

3 Introduction aux listes et aux chaînes de caractères

3.1 Tableaux et listes

3.1.1 Deux structures de données en informatique.

Les données manipulées en informatique sont organisées, que ce soit dans la mémoire où dans la manière d'y accéder, de les manipuler. Une telle organisation porte le nom de **structure de données**, en voici deux grandes.

Définition – Tableaux

Les données sont stockées dans des cases contiguës de la mémoire de l'ordinateur, chaque emplacement étant souvent indicé par un entier. A priori, il n'est pas possible d'en rajouter autant que possible : cette place a été pré-allouée. Pour accéder au contenu de l'emplacement numéro k , la machine a seulement besoin de connaître l'adresse de la première case mémoire et de la largeur de chaque case. L'accès en lecture et en écriture à une donnée (à partir du numéro de son emplacement) se fait donc en temps constant (on dit $\mathcal{O}(1)$). L'ajout d'une nouvelle donnée à un tableau peut alors être problématique !

Définition – Liste (chaînée)

Les données ne sont pas stockées de manière organisée dans la mémoire, mais de chaque emplacement on pointe l'emplacement suivant. Ainsi, l'accès (en lecture ou en écriture) ne se fait plus en temps constant, mais l'ajout d'une nouvelle donnée se fait simplement en temps constant.

3.1.2 Et en python...

Les objets de type `list` en Python sont des tableaux : c'est une dénomination fâcheuse car, partout ailleurs en informatique, le terme **liste** désigne en fait une liste chaînée. Prenez donc l'habitude de dire « tableau », ou « liste Python » et non « liste » quand vous parlez de cette structure.

Cependant, il y a une raison à cette dénomination : ces tableaux en Python possèdent presque la propriété des listes, dans le sens où l'ajout d'un nouvel élément se fait en temps constant **amorti**. Cela signifie que la plupart de ces ajouts se font en temps



3.1	Tableaux et listes	1
3.2	Les listes	2
3.3	Chaînes et type <code>str</code> en python	6

- Choisir un type de variable.
- Concevoir un algorithme utilisant une structure conditionnelle (Si), une structure itérative (while), une structure itérative (for).
- Manipuler des listes.

constants car en fait Python a « gardé des cellules en réserve », mais parfois l'ajout d'un élément force la création d'un nouveau tableau avec plus de cellules de réserve, et la copie de l'ancien tableau dans le nouveau. Cette opération est coûteuse, mais elle est rare. En moyenne, chaque ajout à un coût constant.

On parle alors de **tableaux dynamiques**.

3.2 Les listes

3.2.1 Déclaration et initialisation

```
1 # Déclaration d'une liste vide
2 L = []
3 # Déclaration d'une liste avec des éléments connus
4 L = [10, -12, 20, 54, 4]
```

3.2.2 Accès et modification aux éléments d'une liste

Les éléments d'une liste sont indexés. Cela signifie qu'ils sont numérotés par un numéro. Le premier élément de la liste est porte le numéro 0.

L = [10,	-12,	20,	54,	4]
	↑	↑	↑	↑	↑
Indice	0	1	2	3	4

On peut alors accéder aux éléments par leurs indices.

```
1 >>> L[0] = 0
2 >>> print(L)
3 [0, -12, 20, 54, 4]
```

Attention à utiliser un indice existant!

```
1 >>> L[12]
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4   IndexError: list index out of range
```

Les indices appartiennent à $\llbracket 0, \text{len}(t) \rrbracket$. Mais on peut aussi compter les éléments à partir de la fin, en utilisant des indices négatifs (plutôt déconseillé pour le moment, à part pour avoir accès au dernier élément)

```
1 L[-1] # dernier élément
2 L[-2] # avant-dernier
```

Attention, là aussi, à ne pas sortir du tableau!

```
1 >>> t[-4]
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4   IndexError: list index out of range
```



```
1 42 in t
2 [] in [[]]
```

Enfin, on peut recopier plusieurs fois un **même** tableau avec l'opérateur `*`.

```
1 4*t
2 t*4
```

3.2.4 Tranchage – Slicing

On peut directement accéder à une sous-liste Python (ou tranche – **slice** en anglais – c'est-à-dire bloc de cases consécutives) d'une liste, c'est ce que l'on appelle le tranchage (*slicing*).

