Chapitre 7
Paradigmes de programmation

Jusqu'à présent, nous avons toujours programmé de la même façon : des instructions exécutées les unes après les autres, des variables modifiées, des fonctions qui font des calculs. Or il existe en réalité plusieurs **manières d'aborder la programmation**, plusieurs points de vue sur la façon de concevoir et d'écrire un programme : ce sont les **paradigmes de programmation**. Dans ce chapitre, nous allons découvrir les principaux paradigmes, les reconnaître sur des exemples, et apprendre à choisir celui qui convient le mieux à un problème donné.

I. Concept : généralités

Définition 7.1.
Un **paradigme de programmation** est une façon d'approcher la programmation informatique et de traiter les solutions aux problèmes et leur formulation dans un langage de programmation approprié.

Un paradigme de programmation fournit la **vue qu'a le développeur de l'exécution de son programme** : il détermine la manière dont le programmeur pense son programme, le découpe et l'organise. Le même problème peut être traité dans des styles très différents selon le paradigme choisi.

■ **Exemple 7.1.**
En **programmation orientée objet**, les développeurs peuvent considérer le programme comme une collection d'objets en interaction : un jeu vidéo est alors vu comme des personnages, des objets, des décors qui s'envoient des messages. ■

■ **Exemple 7.2.**
En **programmation fonctionnelle**, un programme peut être vu comme une suite d'évaluations de fonctions sans états : le même jeu vidéo est alors vu comme une suite de transformations de données (l'état du jeu à l'instant t produit l'état du jeu à l'instant $t+1$). ■

■ **Exemple 7.3.**
Lors de la programmation d'ordinateurs ou de systèmes multi-processeurs, la **programmation orientée processus** permet aux développeurs de voir les applications

comme des ensembles de processus agissant sur des structures de données localement partagées : chaque processus exécute sa partie du travail en parallèle des autres. ■

R Ces paradigmes ne sont pas exclusifs : un même langage de programmation peut permettre d'en utiliser plusieurs, parfois même dans un même programme. C'est le cas de Python, comme nous le verrons dans ce chapitre.

II. Exemples de paradigmes de programmation

1. Programmation impérative

La **programmation impérative** décrit les opérations en séquences d'instructions exécutées par l'ordinateur pour modifier l'état du programme. Ce type de programmation est le plus répandu parmi l'ensemble des langages de programmation existants.

■ **Exemple 7.4.**
On déclare différentes variables auxquelles on attribue des valeurs : chaque instruction modifie l'état du programme. ■

```
x = 1
y = x + 2
z = x * y
```

Les opérations sont effectuées sur les variables elles-mêmes. On peut aussi déclarer une fonction et l'exécuter sur des variables :

■ **Exemple 7.5.**
Une fonction impérative : elle prend des valeurs en entrée, les combine, et renvoie un résultat. ■

```
def un_exemple_de_fonction(nom, prenom, salle):
    return f"{prenom} {nom} est en salle {salle}"

un_exemple_de_fonction("Atbi", "Isma", "K223")
```

R L'inconvénient immédiat de la programmation impérative est la **multiplication des variables**. On doit souvent se questionner sur l'état d'une variable durant le déroulé des instructions : les fonctions ont-elles connaissance de cet état ? Est-ce une variable globale ? Pourrait-on regrouper certaines de ces variables en un . . .objet ?

Un peu d'histoire. Fortran (IBM, 1954) est un des premiers langages de programmation impérative. À ce jour il est toujours utilisé par les scientifiques pour la qualité de ses librairies. Plus tard, C est devenu la référence de la programmation impérative : la majorité des applications de bureau que vous utilisez sont écrites en

C, tout comme le noyau des systèmes d'exploitation les plus courants. Python, Java, JavaScript... implémentent tous la programmation impérative.

■ **Exemple 7.6.**
Un programme impératif classique : une boucle qui accumule un résultat dans une variable. La variable `total` change de valeur à chaque tour de boucle : c'est l'**état** du programme qui évolue. ■

```
def somme_entiers(n):
    total = 0
    for i in range(1, n + 1):
        total = total + i
    return total

print(somme_entiers(10))
```

Ce programme affiche `55` : la somme des entiers de 1 à 10.

