Programmation fonctionnelle

Langages et programmation

Objectifs.
Prérequis : fonctions en Python, récursivité
À l'issue de ce chapitre, vous saurez :

- identifier les caractéristiques du paradigme fonctionnel (fonctions pures, absence d'effets de bord, récursivité) ;
- écrire des fonctions en style fonctionnel en Python (expressions conditionnelles, composition, récursivité) ;
- utiliser les fonctions d'ordre supérieur (`map`, `filter`, `sorted` avec `key`, `lambda`) ;
- comparer les paradigmes impératif, fonctionnel et objet sur un même problème.

Histoire

Le premier langage fonctionnel, Lisp (pour *List Processing*), est conçu en 1958 par John McCarthy. Il met en pratique le lambda-calcul d'Alonzo Church, un modèle de calcul fondé uniquement sur des fonctions. Lisp a donné naissance à toute une famille de langages, dont Common Lisp et Scheme.

Une seconde famille part du langage ML, conçu par Robin Milner au début des années 1970 pour écrire des logiciels de démonstration mathématique. Son système de types, très strict, détecte de nombreuses erreurs avant l'exécution. De ML descendent notamment Standard ML, OCaml (issu de CAML, développé à l'INRIA à partir de 1985) et Haskell (1990).

Aujourd'hui, Python, JavaScript, Java ou C# ont tous emprunté des idées à ces langages : on peut y écrire des programmes dans un **style fonctionnel**, même s'ils ne sont pas des langages fonctionnels purs.

Exemple introductif - Traitement de listes

Calculons la somme des éléments d'une liste.

En **Lisp**, une liste s'écrit entre parenthèses et deux fonctions de base permettent de la parcourir : `car` donne son premier élément, `cdr` la liste privée de ce premier élément. Les opérateurs se placent **avant** leurs arguments : `(+ 2 3)` vaut 5.

```
(defun somme (liste)
  (if (null liste)
      0
      (+ (car liste) (somme (cdr liste)))))

(somme '(4 7 1 8))
```

```
20
```

La définition se lit ainsi : la somme d'une liste vide vaut 0 ; sinon, c'est le premier élément ajouté à la somme du reste.

En **Python**, avec trois petites fonctions qui jouent le rôle de `null`, `car` et `cdr`, on écrit `somme` exactement de la même façon :

```python
def est_vide(liste):
    return liste == []

def premier(liste):
    return liste[0]

def reste(liste):
    return liste[1:]

def somme(liste):
    return 0 if est_vide(liste) else premier(liste) + somme(reste(liste))

print(somme([4, 7, 1, 8]))
```

```
20
```

Exemple introductif - Calcul arithmétique

Comptons les chiffres d'un entier positif. En **style impératif**, on divise le nombre par 10 tant qu'il a plus d'un chiffre, en comptant les divisions dans une variable :

```python
def nombre_de_chiffres(n):
    chiffres = 1
    while n >= 10:
        n = n // 10
        chiffres = chiffres + 1
    return chiffres

print(nombre_de_chiffres(2026))
```

```
4
```

En **style fonctionnel**, on décrit directement le résultat : un nombre inférieur à 10 a un chiffre ; sinon, il a un chiffre de plus que le nombre obtenu en retirant son chiffre des unités.

```python
def nombre_de_chiffres(n):
    return 1 if n < 10 else 1 + nombre_de_chiffres(n // 10)

print(nombre_de_chiffres(2026))
```

```
4
```

Le paradigme fonctionnel

Les deux versions fonctionnelles ci-dessus reposent sur quatre outils :

— des **fonctions**, qui reçoivent des valeurs et renvoient un résultat ;
— la **composition** : le résultat d'une fonction devient l'argument d'une autre, comme dans `somme(reste(liste))`, qui calcule d'abord `reste(liste)` puis applique `somme` à ce résultat ;
— les **expressions conditionnelles**, qui s'écrivent en Python `valeur1 if condition else valeur2` et **valent** quelque chose, contrairement à une instruction `if` ;
— la **récursivité**, qui ramène le calcul à un cas plus petit.

