Les listes

Doc. 3,2

Objectif

Ranger des valeurs de même type dans une liste, modifier et parcourir ses éléments, puis construire des listes, y compris des listes de listes.

§1. Quand un tuple ne suffit plus

Un professeur range les notes d'un devoir dans un tuple :

```python
notes = (12, 15, 9, 14)
```

En relisant les copies, il s'aperçoit que la troisième note est fausse : c'est 11 et non 9. Avec un tuple, l'instruction `notes[2] = 11` provoque une erreur, puisqu'un tuple n'est pas mutable (Doc. 3,1). Il faudrait reconstruire un tuple complet, `(notes[0], notes[1], 11, notes[3])`, pour changer une seule valeur. Avec 35 élèves, ce serait pénible.

Python propose pour cela un autre type construit, la **liste**, qui s'écrit entre crochets.

Q1. Saisir les instructions suivantes dans la console :

```python
>>> notes = [12, 15, 9, 14]
>>> type(notes)
>>> notes[2] = 11
>>> notes
```

La modification a-t-elle été acceptée ?

À retenir

Une **liste** est une suite ordonnée de valeurs, écrite entre crochets et séparées par des virgules : `[12, 15, 9, 14]`. Son type est `list`.

Une liste est **mutable** : on peut modifier ses éléments après sa création, alors qu'un tuple ne le permet pas.

Dans cette activité, tous les éléments d'une même liste ont le même type.

Remarque. Il ne faut pas confondre une liste avec un **tableau**. Un tableau est un ensemble de valeurs de même type, rangées les unes à la suite des autres dans la mémoire : on dit qu'elles sont **contiguës**. Une liste Python ne répond pas à cette définition : ses éléments peuvent être de types différents, et ils ne sont pas rangés côte à côte dans la mémoire. Python se sert de tableaux pour fabriquer ses listes, mais une liste n'est pas un tableau.

§2. Lire et modifier les éléments d'une liste

Bonne nouvelle : tout ce qu'on a appris sur les indices des tuples reste vrai pour les listes.

Q2. Saisir `notes = [12, 15, 11, 14]` dans la console. Prévoir le résultat de chacune des instructions suivantes, puis vérifier :

```python
>>> notes[0]
```

```
>>> len(notes)
>>> notes[len(notes) - 1]
>>> notes[4]
>>> 15 in notes
>>> []
>>> len([])
```

Ce qui est nouveau, c'est qu'on peut écrire `notes[i]` à gauche du signe `=`. L'élément d'indice `i` est alors remplacé par une nouvelle valeur. Les opérateurs d'affectation du Doc. 2,2 fonctionnent aussi : `notes[1] += 2` ajoute 2 à l'élément d'indice 1.

Q3. Le professeur décide d'ajouter un point à chaque note. Un élève propose :

```python
notes = [12, 15, 11, 14]
for n in notes:
    n += 1
```

Un autre élève propose :

```python
notes = [12, 15, 11, 14]
for i in range(len(notes)):
    notes[i] += 1
```

1. Exécuter les deux propositions et afficher `notes` après chacune d'elles. Laquelle fonctionne ?
2. Expliquer pourquoi l'autre ne modifie pas la liste.

À retenir

`t[i] = valeur` remplace l'élément d'indice `i` de la liste `t`.

Pour modifier les éléments d'une liste dans une boucle, on parcourt les **indices** : `for i in range(len(t)):`. La boucle `for element in t:` permet de lire les éléments, mais pas de les modifier.

§3. Construire une liste

§3.1. Une liste de taille fixée à l'avance

L'opérateur `*` appliqué à une liste et à un entier répète la liste. Ainsi `[0] * 6` vaut `[0, 0, 0, 0, 0, 0]`. On obtient une liste de 6 éléments, qu'on peut ensuite remplir par indice.

Q4. Recopier et compléter la fonction suivante, qui doit construire la liste des carrés des entiers de 0 à `n - 1` :

```python
def carres(n: int) -> list[int]:
    """Retourne la liste des carrés des entiers de 0 à n - 1.

    >>> carres(5)
    [0, 1, 4, 9, 16]
    """
    resultat = [0] * n
    for i in range(n):
        ...
```

```
    return resultat
```

§3.2. Une liste en compréhension

En mathématiques, on décrit parfois un ensemble par une règle plutôt qu'en écrivant tous ses éléments : $\{k^2 \mid k \in \{0,1,2,3,4\}\}$ se lit « l'ensemble des k^2 pour k appartenant à $\{0,1,2,3,4\}$ ». Python permet d'écrire une liste de la même façon :

```python
[k * k for k in range(5)]
```

Cette expression se lit presque comme la phrase : « la liste des `k * k` pour `k` parcourant `range(5)` ». On dit que la liste est construite **en compréhension**. Elle remplace les quatre lignes du corps de la fonction `carres`.

