Listes et compréhensions de listes en Python, introduction à la notion de coût en temps

1. Les tableaux

Tableaux

En informatique, un **tableau** est une structure de données représentant une séquence finie d'éléments auxquels on peut accéder efficacement par leur position (ou indice), dans la séquence. C'est un type de conteneur que l'on retrouve dans un grand nombre de langages de programmation. On parle aussi de tableau indexé.

Dans les langages à **typage statique** (comme C, Java et OCaml), tous les éléments d'un tableau doivent être du même type. Certains langages à **typage dynamique**, tels que Python, permettent des tableaux hétérogènes (donc avec des données de natures différentes).

Les tableaux en Python

En Python, un tableau est représenté par un objet de type `list`. Les principales différences entre les types `list` et `tuple` sont :

- un objet de type `list` est une séquence **entre crochets** :

```
>>> mon_tab = [45, 24, -35, -12]
>>> type(mon_tab)
<class 'list'>
```

- un objet de type `list` est **mutable**, ce qui signifie qu'on peut changer sa valeur *après sa création*[1], ce qui n'est pas le cas d'un `tuple` :

Code

```
>>> mon_tab = [45, 24, -35, -12]
>>> mon_tab[2]  = 1000
>>> mon_tab
[45, 24, 1000, -12]
>>> mon_tuple=(45, 24, -35, -12)
>>> mon_tuple[2] = 1000
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Python Tutor

Les types `list` et `tuple` partagent sinon les mêmes propriétés : indices, accès aux éléments, longueur avec `len()`, parcours séquentiel avec `for` ...

En particulier les algorithmes vus précédemment sont aussi valable pour un tableau.

Algorithme de recherche séquentiel

Enoncé

On veut construire une fonction `chercheElement(elem, monTab)` qui renvoie :

- l'indice de *la première occurrence* de `elem` si `elem` est présent dans le tableau `monTab` ;
- la longueur du tableau si l'élément `elem` n'est pas présent dans le tableau `monTab`.

Répondre aux questions suivantes :

1. Décrire en pseudo-code un algorithme définissant cette fonction.
2. Proposer une implémentation de ce pseudo-code en Python.
3. Quel est le **coût en temps** de cet algorithme ? (*à faire avec le professeur.*)

Solution

A venir !

Méthode *built-in* `index` en Python

2. Spécificité des listes en Python

Méthodes et utilisations des listes

1. **Construire une liste vide :**

```
>>> monTab = [ ] # ou bien
>>> monTab = list()
```

2. **Ajouter un élément à la fin d'une liste :**

```
>>> monTab.append(3)
>>> monTab
[3]
>>> monTab.append(5)
>>> monTab.append(7)
>>> monTab
[3, 5, 7]
```

3. **Supprimer et récupérer le dernier élément du tableau :**

```
>>> dernier = monTab.pop()
>>> dernier
7
>>> monTab
[3, 5]
```

La méthode `pop` possède d'autres propriétés que je vous laisse rechercher.

4. **Convertir un tableau en tuple :**

```
>>> monTuple = tuple(monTab)
>>> monTuple
(3, 5)
```

5. **Convertir un tuple en tableau :**

```
>>> monTab = list(monTuple)
>>> monTab
[3, 5]
```

6. **Extraire des parties d'un tableau grâce aux *slices* :**

```
>>> monTab = [35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47]
>>> monTab[ :4]
[35, 36, 37, 38]
>>> monTab[7: ]
[42, 43, 44, 45, 46, 47]
>>> monTab[4:7]
[39, 40, 41]
```

7. **Concaténer des tableaux:**

```
>>> [1, 2, 3] + [7, 8, 9]
[1, 2, 3, 7, 8, 9]
```

Exercice

Enoncé

En utilisant les méthodes présentées ci-dessus :

1. Écrire une fonction `carre(n)` qui renvoie le tableau des carrés des nombres entiers compris entre 0 et $n-1$.

```
>>> carre(4)
[0, 1, 4, 9]
```

2. Écrire une fonction `imagesf(deb,fin)` qui renvoie le tableau des images des nombres entiers compris entre `deb` et `fin` par la fonction $f : x \mapsto 3x^2 - 2x + 1$.