On y chercherait en vain une variable dont la valeur change, une affectation, un affichage, une boucle, une suite d'instructions exécutées l'une après l'autre ou une instruction `if` : tous ces éléments appartiennent au **paradigme impératif**, celui de la version avec `while`.

Correspondances

D'un paradigme à l'autre, les outils se répondent :

— là où l'impératif **enchaîne des instructions**, le fonctionnel **compose des fonctions** ;
— là où l'impératif **exécute des instructions**, le fonctionnel **calcule des expressions**, y compris conditionnelles ;
— là où l'impératif **répète avec une boucle**, le fonctionnel **rappelle la fonction** (récursivité).

Activité. Écrire en Python, selon le paradigme fonctionnel, une fonction `inverse(s)` qui renvoie une chaîne de caractères inversée, en n'utilisant que `s[0]`, `s[1:]`, la chaîne vide `''` et l'opération `+`. *Voir la fiche d'exercices pour d'autres problèmes de réécriture impératif vers fonctionnel.*

Les fonctions d'ordre supérieur

En programmation fonctionnelle, une fonction est une **valeur** comme un nombre ou une liste : on peut la ranger dans une variable, la donner en argument à une autre fonction, ou la renvoyer comme résultat. Une fonction qui reçoit ou renvoie une fonction est dite **d'ordre supérieur**.

En Python, le mot-clé `lambda` crée une fonction sans lui donner de nom ; on peut ensuite la ranger dans une variable :

```
1 double = lambda x: 2 * x
2
3 print(double(16))
```

```
32
```

Deux usages reviennent sans cesse : les fonctions de la bibliothèque standard qui attendent une fonction en argument, comme `sorted`, et les traitements qui appliquent un même calcul à tous les éléments d'une liste.

Utiliser des fonctions en paramètres

La fonction `sorted` renvoie une nouvelle liste triée. Son paramètre facultatif `key` reçoit une fonction : `sorted` calcule l'image de chaque élément par cette fonction, puis range les éléments selon ces images.

Sans `key`, des chaînes de caractères sont rangées par ordre alphabétique :

```
villes = ['Lyon', 'Paris', 'Nice', 'Bordeaux', 'Lille', 'Rennes', 'Marseille', 'Brest']
print(sorted(villes))
```

```
['Bordeaux', 'Brest', 'Lille', 'Lyon', 'Marseille', 'Nice', 'Paris', 'Rennes']
```

Avec `key=len`, elles sont rangées selon leur nombre de lettres ; à longueur égale, l'ordre de départ est conservé :

```
print(sorted(villes, key=len))
```

```
['Lyon', 'Nice', 'Paris', 'Lille', 'Brest', 'Rennes', 'Bordeaux', 'Marseille']
```

La clé peut être une fonction écrite avec `lambda`. Ici, chaque élève est un couple (prénom, note), et l'on trie selon la note, donc selon l'élément d'indice 1 du couple :

```
eleves = [("Inès", 14), ("Hugo", 9), ("Léa", 17), ("Tom", 12)]
print(sorted(eleves, key=lambda e: e[1]))
```

```
[('Hugo', 9), ('Tom', 12), ('Inès', 14), ('Léa', 17)]
```

Activité. Trier la liste d'enregistrements `[[5012, 'Jean', 'Dupond'], [4086, 'Jacques', 'Dupond'], ...]` par ordre alphabétique de nom, puis de prénom si les noms sont identiques. *Voir la fiche d'exercices pour d'autres problèmes de tri avec `sorted` et `lambda`.*

Utiliser une fonction d'ordre supérieur

Doubler chaque nombre d'une liste, élever chaque nombre au cube, mesurer la longueur de chaque mot : ces trois calculs suivent le même schéma, qui consiste à appliquer **une même fonction à chaque élément** et à recueillir les résultats dans une nouvelle liste. Écrit récursivement pour la fonction `double` :

```
double = lambda x: 2 * x

def doubler_tout(liste):
    return [] if liste == [] else [double(liste[0])] + doubler_tout(liste[1:])

print(doubler_tout([3, 5, 8, 11]))
```

```
[6, 10, 16, 22]
```

