Cours : Listes

Les programmes que nous avons écrits jusqu'à présent ne travaillaient qu'avec un nombre limité de valeurs. Nous avons par exemple écrit une fonction calculant le minimum de deux nombres flottants. Si notre algorithme doit travailler non pas avec deux nombres flottants, mais avec une dizaine de nombres, des milliers, voire des milliards, nous devons stocker cette collection de valeurs dans une *structure de données*. Différentes méthodes existent pour stocker en mémoire ces données. Selon l'algorithme que l'on souhaite utiliser pour traiter ces données, certaines sont plus performantes que d'autres. Dans ce chapitre, nous allons introduire la structure de donnée concrète la plus importante en informatique : *le tableau dynamique*. C'est avec la *liste chainée*, la structure de donnée concrète le plus utilisée pour stocker une collection ordonnée d'objets. La première est fondamentale en programmation impérative alors que la seconde est fondamentale en programmation fonctionnelle. Python propose un tableau dynamique qu'il a eu le très mauvais goût d'appeler liste et qu'il ne faudra pas confondre avec la liste chainée OCaml qui est aussi appelée liste, avec bon goût cette fois.

1 Liste

Une liste est une succession ordonnée de n valeurs. Si `a` est une liste de longueur n, ses valeurs sont indexées de 0 à $n-1$ et il est possible d'accéder à la valeur d'indice k avec la notation `a[k]`. En mémoire, les valeurs sont stockées les unes à la suite des autres. On en déduit donc que si p est l'adresse mémoire de la valeur d'indice 0, la valeur d'indice k est située à l'adresse $p+k\delta$ où δ est le nombre d'octets utilisé pour stocker une valeur. Une liste a donc l'avantage d'avoir un accès direct à son k-ième élément. La longueur d'une liste s'obtient avec la fonction `len`.

```
>>> note = [9, 10, 14]
>>> note[0]
 9
>>> (note[0] + note[1] + note[2]) / len(note)
 11.0
```

Les éléments d'une sont accessibles en lecture et en écriture. Il est donc possible de changer leurs éléments. Si on veut changer la dernière note :

```
>>> note = [9, 10, 14]
>>> note[2] = 16
>>> note
 [9, 10, 16]
```

Si on dépasse les bornes d'une liste, Python lance l'exception « list index out of range ». Par exemple `note[3]` va renvoyer une exception.

Exercice 1

1. Écrire un programme qui prend une liste de nombres en entrée et qui renvoie la somme de ses éléments.

Si on souhaite ajouter un point à toutes les notes, il suffit d'effectuer une boucle sur l'indice de la liste.

```
>>> note = [9, 10, 14]
>>> for k in range(len(note)):
        note[k] = note[k] + 1
>>> note
 [10, 11, 15]
```

Nous avons vu comment créer une liste contenant quelques éléments. Mais si nous souhaitons créer une liste contenant n éléments, on utilise la notation suivante qui permet de créer ce que l'on appelle une liste en compréhension.

```
>>> n = 5
>>> a = [k**2 for k in range(n)]
>>> a
 [0, 1, 4, 9, 16]
```

On peut aussi restreindre les éléments que l'on utilise en ajoutant une condition. On dit qu'on filtre la liste.

```
>>> a = [k**2 for k in range(n) if k % 3 != 1]
>>> a
 [0, 4, 9]
```

Les listes sont dynamiques, c'est-à-dire qu'il est possible d'augmenter ou de diminuer leur taille une fois créées. Pour ajouter un élément en fin de liste, on utilise la méthode `append` qui va agir par effet de bord et ajouter 18 à la fin de la liste.

```
>>> note = [9, 10, 14]
>>> note.append(18)
>>> note
 [9, 10, 14, 18]
```

Il est aussi possible d'enlever le dernier élément d'une liste. On utilise pour cela la méthode `pop` qui supprime le dernier élément de la liste et la renvoie.

```
>>> chapitre_du_cours_a_apprendre = [9, 8, 7, 6, 5]
>>> appris = chapitre_du_cours_a_apprendre.pop()
```

Exercice 2

Écrire un programme prenant en entrée une base $b \geqslant 2$ et un entier $m \in \mathbb{N}$ et qui renvoie la décomposition en base b de n sous forme de liste, en commençant par les chiffres de poids faible.

