ICC/Programmation — Exercices Semaine 11

4 mai 2026

Objectif. Dans ces exercices, vous pratiquerez les trois piliers de la programmation fonctionnelle en Python introduits lors du cours d'aujourd'hui : les compréhensions de listes, les fonctions d'ordre supérieur et les expressions lambda. Vous commencerez par des exemples simples et autonomes, puis combinerez les trois concepts pour écrire du code concis et expressif.

Ces exercices sont autonomes. Aucune bibliothèque supplémentaire n'est requise au-delà de la bibliothèque standard de Python.

Mise en place. Créez un nouveau fichier **`functional.py`** et travaillez chaque exercice en ajoutant du code à la fin du fichier. Exécutez-le à tout moment avec :

```
python3 functional.py
```

Exercice 1. Compréhensions de listes

Une compréhension de liste construit une nouvelle liste en appliquant une expression à chaque élément d'une séquence existante, en filtrant optionnellement les éléments avec **`if`**.

(a) **Transformation de base.** La boucle ci-dessous convertit une liste de températures de Celsius en Fahrenheit. Réécrivez-la sous forme d'une seule compréhension de liste :

```python
celsius = [0, 20, 37, 100]
fahrenheit = []
for c in celsius:
    fahrenheit.append(c * 9 / 5 + 32)
print(fahrenheit)  # [32.0, 68.0, 98.6, 212.0]

# Votre version en une ligne :
fahrenheit = [... for c in celsius]
print(fahrenheit)
```

(b) **Filtrage.** À l'aide d'une compréhension avec une clause **`if`**, extrayez uniquement les mots de la liste qui comportent plus de quatre caractères :

```python
words = ["cat", "elephant", "dog", "butterfly", "ox", "penguin"]

# Attendu : ['elephant', 'butterfly', 'penguin']
long_words = [... for word in words if ...]
print(long_words)
```

(c) **Compréhension imbriquée.** La table de multiplication ci-dessous est construite avec des boucles imbriquées. Remplacez-la par une compréhension de liste imbriquée (une compréhension qui produit une liste de listes) :

```python
table = []
for x in range(1, 6):
    row = []
    for y in range(1, 6):
        row.append(x * y)
    table.append(row)

for row in table:
    print(row)
# [1, 2, 3, 4, 5]
# [2, 4, 6, 8, 10]
# ...

# Votre version :
table = [[... for y in range(1, 6)] for x in range(1, 6)]
```

(d) **Compréhension de dictionnaire.** Les compréhensions fonctionnent aussi avec les dictionnaires en utilisant la syntaxe **`{clé: valeur for ...}`**. Construisez un dictionnaire qui associe chaque mot à sa longueur :

```python
words = ["apple", "banana", "kiwi", "strawberry"]

# Attendu : {'apple': 5, 'banana': 6, 'kiwi': 4, 'strawberry': 10}
lengths = {word: ... for word in words}
print(lengths)
```

(e) **Question.** Considérez la compréhension suivante :

```
result = [x * y for x in range(1, 4) for y in range(1, 4) if x != y]
print(result)
```

Avant d'exécuter le code, prédisez ce que **`result`** contiendra. Exécutez-le ensuite et vérifiez. Combien d'éléments la liste contient-elle, et pourquoi ?

Exercice 2. Fonctions d'ordre supérieur

Une fonction d'ordre supérieur est une fonction qui prend une autre fonction en argument ou qui retourne une fonction comme résultat. Les fonctions intégrées **`map`**, **`filter`** et **`sorted`** en sont des exemples courants.

(a) **Les fonctions comme valeurs.** En Python, une fonction est un objet comme un autre. Exécutez le code suivant et assurez-vous de comprendre chaque ligne :

```
def double(x: int) -> int:
    return x * 2

def triple(x: int) -> int:
    return x * 3

# Stocker des fonctions dans une liste et les appeler dans une boucle
operations = [double, triple]
for f in operations:
    print(f(10))   # affiche 20, puis 30
```

Quel est le type de la variable **`f`** à l'intérieur de la boucle ?

