Travaux pratiques d'introduction 6

Listes/Tableaux – 2

Informatique tronc commun MPSI et PCSI

I Introduction

I.1 Rappels

Des séances précédentes nous avons retenus que nous pouvions mémoriser une valeur – un nombre entier, un nombre réel, une chaine de caractère, etc – en l'associant à une variable.

I.2 Objectifs

Dans ce TP nous approfondissons les thématiques suivantes :

1. Extraire ou modifier une liste, non plus par valeur unique, mais par tranche. On parle de *slicing*. Nous étendons ce point à des tableaux spécifiques, les tableaux `numpy`, utilisés dans les Data Science.
2. Comment créer des listes dans une syntaxe propre à `Python` : les listes par compréhension.
3. `Python` offre aussi une syntaxe plus « naturelle » de parcours des listes. Nous la mettrons en œuvre.

II Extraire ou modifier des valeurs d'un tableau par tranche (Slicing)

Comment récupérer des éléments d'une séquence : liste, chaine de caractères, ...

- ... sans l'aide d'une boucle `for` ou `while`,
- ... mais par "tranche" ?

II.1 Indices positifs de position

On va travailler avec la chaine de caractères, `s = 'egg, bacon'` ou le tableau `t = [-10, 2, 23, 8, 5, 4, 1, 6, 12, 7]`
Ces deux structures ont comme point commun d'être ordonnées et donc repérer par un indice de position `i` débutant à `0`.

e	g	g	,		b	a	c	o	n
0	1	2	3	4	5	6	7	8	9

-10	2	23	8	5	4	1	6	12	7
0	1	2	3	4	5	6	7	8	9

II.2 Comment extraire une séquence de caractères ou de valeurs ?

II.2.a Comment extraire `'egg'` ou `[-10, 2, 23]` ?

La syntaxe est la suivante `s[0:3]`
Le premier indice `0` est inclus, le dernier `3` exclu.

```
>>> s[0:3]
'egg'
```

	↓	↓	↓							
	e	g	g	,		b	a	c	o	n
	0	1	2	3	4	5	6	7	8	9

```
>>> t[0:3]
[-10, 2, 23]
```

	↓	↓	↓							
	-10	2	23	8	5	4	1	6	12	7
	0	1	2	3	4	5	6	7	8	9

II.2.b Comment extraire `'bacon'` ou `[4, 1, 6, 12, 7]` ?

La syntaxe serait `s[5:10]` ou `t[5:10]`

```
>>> s[5:10]
'bacon'
```

						↓	↓	↓	↓	↓
	e	g	g	,		b	a	c	o	n
	0	1	2	3	4	5	6	7	8	9

```
>>> t[5:10]
[4, 1, 6, 12, 7]
```

						↓	↓	↓	↓	↓
	-10	2	23	8	5	4	1	6	12	7
	0	1	2	3	4	5	6	7	8	9

II.2.c Comment extraire toute la chaine de caractères ou toutes les valeurs du tableau ?

A priori `s[0:10]` ou `t[0:10]`, mais `Python` autorise à ne pas indiquer l'indice de début de liste (`0`) et de fin de liste `(le dernier + 1)` qui n'est autre que la longueur de la liste `len(s)`. On peut donc écrire, `s[:]` ou `t[:]` ce qui évite de connaître la longueur du tableau.

```
>>> s[0:10]
'egg bacon'
>>> s[:]
'egg bacon'
```

	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
	e	g	g	,		b	a	c	o	n
	0	1	2	3	4	5	6	7	8	9

```
>>> t[:]
[-10, 2, 23, 8, 5, 4, 1, 6, 12, 7]
```

	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
	-10	2	23	8	5	4	1	6	12	7
	0	1	2	3	4	5	6	7	8	9

II.2.d Comment extraire un élément sur deux ?

On complète l'écriture précédente en ajoutant un argument : le `pas`, qui par défaut vaut 1 et qui vaut donc maintenant 2.
`s[0:10:2]` ou `t[0:10:2]` ou encore `s[::2]` ou `t[::2]`.

```
>>> s[0:10:2]
'eg ao'
>>> s[::2]
'eg ao'
```

↓	↓	↓	↓	↓					
e	g	g	,		b	a	c	o	n
0	1	2	3	4	5	6	7	8	9

```
>>> t[::2]
[-10, 23, 5, 1, 12]
```

↓	↓	↓	↓	↓					
-10	2	23	8	5	4	1	6	12	7
0	1	2	3	4	5	6	7	8	9

À retenir : Structure générale du slicing

`t` est une structure de données *ordonnées* de `Python`

`t[debut : fin : pas (par defaut 1)]`

Rappelons-nous que :