Q5. Prévoir la liste obtenue pour chacune des expressions suivantes, puis vérifier dans la console :

```python
>>> [2 * k + 1 for k in range(5)]
>>> [n / 2 for n in [12, 15, 11, 14]]
>>> [len(mot) for mot in ["tuple", "liste", "indice"]]
>>> [k for k in range(1, 21) if k % 3 == 0]
```

À quoi sert la partie `if k % 3 == 0` dans la dernière expression ?

Q6. Écrire une liste en compréhension qui donne :

1. les 10 premiers multiples de 7 : `[7, 14, 21, ..., 70]` ;
2. les notes de la liste `notes = [12, 15, 8, 14, 6, 17]` qui sont inférieures à 10 ;
3. la liste `[True, False, True, False, True, False]` qui indique, pour chaque entier de 0 à 5, s'il est pair.

À retenir

On peut construire une liste :

— de taille fixée, avec `[valeur] * n`, puis la remplir par indice ;
— **en compréhension**, avec `[expression for element in iterable]`. On peut ajouter un filtre : `[expression for element in iterable if condition]`.

§4. Une liste, plusieurs noms

§4.1. Une surprise

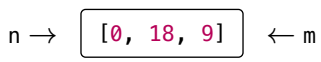
Q7. Saisir les instructions suivantes dans la console, en prévoyant le résultat de la dernière ligne avant de la valider :

```python
>>> n = [12, 18, 9]
>>> m = n
>>> m[0] = 0
>>> n
```

Le résultat est-il celui que vous attendiez ?

Ce résultat surprend presque tout le monde la première fois. Pour le comprendre, il faut revenir à ce que fait vraiment une affectation (Doc. 2,2) : une variable **référence** une valeur. Elle ne contient pas la liste, elle indique où elle se trouve, un peu comme une étiquette collée dessus.

L'instruction `m = n` ne fabrique donc pas une deuxième liste. Elle colle une deuxième étiquette, `m`, sur la **même** liste :



Quand on écrit `m[0] = 0`, on modifie cette liste unique. En la regardant par l'étiquette `n`, on voit forcément la modification. On dit que `m` et `n` sont des **alias**.

Avec les nombres et les tuples, ce phénomène existe aussi, mais il ne se voit jamais : comme on ne peut pas modifier un nombre ou un tuple, on ne peut pas constater que deux étiquettes désignent le même objet. C'est la mutabilité des listes qui le rend visible.

Q8. Pour obtenir une vraie copie, on peut construire une nouvelle liste en compréhension. Prévoir puis vérifier le contenu de `n` et de `c` :

```python
>>> n = [12, 18, 9]
>>> c = [x for x in n]
>>> c[0] = 0
>>> n
>>> c
```

§4.2. Une fonction peut modifier la liste qu'on lui donne

Lors d'un appel, le paramètre reçoit la valeur de l'argument, exactement comme dans une affectation. Si l'argument est une liste, le paramètre devient donc un alias de cette liste.

Q9. Recopier les deux fonctions suivantes dans l'éditeur :

```python
def reinitialiser_1(notes: list[float]) -> None:
    notes = [0] * len(notes)

def reinitialiser_2(notes: list[float]) -> None:
    for i in range(len(notes)):
        notes[i] = 0
```

Prévoir, puis vérifier, le contenu de `releve` après chaque appel :

```python
>>> releve = [12, 15, 9, 14]
>>> reinitialiser_1(releve)
>>> releve
>>> reinitialiser_2(releve)
>>> releve
```

Expliquer la différence de comportement entre les deux fonctions.

⚠ Effet de bord

Une fonction qui modifie une liste reçue en argument agit sur la liste du programme appelant. On parle d'**effet de bord**. C'est parfois ce qu'on veut, mais il faut alors l'annoncer clairement dans la documentation de la fonction. Si on ne le veut pas, la fonction doit travailler sur une copie, ou construire et retourner une nouvelle liste.