Les listes Python peuvent contenir des valeurs de types différents. Ils peuvent même contenir d'autres listes.

```
>>> a = [9, 3.14159, "Hello", True, []]
```

Cependant, en pratique, contrairement aux `tuple`, on n'utilisera que des listes contenant des valeurs ayant le même type.

Exercice 3

On choisit dans cet exercice de représenter un polynôme par la liste de ces coefficients : le polynôme

$$P = \sum_{k=0}^{n} a_k X^k$$

sera représenté par la liste $[a_0, \ldots, a_n]$. Un même polynôme admet donc plusieurs représentations distinctes puisque que l'on n'impose pas de condition sur a_n. Par exemple, les listes `[1, 2, 6]` et `[1, 2, 6, 0]` représentent le même polynôme.

1. Écrire la fonction `somme(P, Q)` qui renvoie une liste représentant le polynôme $P+Q$.
2. Justifier qu'il existe une suite (P_n) de polynômes de degré n à coefficients entiers tels que
$$\forall x \in \mathbb{R}^* \quad P_n\left(x - \frac{1}{x}\right) = x^n + \left(-\frac{1}{x}\right)^n$$
et donner une relation entre P_{n+1}, P_n et P_{n-1}.
3. Écrire une fonction d'en-tête `P(n)` qui renvoie, lorsque l'entier n est donné en paramètre, une liste représentant le polynôme P_n.

2 Recherche dans une liste

Il est courant de se demander si une valeur est déjà présente dans une liste. Si l'on souhaite écrire un algorithme s'arrêtant dès que la valeur est trouvée, il y a deux manières courantes de programmer cet algorithme.

```
>>> def est_present_v1(x, lst):
        for k in range(len(lst)):
            if lst[k] == x:
                return True
        return False
```

Si on souhaite avoir un programme avec un seul point de sortie, on utilise plutôt la version suivante :

```
>>> def est_presentV2(x, lst):
        k = 0
        while k < len(lst) and lst[k] != x:
            k = k + 1
        return k != len(lst)
```

Cette version utilise le côté paresseux de l'opérateur `and`. En effet, cet opérateur possède une subtilité. Il évalue d'abord le terme à sa gauche et si il obtient `False`, il n'évalue pas le terme à sa droite, car on sait que le résultat de l'opération est `False`. Ici, si jamais la valeur cherchée ne se trouve pas dans la liste de longueur n, la variable `k` va prendre la valeur n et l'évaluation paresseuse de la condition va faire que la condition `k < len(lst)` (qui ne sera plus vérifiée) empêchera d'effectuer le test `lst[k] != x`, opération qui n'a aucun sens car l'index `k` est sorti des bornes de la liste. Au passage, l'opérateur `or` est aussi paresseux en Python : si son membre de gauche s'évalue en `True`, le membre de droite n'est pas évalué.

Exercice 4

1. Écrire une fonction qui recherche et renvoie le minimum d'une liste de nombres.
2. Écrire une fonction qui recherche et renvoie l'indice du minimum d'une liste de nombres.

Exercice 5

Soit L une liste d'entiers de longueur n. On dit que la liste L est bicolore, si l'on peut la séparer en deux sous-listes monotones de sens contraires. Par exemple, la liste $L = [1, 2, 3, 5, 6, 10, 8, 3, 2]$ est bicolore, car on peut la séparer en $[1, 2, 3, 5, 6] + [10, 8, 3, 2]$. Tandis que la liste $L' = [4, 5, 6, 5, 2, 4]$ n'est pas bicolore. Par convention, les listes croissantes et décroissantes sont bicolores. Écrire une fonction `bicolore(L)` qui prend comme argument la liste L et renvoie un booléen de valeur `True` si L est bicolore et un booléen de valeur `False` si L n'est pas bicolore.

Les chaînes de caractères fonctionnant de manière similaire aux listes, voici quelques exercices de recherche sur ces dernières.

Exercice 6

La seule commande sur les strings autorisée dans cet exercice est la lecture du caractère d'indice i par le biais de `c[i]`.

1. Écrire une fonction `egal(w1, w2)` qui, lorsque `w1` et `w2` sont deux chaînes de caractères renvoie un booléen de valeur `True` si ces chaînes sont égales et de valeur `False` sinon.
2. En déduire une fonction `sousMot(w, s)` qui, lorsque `w` et `s` sont deux chaînes de caractères renvoie un booléen de valeur `True` si `w` est une sous-chaîne de `s` et de valeur `False` sinon.

