TRAVAUX PRATIQUES n° 6
Introduction aux listes

Dans ce TP, nous allons apprendre à créer des listes, accéder à leurs éléments, les modifier, puis écrire les premiers algorithmes de parcours (somme, minimum, etc.). L'objectif est de bien maîtriser indices et boucles avant d'aborder, au TP suivant, des motifs plus généraux (compter, tester, chercher).

I. Définition de listes

Une liste est un type de donnée de Python (type `list`) permettant de stocker plusieurs données en une seule variable.

Il existe plusieurs manières de définir une liste :

⋆ **Définition en extension** :

```
nom_liste = [e_1, e_2, ..., e_n]
```

⋆ **Définition d'une liste contenant plusieurs fois la même valeur** :

```
nom_liste = nombre * [valeur]
```

⋆ **Définition en compréhension** :

- `nom_liste = [f(k) for k in range(a, b)]`
- `nom_liste = [f(k) for k in autre_liste]`

Voici quelques exemples que vous pouvez tester dans le shell pour mieux comprendre chaque syntaxe. Pour chaque liste, on doit recopier la définition ci-dessous **et** retaper le nom de la liste dans le shell. **N'hésitez pas à modifier ces exemples pour voir les effets des modifications !**

```
>>> liste_1 = [1, 3, 5]
>>> liste_2 = [5, "Bonjour !", 4.56, True]
>>> liste_3 = [k**2 + 2*k for k in range(4, 10)]
>>> liste_4 = [i**3 for i in range(100)]
>>> liste_5 = [2*e-4*e**2 for e in liste_3]
>>> liste_6 = 33 * [2.3]
```

Exercice 1 : Pour chacune des questions suivantes, écrire la définition de la liste dans l'éditeur, exécuter le code, puis vérifier le contenu de la liste en tapant son nom dans le shell.

1) Définir une liste vide nommée `vide`.
2) Définir la liste `premiers` des 10 premiers nombres premiers (les écrire un par un).
3) Définir la liste `zeros` contenant 100 fois le nombre 0.
4) Définir la liste `gauss` des nombres de la forme $\frac{n(n+1)}{2}$ où $n \in [\![-10, 10]\!]$ (il doit donc y avoir 21 éléments). Faire en sorte que les éléments de cette liste soient bien des entiers (`int`) et non des flottants (`float`).
5) Définir la liste `carres` des carrés des nombres de la liste `gauss`.

II. Accéder aux éléments et modifier un élément

Une fois qu'une liste est définie, on doit pouvoir récupérer la valeur d'un élément en particulier. Pour cela, il faut donner à Python deux informations :

- ⋆ le nom de la liste dans laquelle se trouve l'élément ;
- ⋆ la position de l'élément dans cette liste : on appelle cette position **l'indice de l'élément dans la liste**.

Ainsi la syntaxe pour accéder à un élément particulier d'une liste est la suivante :

```
nom_liste[indice]
```

Pour une liste ayant n éléments, l'indice varie de 0 (pour le premier élément) à $n-1$ (pour le dernier élément).

Par exemple, pour la liste `L = [5, 3.14, "abc", -3, 0]`, on a

Élément	`5`	`3.14`	`"abc"`	`-3`	`0`
Indice	`0`	`1`	`2`	`3`	`4`

Exercice 2 : Deviner ce qui devrait apparaître après chacune de ces instructions, puis vérifier en exécutant l'instruction dans le shell.

```
>>> L = [5, 3.14, "abc", -3, 0]
>>> L[0]
>>> L[1]
>>> L[2]
>>> L[3]
>>> L[4]
>>> L[5]
```

Que signifie l'erreur `IndexError` et pourquoi apparaît-elle lors de l'exécution de la dernière instruction ?

Variante

Il est aussi possible d'accéder à un élément d'une liste par un indice négatif : dans ce cas, on compte en partant de la fin. Pour une liste à n éléments, le dernier élément est l'élément d'indice -1, l'avant-dernier est l'élément d'indice -2 et le premier élément est l'élément d'indice $-n$.
Par exemple, pour la liste `L = [5, 3.14, "abc", -3, 0]`, on a

Élément	`5`	`3.14`	`"abc"`	`-3`	`0`
Indice	`0`	`1`	`2`	`3`	`4`
Indice négatif	`-5`	`-4`	`-3`	`-2`	`-1`

Exercice 3 : Deviner ce qui devrait apparaître après chacune de ces instructions, puis vérifier en exécutant l'instruction dans le shell.

```
>>> L[-1]
>>> L[-2]
>>> L[-3]
>>> L[-4]
>>> L[-5]
>>> L[-6]
```

Pourquoi observe-t-on encore une erreur `IndexError` ?