✅ À retenir

Après `m = n`, les variables `m` et `n` référencent la **même** liste : ce sont des **alias**. Toute modification faite par l'une est visible par l'autre.

Pour obtenir une copie indépendante, on construit une nouvelle liste, par exemple `[x for x in n]`.

Une fonction qui reçoit une liste peut la modifier (effet de bord) avec `t[i] = ...`. En revanche, une nouvelle affectation du paramètre (`t = ...`) ne change pas la liste de l'appelant.

§5. Des listes de listes

Les éléments d'une liste peuvent eux-mêmes être des listes. C'est ainsi qu'on représente des valeurs rangées en lignes et en colonnes, comme les cases d'une grille de jeu. En mathématiques, des nombres rangés de cette façon forment une **matrice**.

```python
m = [[1, 2, 3],
     [4, 5, 6]]
```

`m` est une liste de deux éléments : `m[0]` est la liste `[1, 2, 3]`, qu'on voit comme la première ligne, et `m[1]` est la liste `[4, 5, 6]`. Pour atteindre un nombre, on applique deux indices l'un après l'autre : `m[1]` donne la ligne d'indice 1, puis `m[1][2]` donne l'élément d'indice 2 de cette ligne.

Q10. Avec la liste `m` ci-dessus, prévoir puis vérifier :

```python
>>> m[0]
>>> m[1][2]
>>> len(m)
>>> len(m[0])
>>> m[0][1] = 0
>>> m
```

Que représentent `len(m)` et `len(m[0])` pour la matrice ?

Q11. Écrire la fonction `somme_matrice` dont la spécification est :

```python
def somme_matrice(m: list[list[int]]) -> int:
    """Retourne la somme de tous les éléments de la matrice m.

    >>> somme_matrice([[1, 2, 3], [4, 5, 6]])
    21
    """
```

Indication. Utiliser deux boucles imbriquées : l'une parcourt les indices des lignes, l'autre ceux des colonnes.

Q12. Dans le Doc. 2,4, la fonction `table_multiplication` retournait une chaîne de caractères. On veut maintenant une matrice, pour pouvoir retrouver chaque produit par ses indices. Écrire la fonction `table_multiplication` dont la spécification est :

```python
def table_multiplication(n: int) -> list[list[int]]:
    """Retourne la table de multiplication des entiers de 1 à n,
    sous forme d'une matrice de n lignes et n colonnes.

    >>> table_multiplication(4)
    [[1, 2, 3, 4], [2, 4, 6, 8], [3, 6, 9, 12], [4, 8, 12, 16]]
    """
```

Indication. Une ligne de la table est une liste en compréhension. La table entière est une liste de lignes, elle aussi en compréhension.

Avec `t = table_multiplication(10)`, que vaut `t[6][7]` ? Quel produit cet élément représente-t-il ?

⚠ Un piège classique

Pour fabriquer une grille de 3 lignes et 3 colonnes remplie de zéros, l'écriture `[[0] * 3] * 3` semble naturelle. Pourtant :

```
>>> g = [[0] * 3] * 3
>>> g[0][0] = 5
>>> g
[[5, 0, 0], [5, 0, 0], [5, 0, 0]]
```
Python

Le `* 3` extérieur ne fabrique pas trois lignes : il place trois fois **la même** ligne dans la liste. Les trois lignes sont des alias. La bonne écriture fabrique une nouvelle ligne à chaque tour :

```
>>> g = [[0] * 3 for i in range(3)]
```
Python

✅ À retenir

Une liste de listes permet de représenter une **matrice**, c'est-à-dire des valeurs rangées en lignes et en colonnes. `m[i]` est la ligne d'indice `i`, et `m[i][j]` l'élément de la ligne `i` et de la colonne `j`.

`len(m)` donne le nombre de lignes et `len(m[0])` le nombre de colonnes. On parcourt une matrice avec deux boucles imbriquées.

§6. Exercices de synthèse

⚠ Consigne

Écrire chaque fonction dans la partie principale de l'éditeur, avec sa spécification complète (signature et documentation). Tester ensuite chacun des exemples de la documentation dans la console. Les fonctions `sum`, `min`, `max` et la méthode `index` de Python ne doivent pas être utilisées.