Exercice 7

Soit A l'ensemble des lettres de l'alphabet latin usuel. Il y a donc $p = 26$ lettres. Soit $n \in \mathbb{N}$. On appellera *mot* de longueur n une chaîne de caractères $t = "b_0 \dots b_{n-1}"$, telle que

$$\forall i \in \{0, \dots, n-1\} \quad b_i \in A.$$

Un mot de longueur 1 sera simplement appelé une *lettre*. Si $n = 0$, on retrouve le mot vide que l'on note "" et qui ne contient aucun caractère. On appellera *sous-mot* de t tout mot obtenu en supprimant des lettres de t, éventuellement aucune ou toutes. Plus formellement, un mot x est sous-mot de $t = "b_0 \dots b_{n-1}"$ si et seulement si il existe des entiers $0 \leqslant i_0 < i_2 < \dots < i_{p-1} \leqslant n-1$ tels que $x = "b_{i_0} \dots b_{i_{p-1}}"$. Par exemple, les sous-mots de $t = "abba"$ sont les mots : "", "a", "b","aa","ab", "ba","bb", "aba", "abb", "bba" et "$abba$".

Étant donnés deux mots $x = "a_0 \dots a_{p-1}"$ et $y = "b_0 \dots b_{n-1}"$, on note

$$\binom{y}{x}$$

le nombre de façons d'écrire x comme sous-mot de y. Par exemple

$$\binom{"abab"}{"ab"} = 3 \text{ et } \binom{"abbab"}{"aba"} = 2.$$

Par définition

$$\binom{y}{""} = 1.$$

1. Calculer $\binom{"aabbaa"}{"aba"}$.
   (a) Que vaut $\binom{y}{x}$, lorsque $\text{len}(x) = \text{len}(y)$? Lorsque $\text{len}(y) < \text{len}(x)$?
   (b) Soit a une lettre et n un entier naturel. On note
   $$a^n = \underbrace{"aa \dots a"}_{n \; fois}.$$
   Que vaut $\binom{a^n}{a^p}$ lorsque n et p sont deux entiers ?
   (c) Soient x et y deux mots et "a" et "b" deux lettres. Donner sans justification une expression de $\binom{y"b"}{x"a"}$ en fonction de $\binom{y}{x"a"}$, $\binom{y}{x}$, de "a" et de "b". On discutera suivant que $a = b$ ou non.
2. Soient $x = "a_0 \dots a_{p-1}"$ et $y = "b_0 \dots b_{n-1}"$ deux mots. On pose pour tout $0 \leqslant i \leqslant p-1$ et pour tout $0 \leqslant j \leqslant n-1$:
   $$h(i,j) = \binom{"b_0 \dots b_{j-1}"}{"a_0 \dots a_{i-1}"}.$$
   (a) Que vaut $h(i,j)$ lorsque $i > j$?
   (b) Que vaut $h(0,j)$?
   (c) Justifier que $h(i,j) = h(i-1,j-1) + h(i,j-1)$ si $b_{j-1} = a_{i-1}$.
   (d) Justifier que $h(i,j) = h(i,j-1)$ si $b_{j-1} \neq a_{i-1}$.
3. À l'aide de la question précédente, on cherche à écrire un algorithme de calcul de $\binom{y}{x}$ dont le nombre de comparaisons entre une lettre de x et une lettre de y est majorée par une expression du type $K(\text{len}(x) + \text{len}(y))$. On note pour $j \in \{0, \dots, \text{len}(y)\}$, H_j la propriété :
   « `res` est la liste `[h(0,j),h(1,j),h(2,j), ..., h(j,j)]` ».
   (a) Comment rendre H_0 vraie ?
   (b) Soit $j \in \{1, \dots, \text{len}(y)\}$. On suppose que H_{j-1} est vraie. Par quelle instruction peut-on rendre H_j vraie ?
   (c) Créer alors la fonction `nbEcrituresSousMot(x, y)` qui calcule le nombre d'écritures de `x` comme sous-mot de `y`. On justifiera que l'algorithme est valide et on majorera sa complexité par une fonction de $\text{len}(x)$ et $\text{len}(y)$.