■ **Exemple 7.7.**
Le problème de l'état se pose dès qu'une fonction utilise une variable définie en dehors d'elle. Ici, la fonction `ajouter` modifie la variable globale `total` : le résultat dépend de l'état dans lequel se trouve le programme avant l'appel. ■

```
total = 0

def ajouter(x):
    global total
    total = total + x
```

2. Programmation orientée objet

La programmation orientée objet est le **paradigme de programmation dominant** en 2020. Elle a vu le jour durant les années 70 et sa première implémentation reconnue est SmallTalk. Python, Java, C++ sont des langages qui implémentent la programmation objet.

La programmation objet consiste en la **définition** et l'**interaction** de briques logicielles appelées **objets** ; un objet représente un concept, une idée ou toute entité du monde physique, comme une voiture, une personne ou encore une page d'un livre. Il possède une structure interne et un comportement, et il sait interagir avec ses pairs. Il s'agit donc de représenter ces objets et leurs relations ; l'interaction entre les objets via leurs relations permet de concevoir et réaliser les fonctionnalités attendues, de mieux résoudre le ou les problèmes. Dès lors, l'étape de **modélisation** revêt une importance majeure et nécessaire pour la POO : c'est elle qui permet de transcrire les éléments du réel sous forme virtuelle.

■ **Exemple 7.8.**
La classe `Eleve` : chaque objet de cette classe possède des attributs (nom, sexe,

spécialités) et une méthode `dire_present`. C'est un exemple de programmation orientée objet. ■

```
class Eleve:
    def __init__(self):
        self.nom = "Mettre un nom"
        self.sexe = "Adolescent(e)"
        self.specialite = []
    def dire_present(self):
        print(f"{self.nom} Present !")

un_eleve = Eleve()
un_eleve.nom = "Aurore"
un_eleve.dire_present()
```

Ici, `Eleve` désigne un objet personne qui a une méthode : `dire_present`. Nom, prénom et ville pourraient eux-mêmes être des objets. Dans certains langages, comme Python, tout est objet (les entiers, les fonctions, les classes elles-mêmes).

> **R** Cet exemple n'est pas très utile : une classe qui n'a que deux méthodes dont `__init__`... n'est qu'une fonction ! En pratique, les classes ont parfois des dizaines de méthodes... et certaines classes héritent les unes des autres.

Concevoir un programme dans ce paradigme consiste donc d'abord à définir les objets dont on aura besoin au travers de leur interaction : que doit faire la personne ? Comment interagit-elle avec l'objet ville ? Ces interactions permettent de définir les **méthodes**.

■ Exemple 7.9.
Pour modéliser une personne et une ville, on commence par se demander quelles sont leurs interactions : la personne peut *se déplacer vers* la ville, la ville peut *compter* ses habitants. Chaque interaction devient une méthode : `se_deplacer(ville)` pour la personne, `compter_habitants()` pour la ville. ■

■ Exemple 7.10.
En Python, tout est objet : l'entier `5` est un objet de la classe `int`, la chaîne `"NSI"` est un objet de la classe `str`, et une fonction est elle-même un objet. C'est pour cela qu'on peut écrire `"NSI".upper()` : la méthode `upper` s'applique à l'objet `"NSI"`. ■

3. Programmation fonctionnelle

La **programmation fonctionnelle** est un paradigme de programmation de type **déclaratif** qui considère le calcul en tant qu'évaluation de fonctions mathématiques. Comme le changement d'état et la mutation des données ne peuvent pas être représentés par des évaluations de fonctions, la programmation fonctionnelle ne les admet pas ; au contraire, elle met en avant l'**application des fonctions**, contrairement au modèle de programmation impérative qui met en avant les changements d'état.

