Chapitre 4. Accès indexé

4.1 Accès indexé dans les listes

Aux chapitres précédents, on accédait aux listes dans l'ordre de leur contenu. On peut cependant aussi utiliser un accès indexé. C'est-à-dire que lorsque l'on écrit :

```
x = L[2]
```

la valeur écrite dans `x` est celle de l'élément d'indice `2` de la liste `L`, on n'a pas besoin de parcourir la liste, on peut directement accéder à l'élément d'indice `2`. Tout comme pour `range`, les indices commencent à partir de `0`, c'est donc sur cet exemple le *troisième* élément de la liste que l'on récupère ainsi, le premier élément étant d'indice 0, le second d'indice 1, etc.

Par exemple la fonction suivante :

```
def sommeListe(L: list) -> int:
    s = 0
    for x in L:
        s = s + x
    return s
```

peut ainsi s'écrire aussi sous la forme :

```
def sommeListe2(L: list) -> int:
    s = 0
    for i in range(len(L)):
        s = s + L[i]
    return s
```

Dans `sommeListe`, on disait à Python de récupérer un à un les éléments de la liste `L`. Dans `sommeListe2` on lui demande seulement que `i` prenne les valeurs `0`, `1`, ..., `len(L)-1`, et l'on récupère alors l'élément d'indice `i` de la liste `L`. Cela revient au final bien au même.

L'intérêt d'utiliser un accès indexé, c'est que l'on a un meilleur contrôle sur le parcours de la liste. Par exemple, on peut calculer la somme de la première moitié de la liste seulement :

```
def sommeListeMoitie(L: list) -> int:
    s = 0
    for i in range(len(L) // 2):
        s = s + L[i]
    return s
```

On peut aussi parcourir deux listes en même temps (supposées de même taille), pour calculer un produit scalaire par exemple :

```
def produitScalaire(L1: list, L2: list) -> int:
    s = 0
    for i in range(len(L1)):
        s = s + L1[i] * L2[i]
    return s
```

On peut également *écrire* dans la liste avec l'affectation :

```
L[2] = 0
```

écrit 0 à l'indice 2 de la liste L. Attention, la liste est *modifiée*, ainsi la fonction :

```
def metAZero(L: list):
    for i in range(len(L)):
        L[i] = 0
```

modifie la liste qui a été passée en paramètre
Enfin, on peut facilement construire une liste ainsi :

```
maListe = [0] * 1000
```

qui "multiplie" donc par 1000 la liste contenant un seul élément 0, produisant ainsi une liste de 1000 éléments tous égaux à 0.

Exercice 4.1.1 Écrire une fonction `maximumListe(L: list) -> int` qui retourne le maximum des nombres contenus dans la liste. Attention, par rapport à l'exercice 2.1.3, cette fois-ci on ne suppose pas que les nombres sont forcément positifs.

Exercice 4.1.2 Écrire une fonction `position(L: list, x: int) -> int` qui retourne l'indice de la première apparition du nombre `x` dans la liste `L`, s'il y apparaît, `-1` sinon.

Exercice 4.1.3 Écrire une fonction `positionDernier(L: list, x: int) -> int` qui retourne l'indice de la *dernière* apparition du nombre `x` dans la liste `L`, s'il y apparaît, `-1` sinon.

Exercice 4.1.4 Écrire une fonction `absListe(L: list)` qui modifie la liste pour remplacer chaque élément par sa valeur absolue (en utilisant la fonction `abs`). Note : puisque la fonction modifie la liste, elle n'a pas besoin de retourner une valeur, il n'y a donc pas d'instruction `return` à mettre, la fonction se termine simplement en retournant rien (`None`).

Exercice 4.1.5 Taper les instructions suivantes dans Python Tutor et prenez le temps d'expliquer précisément l'évolution des variables pendant l'exécution pas à pas de ces instructions.

```
L = [1] * 10
L[1] = 3

for i in range(10):
    L[i] = i*i
x = L[3]
```

Exercice 4.1.6 Écrire une fonction `sommeListes(L1: list, L2: list) -> list` qui calcule la somme de deux listes (supposées de même taille) élément par élément, c'est-à-dire que `sommeListes([34,54],[10,11])` doit retourner la liste `[44,65]`.

