Listes/tableaux (1/2)

D'après des TP de Stéphane Gonnord

Buts du TP

- ☞ Apprivoiser les tableaux/listes : définition, indexation, modification, slicing...
- ☞ Manipuler un petit peu des compréhensions de liste.
- ☞ Apprendre à faire des parcours simples sur des tableaux.
- ☞ Se confronter à un extrait de sujet de concours.

1 Manipulations des tableaux/listes ; slicing et compréhension

EXERCICE 1 Des tableaux/listes-jouets

1. *Créer les listes suivantes, qui pourront servir dans la suite du TP pour vérifier les fonctions qui auront été écrites :*

```
l0 = [42] * 15                  # À éviter
l00 = [42 for i in range(15)]   # À privilégier
l000 = [42 for _ in range(15)]  # Possible également

l1 = [10, 30, 42, 2, 17, 5, 30, -20]

l2 = []
for i in range(-3, 6):
    l2.append(i**2)

l21 = [i**2 for i in range(-3,6)]

l3 = []
for i in range(1000):
    if (i%5) in [0, 2, 4]:
        l3.append(i**3)

l31 = [i**3 for i in range(1000) if (i%5) in [0,2,4]]

l4 = [843.0]
for i in range(20):
    l4.append(l4[i]/3+28)
```

   Ces définitions seront écrites dans le fichier de script (`tp5.py`). Après exécution (F5), on vérifiera les valeurs de ces listes dans l'interpréteur et/ou l'explorateur de variables.

2. *Quelles sont les longueurs de ces listes ?*
3. *Constater enfin que c'est une très mauvaise idée*[1] *de prendre la lettre* `l` *dans un nom de liste* `:-)`

EXERCICE 2 *Promettre de ne jamais appeler une liste* `l` *ou* `l0` *mais plutôt* `liste` *ou* `t0`*... Renommer convenablement les listes précédemment définies.*

1. PEP 8, *Names to Avoid* :
« Never use the characters 'l' (lowercase letter el), 'O' (uppercase letter oh), or 'I' (uppercase letter eye) as single character variable names. In some fonts, these characters are indistinguishable from the numerals one and zero. When tempted to use 'l', use 'L' instead. »

EXERCICE 3 Un peu de slicing

1. *Créer la liste des dix derniers éléments de* `t3`.
2. *Créer la liste constituée des éléments de* `t3` *sauf les 250 premiers et les 250 derniers.*
3. *Créer la liste constituée des cinq premiers éléments de* `t4` *suivie des cinq derniers de cette même liste.*

EXERCICE 4 *Éteindre l'écran. « Deviner » les valeurs des listes* `k,t,m,n` *après avoir tapé les commandes suivantes :*

```
k = [10, 15, 12]
t = k
m = t
n = m[:]
m[1] = 17
n[0] = 19
```

Allumer l'écran, et vérifier !

Aller sur `http://www.pythontutor.com/visualize.html`*, entrer le code précédent, et visualiser pas-à-pas l'exécution.*

EXERCICE 5 *Éteindre l'écran. « Deviner » les valeurs de* `t5`*,* `t6` *et* `t7` *après avoir tapé les commandes suivantes :*

```
t5 = [t2[2*i + 1] for i in range(3)]
t6 = [x**2 for x in t2]
t7 = [(x, y) for x in [1, 2, 3] for y in [3, 1, 4] if x != y]
```

Allumer l'écran, et vérifier.

EXERCICE 6 *Créer à l'aide de compréhensions de liste*[2] *:*

- *la liste constituée des termes d'indice pair de* `t1` *;*
- *celle constituée des termes pairs de* `t1`.

EXERCICE 7 *Écrire une fonction réalisant l'échange de deux valeurs dans un tableau. Cette fonction recevra comme arguments un tableau et deux indices (distincts) correspondant à des positions réelles dans ce tableau, et effectuera l'échange sans rien renvoyer.*

```
>>> t = [10, 42, 1, 3, 5, 7, 1515, -10]
>>> echange(t, 1, 6)
>>> t
[10, 1515, 1, 3, 5, 7, 42, -10]
```

2 Parcours de tableaux

EXERCICE 8

1. *Écrire une fonction calculant la* somme *des éléments d'un tableau.*

```
def somme(tab):
    ....
>>> somme(t1)
116
```