On utilise les syntaxes :

- ▶ `u[i:j]` pour accéder à la tranche de la liste `u` allant des indices `i` à `j-1`;
- ▶ `u[i:]` pour accéder à la tranche allant de l'indice `i` à la fin de la liste `u`;
- ▶ `u[:j]` pour accéder à la tranche allant du début de la liste `u` à l'indice `j-1`;
- ▶ `u[:]` pour accéder à la tranche complète (toute la liste `u`);
- ▶ `u[i:j:p]` pour accéder à la tranche de la liste `u` allant des indices `i` à `j-1`, par pas de `p`.

```
1 u
2 u[2:5]
3 u[2:]
4 u[:5]
5 u[1:8:2]
```

3.2.5 Autres méthodes

Python est un langage **orienté objet** : en python, tout est un **objet**, et ces objets sont regroupées en **classes**. Une **méthode** est une fonction qui s'applique aux objets d'une classe particulière. Si `method` est une méthode de la classe `c`, et si `a` est un objet de cette classe, alors on applique `method` à `a` avec la syntaxe suivante : `a.method()`, les parenthèses pouvant contenir des paramètres.

Les méthodes rassemblées dans le tableau 3.1 ne sont pas exigibles.

TABLE 3.1 – Méthodes applicables aux listes.

Méthode	Description
<code>append(x)</code>	Ajoute <code>x</code> en fin de liste
<code>extend(L)</code>	Concatène la liste <code>L</code> en fin de liste
<code>insert(i, x)</code>	Insère <code>x</code> à la position <code>i</code>
<code>remove(x)</code>	Supprime la première occurrence de <code>x</code> (erreur si impossible)
<code>pop(i)</code>	Supprime l'élément en position <code>i</code> (si vide, le dernier)
<code>index(x)</code>	Renvoie l'indice de la première occurrence de <code>x</code> (erreur si impossible)
<code>count(x)</code>	Renvoie le nombre d'occurrences de <code>x</code>
<code>sort(cmp, key, rev)</code>	Trie la liste (nombreuses options)
<code>reverse()</code>	Renverse la liste

3.2.6 Tableaux multidimensionnels

On peut représenter une matrice avec des listes Python, par exemple en la décrivant

ligne par ligne. Par exemple, on peut représenter la matrice $M = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$ par

```
1 M = [[1,2,3],[4,5,6],[7,8,9]]
```

On accède à la première ligne deuxième colonne par `M[0][1]`. Cependant, cela ne permet pas d'effectuer les opérations classiques sur les matrices. On préférera utiliser les possibilités de la bibliothèque de calcul numérique `numpy`.

3.2.7 Alias

Parfois, lorsque l'on manipule des tableaux, on observe un phénomène étrange qui laisse pantois : c'est l'**aliasing**. C'est une notion assez subtile, mais qu'il est bon de connaître pour éviter les problèmes qui en découlent, ou au moins avoir une parade lorsqu'on s'y trouve confronté.

Commençons par un exemple où tout se passe intuitivement :

```
1 x = 3
2 y = x
3 x = 42
```

y vaut alors 3.

Avec des listes Python, cela ne fonctionne pas exactement de cette façon :

```
1 x = [0]*5
2 y = x
3 x[0] = 42
```

y vaut alors `[42, 0, 0, 0, 0]`.

Tâchons de donner une explication à ce phénomène.

- Pour les objets de type **non mutable** (on ne peut pas les modifier, comme les types `int`, `float`, `bool`, `str`, `tuple`), une instruction du type `x = y` crée une nouvelle variable : la variable `x` pointe vers une case contenant la valeur de `y`, et l'on dit que `x` et `y` sont des **alias** de cette valeur. Réattribuer à `x` ou `y` une nouvelle valeur casse cet alias : les deux variables ne pointent plus vers la même valeur.
- Pour les objets de type **mutable**, les choses sont plus compliquées : on peut modifier leur contenu sans les réassigner. C'est le cas des listes python. Après une instruction du type `x = y`, les variables `x` et `y` pointent vers le même objet : ce sont des **alias** de cet objet là aussi. Mais si l'on modifie le contenu de `x`, sans réaffecter `x`, l'alias n'est pas cassé et l'on modifie à la fois `x` et `y`.

La fonction `id()`, qui affiche pour chaque objet son « numéro d'identité », permet de mettre cela en évidence :

<pre>1 x = 3 2 y = x 3 id(x), id(y) 4 x = 42 5 id(x), id(y)</pre>	<pre>1 x = [0]*5 2 y = x 3 id(x), id(y) 4 x[0] = 42 5 id(x), id(y)</pre>
---	--

Si l'on veut que `x` et `y` ne soient plus des alias, on pourra plutôt utiliser la méthode `copy()`.

```
1 x = [0]*5
2 y = x.copy()
3 id(x), id(y)
4 x[0] = 42
5 y
```

Il existe d'autres manières pour des tableaux de casser cet alias : `y = list(x)`, `y = x[:]` ou encore `y = x+[]`.