Pour modifier un élément d'une liste, on utilise la syntaxe :

```
nom_liste[indice] = nouvelle_valeur
```

Exercice 4 : Deviner ce qui devrait apparaître après chacune de ces instructions, puis vérifier en exécutant l'instruction dans le shell.

```
>>> L[2] = "def"
>>> L
>>> L[-1] = L[-1] + 2
>>> L
>>> L[1] = L[0] + L[-1]
>>> L[-2] = L[2]
>>> L
>>> L[1] = L[0] + L[-1]
>>> L
```

III. Fonctions sur les listes

Il y a 4 fonctions à connaître sur les listes et la première est la plus importante :

- ⋆ Fonction `len` : calcule la longueur (la taille) d'une liste, c'est-à-dire le nombre d'éléments d'une liste.
- ⋆ Fonction `sum` : calcule la somme des éléments d'une liste numérique.
- ⋆ Fonction `max` : calcule le maximum des éléments d'une liste numérique.
- ⋆ Fonction `min` : calcule le minimum des éléments d'une liste numérique.

Exercice 5 : Deviner ce qui devrait apparaître après chacune de ces instructions, puis vérifier en exécutant l'instruction dans le shell.

```
>>> L = [4, 6, -3, 13, 2, 4, 6, 1, -3]
>>> len(L)
>>> sum(L)
>>> max(L)
>>> min(L)
```

Exercice 6 :

1) Calculer, en utilisant la définition en compréhension des listes et la fonction `sum`, la somme $\sum_{k=0}^{100} k^4$.

2) Écrire une fonction `somme_harmonique(n)` qui calcule et renvoie la valeur de la somme $\sum_{k=2}^{n} \frac{1}{k-1}$.

IV. Concaténation et méthodes

On peut « coller » deux listes : c'est la **concaténation**. Syntaxe : `liste_1 + liste_2`.

Les listes possèdent aussi des méthodes, dont la syntaxe est `liste.nom_methode(...)` (et non `nom_fonction(liste)`). Les méthodes marquées d'une étoile dans le résumé **modifient** la liste : ce sont des fonctions à effet de bord.

Testez les exemples suivants **dans le shell** (après chaque ligne, tapez le nom de la liste) :

```
>>> liste_1 = [1, 3, 5]
>>> liste_2 = [2, 4, 6, 8]
>>> liste_1 + liste_2
>>> liste = []
>>> liste.append(5)
>>> liste.append(-3)
>>> liste.append(12)
>>> liste.insert(1, 12)
>>> x = liste.pop(2)
>>> liste.remove(12)
>>> liste.reverse()
>>> liste.sort()
>>> liste.index(5)
>>> liste.count(5)
```

Exercice 7 (*Méthodes*) : Partir de `L = []` et prédire l'état de `L` (et la valeur de `x` le cas échéant) après chaque instruction, puis vérifier dans le shell.

```
>>> L = []
>>> L.append(3)
>>> L.append(7)
>>> L.append(1)
>>> L.insert(1, 5)
>>> x = L.pop(2)
>>> L
>>> x
```

V. Parcourir une liste

Parcourir une liste consiste, à l'aide d'une boucle, à examiner chaque élément. Deux types de parcours :

- ⋆ Parcours des **éléments** : `for e in nom_liste:`
- ⋆ Parcours des **indices** : `for i in range(len(nom_liste)):` puis `nom_liste[i]`