(b) **Écrire une fonction d'ordre supérieur.** Complétez la fonction **`apply_to_all`** ci-dessous. Elle reçoit une fonction **`f`** et une liste, et retourne une nouvelle liste contenant le résultat de l'application de **`f`** à chaque élément :

```
from typing import Callable

def apply_to_all(f: Callable[[int], int], values: list[int]) -> list[int]:
    result = []
    for v in values:
        result.append(...)   # à compléter
    return result

print(apply_to_all(double, [1, 2, 3, 4]))   # [2, 4, 6, 8]
print(apply_to_all(triple, [1, 2, 3, 4]))   # [3, 6, 9, 12]
```

Remarquez que **`apply_to_all`** fait exactement ce que fait la fonction intégrée **`map`** — **`list(map(double, [1, 2, 3, 4]))`** donne le même résultat.

(c) **Retourner une fonction.** La fonction **`make_multiplier`** ci-dessous prend un nombre **`n`** et retourne une nouvelle fonction qui multiplie son argument par **`n`**. Complétez le corps :

```
def make_multiplier(n: int) -> Callable[[int], int]:
    def multiply(x: int) -> int:
        return ...   # à compléter
    return multiply

times5 = make_multiplier(5)
times7 = make_multiplier(7)

print(times5(3))            # 15
print(times7(3))            # 21
print(times5(times7(2)))    # qu'affiche ceci ?
```

(d) **Tri avec une clé.** La fonction intégrée **`sorted`** accepte un argument **`key`** — une fonction qui extrait une valeur de comparaison de chaque élément. Triez la liste de mots d'abord par longueur (le plus court en premier), puis par longueur en ordre décroissant :

```
words = ["banana", "fig", "apple", "kiwi", "strawberry", "plum"]

by_length = sorted(words, key=...)
print(by_length)   # ['fig', 'kiwi', 'plum', 'apple', 'banana', 'strawberry']

by_length_desc = sorted(words, key=..., reverse=True)
print(by_length_desc)
```

(e) **Question.** Considérez cette fonction :

```
def apply_twice(f: Callable[[int], int], x: int) -> int:
    return f(f(x))

print(apply_twice(double, 3))
```

Que retourne **`apply_twice(double, 3)`** ? Que signifierait **`apply_twice(apply_twice(double, ...), 1)`** — est-ce du Python valide ? Essayez et expliquez le résultat.

Exercice 3. Expressions lambda

Un lambda est une fonction anonyme à expression unique. Il est utile quand vous avez besoin d'une courte fonction en argument et ne souhaitez pas lui donner un nom avec **`def`**.

(a) **Syntaxe de base.** Exécutez le code suivant et vérifiez que la sortie correspond à vos attentes :

```
square    = lambda x: x ** 2
add       = lambda x, y: x + y
constant  = lambda: 42

print(square(5))       # 25
print(add(3, 7))       # 10
print(constant())      # 42
```

Chaque expression lambda est équivalente à une définition **`def`**. Réécrivez **`square`** avec un **`def`** normal et vérifiez qu'elle produit le même résultat.

(b) **Lambda comme argument.** Réécrivez les appels **`sorted`** de l'Exercice 2(d) en utilisant des lambdas plutôt qu'une fonction définie séparément :

```
words = ["banana", "fig", "apple", "kiwi", "strawberry", "plum"]

by_length      = sorted(words, key=lambda w: ...)
by_length_desc = sorted(words, key=lambda w: ..., reverse=True)
print(by_length)
print(by_length_desc)
```

(c) **Lambda avec `apply_twice`.** Utilisez la fonction **`apply_twice`** de l'Exercice 2 avec un lambda pour calculer ce qui suit sans définir aucune fonction auxiliaire nommée :

```
# Ajouter 3 deux fois (c'est-à-dire ajouter 6 au total)
result = apply_twice(lambda x: ..., 10)
print(result)   # 16

# Mettre un nombre au carré deux fois (c'est-à-dire élever à la puissance 4)
result2 = apply_twice(lambda x: ..., 3)
print(result2)  # 81
```