Un peu d'histoire. L'origine de la programmation fonctionnelle peut être trouvée dans le **lambda-calcul**. Le langage fonctionnel le plus ancien est Lisp, créé en 1958 par McCarthy. Lisp a donné naissance à de nombreuses variantes telles que

Scheme (1975). Des langages fonctionnels plus récents, tels Haskell (1987), OCaml, Scala (2003), ont vu le jour et sont employés dans l'industrie.

La programmation fonctionnelle présente de nombreux atouts devant la programmation objet. Elle s'affranchit de façon radicale des **effets secondaires** (ou **effets de bord**) en interdisant toute opération d'affectation. On s'assure ici d'éviter un problème majeur : les effets de bord.

> **R** En *programmation impérative*, une **même fonction appelée à différents moments avec les mêmes paramètres peut renvoyer différents résultats**. C'est exactement ce que souhaite éviter la programmation fonctionnelle.

■ **Exemple 7.11.**
La fonction `cumuler` suivante a un effet de bord : elle modifie le contenu de la variable `tableau` à chaque appel. ■

```
tableau = [1]

def cumuler(entier):
    tableau[0] = tableau[0] + entier
    return tableau[0]
```

Lorsqu'on appelle une première fois `cumuler(2)`, on obtient 3. Lorsqu'on la rappelle une seconde fois, on obtient 5. Pourquoi ? Parce que la fonction a un **effet de bord** consistant à modifier le contenu de `tableau`.

> **R** Tester cette fonction demande de définir d'abord l'état de la machine (le contenu de `tableau`) et d'en vérifier l'état après l'exécution de la fonction. Ce n'est pas très gênant quand la « machine » ne contient qu'une variable ; c'est plus difficile quand elle est le résultat de milliers de lignes d'instructions. C'est typiquement ce que souhaite éviter la programmation fonctionnelle.

Retenez simplement que Python n'est pas un langage fonctionnel... mais qu'il en implémente une partie. En particulier, les **listes par compréhension**, les fonctions **lambda** mais aussi la fonction `map` ou encore `enumerate` s'inspirent de la programmation fonctionnelle.

■ **Exemple 7.12.**
Une fonction `lambda` est une fonction anonyme, définie sans le mot-clé `def`. Elle se contente de calculer une valeur à partir de ses paramètres, sans modifier aucun état : c'est une fonction **pure**. ■

```
double = lambda x: 2 * x
print(double(21))
```

Ce programme affiche `42`.

■ **Exemple 7.13.**
Les listes par compréhension permettent de construire une liste en une seule expression, sans boucle ni variable d'accumulation. Les deux programmes suivants construisent la même liste de carrés : le premier en style impératif, le second en style fonctionnel. ■

```
carres = []
for i in range(1, 11):
    carres.append(i ** 2)
```

```
carres = [i ** 2 for i in range(1, 11)]
```

■ **Exemple 7.14.**
La fonction `map` applique une fonction à chaque élément d'une liste ; `enumerate` permet de parcourir une liste en obtenant à la fois l'indice et la valeur. Ces deux outils viennent de la programmation fonctionnelle. ■

```
noms = ["Aya", "Noe", "Isma"]
majuscules = list(map(str.upper, noms))
for i, nom in enumerate(noms):
    print(i, nom)
```

Ce programme affiche :

```
0 Aya
1 Noe
2 Isma
```

R Nous approfondirons la programmation fonctionnelle plus longuement dans la suite de l'année, au travers d'exemples implémentés en Python.

4. Programmation événementielle

La **programmation événementielle** est un paradigme de programmation fondé sur les **événements**. Elle s'oppose à la programmation séquentielle (impérative, objet) : le programme sera principalement défini par ses **réactions** aux différents événements qui peuvent se produire, c'est-à-dire des changements d'état de variable, par exemple l'incrémentation d'une liste, un mouvement de souris ou de clavier.

Elle permet de réaliser des opérations **asynchrones**, comme par exemple réagir à une action venant de l'extérieur : « si la souris quitte l'écran, afficher un popup "ne partez pas, abonnez-vous !" ».

JavaScript est le langage le plus utilisé qui implémente la programmation événementielle. En JavaScript, par exemple, il est totalement impossible de « figer » l'exécution durant un temps précis.

