Chapitre 4
Récursivité

La récursivité est le « sens » de l'informaticien : autant la travailler tôt. Comme tous les sens, il faut du temps et de l'énergie pour la développer. Dans ce chapitre, nous allons apprendre à écrire des fonctions qui s'appellent elles-mêmes, à comprendre comment un ordinateur exécute de telles fonctions, et à reconnaître les situations où cette façon de programmer est naturelle et efficace.

I. Définition et premier exemple

1. Qu'est-ce que la récursivité ?

Définition 4.1.
La **récursivité** est une démarche qui fait référence à l'objet même de la démarche à un moment du processus. En d'autres termes, c'est une démarche dont la description mène à la répétition d'une même règle. Elle se manifeste sous deux formes :
- écrire un algorithme qui s'invoque lui-même ;
- définir une structure à partir de l'une au moins de ses sous-structures.

■ **Exemple 4.1.**
Les **poupées russes** sont une illustration classique de la récursivité : une poupée contient une poupée plus petite, qui contient elle-même une poupée plus petite, et ainsi de suite jusqu'à la plus petite poupée, qui ne contient plus rien. La structure « poupée » est définie à partir d'une sous-structure qui est elle-même une poupée. ■

■ **Exemple 4.2.**
On définit la factorielle $n! = n \times (n-1) \times \cdots \times 2 \times 1$. On peut aussi voir la factorielle de la façon suivante :

$$\begin{aligned} 5! &= 5 \times 4! \\ &= 5 \times 4 \times 3! \\ &= 5 \times 4 \times 3 \times 2! \\ &= 5 \times 4 \times 3 \times 2 \times 1 \end{aligned}$$

Pour calculer 5!, on a besoin de 4!, qui a besoin de 3!, et ainsi de suite : c'est la même règle ($n! = n \times (n-1)!$) qui se répète à chaque étape. La factorielle est une fonction **récursive**. ■

2. Une première fonction récursive : la factorielle

■ **Exemple 4.3.**
La fonction `factorielle` ci-dessous se calcule en s'appelant elle-même : c'est un programme récursif. ■

```
def factorielle(n):
    if n <= 1:
        return 1
    else:
        return n * factorielle(n - 1)
```

■ **Exemple 4.4.**
Que renvoie `factorielle(0)` ? Le paramètre $n = 0$ vérifie $n \leqslant 1$: la condition `if` est vraie et la fonction renvoie 1 sans faire le moindre appel récursif. De même, `factorielle(1)` renvoie 1. On dit que la fonction possède un **cas de base**. ■

> **R** Une fonction récursive ne peut pas s'appeler elle-même indéfiniment : il lui faut au moins une situation qui ne consiste pas en un appel récursif, sinon elle ne se terminerait jamais. Nous verrons dans la section III comment formaliser cette idée.

II. Déroulé d'un programme récursif

1. La pile des appels

Chaque appel récursif s'ajoute à la **pile des appels** successifs de la fonction. Chaque appel possède son propre environnement, donc ses propres variables. La pile est nécessaire pour mémoriser les valeurs propres à chaque appel : quand un appel se termine, il renvoie sa valeur à l'appel qui l'attend, juste en dessous dans la pile.

■ **Exemple 4.5.**
Déroulons l'exécution de `factorielle(4)`. Chaque appel est mis en attente tant que son appel récursif n'a pas renvoyé sa valeur : ■

```
factorielle(4) est appelee
|-- factorielle(3) est appelee
| |-- factorielle(2) est appelee
| | |-- factorielle(1) est appelee
| | | cas d'arret : retourne 1
| | |-- 2 * 1 = 2 : retourne 2
| |-- 3 * 2 = 6 : retourne 6
|-- 4 * 6 = 24 : retourne 24
```

La fonction `factorielle(4)` renvoie donc 24. On remarque que les appels se **superposent** : `factorielle(4)` attend le résultat de `factorielle(3)`, qui attend celui de `factorielle(2)`, etc. C'est exactement le comportement d'une pile (dernier arrivé, premier servi).

> **R** Chaque appel possède son propre environnement, donc sa propre variable n : le n de `factorielle(3)` n'est pas le n de `factorielle(2)`. C'est pour cela que la pile est nécessaire : elle mémorise les valeurs propres à chaque appel.

2. La limite de profondeur de récursivité

Attention ! En Python, la taille de la pile des appels récursifs est **limitée**. Si la récursivité est trop profonde et que l'on atteint cette limite, on déclenche une `RecursionError`.

▪ Exemple 4.6.
La fonction suivante s'appelle elle-même sans jamais s'arrêter : elle déclenche une `RecursionError`. ▪

```
def boucle_infinie(n):
    return boucle_infinie(n)

boucle_infinie(3)
```

Python lève l'erreur `RecursionError: maximum recursion depth exceeded` : la profondeur maximale de récursivité (environ 1000 appels par défaut) a été dépassée.