4.2 Exercices de révisions et compléments

Exercice 4.2.1 Écrire une fonction `inverseListe(L: list)` qui "inverse" l'ordre de la liste : elle crée une nouvelle liste dont le premier élément est le dernier de `L`, le deuxième est l'avant-dernier de `L`, etc. jusqu'au dernier élément qui est le premier de `L`.

Exercice 4.2.2 Écrire une fonction `alterne(L1: list, L2: list) -> list` qui, étant données deux listes `L1` et `L2` supposées de même taille, construit une liste deux fois plus grande, contenant l'alternance des éléments de `L1` et de `L2`.

Exercice 4.2.3 Écrire une fonction `estTrie(L: list) -> bool` qui retourne `True` si la liste `L` est triée dans l'ordre croissante, et `False` sinon.

4.3 Pour aller plus loin : Les chaînes de caractères

En plus des listes, Python permet de manipuler des chaînes de caractères, autrement dit, du texte !

Par exemple :

```
s = "abc"
```

s est alors une chaîne de caractères contenant le caractère `"a"` à l'indice 0, le caractère `"b"` à l'indice 1, et le caractère `"c"` à l'indice 2.

On peut également parcourir la chaîne de caractères comme une liste :

```
for c in s:
    print(c)
```

ou de manière indexée :

```
for i in range(len(s)):
    print(s[i])
```

En anglais les chaînes de caractères s'appellent *string*, Python affichera donc pour elles le type `str`.

4.3.1 Parcours de chaînes de caractères

Exercice 4.3.1 Écrire une fonction `nombreA(s: str) -> int` qui retourne le nombre d'apparitions du caractère `"a"` dans la chaîne de caractères `s`.

Exercice 4.3.2 Écrire une fonction `nombreQU(s: str) -> int` qui retourne le nombre d'apparitions des caractères `"q"` et `"u"` l'un après l'autre consécutivement dans la chaîne de caractères `s`.

4.3.2 Fabrication de nouvelles chaînes de caractères

Contrairement aux listes, les chaînes de caractères ne sont pas modifiables :

```
s[0] = "a"
```

nous répond

```
TypeError: 'str' object does not support item assignment
```

Du coup pour produire une nouvelle chaîne de caractères, on est obligé d'en fabriquer une nouvelle, morceau par morceau, par exemple ce code qui duplique les lettres de la chaîne `s` :

```
s = "abc"
s2 = ""
for c in s:
   s2 = s2 + c + c
```

Exercice 4.3.3 Écrire une fonction `padIpadO(s: str) -> str` qui retourne une nouvelle chaîne contenant la même chose que `s` sauf que tous les caractères `"i"` et `"o"` sont remplacés par le caractère `"a"`.

Exercice 4.3.4 On peut aussi créer des listes de chaînes de caractères !

Taper les instructions suivantes et observez :

```
L = [ "zéro", "un", "deux", "trois" ]
x = L[1]
L[3] = "quatre!"
```

Écrire une fonction `nomJour(i: int) -> str` qui reçoit en paramètre le numéro d'un jour de la semaine (entre 1 et 7) et qui renvoie une chaîne de caractères contenant le nom de ce jour.

Exercice 4.3.5 Écrire une fonction `inverseChaine(s: str) -> str` qui retourne une nouvelle chaîne contenant le même contenu que `s`, mais dans l'ordre inverse : le premier caractère de `s` se retrouve à la fin de la chaîne retournée, le deuxième de `s` se retrouve en avant-dernière position de la chaîne, etc. et inversement le dernier caractère de `s` se retrouve au début de la chaîne retournée, l'avant-dernier caractère de `s` se retrouve en deuxième position de la chaîne retournée, etc.