■ **Exemple 7.15.**
En Python, il suffit d'utiliser la fonction `sleep` du module `time` pour suspendre le programme pendant une durée fixée. ■

```
from time import sleep

print("Dans 3 secondes ca explose !")
sleep(3)
print("BOOM !")
```

En JavaScript, le moteur continuera toujours à effectuer des calculs en attendant la réalisation d'un événement. Par exemple, si la page a fini de charger les images et qu'elle attendait ça pour les afficher, vos images peuvent apparaître soudainement durant le déroulement d'une instruction.

La programmation événementielle peut être modélisée dans les jeux vidéos, par exemple, avec une boucle infinie :

Exemple 7.16.
La structure générale d'un jeu vidéo : une boucle infinie qui lit les actions du joueur (les événements), calcule le nouvel état du jeu, puis dessine le monde. ■

```
Tant que VRAI :
    lire les actions du joueur
    calculer le nouvel etat du jeu
    dessiner le monde
Fin tant que
```

Exemple 7.17.
Un programme événementiel est organisé autour d'une **file d'événements** : chaque événement (clic, touche, déplacement) est mis en attente, puis traité un par un par le programme, qui déclenche la réaction associée. ■

Exemple 7.18.
Un distributeur de billets est un bon exemple de programme événementiel : il ne fait rien tant qu'aucun événement externe ne se produit (insertion de la carte, saisie du code, choix du montant). Chaque événement déclenche une réaction du programme.
■

Le tableau suivant donne quelques exemples de couples « événement / réaction » :

Événement	Réaction du programme
Clic sur le bouton « Valider »	Envoi du formulaire
Touche flèche droite enfoncée	Déplacement du personnage
Souris qui quitte la fenêtre	Affichage d'un message
Chargement des images terminé	Affichage des images

III. Comparer et choisir un paradigme

Les quatre paradigmes que nous venons de voir ne s'excluent pas : avec un même langage de programmation, on peut utiliser des paradigmes différents, et dans un même programme, on peut utiliser des paradigmes différents. Python est un langage **multiparadigme** : on peut y écrire des programmes impératifs, orientés objet ou fonctionnels, et les mélanger.

Paradigme	Vision du programme	État du programme	Exemples de langages	Domaines d'application
Impératif	Suite d'instructions exécutées dans l'ordre	Modifié en permanence par les instructions	Fortran, C, Python, Java, JavaScript	Scripts, calculs, systèmes
Objet	Collection d'objets qui interagissent	Réparti dans les attributs des objets	SmallTalk, Java, C++, Python	Applications, simulations, jeux
Fonctionnel	Évaluation de fonctions mathématiques	Aucun : pas de mutation ni d'affectation	Lisp, Scheme, Haskell, OCaml, Scala	Calcul scientifique, traitement de données
Événementiel	Réactions à des événements extérieurs	Partagé entre les gestion-naires d'événements	JavaScript	Interfaces web, jeux vidéo

■ **Exemple 7.19.**
Le même problème — calculer la somme des carrés des entiers de 1 à n — peut être traité dans trois paradigmes différents, en Python. ■

En **impératif**, on accumule le résultat dans une variable, modifiée à chaque tour de boucle :

```
def somme_carres(n):
    total = 0
    for i in range(1, n + 1):
        total = total + i ** 2
    return total
```

En **fonctionnel**, on exprime le calcul directement, sans variable modifiée :

```
def somme_carres(n):
    return sum(i ** 2 for i in range(1, n + 1))
```

En **orienté objet**, on encapsule la valeur n dans un objet, et le calcul devient une méthode de cet objet :

```
class Calcul:
    def __init__(self, n):
        self.n = n

    def somme_carres(self):
        return sum(i ** 2 for i in range(1, self.n + 1))
```

Les trois versions renvoient le même résultat, mais la manière de penser le problème est différente.