3 Recherche dichotomique dans une liste triée

Lorsqu'il existe une relation d'ordre totale sur les valeurs avec lesquelles on travaille et que la liste est *triée*, il est possible d'effectuer une recherche beaucoup plus rapide. Notons tout d'abord que si nous possédons un tableau, il est facile de le trier avec la méthode `sort`.

```
>>> liste = [8, 3, 7]
>>> liste.sort()
>>> liste
 [3, 7, 8]
```

L'algorithme de recherche sur une liste triée que nous allons présenter est appelé algorithme de recherche dichotomique.

```
>>> def est_present_dichotomie(x, tab):
        g = 0
        d = len(tab) - 1
        while g <= d:
            m = (g + d) // 2
            if x < tab[m]:
                d = m - 1
            elif x > tab[m]:
                g = m + 1
            else:
                return True
        return False
```

Remarquons au passage qu'il existe de nombreuses versions similaires de l'algorithme de recherche dichtomique. En voici une autre :

```
>>> def est_present_dichotomie_v2(x, tab):
        g = 0
        d = len(tab) - 1
        while g < d:
            m = (g + d) // 2
            if x > tab[m]:
                g = m + 1
            else:
                d = m
            return tab[g] == x
```

Exercice 8

Que penser du programme suivant :

```
>>> def est_present_dichotomie_v3(x, tab):
        g = 0
        d = len(tab) - 1
        while g < d:
            m = (g + d) // 2
            if x < tab[m]:
                d = m - 1
            else:
                g = m
        return tab[g] == x
```

4 Objets, types mutables et immutables

Les quelques expériences ci-dessous vont mettre à mal l'idée que l'on se fait des variables.

```
>>> a = [3, 9, 16]
>>> b = a
>>> a.append(25)
>>> a
 [3, 9, 16, 25]
>>> b
 [3, 9, 16, 25]
```

Ainsi, l'idée selon laquelle une variable est une boîte dans laquelle on place des valeurs est mise à mal. On voit en effet que les listes contenues dans les variables `a` et `b` ne sont pas indépendantes. Nous allons donc revenir sur certaines notions présentées dans les chapitres précédents qui ne correspondent pas à la réalité.

En Python, toutes les données sont représentées par des objets : les entiers, les nombres flottants, les booléens, les chaînes de caractères, les tuples, les listes et même les fonctions. Un objet possède un identifiant, un type et une valeur. L'identifiant d'un objet est l'adresse mémoire à laquelle est stocké l'objet. Sur un ordinateur 64 bits, c'est donc un entier compris entre 0 et $2^{64} - 1$. L'identifiant d'un objet ne change pas au cours de sa vie. De plus, deux objets distincts ont des identifiants différents. Un objet possède aussi un type qui ne change pas au cours de la vie de l'objet. Enfin, un objet possède une valeur. Cette valeur peut ou non changer selon le type de l'objet. Les types élémentaires `bool`, `int`, `float` et `string` sont *immutables*. C'est aussi le cas des `tuple`. Pour les objets de types immutables, leur valeur ne peut pas changer. Le seul moyen d'avoir un une nouvelle valeur est de créer un nouvel objet. Les listes sont quant à elles de type *mutable*. Pour les objets de types `list`, leur valeur peut changer au cours de leur vie.

Contrairement à ce que nous avons annoncé, une variable n'est pas une boîte dans lequel on met une valeur. C'est plutôt une boîte dans laquelle on met l'identifiant de l'objet auquel il est associé. Par exemple, l'instruction

```
>>> a = 1
```

crée un objet de type `int` dont la valeur est 1 et associe le nom `a` à cet objet. En pratique, on peut voir une variable non pas comme une boîte contenant une valeur, mais comme une boîte contenant l'identifiant d'un objet. Lorsque l'on écrit

```
>>> a = a + 1
```

Python crée un nouvel objet de type `int` dont la valeur est 2 et associe le nom `a` à ce nouvel objet. La fonction `id`, donnant l'identifiant d'un objet nous permet d'observer cela.

```
>>> a = 1
    print(id(a))
    a = a + 1
    print(id(a))
 4326372464
 4326372496
```

On voit qu'après avoir écrit `a = 1`, l'adresse mémoire de l'objet associé à `a` contenant la valeur 1 est différent de la seconde adresse mémoire qui est celle d'un objet qui contient la valeur 2. Lorsque l'on écrit

```
>>> a = 1
    b = a
    print(id(a))
    print(id(b))
 4326372464
 4326372464
```

on voit que les variables `a` et `b` sont associées au même objet contenant la valeur 1. Mais puisque `int` est un type *immutable*, il est impossible de changer la valeur de cet objet. Il est donc impossible de changer la valeur associée à `b` en manipulant la variable `a`. Tout se passe donc comme si `a` et `b` étaient deux boîtes contenant directement la valeur 1.

