Chapitre 7
Compréhension

Ce chapitre introduit la **compréhension** : une syntaxe compacte pour construire une nouvelle liste (ou un nouveau dictionnaire) en transformant les éléments d'un objet déjà existant. Au lieu d'écrire une boucle puis des appels à `append`, on écrit directement l'expression entre crochets. La compréhension est utilisée partout en programmation Python, car elle rend le code plus court et plus lisible.

I. Qu'est-ce que la compréhension ?

Faire des objets en **compréhension** signifie transformer les éléments de cet objet **terme à terme**. La seule contrainte est que l'on puisse **faire une boucle** sur cet objet : on appelle ces objets des **itérables**. Les listes, les chaînes de caractères, les objets `range` et les dictionnaires sont des itérables.

Définition 7.1.
Un **itérable** est un objet sur lequel on peut faire une boucle `for` : une liste, une chaîne de caractères, un objet `range`, un dictionnaire, etc. **Construire un objet par compréhension**, c'est transformer les éléments d'un itérable **terme à terme** pour créer un nouvel objet.

■ **Exemple 7.1.**
On part d'une liste et d'un dictionnaire quelconques. ■

```
liste = [1,2,3,4,5]
dictionnaire = {1:"a",2:"b",3:"c"}
```

■ **Exemple 7.2.**
On crée une nouvelle liste en compréhension : chaque élément i de la liste est transformé en $i + 1$. ■

```
nouvelle_liste = [i+1 for i in liste]
print(nouvelle_liste)
```

Exemple 7.3.
On crée un nouveau dictionnaire en compréhension : pour chaque couple (clé, valeur) du dictionnaire, on crée la nouvelle entrée `clé + 1 : valeur * 2`. La méthode `items()` fournit les couples (clé, valeur). ■

```
nouveau_dictionnaire = {clé + 1:valeur*2 for (clé,valeur) in dictionnaire.
    items()}
print(nouveau_dictionnaire)
```

R Syntaxe d'une liste en compréhension : `[expression for variable in itérable]`. L'expression est évaluée pour chaque élément de l'itérable ; le résultat est la liste des valeurs obtenues. Pour un dictionnaire : `{expression_clé : expression_valeur for variable in itérable}`.

Exemple 7.4.
Une compréhension peut simplement recopier les éléments d'un objet `range` dans une liste. ■

```
entiers = [x for x in range(5)]
print(entiers)
```

Exemple 7.5.
La transformation s'applique à tous les itérables : ici, chaque caractère d'une chaîne est doublé. ■

```
doubles_caracteres = [c * 2 for c in "ab"]
print(doubles_caracteres)
```

Exercice 7.1.
À vous de modifier (dans le bloc suivant) ces dernières lignes de code pour sentir comment fonctionne la compréhension : changer l'expression `i+1`, la liste de départ, l'opération effectuée sur les valeurs, etc. Tester plusieurs variantes et observer les résultats affichés.

```
nouvelle_liste = [i+1 for i in liste]
nouveau_dictionnaire = {clé + 1:valeur*2 for (clé,valeur) in
    dictionnaire.items()}
print(nouvelle_liste)
print(nouveau_dictionnaire)
```

■

II. Filtrer et transformer : if et if/else

Il est possible d'utiliser une clause `if` ou `if/else` dans les constructions par compréhension. Deux places sont possibles, et elles n'ont pas le même rôle.

Exemple 7.6.
On ne garde que les éléments qui vérifient une condition : le `if` se place **après** le `for`. Ici, on ne garde que les éléments i strictement supérieurs à 2, puis on les élève au cube. ■

```
if_liste = [i**3 for i in liste if i>2]
print(if_liste)
```

Exemple 7.7.
On peut aussi transformer différemment selon une condition, avec un `if/else` **dans l'expression** : si $i < 3$ on prend $i + 2$, sinon on prend i^i. ■

```
if_else_liste = [ i+2 if i < 3 else i**i for i in liste]
print(if_else_liste)
```

R Ne pas confondre les deux places du `if` : après le `for`, le `if` **filtre** (on garde ou on jette un élément) ; dans l'expression, le `if/else` **choisit la transformation** (on produit une valeur ou une autre).