2. *Écrire une fonction calculant le* produit *des éléments d'un tableau.*

```
def produit(tab):
    ....
>>> produit(t1)
-1285200000
```

2. Ou « listes en compréhension » : les deux existent vraiment !

EXERCICE 9 *Écrire une fonction calculant la* moyenne *des éléments d'un tableau, puis une autre calculant l'*écart-type *:*

$$\mu := \frac{1}{|T|}\sum_{x\in T} x, \qquad \sigma := \left(\frac{1}{|T|}\sum_{x\in T}(x-\mu)^2\right)^{1/2} \underset{exercice!}{=} \left(\left(\frac{1}{|T|}\sum_{x\in T} x^2\right) - \mu^2\right)^{1/2}$$

```
>>> moyenne(t2)
7.666666666666667
>>> ecart_type(t2)
7.803133273813083
```

EXERCICE 10 *Écrire une fonction calculant le* maximum *des éléments d'un tableau. De même, écrire une fonction renvoyant la* position *de ce maximum.*

```
>>> maximum(t1), position_maximum(t1)
42, 2
```

EXERCICE 11 *Dans la fonction écrite à l'exercice précédent, que se passe-t-il lorsque le maximum est pris plusieurs fois ? Comment changer ce comportement ?*

Conclure quand à l'importance de la précision des énoncés et des mots utilisés : la *et* une *ont des sens différents...*

EXERCICE 12 *Écrire une fonction renvoyant* `True` *si la liste donnée en argument est* croissante, *et* `False` *sinon. Même chose pour cette fois la* stricte décroissance.

Tester ces fonctions !

EXERCICE 13 *Écrire une fonction prenant en entrée un tableau, et renvoyant le tableau des sommes cumulées (depuis le premier terme) :*

```
>>> sommes_cumulees(t2)
[9, 13, 14, 14, 15, 19, 28, 44, 69]
```

Évaluer le nombre d'additions réalisées, en fonction de la longueur n du tableau. Si ce nombre est de l'ordre de n^2, essayer de réécrire la fonction pour arriver à un nombre de l'ordre de n.

EXERCICE 14 *Écrire une fonction prenant en entrée un tableau possédant au moins deux éléments, et renvoyant les deux plus gros éléments de ce tableau.*

```
>>> deux_gros(t1)
42, 30
```

EXERCICE 15 *L'exercice précédent était mal spécifié : que se passe-t-il si le plus gros élément est présent deux fois ? Réécrire la fonction pour que le comportement soit différent !*

EXERCICE 16

1. *Écrire une fonction* `difference(t)` *qui prendra en argument un tableau* `t` *et qui ne retourne rien mais qui modifie son argument en remplaçant chaque élément sauf le premier par sa différence avec l'élément précédent. Interdiction d'utiliser un autre tableau que l'argument* `t` *!*

   ```
   >>> t = [4,5,3,7] ; difference(t)
   >>> t
   [4, 1, -2, 4]
   ```

2. *Peut-on écrire une fonction réciproque de la précédente qui permet de retrouver le tableau initial ?*

EXERCICE 17 Recherche de doublons et permutations

1. *Écrire une fonction* `distincts(t)` *permettant de tester si les éléments d'un tableau* `t` *sont deux à deux distincts. Proposer une version avec deux boucles* `for` *imbriquées puis une autre avec une deux boucles* `while`*.*
2. *Déterminer le nombre de comparaisons effectuées dans le pire des cas (qui est ?).*

3. *On suppose que le tableau* `t` *à tester est de longueur* n *et ne contient que des entiers de* $[\![0; n-1]\!]$. *Il s'agit donc de savoir si ce tableau est une permutation de l'ensemble* $[\![0; n-1]\!]$.

   Pour améliorer l'algorithme précédent, on va utiliser le fait que les valeurs de `t` *sont dans un ensemble connu (*$[\![0; n-1]\!]$*) et essayer de ne faire qu'un seul parcours de la liste. Pour celà, on va d'abord construire un tableau* `memo` *de* n *booléens initialisés tous à* `False`*, puis on va parcourir le tableau* `t` *en mettant à jour* `memo[k]` *(à* `True`*) si l'on rencontre l'élément* `k`.