Cependant, en insérant des tableaux dans d'autres tableaux, on a une notion de « profondeur ». Les techniques données ci-dessus ne permettent de rompre un alias qu'au niveau de l'enveloppe externe. Il existe la fonction `deepcopy` de la bibliothèque `copy`, qui effectue une copie totale d'un objet, en profondeur comme son nom l'indique. Cela dit il est peu probable que nous en ayons besoin.

3.2.8 Définir une liste en compréhension

Afin d'accélérer la création de listes en Python, il est possible de les créer *en compréhension*. Il n'est pas indispensable de connaître cette syntaxe, mais elle peut s'avérer pratique de temps à autres. La syntaxe générique est la suivante :

```
1 L = [fonction(i) for i in list if condition(i)]
```

Voici quelques exemples.

Créer la liste des 10 premiers entiers, compris entre 0 et 9 (inclus).

```
1 L = [x for x in range(10)]
```

Créer la liste des entiers pairs, compris entre 0 et 9 (inclus).

```
1 L = [x for x in range(10) if x%2 == 0]
```

Créer la liste des 10 premiers carrés, compris entre 0 et 9 (inclus).

```
1 L = [x**2 for x in range(10)]
```

3.3 Chaînes et type `str` en python

3.3.1 Définition

Pour représenter des textes, on utilise la structure de données de « chaîne de caractères » (*string* en anglais). En python, cela correspond aux objets de type `str`.

Ces objets sont non-mutables, c'est-à-dire qu'une fois créés, on ne peut pas les modifier. Comme il ne peut y avoir des problèmes d'alias (par exemple, comme pour les listes python), nous ne détaillerons pas ici leur représentation en mémoire.

3.3.2 Création et lecture

On définit une chaîne de caractère en entourant ces caractères par des guillemets simples, doubles, ou trois guillemets simples ou doubles. L'utilisation de guillemets simples permet d'utiliser des guillemets doubles dans la chaîne de caractère, et vice-versa.

```
1 'Hallo Welt'
2 'It is "Hello World" !'
3 "Ja, aber 'not in germany' ..."
```

On peut signaler une tabulation par `\t` et créer une nouvelle ligne par `\n`.

```
1 s = u"\t Il faut une science politique nouvelle à un monde tout nouveau.\n
   [...]"
2 print(s)
```

Remarque

La présence de `u` devant les guillemets permet de s'assurer de prendre les comptes les caractères spéciaux liés à l'encodage utf-8.

On accède alors aux caractères d'une chaîne en donnant son indice, comme pour les listes.

```
1 s[0]
2 s[-1]
```

Remarquons que le tranchage fonctionne similairement aux listes.

```
1 s[14:21]
2 s[:11]
3 s[37:]
```

On peut aussi créer une chaîne de caractères à partir de nombres ou de booléens avec la fonction `str`.

```
1 str(42)
2 str(-3.5)
3 str(4==4.)
```

Enfin, rappelons que ce type est non mutable.

```
1 >>> s[0] = "a"
2 Traceback (most recent call last):
3   File "<pyshell#1>", line 1, in <module>
4     s[0] = "a"
5 TypeError: 'str' object does not support item assignment
```

3.3.3 Opérations sur les chaînes

Comme pour les listes, on peut concaténer deux chaînes de caractères avec l'opérateur `+`.

```
1 'Hello'+'World'
```

On peut aussi dupliquer une chaîne de caractères plusieurs fois avec l'opérateur `*`.

```
1 print('GA-BU-ZO-MEU\n'*5)
```

La fonction `len` permet là encore de calculer la longueur d'une liste.

```
1 len(s)
```

On peut aussi chercher la présence d'un motif dans une chaîne avec `in`, son absence avec `not in`

```
1 'Science' in s
2 'science' not in s
```

3.3.4 Chaînes et méthodes

De nombreuses méthodes existent sur les chaînes, voici les plus utiles.

Méthode	Description
<code>count(m)</code>	Compte le nombre d'occurrences du motif <code>m</code> , sans chevauchement.
<code>find(m)</code>	Renvoie la première occurrence du motif <code>m</code> .
<code>islower()</code> (et autres)	Renvoie un booléen indiquant si la chaîne est en minuscules.
<code>join(bout)</code>	Concatène les éléments de la liste <code>bout</code> , séparés par la chaîne.
<code>lower()</code>	Met en minuscule la chaîne de caractères.
<code>split(sep)</code>	Sépare la chaîne selon le séparateur <code>sep</code>
<code>strip(char)</code>	Enlève les lettres en début/fin si elles sont dans la chaîne <code>char</code> .
<code>upper()</code>	Mets en majuscules la chaîne.

```
1 s = "    que ; d'espaces ; mes ; amis ! ;    "
2 s.count('es')
3 s.find('es')
4 s.islower()
5 '+'.join([str(i) for i in range(14)])
6 'HAHAHAHAHA !'.lower()
7 s.split(';')
8 s.split()
9 s.strip()
10 s.strip('q; ')
11 s.upper()
```


Application 1

Sujet

Exercice – Liste de zéros

Question 1 Implémenter la fonction `zeros_01(n:int) -> []` permettant de générer une liste de `n` 0. On utilisera une boucle `while`.

```
1 >>> zeros_01(4)
2 [0,0,0,0]
```

Question 2 Implémenter la fonction `zeros_02(n:int) -> []` permettant de générer une liste de `n` 0. On utilisera une boucle `for`.