Exemple 7.8.
On peut combiner les deux : ici, on double les nombres pairs et on laisse les impairs inchangés. ■

```
transformes = [x * 10 if x % 2 == 0 else x for x in range(5)]
print(transformes)
```

Exemple 7.9.
Le filtre fonctionne aussi sur les caractères d'une chaîne : on ne garde que les voyelles du mot. ■

```
voyelles = [c for c in "python" if c in "aeiouy"]
print(voyelles)
```

Exemple 7.10.
Pour les dictionnaires, la structure est différente : le `if` se place **après** le `for` (et non dans l'expression), et on ne peut pas mettre de `else`. Ici, on ne garde que les clés supérieures à 1, puis on transforme clés et valeurs. ■

```
if_dictionnaire = {clé*3:valeur*5 for clé,valeur in dictionnaire.items()
    if clé > 1}
print(if_dictionnaire)
```

R Pour les dictionnaires : le filtre `if` se place après le `for` ; un `else` n'est pas autorisé comme filtre. On peut en revanche utiliser un `if/else` dans l'expression de la clé ou de la valeur.

■ **Exercice 7.2.**
Recopier et modifier les codes précédents pour vous familiariser avec la compréhension et les `if/else`. Par exemple : ne garder que les valeurs paires d'une liste, transformer les valeurs négatives en 0, ne garder que les mots de plus de trois lettres, etc. ■

III. De la boucle à la compréhension

Toute compréhension peut s'écrire avec une boucle classique : on crée une liste vide, on parcourt l'itérable, on ajoute le résultat à chaque tour. La compréhension condense ces trois étapes en une seule ligne.

■ **Exemple 7.11.**
Version avec une boucle classique : on construit la liste des carrés des entiers de 0 à 5. ■

```
carres = []
for x in range(6):
    carres.append(x ** 2)
print(carres)
```

■ **Exemple 7.12.**
La même construction par compréhension : plus courte et plus lisible. ■

```
carres = [x ** 2 for x in range(6)]
print(carres)
```

R Une compréhension parcourt un itérable : le nombre de tours est connu d'avance. On ne peut donc pas utiliser `while` dans une compréhension ; pour une boucle conditionnelle, on garde une boucle classique.

■ **Exemple 7.13.**
La compréhension fonctionne sur tous les itérables, pas seulement les listes. Ici, on transforme les caractères d'une chaîne en majuscules avec la méthode `upper`. ■

```
majuscules = [c.upper() for c in "nsi"]
print(majuscules)
```

■ **Exemple 7.14.**
On transforme des entiers en chaînes de caractères avec `str`. ■

```
chaines = [str(x) for x in [1,2,3]]
print(chaines)
```

■ **Exemple 7.15.**
On peut combiner transformation et filtre : les multiples de 3 parmi les entiers de 0 à 9, doublés. ■

```
doubles = [x * 2 for x in range(10) if x % 3 == 0]
print(doubles)
```

■ **Exemple 7.16.**
On calcule la longueur de chaque mot d'une liste : la fonction `len` est appliquée terme à terme. ■

```
longueurs = [len(mot) for mot in ["chat", "python"]]
print(longueurs)
```

■ **Exemple 7.17.**
On construit un dictionnaire par compréhension : les entiers de 0 à 2 associés à leur double. ■

```
d_doubles = {x: x * 2 for x in range(3)}
print(d_doubles)
```

■ **Exercice 7.3.**
Écrire en compréhension la liste des entiers de 0 à 9 élevés au cube, puis l'afficher.
■

IV. Exercices

Les exercices suivants reprennent et complètent les notions du chapitre. Pensez à bien lire les énoncés, à tester vos programmes, et à interpréter les messages d'erreur éventuels.