> **R** Comment choisir le paradigme ? Tout dépend du **champ d'application** du programme : un script simple et court sera naturellement écrit en impératif ; un gros système manipulant des entités du monde réel (clients, commandes, comptes) se prête bien à l'objet ; un calcul sur des données, sans effet de bord, se prête bien au fonctionnel ; une interface réagissant à l'utilisateur se prête bien à l'événementiel. Il n'y a pas de « meilleur » paradigme en général, seulement des paradigmes mieux adaptés à chaque situation.

■ **Exemple 7.20.**
Quelques situations et le paradigme le plus naturel pour les traiter : une application bancaire (objets `Compte`, `Client` et leurs interactions), un calcul de statistiques sur de grandes listes (fonctionnel, avec `map` et les listes par compréhension), un site web interactif (événementiel), un petit script de conversion d'unités (impératif). ■

IV. Exercices

Les exercices suivants reprennent et complètent les notions du chapitre. Pensez à bien lire les énoncés, à identifier le paradigme utilisé dans chaque extrait de code, et à écrire des programmes propres.

Exercices d'automatismes

Les exercices d'automatismes exercent les réflexes fondamentaux : vocabulaire des paradigmes, reconnaissance d'un paradigme dans un extrait de code, rappels de Première sur les fonctions et les listes par compréhension. Chaque question doit pouvoir être traitée en moins de trente secondes.

Exercice 7.1.

Dire si les affirmations suivantes sont vraies ou fausses.

1. Un paradigme de programmation est une façon d'aborder la programmation et de formuler les solutions d'un problème.
2. Python est un langage exclusivement fonctionnel.
3. La programmation fonctionnelle admet les changements d'état et la mutation des données.
4. JavaScript implémente la programmation événementielle.
5. En programmation impérative, une même fonction appelée à différents moments avec les mêmes paramètres peut renvoyer différents résultats.

■ **Exercice 7.2.**
Identifier le paradigme utilisé dans chacun des extraits de code suivants (impératif, objet, fonctionnel ou événementiel).

```
x = 1
y = x + 2

class Chien:
    pass

carres = [i ** 2 for i in range(10)]

from time import sleep
sleep(2)

def double(x):
    return 2 * x
```

■

■ **Exercice 7.3.**
On exécute le programme suivant (rappel de Première sur les fonctions) :

```
def bonjour(nom):
    return "Bonjour " + nom

print(bonjour("Aya"))
```

Que va afficher ce programme ? ■

■ **Exercice 7.4.**
Écrire en une seule ligne une liste par compréhension qui construit la même liste que la boucle suivante (rappel de Première) :

```
carres = []
for i in range(1, 11):
    carres.append(i ** 2)
```

■

■ **Exercice 7.5.**

Associer chaque événement à la réaction du programme qui convient : clic sur un bouton, touche flèche droite enfoncée, chargement des images terminé, souris qui quitte la fenêtre. Réactions proposées : affichage des images, déplacement du personnage, envoi du formulaire, affichage d'un message. ■

Exercices d'application

Les exercices d'application demandent d'écrire des programmes complets dans les différents paradigmes, et de comparer les styles.

■ **Exercice 7.6.**

On veut calculer la somme des carrés des entiers de 1 à n.

1. Écrire une version **impérative** : une fonction `somme_carres(n)` avec une boucle et une variable d'accumulation.
2. Écrire une version **fonctionnelle** : la même fonction en une seule expression, avec `sum` et une liste par compréhension.
3. Vérifier que les deux fonctions renvoient le même résultat pour $n = 10$.

```
def somme_carres(n):
    total = 0
    for i in range(1, n + 1):
        total = total + i ** 2
    return total
```

■

■ **Exercice 7.7.**

On considère la fonction suivante :

```
tableau = [1]

def cumuler(entier):
    tableau[0] = tableau[0] + entier
    return tableau[0]
```

1. Que renvoie `cumuler(2)` si on l'appelle une première fois ? Et une seconde fois ? Justifier.
2. Expliquer pourquoi cette fonction a un effet de bord.
3. Écrire une version de cette fonction **sans effet de bord** : elle prendra le total en paramètre et renverra le nouveau total, sans modifier aucune variable globale.