   C'est une technique de mémoïsation *par criblage.*

   On écrira un code avec une boucle `for` *puis un code avec une boucle* `while`.

EXERCICE 18 *Les instructions suivantes permettent de générer un tableau de* 10 *entiers distincts choisis aléatoirement dans l'intervalle* $[\![0; 999]\!]$ *puis de créer un second tableau avec les éléments du premier dans l'ordre croissant.*

```
>>> import random
>>> ta = [random.randint(0, 999) for i in range(10)]
>>> tb = sorted(ta)
```

Écrire une fonction `grand_et_petit(t)` *qui prend un tableau de* $2n$ *entiers distincts et retourne un tableau qui contient le plus grand élément suivi du plus petit, puis le second plus grand avec le second plus petit etc . . .*

```
>>> t
[539, 995, 701, 112, 711, 589, 321, 916, 363, 754]
>>> grand_et_petit(t)
[995, 112, 916, 321, 754, 363, 711, 539, 701, 589]
```

EXERCICE 19 D'après un exercice du site `http://www.france-ioi.org/`

Pour lisser tableau de mesures (des flottants) on remplace chaque valeur, sauf la première et la dernière, par la moyenne des deux valeurs qui l'entourent.

Par exemple, un lissage de `[2, 4, 6, 10, 5]` *est* `[2, 4, 7, 5.5, 5]`.

1. *Écrire une fonction* `lissage(t)` *qui prend en argument un tableau de mesures et retourne le tableau obtenu après lissage.*

   ```
   >>> import random
   >>> mesures = [random.uniform(-5, 5) for k in range(5)]
   >>> mesures
   [-4.689373644740465, -2.0935061308458525, 2.2664333970666224, 0.8771467374427662, -1.7411383260674649]
   >>> lissage(mesures)
   [-4.689373644740465, -1.2114701238369214, -0.6081796967015431, 0.26264753549957875, -1.7411383260674649]
   ```

2. *Écrire une fonction* `ecartmax(t)` *qui prend en argument un tableau de flottants et retourne l'écart-absolu maximal entre deux valeurs consécutives.*

3. *Écrire une fonction* `nblissages(tmes, diffMax, itermax)` *qui prend en argument un tableau de mesures* `tmes`*, un flottant* `diffMax` *strictement positif, un nombre d'itérations maximal* `itermax` *et qui retourne le nombre de lissages nécessaires pour obtenir un écart absolu maximal entre deux mesures consécutives inférieur à* `diffMax`*. Le nombre de lissages doit être inférieur à* `itermax` *sinon la fonction retourne* `None`.

   ```
   >>> nblissages(mesures, 1, 5000)
   8
   >>> nblissages(mesures, 0.8, 5000)
   12
   ```

EXERCICE 20 *Écrire une fonction* `maximum_intervalle(t, n)` *qui prend en arguments un tableau d'entiers* `t` *et un entier* `n` *inférieur ou égal à la longueur de* `t` *et qui retourne la somme maximale sur tous les sous-tableaux de longueur* `n` *inclus dans* `t`.

```
>>> maximum_intervalle([10,2,8,4,7,5], 3)
16
```

3 Extrait du sujet posé au concours Centrale 2015

3.1 Quelques fonctions utilitaires

EXERCICE 21 *Donner la valeur des expressions Python suivantes :*

```
>>> [1, 2, 3] + [4, 5, 6]
>>> 2 * [1, 2, 3]
```

EXERCICE 22 1. *Écrire une fonction Python* `smul` *à deux paramètres, un nombre et une liste de nombres, qui multiplie chaque élément de la liste par le nombre et renvoie une nouvelle liste :*

```
>>> smul(2, [1, 2, 3])
[2, 4, 6]
```

2. *Écrire une fonction Python* `vsom` *qui prend en paramètre deux listes de nombres de même longueur et qui renvoie une nouvelle liste constituée de la somme terme à terme de ces deux listes :*

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

3. *Écrire une fonction Python* `vdif` *qui prend en paramètre deux listes de nombres de même longueur et qui renvoie une nouvelle liste constituée de la différence terme à terme de ces deux listes (la deuxième moins la première).*

```
>>> vdif([1, 2, 3], [4, 5, 6])
[3, 3, 3]
```