Exercices d'automatismes

Les exercices d'automatismes exercent les réflexes fondamentaux : calcul mental d'expressions Python, lecture de code court, prédiction d'affichage, complétion. Chaque question doit pouvoir être traitée en moins de trente secondes.

■ Exercice 7.4.

Types et valeurs : répondre aux questions suivantes sans exécuter les programmes.

1. Quel est le type de la valeur renvoyée par l'expression suivante ?

```
12 / 4
```

2. Quel est le type de la valeur renvoyée par l'expression suivante ?

```
12 // 4
```

3. Quel est le type de la variable `a` après l'affectation suivante ?

```
a = "12"
```

4. Quel est le type de la valeur renvoyée par l'expression suivante ?

```
(3 > 1)
```

5. Que vaut l'expression suivante ?

```
6 + 4 * 2
```

■

■ Exercice 7.5.

Compréhension simple : répondre aux questions suivantes.

1. Qu'affiche le programme suivant ?

```
print([x + 10 for x in [1,2,3]])
```

2. Qu'affiche le programme suivant ?

```
print([x * 3 for x in range(3)])
```

3. Qu'affiche le programme suivant ?

```
print([str(x) for x in [5,6]])
```

4. Qu'affiche le programme suivant ?

```
print([x ** 2 for x in [3,4]])
```

5. Qu'affiche le programme suivant ?

```
print([x for x in range(4)])
```

■

■ **Exercice 7.6.**

Listes et indices : répondre aux questions suivantes.

1. Qu'affiche le programme suivant ?

```
t = [4,5,6]
print(t[1])
```

2. Qu'affiche le programme suivant ?

```
t = [4,5,6]
print(t[-1])
```

3. Que renvoie l'expression suivante ?

```
len([4,5,6])
```

4. Que se passe-t-il à l'exécution du programme suivant ?

```
t = [4,5,6]
print(t[3])
```

5. Quelle instruction permet d'ajouter la valeur 9 à la fin d'une liste `t` ?

■

■ **Exercice 7.7.**

Compréhension avec condition : répondre aux questions suivantes.

1. Qu'affiche le programme suivant ?

```
print([x for x in range(6) if x % 2 == 1])
```

2. Qu'affiche le programme suivant ?

```
print([x for x in [3,8,12,5] if x > 6])
```

3. Qu'affiche le programme suivant ?

```
print([c for c in "abc" if c != "b"])
```

4. Qu'affiche le programme suivant ?

```
print([x * 2 for x in range(5) if x > 2])
```

■

■ Exercice 7.8.
Boucles et fonctions : répondre aux questions suivantes.

1. Combien d'itérations effectue la boucle `for i in range(4, 7)` ?
2. Que renvoie l'appel `f(4)` ?

```
def f(x):
    return x + 1

print(f(4))
```

3. Que vaut l'expression `13 % 4` ?
4. Que vaut l'expression `True and True` ?
5. Que vaut l'expression `3 ** 2` ?

■

■ Exercice 7.9.
Chaînes et transformation : répondre aux questions suivantes.

1. Qu'affiche le programme suivant ?

```
print([len(c) for c in ["a", "bb"]])
```

2. Qu'affiche le programme suivant ?

```
print([c.upper() for c in "ok"])
```

3. Que vaut l'expression `"ab" * 3` ?
4. Que renvoie `len("NSI")` ?

■

■ Exercice 7.10.
Dictionnaires : répondre aux questions suivantes.

1. Que renvoie `d["a"]` pour `d = {"a": 1, "b": 2}` ?
2. Que renvoie `len({"x": 1, "y": 2, "z": 3})` ?
3. Qu'affiche le programme suivant ?

```
print({k: 0 for k in ["a", "b"]})
```

4. Pour construire un dictionnaire par compréhension, quelle méthode fournit les couples (clé, valeur) à parcourir ?

■

Exercices d'application

Les exercices d'application construisent des listes et des dictionnaires par compréhension, du plus simple au plus difficile. Les trois premiers reprennent les exercices du chapitre.