```
>>> imagef(-2,3)
[17, 6, 1, 2, 9, 22]
```

3. Écrire une fonction `genereListe(n)` qui renvoie un tableau de n nombres aléatoires compris entre 1 et n^2. (On pourra importer une fois le module `random` et utiliser la fonction `random.randint(a,b)` qui renvoie un nombre aléatoire entre a et b inclus).
4. Ecrire une fonction `insere(monTab, val, i)` qui renvoie une liste dans laquelle est inseré l'élément `val` à l'indice `i` au sein des éléments de `monTab`, en supposant que `i<len(monTab)`.

```
>>> insere([1, 2, 3, 4],5,2)
[1, 2, 5, 3, 4]
```

5. Écrire une fonction `compter(monTab,val)` permettant de compter le nombre d'occurrences de `val` dans `monTab`.

```
>>> compter([4, 6, 8, 6, 7, 6, 9], 6)
3
>>> compter([2, 4, 6],3)
0
```

6. Écrire une fonction `compterIndices(monTab,val)` permettant de renvoyer un tableau des indices des occurrences de `val` dans `monTab`.

```
>>> compter([4, 6, 8, 6, 7, 6, 9], 6)
[1, 3, 5]
>>> compter([2, 4, 6],3)
[]
```

7. Écrire une fonction `separer(monTab,val)` permettant, à partir d'une liste de nombres `monTab` d'obtenir deux listes. La première comporte les nombres inférieurs ou égaux à un nombre donné, la seconde les nombres qui lui sont strictement supérieurs :

```
>>> separer([45, 21, 56 ,12, 1, 8, 30, 22, 6, 33], 30)
[21, 12, 1, 8, 30, 22, 6], [45, 56, 33]
```

8. Écrire une fonction `plusProche(monTab,val)` permettant de rechercher la plus proche valeur d'un nombre dans une liste :

```
>>> plusProche([45, 21, 56 ,12, 1, 8, 30, 22, 6, 33], 20)
21.
```

Solution

1.
```
def carre(n) :
    t = []
    for i in range(n+1) :
```

```
            t.append(i**2)
        return t
```

2. `python`

```
    def imagesf(deb,fin) :
        t =[]
        for i in range(deb, fin+1) :
            t.append(3*i**2 -2*i+1)
        return t
```

3. Construction de listes par compréhension

Jusqu'à présent nous avons défini nos listes **par extension**, c'est-à-dire en donnant exactement les éléments de la liste. Cependant, cette méthode n'est pas efficace pour de très grandes listes. Il faudra donc donner une méthode de construction de la liste, en essayant de *comprendre* les liens entre les différents éléments de cette liste. On parle alors de listes définies par **compréhension**.

Compréhensions de listes

Une des spécificités de Python est la capacité à construire des listes (et des tuples) par compréhension. Cette capacité est partagée avec d'autres langages, comme `Haskell`.

premières compréhensions

Enoncé

1. Quel est le tableau associé à `monTab` ?

```
monTab = [4*nb for nb in range(100)]
```

2. Quel est le tableau associé à `monTab` ? Pourquoi ?

```
monTab = [3*nb for nb in range(100) if nb%2==0]
```

3. Quel est le tableau associé à `monTab` ?

```
monTab = [let for let in 'Abracadabra']
```

4. Quel est le tableau associé à `monTab` ? Pourquoi ?

```
monTab = [let for let in 'Abracadabra' if let.upper()!='A']
```

5. Quel est le tableau associé à `monTab` ? Pourquoi ?

```
monTab = [ord(let) for let in 'Abracadabra']
```

Réduire le code

Enoncé

Comme vous avez pu le constater, les compréhensions sont rapides à écrire. Et certaines fonctions de l'exercice n°2 peuvent être considérablement réduites : les fonctions `carre`, `imagef` et `genereListe`. Utilisez les compréhensions pour réduire leur taille.

Solution

A venir !

1. Voir ici ↩