Les choses changent si on travaille avec des types immutables. Si on écrit

```
>>> a = []
    b = a
    print(id(a))
    print(id(b))
 4547981064
 4547981064
```

alors `a` et `b` sont associées à un même objet de type `list`, dont la valeur est une liste vide. Si on ajoute 1 à cette liste en utilisant la variable `a`, cet ajout se verra lorsque l'on travaille avec la variable `b`.

```
>>> a = []
>>> b = a
>>> a.append(1)
>>> b
 [1]
```

Puisque `a` et `b` sont associées au même objet, on dit que ces variables présentent un *aliasing*.

Tout comme une variable, une liste ne contient pas des valeurs, mais des identifiants d'objets. Il est donc possible d'avoir un aliasing entre différents éléments d'un tableau. Par exemple :

```
>>> a = []
>>> b = [a, a]
>>> b
 [[], []]
>>> a.append(42)
>>> b
 [[42], [42]]
```

Lorsque l'on crée une liste en compréhension, ce sont des objets distincts qui sont associés au éléments d'un tableau.

```
>>> n = 3
>>> a = [[] for k in range(3)]
>>> a
 [[], [], []]
>>> a[0].append(42)
>>> a
 [[42], [], []]
```

Exercice 9

On souhaite créer une liste de listes pour représenter une matrice. Le but est de pouvoir accèder à l'élément d'indice $(i, j) \in [\![0, n-1]\!]^2$ en utilisant la syntaxe `a[i][j]`. Que pensez-vous des méthodes suivantes pour créer la matrice nulle ?

```
>>> a = [[0 for j in range(n)] for i in range(n)]
>>> a = [[0] * n for i in range(n)]
>>> a = [[0] * n] * n
```

En ce qui concerne les `tuple`, leur caractère immutable signifie que les identifiants des objets le formant ne peuvent pas être changés. Si ces objets sont mutables, il est possible de changer leur valeur.

```
>>> t = ([], [])
>>> t[0].append(3)
>>> t
 ([3], [])
```

Enfin ce n'est pas une valeur que l'on passe à une fonction, mais l'identifiant d'un objet. Si la fonction modifie cet objet, elle aura un effet sur l'objet passé à la fonction. On dit alors que la fonction agit par effet de bord. Par exemple :

```
>>> def f(lst):
        lst.append(42)
>>> a = []
>>> f(lst)
>>> a
 [42]
```

Par contre, si on associe la variable locale à un nouvel objet, cela n'aura aucune influence sur l'objet initial.

```
>>> def f(lst):
        lst = [42]
>>> a = []
>>> f(lst)
>>> a
 []
```

On dit que le passage de paramètre se fait *par objet* en Python. Notons que selon les communautés, le nom utilisé pour ce type de passage diffère. Mais c'est inutile de lancer une nouvelle guerre informatique : le nom donné à ce type de passage diffère d'une communauté à l'autre. L'essentiel est de savoir que les personnes qui n'utilisent pas le terme de « passage par objet » ont tort. La dernière phrase est bien sûr à prendre au second degré.

En résumé, on retiendra les points essentiels suivants :

- En Python, les données sont représentées par des *objets*.
- Certains objets sont *immutables*. Ce sont les objets de type `bool`, `int`, `float`, `str`, `tuple`. D'autres objets sont de type *mutable*. C'est le cas par exemple du type `list`.
- Si on travaille avec des objets immutables, tout se passe comme si une variable était une boîte contenant la valeur d'un objet. Cependant, avec des objets mutables, il est essentiel de réaliser qu'une variable contient non pas une valeur, mais l'identifiant d'un objet. Deux variables peuvent contenir le même identifiant et donc offrir un accès au même objet. On dit dans ce cas qu'il y a *aliasing*.

— En Python, le passage des paramètres se fait *par objet*. En pratique, avec des types immutables, tout se fait comme si on copiait la valeur. Mais avec les types mutables, il est essentiel de réaliser que c'est l'identifiant de l'objet et non sa valeur qui est passée à la fonction. Quand on passe un objet mutable à une fonction, cette dernière peut donc potentiellement le modifier. Si c'est le cas, on dit que la fonction agit par effet de bord.