■ **Exercice 7.11.**
Créer une liste `L = range(20)` puis faire une liste en compréhension où il y a les nombres x qui vérifient $x^2 > 15$.

```
L = range(20)
# Écrire ici la liste en compréhension
```

■

■ **Exercice 7.12.**
Créer une liste `L = range(20)` puis faire une liste en compréhension où les nombres sont transformés en `string`.

```
L = range(20)
# Écrire ici la liste en compréhension
```

■

■ **Exercice 7.13.**
Grâce aux méthodes `.keys()` et `.values()`, faire une liste par compréhension avec toutes les clés du dictionnaire et une avec toutes les valeurs du dictionnaire.

```
dictionnaire = {1:"a",2:"b",3:"c"}
# Liste des clés, puis liste des valeurs, par compréhension
```

■

■ **Exercice 7.14.**
Écrire en compréhension la liste des carrés des entiers de 0 à 9, puis l'afficher. ■

■ **Exercice 7.15.**
Écrire en compréhension la liste des multiples de 7 compris entre 1 et 50, puis l'afficher. ■

■ **Exercice 7.16.**
À partir de la liste `mots = ["pomme", "poire", "cerise"]`, écrire en compréhension la liste des premières lettres de chaque mot, puis l'afficher. ■

■ **Exercice 7.17.**
Écrire en compréhension la liste des longueurs des mots de la liste `["chat", "python", "programmation"]`, puis l'afficher. ■

■ **Exercice 7.18.**
Le programme suivant construit une liste avec une boucle classique. Le réécrire en une seule ligne avec une compréhension.

```
multiples = []
for n in range(1, 6):
    multiples.append(n * 7)
```

■

■ **Exercice 7.19.**
Écrire en compréhension une liste contenant la valeur absolue de chaque nombre de la liste `[-3, 5, -1, 0]`, puis l'afficher. On rappelle que la valeur absolue de x est x si $x \geq 0$, et $-x$ sinon. ■

■ **Exercice 7.20.**
Écrire un dictionnaire par compréhension associant à chaque entier de 0 à 4 son carré, puis l'afficher. ■

Exercices de réflexion

Les exercices de réflexion demandent plus d'analyse : déroulement de programmes, repérage d'erreurs, questions ouvertes sur la compréhension. Ils sont classés du plus abordable au plus exigeant.

■ **Exercice 7.21.**
Prédire ce qu'affiche le programme suivant, sans l'exécuter, puis expliquer l'ordre

des résultats.

```
print([x + y for x in [1,2] for y in [10,20]])
```

1. Qu'affiche ce programme ?
2. Pourquoi les deux valeurs de y sont-elles parcourues pour chaque valeur de x ?

■

■ **Exercice 7.22.**
Prédire ce qu'affiche le programme suivant, sans l'exécuter. Chaque élément du résultat est lui-même une liste.

```
print([[i, i * 2] for i in range(3)])
```

■

■ **Exercice 7.23.**
Le programme suivant provoque une erreur de syntaxe.

```
print([x if x > 2 for x in range(5)])
```

1. Expliquer pourquoi la syntaxe est incorrecte.
2. Écrire la version correcte de cette compréhension.

■

■ **Exercice 7.24.**
Le programme suivant provoque une erreur.

```
d = {1: "a", 2: "b", 3: "c"}
print({k: v for k, v in d.items() if k > 1 else k})
```

1. Expliquer pourquoi un `else` n'est pas accepté à cette place dans une compréhension de dictionnaire.
2. Écrire une version correcte qui ne garde que les clés supérieures à 1.

■

■ **Exercice 7.25.**
On exécute le programme suivant.

```
for i in range(3):
    pass

liste = [i for i in range(3)]
print(i)
```

1. Qu'affiche ce programme ?
2. Que se passe-t-il si l'on tente d'utiliser la variable `i` après la compréhension, alors qu'aucune boucle classique ne l'a définie ?

■

■ **Exercice 7.26.**
Pourquoi ne peut-on pas écrire une compréhension avec une boucle `while` ? Que doit-on utiliser à la place pour construire une liste avec une boucle conditionnelle ?
■