La fonction `map` réalise ce schéma pour n'importe quelle fonction : `map(f, liste)` applique `f` à chaque élément. Elle renvoie un **itérateur**, qui ne calcule les valeurs qu'à la demande ; `list` le transforme en liste.

```
print(list(map(double, [3, 5, 8, 11])))
```

```
[6, 10, 16, 22]
```

```
print(list(map(lambda x: x ** 3, [3, 8, 2, 1])))
```

```
[27, 512, 8, 1]
```

```
print(list(map(len, ['Lyon', 'Paris', 'Bordeaux'])))
```

```
[4, 5, 8]
```

En Python, une liste en compréhension donne le même résultat, souvent de façon plus lisible :

```
print([double(x) for x in [3, 5, 8, 11]])
```

```
[6, 10, 16, 22]
```

La fonction `filter`

`filter(predicat, liste)` garde les éléments pour lesquels la fonction `predicat` renvoie `True`. Un **prédicat** est une fonction qui renvoie un booléen. Comme `map`, `filter` renvoie un itérateur.

```
nombres = [3, 7, 2, 9, 4, 1, 8, 5]
grands = list(filter(lambda x: x > 4, nombres))
print(grands)
```

```
[7, 9, 8, 5]
```

On peut d'abord sélectionner, puis transformer, en donnant le résultat de `filter` à `map` :

```
nombres = [-3, 5, -1, 8, -2, 7]
absolus_negatifs = list(map(abs, filter(lambda x: x < 0, nombres)))
print(absolus_negatifs)
```

```
[3, 1, 2]
```

La même liste s'écrit en compréhension :

```python
absolus_negatifs = [abs(x) for x in nombres if x < 0]
```

La fonction `reduce`

Complément hors programme. La fonction `reduce` n'est pas exigible au baccalauréat ; elle complète `map` et `filter`.

La fonction `reduce` du module `functools` **combine** tous les éléments d'une liste en une seule valeur : elle part d'une valeur initiale, puis applique une fonction à deux arguments au résultat obtenu jusque-là et à l'élément suivant.

```python
from functools import reduce
somme = reduce(lambda acc, x: acc + x, [3, 7, 2, 9], 0)
print(somme)
```

```
21
```

Ici, la valeur initiale est `0`, et les calculs successifs sont `0+3=3`, puis `3+7=10`, puis `10+2=12`, puis `12+9=21`.

Là où `map` et `filter` produisent une suite de valeurs, `reduce` n'en produit qu'**une**.

Fonctions pures et effets de bord

Le style fonctionnel privilégie les **fonctions pures**. Une fonction est pure quand :

— son résultat ne dépend **que de ses arguments** : appelée deux fois avec les mêmes arguments, elle renvoie la même chose ;
— elle **ne modifie rien** en dehors d'elle-même : ni variable globale, ni liste reçue en argument, ni fichier, ni écran. Une telle modification s'appelle un **effet de bord**.

```python
# Fonction pure : toujours le même résultat, aucune modification extérieure
def double(x):
    return 2 * x
```

```python
# Fonction impure : modifie une variable globale (effet de bord)
compteur = 0

def incrementer():
    global compteur
    compteur += 1
    return compteur
```

Après deux appels à `incrementer()`, les résultats sont différents (1 puis 2), alors que `double(5)` renvoie toujours 10.

Un piège fréquent en Python concerne la mutation des listes :

```python
# Fonction impure : modifie la liste passée en paramètre
def ajouter(x, L):
```

```
    L.append(x)
    return L

# Fonction pure : crée une nouvelle liste
def ajouter_pur(x, L):
    return L + [x]
```

Avec `ajouter(4, ma_liste)`, la liste originale est modifiée (effet de bord). Avec `ajouter_pur(4, ma_liste)`, l'originale reste intacte. En programmation fonctionnelle, on préfère toujours créer de nouvelles structures plutôt que modifier les existantes.

Synthèse : comparer les trois paradigmes

Le programme de NSI distingue trois grands paradigmes de programmation : **impératif**, **fonctionnel** et **objet**. Pour bien les différencier, résolvons un même problème avec chacun d'eux.