4. *Écrire une fonction Python* `vprod` *qui prend en paramètre deux listes de nombres de même longueur et qui retourne une nouvelle liste constituée des produits terme à terme de ces deux listes.*

```
>>> vprod([1, 2, 3], [4, 5, 6])
[4, 10, 18]
```

5. *Écrire une fonction Python* `prodscal(u, v)` *qui retourne le produit scalaire de deux vecteurs représentés par leurs listes de coordonnées* `u` *et* `v`.

```
>>> prodscal([1, 2, 3], [4, 5, 6])
32
```

3.2 Interaction gravitationnelle entre N corps

On s'intéresse à un système de N corps massifs en interaction gravitationnelle. Dans la suite, les corps considérés sont assimilés à des points matériels P_j de masses m_j où $j \in [\![0; N-1]\!]$, $N \geqslant 2$ étant un entier positif donné. Le mouvement de ces points est étudié dans un référentiel galiléen muni d'une base orthonormée. L'interaction entre deux corps j et k est modélisée par la force gravitationnelle.

L'action exercée par le corps k sur le corps j est décrite par la force $\overrightarrow{F}_{k/j} = G\frac{m_j m_k}{r_{jk}^3}\overrightarrow{P_jP_k}$ où r_{jk} est la distance séparant les corps j et k ($r_{jk} = \|\overrightarrow{P_jP_k}\|$) et $G = 6,67 \times 10^{-11}\text{N.m}^2\text{.kg}^{-2}$ la constante de gravitation universelle.

À tout instant, chaque corps de masse m_j est repéré par ses coordonnées cartésiennes (x_j, y_j, z_j) dans le référentiel de référence.

Deux listes sont utilisées pour représenter ce système en Python :

- `masse` conserve les masses de chaque corps : `masse[j]` $= m_j$.
- `position` contient les positions de chaque corps : `position[j]` $= [x_j, y_j, z_j]$.

EXERCICE 23 1. *Exprimer la force $\overrightarrow{F}_j$ exercée sur le corps P_j par l'ensemble des autres corps P_k avec $k \neq j$.*

2. *Écrire une fonction Python* `force2(m1, p1, m2, p2)` *qui prend en paramètre les masses (*`m1` *et* `m2` *en kilogrammes) et les positions (*`p1` *et* `p2`*, sous forme de listes de trois coordonnées cartésiennes en mètres) de deux corps 1 et 2 et qui renvoie la valeur de la force exercée par le corps 2 sur le corps 1, sous la forme d'une liste à trois éléments représentant les composantes de la force dans la base de référence, en newtons.*

3. *Écrire une fonction* `forceN(j, m, pos)` *qui prend en paramètre l'indice* `j` *d'un corps, la liste des masses des N corps du système étudié ainsi que la liste de leurs positions et qui renvoie* $\overrightarrow{F}$*, la force exercée par tous les autres corps sur le corps* `j`*, sous la forme d'une liste de ses trois composantes cartésiennes.*

4 Pour ceux qui s'ennuient

EXERCICE 24 *Écrire une fonction prenant un entier n en entrée, et retournant le tableau de* $n+1$ *éléments contenant les termes de la suite de Fibonacci* $f_0, f_1, ..., f_n$ *:*

```
>>> fibonacci(10)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

EXERCICE 25 *Écrire une fonction prenant en entrée un entier* $n \geqslant 0$ *et retournant la liste des* $\binom{n}{k}$ *pour* $k \in [\![0, n]\!]$.

```
>>> ligne_binome(2)
[1, 2, 1]
```

EXERCICE 26 *Écrire une fonction prenant en entrée un tableau, et retournant la plus grande suite croissante constituée de termes consécutifs du tableau :*

```
>>> plus_grand_bloc_croissant([2, 10, 1, 3, 5, 7, 6, 8, 9])
[1, 3, 5, 7]
```

Quelle est la complexité de cette fonction « dans le pire des cas » ? On pourra faire quelques tests sur des listes aléatoires :