Exercice 8 : Deviner ce qui devrait apparaître, puis vérifier en exécutant le code.

```
L = [4, 6, -3, 13, 2, 4, 6, 1, -3]

for e in L:
    print(e)

for i in range(len(L)):
    print(i)

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

VI. Premiers algorithmes

Sauf mention contraire, il est interdit d'utiliser `sum`, `max`, `min`, `in`, `liste.index`, `liste.count` et `liste.sort`. Seule la fonction `len` est autorisée. On réécrit donc soi-même ces calculs avec une boucle : c'est l'objet de cette section. (Dans les sections précédentes, `sum`, `max` et `min` restaient bien sûr autorisés.)

Exercice 9 (*Liste aléatoire*) : Écrire une fonction `liste_aleatoire(n, a, b)` renvoyant une liste de `n` entiers choisis au hasard entre `a` et `b` (inclus).

On utilisera `randint`. *Cette fonction servira à tester les suivantes.*

Exercice 10 (*Sommes et produits*) :

1) Écrire une fonction `somme(liste)` renvoyant la somme d'une liste de nombres.
2) Écrire une fonction `moyenne(liste)` renvoyant la moyenne d'une liste de nombres (on suppose la liste non vide).
3) Écrire une fonction `produit(liste)` renvoyant le produit d'une liste de nombres.

```
>>> somme([1, 2, 3, 4])
10
>>> moyenne([1, 2, 3, 4])
2.5
>>> produit([1, 2, 3, 4])
24
```

Exercice 11 (*Présence et extrêmes*) :

1) Écrire une fonction `est_present(element, liste)` renvoyant `True` si `element` est dans `liste`, `False` sinon.
2) Écrire une fonction `minimum(liste)` et une fonction `maximum(liste)` (on suppose la liste non vide).

```
>>> est_present(3, [1, 5, 3, 2])
True
>>> est_present(4, [1, 5, 3, 2])
False
>>> minimum([4, 6, -3, 13, 2])
-3
>>> maximum([4, 6, -3, 13, 2])
13
```

Exercice 12 (*Tester un algorithme*) : Voici quatre fonctions dont le but est de déterminer si une liste n'est constituée que de nombres positifs ou nuls :

```
1  def tous_positifs_1(liste):
2      for i in range(len(liste)):
3          if liste[i] >= 0:
4              return True
5          else:
6              return False
7
8  def tous_positifs_2(liste):
9      for i in range(len(liste)):
10         if liste[i] >= 0:
11             return True
12     return False
13
14 def tous_positifs_3(liste):
15     for i in range(len(liste)):
16         if liste[i] < 0:
17             return False
18         else:
19             return True
20
21 def tous_positifs_4(liste):
22     for i in range(len(liste)):
23         if liste[i] < 0:
24             return False
25     return True
```

Parmi ces fonctions, lesquelles sont correctes ? Pour les autres, donner un contre-exemple.

Exercice 13 (* *Crible d'Ératosthène*) : Écrire une fonction `eratosthene(N)` renvoyant la liste des nombres premiers $\leqslant N$ ($N \geqslant 2$) en suivant l'algorithme :

- ⋆ `premiers = []` et `est_premier` liste de taille `N+1`, avec `est_premier[0] = est_premier[1] = False` et `True` ailleurs ;
- ⋆ pour `i` de `0` à `N` : si `est_premier[i]` est vrai, marquer `False` tous les multiples `k*i` pour `k \geqslant 2`, puis ajouter `i` à `premiers` ;
- ⋆ renvoyer `premiers`.

VII. Résumé des syntaxes

- ⋆ **Définition :** `[e_1, ..., e_n]`, `n * [valeur]`, `[f(k) for k in range(a, b)]`
- ⋆ **Accès / modification :** `liste[i]`, `liste[i] = x`, concaténation `liste_1 + liste_2`
- ⋆ **Fonctions :** `len`, `sum`, `max`, `min`
- ⋆ **Parcours :** `for e in L:` ou `for i in range(len(L)):`

Méthode	Syntaxe	Effet
`append`*	`liste.append(valeur)`	Ajoute `valeur` à la fin
`insert`*	`liste.insert(i,valeur)`	Insère `valeur` en indice `i`
`pop`*	`liste.pop(i)`	Supprime et renvoie l'élément d'indice `i`
`remove`*	`liste.remove(valeur)`	Supprime la première occurrence de `valeur`
`reverse`*	`liste.reverse()`	Inverse l'ordre
`sort`*	`liste.sort()`	Trie par ordre croissant
`index`	`liste.index(valeur)`	Indice de la première occurrence
`count`	`liste.count(valeur)`	Nombre d'occurrences

Les méthodes avec une étoile **modifient** la liste.