Problème. On dispose d'une liste de notes (entiers entre 0 et 20). On souhaite calculer la moyenne des notes supérieures ou égales à 10.

Style impératif

On utilise des variables, des boucles et des instructions d'affectation. Le programme décrit **étape par étape** comment obtenir le résultat.

```
def moyenne_au_dessus_imperatif(notes):
    somme = 0
    compteur = 0
    for note in notes:
        if note >= 10:
            somme = somme + note
            compteur = compteur + 1
    if compteur == 0:
        return 0
    return somme / compteur

print(moyenne_au_dessus_imperatif([8, 15, 12, 7, 18, 9, 14]))
```

```
14.75
```

Le paradigme impératif se caractérise par l'utilisation de **variables modifiables**, de **boucles** et d'**instructions séquentielles**. C'est le style le plus intuitif pour les débutants.

Style fonctionnel

On compose des fonctions sans modifier de variables. Le programme décrit **ce que l'on veut obtenir** plutôt que comment l'obtenir.

```
def moyenne_au_dessus_fonctionnel(notes):
    au_dessus = list(filter(lambda n: n >= 10, notes))
    return sum(au_dessus) / len(au_dessus) if au_dessus else 0

```

```
print(moyenne_au_dessus_fonctionnel([8, 15, 12, 7, 18, 9, 14]))
```

```
14.75
```

Le paradigme fonctionnel repose sur la **composition de fonctions**, l'absence d'**effets de bord** (pas de modification de variables) et l'utilisation de **fonctions d'ordre supérieur** comme `filter`. On aurait pu aussi utiliser une liste en compréhension : `[n for n in notes if n >= 10]`.

Style objet

On modélise le problème à l'aide d'un objet qui encapsule les données et les traitements associés.

```
class ReleveDeNotes:
    def __init__(self, notes):
        self.notes = notes

    def au_dessus(self, seuil):
        return [n for n in self.notes if n >= seuil]

    def moyenne_au_dessus(self, seuil):
        selection = self.au_dessus(seuil)
        if len(selection) == 0:
            return 0
        return sum(selection) / len(selection)

releve = ReleveDeNotes([8, 15, 12, 7, 18, 9, 14])
print(releve.moyenne_au_dessus(10))
```

```
14.75
```

Le paradigme objet organise le code autour d'**objets** qui regroupent des **attributs** (les données) et des **méthodes** (les traitements). L'avantage est la **modularité** : on peut réutiliser la classe `ReleveDeNotes` dans d'autres contextes, l'enrichir de nouvelles méthodes, etc.

Récapitulatif

Critère	Impératif	Fonctionnel	Objet
Brique de base	instruction, variable	fonction, expression	objet, méthode
Contrôle du flux	boucles, conditions	récursivité, composition	messages entre objets
Modification de l'état	oui (variables mutables)	non (pas d'effet de bord)	oui (attributs modifiables)
Avantage principal	simplicité, intuitivité	concision, preuves plus faciles	modularité, réutilisabilité
Exemple de langage pur	C	Haskell	Smalltalk

Python est un langage **multiparadigme** : il permet de programmer dans les trois styles, et même de les combiner dans un même programme. Le choix du paradigme dépend du problème à résoudre et du contexte.

À retenir.

— Le style **fonctionnel** construit un programme en composant des fonctions, sans modifier de variables, et répète par récursivité ; le style impératif enchaîne des instructions, modifie des variables et répète par des boucles.
— Une **fonction d'ordre supérieur** prend une fonction en paramètre ou renvoie une fonction (`map`, `filter`, `sorted` avec `key`).
— La notation `lambda` permet de définir des fonctions anonymes courtes, utiles comme arguments de fonctions d'ordre supérieur.
— Python est **multiparadigme** : on peut combiner styles impératif, fonctionnel et objet dans un même programme. Le choix du paradigme dépend du problème.
— L'écriture par **compréhension** (`[f(x) for x in L]`) est l'alternative pythonique à `map` et `filter`.
— Une **fonction pure** ne dépend que de ses arguments et ne modifie rien autour d'elle ; plutôt que de modifier une liste existante, elle en construit une nouvelle.