- les indices commencent comme toujours à zéro,
- `debut` est inclus, `fin` est exclu, `pas`, par défaut, vaut 1.
- On peut omettre `debut` s'il vaut 0, comme on peut omettre `fin` s' il vaut la longueur `n = len(t)` du tableau.
- On obtient en tout `(fin-debut)` valeurs dans le résultat

II.3 Indices de position négatifs

1. La numérotation commence à `-1` pour la dernière valeur de la séquence.
2. ... pour aller vers la gauche ...
3. mais est toujours parcourue dans le sens de lecture, de gauche à droite, si le pas est positif.

e	g	g	,		b	a	c	o	n
0	1	2	3	4	5	6	7	8	9
−10	−9	−8	−7	−6	−5	−4	−3	−2	−1

ou encore, pour un tableau.

-10	2	23	8	5	4	1	6	12	7
0	1	2	3	4	5	6	7	8	9
−10	−9	−8	−7	−6	−5	−4	−3	−2	−1

II.4 Comment extraire une séquence de caractères ou de valeurs

Donc, le parcours de la séquence se fait de gauche à droite si le `pas` reste positif, même si les indices sont négatifs.

II.4.a Extraire la dernière valeur de la séquence, sans connaître sa longueur.

Sans même connaître la longueur de la séquence, son indice est `-1`
`s[-1]` ou `t[-1]` donne respectivement, `'n'` et `[7]`

II.4.b Comment extraire `'egg'` ou `[-10, 2, 23]` ?

`s[-10:-7]` ou `t[-10:-7]`, ou encore `s[:-7]` ou `t[:-7]`.

↓	↓	↓							
e	g	g	,		b	a	c	o	n
0	1	2	3	4	5	6	7	8	9
−10	−9	−8	−7	−6	−5	−4	−3	−2	−1

↓	↓	↓							
-10	2	23	8	5	4	1	6	12	7
0	1	2	3	4	5	6	7	8	9
−10	−9	−8	−7	−6	−5	−4	−3	−2	−1

II.4.c Parcourir la séquence de droite à gauche

Il suffit d'attribuer une valeur négative au `pas`, par exemple `-1`.

III Création de tableaux *par compréhension*

III.1 Exemple et syntaxe

Si des tableaux peuvent être donnés, ils peuvent aussi être générés, soit à partir d'une liste déjà existante, soit à partir de la fonction **`range()`**. Dans l'exemple suivant on crée le tableau des carrés de chaque élément d'un tableau initial.

```
L = [-10, -2, 3, 2]
n = len(L)
L_carre = [0] * n
for i in range(n):
    L_carre[i] = L[i]**2
```

La méthode *par compréhension* s'écrit pour le cas précédent ainsi :

```
L_carre2 = [val**2 for val in L]
```

La création de ce tableau de carrés n'a pas nécessité l'utilisation des indices de position, même s'il n'était pas interdit de les utiliser. Les valeurs `val` ont directement été extraites du tableau initial et traitées pour générer ce nouveau tableau.

À retenir : Création de liste/tableau *par compréhension*

La syntaxe est la suivante :

Fais ceci	pour cette collection	dans cette situation
`[ x**2`	**`for x in L`**	**`if x < 0 ]`**

ou encore, si les indices de positions sont nécessaires

Fais ceci	pour cette collection	dans cette situation
`[L[i]**i`	**`for i in range(len(L))`**	**`if L[i] < 0 ]`**

Nous verrons dans un autre chapitre des méthodes `Python` pour travailler sur les listes.

III.2 A partir d'une fonction

III.2.a Codage classique

On se donne une fonction polynomiale :

$$P(x) = 2x^2 - 3x - 2$$

III.2.b Codage avec le module `numpy`

Le module `numpy` nous *évite* d'extraire chaque élément de la séquence pour lui appliquer la fonction souhaitée et recomposer la séquence finale.
Saisir le code suivant. La première ligne charge le module `numpy` et le renomme `np` pour raccourcir la saisie. `np.array` fabrique le tableau au format spécifique `numpy`.

```
import numpy as np

L = np.array([0, 1, 2, 3, 4])

L_carre = L**2

print(L_carre)
```

La puissance de la méthode réside dans la vectorisation de la fonction, ici le carré, qui applique à chaque élément de la liste la fonction sans devoir faire l'extraction de chaque élément à l'aide d'une boucle.

III.3 Tableaux à plusieurs dimensions

Les tableaux de `Python` peuvent contenir des valeurs arbitraires, par exemple : `tab = ['egg', 2, [5, 3], 10.235]`.
En particulier, rien n'interdit de construire des listes dont les éléments sont eux-mêmes des listes, par exemple : `t = [[1, 0, 0, 0], [1, 1, 0, 0], [1, 2, 0, 2]]`.