Question 3 Implémenter la fonction `zeros_03(n:int) -> []` permettant de générer une liste de `n` 0. On générera cette liste « en compréhension » (sans boucle `for` ou `while` explicite).

Exercice – Liste d'entiers

Question 4 Implémenter la fonction `entiers_01(n:int) -> []` permettant de générer la liste des entiers compris entre 0 inclus et `n` exclus. On utilisera une boucle `while`.

```
1 >>> entiers_01(4)
2 [0,1,2,3]
```

Question 5 Implémenter la fonction `entiers_02(n:int) -> []` permettant de générer la liste des entiers compris entre 0 inclus et `n` exclus. On utilisera une boucle `for`.

Question 6 Implémenter la fonction `entiers_03(n:int) -> []` permettant de générer la liste des entiers compris entre 0 inclus et `n` exclus. On générera cette liste « en compréhension » (sans boucle `for` ou `while` explicite).

Question 7 Écrire une fonction `carres(n:int) -> []` qui prend en argument un entier naturel `n` et qui renvoie la liste des `n` premiers carrés d'entiers, en commençant par 0.

Question 8 Écrire une fonction `somme_racine(n:int) -> float` permettant de calculer la somme des racines carrées des `n` premiers entiers naturels non nuls.

Exercice – Notion d’effet de bords

On cherche à écrire une fonction prenant en argument une liste d’entiers (non vide) et incrémentant de 1 le premier élément de cette liste.

Question 9 Écrire une telle fonction `incr_sans_effet_de_bord(L:[]) -> []`, qui ne modifie pas la liste initiale et renvoie en sortie une nouvelle liste.

```
1 >>> L = [1,2,3]
2 >>> LL = incr_sans_effet_de_bord(L)
3 >>> print(L)
4       [1,2,3]
5 >>> print(LL)
6       [2,2,3]
```

Question 10 Écrire une telle fonction `incr_avec_effet_de_bord(L:[])-> None`, qui modifie la liste initiale et ne renvoie rien en sortie (ponctuer par un `return None`).

```
1 >>> L = [1,2,3]
2 >>> LL = incr_sans_effet_de_bord(L)
3 >>> print(L)
4       [2,2,3]
5 >>> print(LL)
6       None
```

Exercice – Un petit peu de géométrie

Question 11 Implémenter la fonction `norme(A:[],B:[])->float` permettant calculer la norme du vecteur \overrightarrow{AB} dans \mathbb{R}^3 . Chacun des points sera constitué de la liste de ses coordonnées (par exemple `A=[xA,yA,zA]`).

Question 12 Implémenter la fonction `prod_vect(u:[], v:[])->float` permettant calculer le produit vectoriel $\overrightarrow{u} \wedge \overrightarrow{v}$ dans \mathbb{R}^3 .

Exercice – Suites d’entiers

Question 13 Implémenter la fonction `impairs(n:int) -> []` permettant de générer la liste des `n` premiers entiers naturels impairs.

Question 14 Implémenter la fonction `multiples_5(d:int, f:int) -> []` permettant de générer la liste de tous les multiples de 5 compris entre `d` et `f` (bornes incluses si se sont des multiples de 5).

Question 15 Implémenter la fonction `cube(f:int) -> []` permettant de générer la liste de tous les cubes d’entiers naturels inférieurs ou égaux à `f` (inclus si `f` est un cube).

Exercice – Un petit peu de statistiques

Question 16 Écrire une fonction `moy_extr(L:[])->float` qui prend en argument une liste `L` et renvoie en sortie la moyenne du premier et du dernier élément de `L`.

Question 17 Écrire une fonction `moyenne(L:[])->float` qui prend en argument une liste `L` et renvoie la moyenne des éléments de `L`.

Question 18 Écrire une fonction `ecart_type(L:[])->float` qui prend en argument une liste `L` et renvoie l'écart type des données : en notant \bar{x} la moyenne des échantillons, on

$$\sigma = \sqrt{\frac{1}{n} \sum_{i=1}^n (x_i^2 - \bar{x})}.$$

Exercice – Chaînes de caractères

Question 19 Écrire une fonction `lettre(i:int)->str` qui prend en argument un entier `i` et renvoie la `i`^e lettre de l'alphabet.