```
from random import randint
resultats = []
for _ in range(100):
    t = [randint(0, 10**6) for _ in range(10**5)]
    resultats.append(len(plus_grand_bloc_croissant(t)))
```

5 Besoin d'indications ?

- Exercice 8. Sans génie excessif :

  Entrées : T
  $s \leftarrow 0$ # la somme provisoire
  pour x *décrivant* T **faire**
  └ $s \leftarrow s + x$
  Résultat : s

- Exercice 10. Une variable va contenir le maximum provisoire, qui sera mis à jour à chaque visite d'un nouvel élément du tableau :

  Entrées : T
  $maxi \leftarrow T[0]$ # le maximum provisoire
  pour x *décrivant* T **faire**
  si $x > maxi$ **alors**
  $maxi \leftarrow x$
  Résultat : $maxi$

  Si on s'intéresse à la *position* du maximum, il suffit cette fois de mettre à jour une variable qui représentera l'indice du maximum provisoire. *Et on évitera de comparer des* valeurs *du tableau et des* indices*...*

- Exercice 12. On peut renvoyer le résultat dès qu'on constate un défaut de croissance :

  Entrées : T
  pour i *allant de* 0 *à* $|T|-2$ **faire**
  | **si** $T[i] > T[i+1]$ **alors**
  | | **Résultat :** `False`
  Résultat : `True`

- Exercice 13. On peut calculer naïvement chaque somme partielle, ce qui fait $n = |T|$ calculs de sommes de longueurs $1, 2, ..., n$, soit de l'ordre de n^2 sommes ($\frac{n(n-1)}{2}$ plus précisément...). On peut être plus malin en notant que que k-ème somme cumulée est égale à la $(k-1)$-ème à laquelle on ajoute le k-ème élément du tableau. Ceci permet de calculer toutes les sommes en n additions.

  Entrées : T
  $s \leftarrow 0$ # la somme partielle
  $SP \leftarrow []$ # le tableau des sommes partielles
  pour i *allant de* 0 *à* $|T|-1$ **faire**
  | $s \leftarrow s + T[i]$
  | Adjoindre s à SP # via `SP.append(s)`
  Résultat : `SP`

- Exercice 14. On peut noter m_1 et m_2 le maximum et celui juste après. En première approximation, ça peut donner quelque chose comme :

  Entrées : T
  si $T[0] \geqslant T[1]$ **alors**
  | $m_1, m_2 \leftarrow T[0], T[1]$
  sinon
  | $m_1, m_2 \leftarrow T[1], T[0]$
  pour i *allant de* 2 *à* $|T|-1$ **faire**
  | **si** $T[i] \geqslant m_1$ **alors**
  | | $m_1, m_2 \leftarrow T[i], m_1$ # un nouveau maximum
  | **sinon**
  | | **si** $T[i] \geqslant m_2$ **alors**
  | | | $m_2 \leftarrow T[i]$ # un nouveau deuxième
  Résultat : m_1, m_2

- Exercice 24. On pourra écrire une version avec `append`, et une autre où le tableau de bonne longueur est tout de suite créé.

- Exercice 25. Il suffit d'appliquer les règles de construction du triangle de Pascal. Pour passer de $[1,2,1]$ à $[1,3,3,1]$ puis $[1,4,6,4,1]$, si on a calculé `ligne` la ligne au rang $n-1$:

  – on commence avec un 1 ;
  – on continue avec les `ligne[i]+ligne[i+1]` pour i allant de 0 à $n-1$.
  – on termine avec un 1.

- Exercice 26. On tient à jour une variable lm représentant la longueur maximale des blocs croissants rencontrés jusqu'ici. Pour chaque nouvel indice i du tableau, on trouve le plus grand bloc $T[i, j[$ croissant, et on compare alors $j-i$ (la longueur dudit bloc) et lm.

  Entrées : T
  $lm \leftarrow 0$
  pour i *allant de* 0 *à* $|T|-1$ **faire**
  | $j \leftarrow i+1$
  | **tant que** $T[j] \geqslant T[j-1]$ **faire**
  | | $j \leftarrow j+1$
  | **si** $j - i > lm$ **alors**
  | | $lm \leftarrow j-i$
  Résultat : lm

Attention : si `T` est croissant, le temps d'exécution sera quadratique (donc ça ne passera pas pour $n = 10^6$). Si les blocs croissants sont de longueur maximale l, le temps d'exécution sera en $O(l.n)$, avec n la longueur de la liste : ça passe pour une liste aléatoire (pour laquelle l est de l'ordre de $\ln n$). On trouvera dans le corrigé une version linéaire. Le lire et le comprendre constituera un bon exercice !