III.3.a Comment accéder à une valeur de ce tableau de tableaux ?

```
>>> t = [[1, 0, 0, 0], [1, 1, 3, 0], [4, 2, 0, 1]]
>>> t[2]
[4, 2, 0, 1]
```

`t[2]` renvoie le troisième (indice 2) élément du tableau qui est donc la liste `[4, 2, 0, 1]`

```
>>> t = [[1, 0, 0, 0], [1, 1, 3, 0], [4, 2, 0, 1]]
>>> t[2]
[4, 2, 0, 1]
>>> t[2][0]
4
```

`t[2][0]` renvoie le premier (indice 0) élément de la liste précédente qui est donc `4`.

Un tel tableau de tableaux est un tableau à plusieurs dimension et ouvre à l'écriture des matrices. L'exemple précédent peut se visualiser ainsi avec une matrice (3×4) : 3 lignes et 4 colonnes.

	0	1	2	3
0	1	0	0	0
1	1	1	3	0
2	4	2	0	1

À retenir : Tableau de tableaux

On peut représenter une matrice `t` de n lignes et m colonnes ($n \times m$) par un tableau n de listes de m éléments.
La syntaxe suivante permet d'extraire l'élément de la ligne `i` de la colonne `j` : `t[i][j]`.

III.3.b Codage avec le module `numpy`

Le tableau de tableaux précédant s'écrit dans la syntaxe propre à `numpy` :

```
import numpy as np
ta = np.array([[1, 0, 0, 0], [1, 1, 3, 0], [4, 2, 0, 1]])
```

Pour accéder à une valeur à la ligne `i` et à la colonne `j`, la syntaxe diffère de la précédente : `t[i,j]`

```
import numpy as np
ta = np.array([[1, 0, 0, 0], [1, 1, 3, 0], [4, 2, 0, 1]])
>>> ta[2,0]
4
```

Si l'on souhaite extraite une ligne entière, soit toutes les valeurs de `j` d'une valeur `i` fixée : `t[i,:]`
Ou encore, le contraire, extraite une colonne entière, soit toutes les valeurs de `i` d'une valeur `j` fixée : `t[:,j]`

À retenir : Représentation des matrices avec `numpy`

```
import numpy as np
ta = np.array([[1, 0, 0, 0], [1, 1, 3, 0], [4, 2, 0, 1]])
```

`ta[i,j]` : donne accès à la valeur de la ligne `i` et de la colonne `j`.
`ta[i,:]` renvoie la ligne `i`
`ta[:,j]` renvoie la colonne `j`
La syntaxe du *slicing* est fonctionnelle.

IV Quand on peut se passer des indices de position

Dans cette dernière partie, on travaille sur les tableaux en essayant au maximum de se passer des indices de position.
On va donc

- parcourir un tableau en extrayant directement ses valeurs. Nous avons utilisez cette écriture proche du langage naturelle dans les listes par compréhension.
- faire des tests logiques avec les opérateurs habituels `==`, `!=`, `<`, `>`, `<=`, `>=`, `in`, `not in` ...

IV.1 Exemples et syntaxe

On va travailler avec la chaine de caractères, `s = 'egg, bacon'` ou le tableau `t = [-10, 2, 23, 8, 5, 4, 1, 6, 12, 7]`

Le test d'appartenance ou non est réduit à cette expression « naturelle ».

```
>>> 'egg' in s
True
>>> 23 in t
True
>>> 'ego' not in s
True
```

La parcours dans un tableau, s'écrit :

```
print("Directions possibles :")
for d in ["Nord", "Sud", "Est", "Ouest"]:
    print("*", d)
```

le programme affiche

```
Directions possibles :
* Nord
* Sud
* Est
* Ouest
```

À retenir : La liste vue comme une collection

L'itération se fait directement sur les éléments du tableau qui se trouve alors réduit à une collection d'éléments sans l'indice qui lui est associé. Rappellons que cet indice ordonne les éléments et fait le cœur d'une séquence.
Si cette forme est plus simple à lire comme à écrire, elle n'est applicable que lorsqu'effectivement nous n'avons pas besoin de connaître l'indice de chaque élément du tableau.

- Test d'appartenance ou non d'un élément `ele` dans le tableau `L` :

  ```
  ele in L
  ele not in L
  ```

  Comme chaque test, le résultat est un `booleen`
- Parcours des éléments d'une liste avec une boucle **`for`**.

  ```
  for val in L:
  ```

À retenir

Un tableau permet de regrouper plusieurs valeurs sous la forme d'une séquence ordonnée dans laquelle chaque valeur est associée à un indice ou numéro. Les indices permettent d'accéder aux valeurs contenues dans un tableau, pour les consulter ou les modifier. On peut utiliser une boucle pour examiner tour à tour les différentes valeurs contenues dans un tableau.