Chapitre 6. Les conteneurs, itérateurs et collections d'objets

Objectifs

À la fin de ce chapitre, tu sauras :

- Manipuler les conteneurs standards de Python (listes, tuples, ensembles, dictionnaires) ;
- Créer tes propres itérateurs **et** générateurs **;**
- Utiliser les coroutines pour un flux de données interactif ;
- Exploiter les collections utiles (deque, Counter, defaultdict, namedtuple, etc.) ;
- Appliquer les fonctions fonctionnelles (map, filter, reduce) sur des séquences.

1 Les conteneurs en Python

Un **conteneur** est un objet qui contient d'autres objets.

Exemples : list, tuple, set, dict.

```
# Liste : ordonnée, modifiable
fruits = ["pomme", "banane", "orange"]

# Tuple : ordonné, non modifiable
coord = (10, 20)

# Ensemble : non ordonné, sans doublons
unique = {"a", "b", "a", "c"}

# Dictionnaire : clé → valeur
personne = {"nom": "Ali", "age": 25}
```

Itération sur un conteneur :

```
for fruit in fruits:
    print(fruit)
```

2 Les itérateurs

Un **itérateur** est un objet qui permet de parcourir un conteneur **élément par élément**.

Principe :

Un objet est itérable s'il possède une méthode __iter__(). Un itérateur possède une méthode __next__().

Exemple :

```
liste = [10, 20, 30]
it = iter(liste)

print(next(it))  # 10
print(next(it))  # 20
print(next(it))  # 30
# next(it)  # StopIteration
```

Créer un itérateur personnalisé :

```
class Compteur:
    def __init__(self, debut, fin):
        self.debut = debut
        self.fin = fin

    def __iter__(self):
        self.current = self.debut
        return self

    def __next__(self):
        if self.current > self.fin:
            raise StopIteration
        val = self.current
        self.current += 1
        return val

for i in Compteur(1, 5):
    print(i)
```

3 Les générateurs

Un **générateur** est une fonction spéciale qui produit une **suite de valeurs**, sans les stocker toutes en mémoire. Il utilise le mot-clé yield.

Exemple :

```
def compte(a, b):
    while a <= b:
        yield a
        a += 1
```

```
for x in compte(1, 4):
    print(x)
```

✓ **Avantage** : économise la mémoire, permet un flux de données infini.

4 Les coroutines

Une **coroutine** ressemble à un générateur, mais elle reçoit aussi des valeurs avec send().

Exemple :

```
def echo():
    print("Coroutine démarrée")
    while True:
        message = yield
        print("Reçu:", message)

co = echo()
next(co)          # démarre la coroutine
co.send("Bonjour")
co.send("Python")
co.close()
```

5 Les collections avancées (collections module)

Python fournit plusieurs **types de conteneurs optimisés** dans le module collections.

deque — file double

```
from collections import deque
dq = deque([1, 2, 3])
dq.appendleft(0)
dq.append(4)
print(dq)  # deque([0,1,2,3,4])
```

Counter — compteur d'éléments

```
from collections import Counter
c = Counter("abracadabra")
print(c.most_common(2))  # [('a', 5), ('b', 2)]
```

defaultdict — dictionnaire avec valeur par défaut

```
from collections import defaultdict
d = defaultdict(int)
d["a"] += 1
```

```
print(d)  # {'a': 1}
```

namedtuple — tuple nommé

```
from collections import namedtuple
Point = namedtuple("Point", ["x", "y"])
p = Point(3, 4)
print(p.x, p.y)
```

6 Les fonctions fonctionnelles : map, filter, reduce

map() — applique une fonction à chaque élément

```
nums = [1, 2, 3]
res = list(map(lambda x: x**2, nums))
print(res)  # [1, 4, 9]
```

filter() — filtre les éléments selon une condition

```
pairs = list(filter(lambda x: x % 2 == 0, range(10)))
print(pairs)  # [0, 2, 4, 6, 8]
```

reduce() — réduction à une seule valeur

```
from functools import reduce
somme = reduce(lambda x, y: x + y, [1, 2, 3, 4])
print(somme)  # 10
```

Exercices et solutions

✓ Exercice 1 :

Créer un générateur fibonacci(n) qui génère les n premiers termes de la suite de Fibonacci.

Solution :

```
def fibonacci(n):
    a, b = 0, 1
    for _ in range(n):
        yield a
        a, b = b, a + b

print(list(fibonacci(7)))  # [0,1,1,2,3,5,8]
```

✓ Exercice 2 :

Créer une classe Pile (stack) et File (queue) avec append() et pop().

Solution :

```
class Pile:
```

```
    def __init__(self):
        self.data = []

    def push(self, x):
        self.data.append(x)

    def pop(self):
        return self.data.pop()

class File:
    def __init__(self):
        from collections import deque
        self.data = deque()

    def enfiler(self, x):
        self.data.append(x)

    def defiler(self):
        return self.data.popleft()

p = Pile()
p.push(10)
p.push(20)
print(p.pop())  # 20

f = File()
f.enfiler("A")
f.enfiler("B")
print(f.defiler())  # "A"
```

✅ **Exercice 3 :**

Créer une **coroutine** filtre_positifs() qui reçoit des nombres et affiche seulement les positifs.

Solution :

```
def filtre_positifs():
    print("Coroutine active")
    while True:
```

```
    val = yield
    if val > 0:
        print("Positif:", val)

co = filtre_positifs()
next(co)
co.send(5)
co.send(-2)
co.send(10)
co.close()
```

Récapitulatif

Concept	**Description**	**Exemple clé**
Itérateur	Objet parcourable avec __next__()	iter(liste)
Générateur	Fonction avec yield	def g(): yield x
Coroutine	Générateur recevant avec send()	co.send(val)
Collections	Structures spécialisées	deque, Counter
map/filter/reduce	Programmation fonctionnelle	map(f, L)