Exercice 10

Les fonctions suivantes ont-elles un effet de bord, si oui décrire cet effet de bord :

```
def pas_de_zero_v0(liste):
    for i in range(len(liste)):
        if liste[i] == 0:
            liste[i] = 1
```

```
def pas_de_zero_v2(liste):
    listeBis = liste
    for i in range(len(listeBis)):
        if listeBis[i] == 0:
            listeBis[i] = 1
    return listeBis
```

```
def pas_de_zero(x):
    if x == 0:
        x = 1
```

5 Concaténation, Slicing

Comme pour les chaînes de caractères, il est possible de concaténer deux listes. On peut même multiplier des listes par un entier.

```
>>> [2, 3, 4] + [5, 6, 7]
 [2, 3, 4, 5, 6, 7]
>>> [1, 2, 3] * 3
 [1, 2, 3, 1, 2, 3, 1, 2, 3]
```

Faites cependant très attention à ces opérations qui sont très dangereuses dès lors que votre liste contient des objets mutables. Étant donné qu'une liste, tout comme une variable, contient en fait les identifiants des objets, on peut avoir quelques surprises dues à l'aliasing. C'est par exemple une très mauvaise idée de construire une liste de listes vides de cette manière.

```
>>> n = 2
>>> liste = [[]] * 2
>>> liste[0].append(0)
>>> liste
 [[0], [0]]
```

On préférera utiliser la compréhension qui provoque moins de soucis :

```
>>> n = 2
>>> liste = [[] for k in range(n)]
>>> liste.append(0)
>>> liste
 [[0], []]
```

Il est possible de construire des sous-listes avec une opération que l'on appelle le slicing.

```
>>> liste = ["Zero", "Un", "Deux"]
>>> a = liste[0:2]
>>> a
 ["Zero", "Un"]
```

Plus généralement, si `liste` est de longueur n et $(a, b) \in [\![0, n]\!]^2$ sont tels que $a \leqslant b$, alors `liste[a:b]` est la sous-liste composée des objets `liste[a]`, ..., `liste[b-1]`. Encore une fois, méfiez-vous de l'aliasing si vous travaillez avec des types mutables :

```
>>> liste = [[], [0], [0, 1]]
>>> sous_liste = liste[0:2]
>>> sous_liste[0].append(3)
>>> sous_liste
 [[3], [0]]
>>> liste
 [[3], [0], [0, 1]]
```

Enfin, Python possède une particularité qui permet d'accéder facilement aux listes par la fin. En effet, l'index -1 peut être utilisé pour accéder au dernier élément, l'index -2 à l'avant-dernier, etc. Par exemple :

```
>>> liste = ["Zero", "Un", "Deux"]
>>> liste[-1]
 "Deux"
>> liste[1:-1]
 ["Zero", "Un"]
```

Les mêmes opérations, excepté les modifications, sont disponibles pour les chaînes de caractères.

Exercice 11

On suppose que n est un entier naturel non nul.

1. Écrire une fonction `barre(L, j)`, qui lorsque L est une liste et $j \in \mathbb{N}^*$, remplace par un 0 le contenu de toutes les cases de L d'indice strictement supérieur à j et divisible par j.
2. En s'appuyant sur le principe du crible d'Eratosthène, écrire une fonction `listePremier(n)` qui lorsque $n \in \mathbb{N}^*$, renvoie la liste de tous les nombres premiers inférieurs ou égaux à n.

6 Types itérables

Les listes sont des objets itérables, c'est-à-dire qu'il est possible d'écrire des boucles `for` où l'indice prend les valeurs successives de la liste.

```
>>> liste = [2, 3, 5, 7, 11]
>>> somme = 0
    for i in liste:
        somme = somme + i
>>> somme
```

Il existe de nombreux types Python qui sont itérables. C'est le cas par exemple des chaînes de caractères. Un peu comme un croupier qui distribue les cartes, les types itérables vous donnent les éléments un à un.

Si votre liste contient des tuples, il est possible d'utiliser plusieurs variables dans la boucle. Par exemple :

```
>>> liste = [("Lyon", 515695), ("Marseille", 869815), ("Paris", 2140526)]
>>> for ville, population in liste:
        print(ville, "a", population, "habitants.")
```

On peut aussi utiliser la fonction `enumerate` sur une liste pour obtenir l'effet suivant :

```
>>> notes = [6, 12, 13]
>>> for k, note in enumerate(notes):
        print("Au devoir", k + 1, "ma note est", note + ".")
```