien contenu. De plus en plus coûteux au fur et à mesure que le tableau croît.

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, ...

```
[38]: 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)
40	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

```
[60]: print(sof(0), sof(1), sof(2**30), sof(2**60), sof(2**90))
      print(sof(""), sof("a"), sof("aa"), sof("aaa"), sof("aaaa"))
      print(sof(tuple()), sof((1,)), sof((1,2)), sof((1,2,3)), sof((1,2,3,4)))
```

```
24 28 32 36 40
49 50 51 52 53
40 48 56 64 72
```

Que faire si la taille initialement prévue est trop grand ?

La décroissance de la taille d'une liste est gérée par la [règle](#):

```
si new_size < allocated //2 alors on réalloue
sinon on garde l'allocation inchangée.
```

0.3.3 Ajout en tête

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

- Insertion systématique en position 0. Non !
- Append pour chaque élément puis reverse à la fin.

0.3.4 Conclusion (le transparent le plus important du cours 2)

Quelles sont les méthodes lentes et rapides parmi:

`L[index]`, `insert(index, objet)`,

`pop()`, `pop(index)`,

`remove(valeur)`, `L.append(valeur)`,

`index(valeur)`

Rapide: * `L[index]`: simple calcul arithmétique * `L.append`: ajout en fin + ev. réallocation (rare)

* `pop()`: supprime en fin + ev. réallocation (rare)

Lent: * `insert(index, objet)` : décalage * `index(valeur)`: parcours jusqu'à trouve * `remove(valeur)`:
parcours jusqu'à trouve + décalage (parcours entier) * `pop(index)` ou `del L[index]`: décalage

0.4 Compléments sur les listes

0.4.1 Listes en compréhension

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

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

- 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.

```
[4]: chaine = "le petit chat"
L = [c for c in chaine]
print(L)
```

```
['l', 'e', ' ', 'p', 'e', 't', 'i', 't', ' ', 'c', 'h', 'a', 't']
```

```
[3]: L_pairs = [x for x in range(101) if x % 2 == 0]
print(L_pairs)
L_mul3 = [x for x in L_pairs if x % 3 == 0]
print(L_mul3)
```

```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40,
42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70, 72, 74, 76, 78, 80,
82, 84, 86, 88, 90, 92, 94, 96, 98, 100]
[0, 6, 12, 18, 24, 30, 36, 42, 48, 54, 60, 66, 72, 78, 84, 90, 96]
```

```
[6]: L = [[x] for x in range(5)]
print(L)
L[0].append(3)
print(L)
```

```
[[0], [1], [2], [3], [4]]
[[0, 3], [1], [2], [3], [4]]
```

0.4.2 Parcours simultané de plusieurs listes avec *zip*

`zip(*iterables)`

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

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

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

```
[11]: zip
```

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

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

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

```
[13]: 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:

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

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

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

```
[ ]: for i in range(min([len(L1), len(L2), ..., len(Ln)])):
      e1 = L1[i]
      e2 = L2[i]
      ...
      en = Ln[i]
```

Autre approche possible (sans zip):

```
[15]: 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])
```

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

Cela devient difficile de faire mieux que:

```
[18]: 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, sep=' --- ')
```



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

```
[ ]:
```