(d) **Tri d'une liste de tuples.** Chaque tuple ci-dessous représente un étudiant sous la forme **`(nom, note)`**. Triez la liste par note en ordre croissant à l'aide d'une clé lambda :

```
students = [
    ("Alice", 5.5),
    ("Bob",   4.0),
    ("Carol", 5.75),
    ("Dave",  4.5),
]

by_grade = sorted(students, key=lambda s: ...)
for name, grade in by_grade:
    print(f"{name}: {grade}")
# Bob: 4.0
# Dave: 4.5
# Alice: 5.5
# Carol: 5.75
```

(e) **Ce que les lambdas ne peuvent pas faire.** Essayez de prédire si chaque extrait ci-dessous est du Python valide. Exécutez ensuite chacun et expliquez l'erreur s'il échoue :

```
# Extrait 1
f = lambda x: x + 1
print(f(4))
```

```python
# Extrait 2
g = lambda x:
        x + 1
print(g(4))

# Extrait 3
h = lambda x: y = x + 1; return y
print(h(4))
```

Exercice 4. Tout assembler

Dans ce dernier exercice, vous combinerez les compréhensions, les fonctions d'ordre supérieur et les lambdas pour résoudre une petite tâche de traitement de données.

(a) **Mise en place.** Copiez le jeu de données ci-dessous dans **`functional.py`** :

```python
transactions = [
    {"id": 1, "amount": 120.0,  "category": "food"},
    {"id": 2, "amount":  45.5,  "category": "transport"},
    {"id": 3, "amount": 200.0,  "category": "food"},
    {"id": 4, "amount":  15.0,  "category": "transport"},
    {"id": 5, "amount": 340.0,  "category": "electronics"},
    {"id": 6, "amount":  80.0,  "category": "food"},
    {"id": 7, "amount": 500.0,  "category": "electronics"},
    {"id": 8, "amount":  22.5,  "category": "transport"},
]
```

(b) **Filtrer.** À l'aide d'une compréhension de liste, extrayez uniquement les transactions de la catégorie alimentation :

```python
food = [t for t in transactions if ...]
print(food)   # trois dicts avec category "food"
```

(c) **Transformer.** À l'aide d'une compréhension de liste, construisez une liste de tuples **`(id, montant)`** pour chaque transaction dont le montant dépasse 100 :

```python
expensive = [(t["id"], t["amount"]) for t in transactions if ...]
print(expensive)   # [(1, 120.0), (3, 200.0), (5, 340.0), (7, 500.0)]
```

(d) **Trier.** Triez toutes les transactions par montant en ordre décroissant en utilisant **`sorted`** et une clé lambda :

```python
by_amount = sorted(transactions, key=lambda t: ..., reverse=True)
for t in by_amount:
    print(t["id"], t["amount"])
```

(e) **Totaux par catégorie.** À l'aide d'une compréhension de dictionnaire, calculez le montant total dépensé par catégorie. *Indice :* utilisez **`sum()`** avec une expression génératrice à l'intérieur.

```python
categories = {"food", "transport", "electronics"}

totals = {
    cat: sum(t["amount"] for t in transactions if t["category"] == cat)
    for cat in categories
}
print(totals)
# {'food': 400.0, 'transport': 83.0, 'electronics': 840.0}
```

(f) **Pipeline.** Combinez tout en une seule expression : partez de **`transactions`**, ne conservez que celles dont le montant dépasse 50, triez-les par montant décroissant, et extrayez uniquement l'**`id`** de chacune. Écrivez cela en une seule ligne à l'aide d'une compréhension et de **`sorted`** :

```python
result = [t["id"] for t in sorted(
    [...],          # filtrer ici
    key=lambda t: ...,
    reverse=True
)]
print(result)   # [7, 5, 3, 6, 1]
```