Exercice 4.3.6 Chiffre de César

Le principe du chiffre de César est de chiffrer un texte en remplaçant chacun de ses caractères par le caractère qui est 13 positions plus loin dans l'alphabet. Ainsi tous les caractères `"a"` deviennent `"n"`, tous les caractères `"b"` deviennent `"o"`, etc., et inversement tous les caractères `"n"` deviennent `"a"`, tous les caractères `"o"` deviennent `"b"`, etc.

Écrire une fonction `cesar(s: str) -> str` qui retourne une version chiffrée de la chaîne `s` avec le chiffre de César. On supposera pour simplifier que la chaîne ne contient que des lettres minuscules ou des espaces (ces derniers ne seront pas modifiés par le chiffrage). Que se passe-t-il si l'on l'appelle deux fois sur la même chaîne ? Que signifie le message chiffré `obawbhe` ?

Note : plutôt qu'énumérer tous les caractères possibles, on peut utiliser d'une part la fonction prédéfinie `ord(c: str) -> int` qui retourne le numéro du caractère `c`, et d'autre part `chr(i: int) -> str` qui retourne le caractère qui a pour numéro `i`.

Exercice 4.3.7 Longueur de monotonie

- Écrire une fonction `compte(L: list, n: int) -> int` qui compte le nombre d'apparition du nombre `n` dans la liste `L`.

Une monotonie est une succession de valeurs identiques (possiblement une seule valeur).

- Écrire une fonction `tailleMonotonie(L: list, n: int) -> int` qui retourne la longueur de la plus grande monotonie de `n` dans la liste `L`.

- Écrire une fonction `plusGrandMonotonie(L: list) -> int` qui retourne l'entier de la liste ayant la plus grande monotonie. En cas d'égalité, elle doit retourner celui qui apparaît en premier dans la liste.

Exercice 4.3.8 Écrire une fonction `nbMots(s: str) -> int` qui retourne le nombre de mots contenus dans la chaîne `s`. On supposera pour simplifier qu'ils sont simplement séparés par une espace.

Exercice 4.3.9 Écrire une fonction `niOuiNiNon(s: str) -> bool` qui retourne `True` si la chaîne `s` ne contient ni le mot `oui` ni le mot `non`, et `False` sinon. Pour simplifier on ne cherchera que ces mots en minuscule.

4.4 L'essentiel du chapitre

4.4.1 Listes

On peut accéder à l'élément d'indice `i` d'une liste `L` avec la notation `L[i]`. On a ainsi deux manières de parcourir une liste :

```
for x in L:
    ... x ...
```

```
for i in range(len(L)):
    ... L[i] ...
```

La deuxième forme est nécessaire lorsque l'on a besoin de jouer sur les indices. Sinon, la première forme est plus simple à écrire.

On peut construire une liste en "multipliant" simplement une liste contenant un seul élément, par exemple pour construire une liste contenant 1000 fois l'élément 0 :

```
maListe = [0] * 1000
```

Attention : il y a donc deux significations pour les caractères `[]` :

- `[i]` seul, construit la liste contenant le nombre `i`.
- `L[i]` avec `L` qui est une liste, récupère l'élément d'indice `i` dans la liste `L`.

4.4.2 (Pour aller plus loin : Chaînes de caractères)

Une chaîne de caractères se comporte essentiellement comme une liste :

```
for c in s:
    ... c ...
```

```
for i in range(len(s)):
    ... s[i] ...
```

Sauf que l'on ne peut pas la modifier, on ne peut pas écrire `s[i] = ...`, il faut alors la reconstruire depuis zéro :

```
s2 = ""
for c in s:
    s2 = s2 + c
```

4.4.3 Types

Il ne faut pas confondre les différents types :

- Les indices au sein des listes et chaînes de caractères (que l'on appelle en général plutôt `i`)
- Les listes (que l'on appelle en général plutôt `L`)
- Les éléments contenus dans les listes (que l'on appelle en général plutôt `x`, `element`, ...)
- Les chaines de caractères (que l'on appelle en général plutôt `s`).
- Les caractères eux-même (que l'on appelle en général plutôt `c`).