Q13. **Moyenne.** Écrire la fonction `moyenne` dont la spécification est :

```
def moyenne(notes: list[float]) -> float:
    """Retourne la moyenne des éléments de notes, qui ne doit pas être vide.

    >>> moyenne([12, 15, 9, 14])
    12.5
    """
```
Python

Vérifier la précondition à l'aide d'une assertion (voir Doc. 3,1).

Q14. **Recherche d'une valeur.** Écrire la fonction `indice` dont la spécification est :

```
def indice(valeurs: list[int], cherchee: int) -> int:
    """Retourne l'indice de la première occurrence de cherchee dans
```
Python

```
    valeurs, ou -1 si cherchee n'est pas dans valeurs.

    >>> indice([4, 8, 15, 16, 23, 42], 16)
    3
    >>> indice([4, 8, 15, 16, 23, 42], 7)
    -1
    >>> indice([5, 3, 5], 5)
    0
    """
```

Dans le pire des cas, combien de comparaisons la fonction effectue-t-elle pour une liste de 1000 éléments ? Et pour une liste deux fois plus longue ?

Q15. **Statistiques d'un devoir.** On veut écrire la fonction `statistiques` dont la spécification est :

```python
def statistiques(notes: list[float]) -> tuple[float, int, int, int]:
    """Retourne un tuple (m, bas, milieu, haut) où m est la moyenne
    des notes, bas le nombre de notes strictement inférieures à m - 3,
    milieu le nombre de notes comprises entre m - 3 et m + 3 (bornes
    incluses), haut le nombre de notes strictement supérieures à m + 3.

    >>> statistiques([12, 15, 8, 14, 6, 17])
    (12.0, 2, 3, 1)
    >>> statistiques([10, 7, 13])
    (10.0, 0, 3, 0)
    """
```

1. Vérifier à la main le premier exemple de la documentation.
2. Écrire la fonction, en utilisant la fonction `moyenne` de la question précédente.
3. Un élève propose de compter les notes de chaque groupe avec une fonction qui compte les valeurs comprises dans un intervalle $[a\ ;\ b]$, bornes incluses, appelée trois fois : sur $[0\ ;\ m-3]$, sur $[m-3\ ;\ m+3]$ et sur $[m+3\ ;\ 20]$. Le deuxième exemple de la documentation montre pourquoi cette idée donne un résultat faux. Expliquer.

Q16. **Tirage du Loto sans répétition.** Dans le Doc. 2,4, la fonction `loto_naif` pouvait tirer deux fois le même numéro, faute de pouvoir mémoriser les numéros déjà sortis. Une liste résout ce problème. Écrire la fonction `loto` dont la spécification est :

```python
def loto() -> list[int]:
    """Retourne une liste de 6 entiers différents, tirés au hasard
    entre 1 et 49 (bornes incluses).
    """
```

Indication. Partir de la liste `[0] * 6`. Tirer un numéro avec `random.randint(1, 49)`. S'il n'est pas déjà dans la liste, le ranger à la première place libre. Le nombre de tirages nécessaires n'est pas connu à l'avance : quelle boucle faut-il utiliser ?

§6.1. Pour aller plus loin

Q17. **Transposée d'une matrice.** La transposée d'une matrice s'obtient en échangeant ses lignes et ses colonnes : la première ligne devient la première colonne, et ainsi de suite. Écrire, en une seule ligne de code, la fonction `transposee` dont la spécification est :

```python
def transposee(m: list[list[int]]) -> list[list[int]]:
    """Retourne la transposée de la matrice m.

    >>> transposee([[1, 2, 3], [4, 5, 6]])
    [[1, 4], [2, 5], [3, 6]]
    """
```

§7. Bilan

À l'issue de cette activité, je sais…

- expliquer la différence entre une liste, mutable, et un tuple, non mutable ;
- lire et modifier un élément d'une liste grâce à son indice ;
- parcourir une liste par ses éléments pour la lire, et par ses indices pour la modifier ;
- construire une liste de taille fixée avec `[valeur] * n` ;
- construire une liste en compréhension, avec ou sans filtre `if` ;
- expliquer pourquoi, après `m = n`, une modification de `m` est visible dans `n`, et faire une vraie copie d'une liste ;
- prévoir si une fonction modifie la liste qu'elle reçoit en argument ;
- représenter une matrice par une liste de listes et utiliser la notation `m[i][j]` ;
- écrire un algorithme de recherche d'une valeur, de calcul d'une moyenne, sur une liste.