■

■ **Exercice 7.8.**
On veut modéliser un compteur avec la programmation orientée objet (rappel du chapitre 0).

1. Écrire une classe `Compteur` avec un attribut `valeur` initialisé à 0, une méthode `incrementer` qui ajoute 1 à la valeur, et une méthode `lire` qui renvoie la valeur.
2. Créer deux compteurs distincts, incrémenter le premier trois fois, et afficher la valeur des deux compteurs. Que remarque-t-on ?

```
class Compteur:
    def __init__(self):
        self.valeur = 0

    def incrementer(self):
        self.valeur = self.valeur + 1

    def lire(self):
        return self.valeur
```

■

■ **Exercice 7.9.**
On dispose d'une liste de notes :

```
notes = [12.5, 8.25, 15.0, 9.75, 17.0]
```

1. Écrire une liste par compréhension qui ne garde que les notes supérieures ou égales à 10.
2. Utiliser `map` avec la fonction `round` pour arrondir toutes les notes à l'entier le plus proche.
3. Calculer la moyenne des notes avec `sum` et `len` (une seule expression).

■

■ **Exercice 7.10.**
On veut modéliser un jeu vidéo simple avec la programmation événementielle.

1. Lister les événements possibles dans ce jeu (clavier, souris, temps).
2. Écrire la boucle principale du jeu en pseudo-code : lire les actions du joueur, calculer le nouvel état du jeu, dessiner le monde.

```
Tant que VRAI :
    lire les actions du joueur
    calculer le nouvel etat du jeu
    dessiner le monde
```

```
Fin tant que
```

■

Exercices de réflexion

Les exercices de réflexion demandent de comparer les paradigmes, d'analyser des situations limites et de choisir un paradigme selon le champ d'application.

■ **Exercice 7.11.**

On veut calculer la moyenne d'une liste de notes.

1. Écrire une version **impérative** : une boucle qui accumule la somme dans une variable.
2. Écrire une version **fonctionnelle** : une seule expression avec `sum` et `len`.
3. Comparer les deux versions : laquelle modifie un état ? Laquelle est la plus courte ? Laquelle est la plus facile à tester ?

```
def moyenne_imperative(notes):
    total = 0
    for note in notes:
        total = total + note
    return total / len(notes)
```

```
def moyenne_fonctionnelle(notes):
    return sum(notes) / len(notes)
```

■

■ **Exercice 7.12.**

On considère le programme suivant, qui utilise une variable globale :

```
tableau = [1]

def cumuler(entier):
    tableau[0] = tableau[0] + entier
    return tableau[0]

print(cumuler(2))
print(cumuler(2))
```

1. Que va afficher ce programme ? Justifier précisément.
2. Expliquer pourquoi cette fonction est difficile à tester : que faut-il connaître avant chaque appel ?
3. Proposer une version de la fonction sans effet de bord, et montrer qu'avec cette version les deux appels produisent le même résultat.

■

■ **Exercice 7.13.**
Pour chacune des situations suivantes, proposer le paradigme de programmation le plus adapté et justifier le choix.

1. Une application de gestion d'une bibliothèque (livres, adhérents, emprunts).
2. Un script qui convertit des températures de degrés Celsius en degrés Fahrenheit.
3. Un site web qui affiche une notification dès que le serveur reçoit un message.
4. Un programme de calcul scientifique manipulant de très grandes listes de mesures.

■

■ **Exercice 7.14.**
Dans un même programme Python, on peut utiliser plusieurs paradigmes. On considère la classe suivante, qui mélange programmation objet (la classe) et programmation fonctionnelle (la liste par compréhension dans la méthode) :

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

    def bonnes_notes(self):
        return [note for note in self.notes if note >= 10]
```

1. Expliquer en quoi cette classe relève de la programmation orientée objet.
2. Expliquer en quoi la méthode `bonnes_notes` relève de la programmation fonctionnelle.
3. Créer un objet `eleve` avec les notes `[12, 8, 15, 9, 17]` et afficher le résultat de `eleve.bonnes_notes()`.

■