3. Visualiser les appels avec la bibliothèque `recviz`

Pour mieux visualiser les appels récursifs des fonctions, on peut utiliser la bibliothèque `recviz`.

> **R** Pour installer la bibliothèque, on utilise la commande `pip install recviz`. En environnement Capytale, on peut l'installer directement dans un notebook avec `micropip` (voir les cellules d'installation du notebook associé à ce chapitre).

On utilise `recviz` en plaçant le décorateur `@recviz` juste au-dessus de la fonction : chaque appel et chaque retour sont alors affichés avec les valeurs des paramètres et du résultat.

■ **Exemple 4.7.**
La fonction `factorielle` décorée avec `@recviz` : on voit s'empiler les appels, puis se dérouler les retours. ■

```
from recviz import recviz

@recviz
def factorielle(n):
    if n <= 1:
        return 1
    else:
        return n * factorielle(n - 1)
```

■ **Exemple 4.8.**
La fonction de **Fibonacci**, définie par $F(0) = 1$, $F(1) = 1$ et $F(n) = F(n-1) + F(n-2)$, se prête très bien à une définition récursive : chaque appel fait deux appels récursifs. ■

```
from recviz import recviz

@recviz
def fibonacci(n):
    if n <= 1:
        return 1
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)
```

L'appel `fibonacci(6)` déclenche 25 appels de la fonction : l'arbre des appels ci-dessous montre pourquoi, sur l'exemple plus lisible de `fibonacci(4)`.

```
fibonacci(4)
|-- fibonacci(3)
| |-- fibonacci(2)
| | |-- fibonacci(1) : retourne 1
| | |-- fibonacci(0) : retourne 1
| | |-- 1 + 1 = 2 : retourne 2
| |-- fibonacci(1) : retourne 1
| |-- 2 + 1 = 3 : retourne 3
|-- fibonacci(2)
| |-- fibonacci(1) : retourne 1
| |-- fibonacci(0) : retourne 1
| |-- 1 + 1 = 2 : retourne 2
|-- 3 + 2 = 5 : retourne 5
```

R L'arbre des appels de `fibonacci` montre qu'un même calcul est répété de nombreuses fois : par exemple, `fibonacci(2)` est calculé deux fois dans `fibonacci(4)`, et `fibonacci(1)` trois fois. C'est le principal **inconvénient** de la récursivité naïve : le nombre d'appels peut devenir très grand. Nous y reviendrons dans les exercices.

III. Les éléments caractéristiques d'une fonction récursive

1. Le cas d'arrêt

Il faut que, dans la fonction, il y ait au moins une situation qui ne consiste pas en un appel récursif (sinon la fonction ne s'arrêterait jamais). Dans la fonction `factorielle`, le cas d'arrêt est :

```
if n <= 1:
    return 1
```

Définition 4.2.
Cette situation est appelée **situation de terminaison**, **situation d'arrêt**, **cas d'arrêt** ou **cas de base** : c'est la situation dans laquelle la fonction répond directement, sans se rappeler elle-même.

■ **Exemple 4.9.**
Pour une fonction qui additionne les éléments d'une liste, le cas d'arrêt naturel est la **liste vide** : s'il n'y a plus aucun élément, la somme vaut 0, sans appel récursif. ■

```
def addition_list(liste):
    if not liste:
        return 0
    else:
        return liste[0] + addition_list(liste[1:])
```

■ **Exemple 4.10.**
Pour vérifier si une chaîne est un **palindrome** (une chaîne qui se lit identiquement dans les deux sens, comme « radar »), on compare les caractères des extrémités. Le cas d'arrêt est alors une chaîne vide ou réduite à un seul caractère : une telle chaîne est toujours un palindrome. ■

■ **Exemple 4.11.**
Pour une fonction qui cherche une valeur dans une liste, le cas d'arrêt est la liste vide : s'il ne reste plus aucun élément à examiner, la valeur n'est pas dans la liste, et la fonction renvoie `False`. ■

2. Il faut se rapprocher du cas d'arrêt

Là où la récursivité et la récurrence sont vraiment **différentes**, c'est le cas suivant. Une récurrence (classique) part d'un cas d'initialisation et va s'en éloigner pour aller vers $+\infty$: visuellement, on déplace le même problème avec juste des nombres plus gros. Une récursivité a plus un **effet de loupe** : à chaque appel, on rapproche la loupe du cas d'arrêt.

■ **Exemple 4.12.**
L'effet de loupe : pour `factorielle(5)`, les paramètres des appels successifs sont 5, puis 4, puis 3, puis 2, puis 1 : on se rapproche à chaque appel du cas d'arrêt $n \leqslant 1$. La loupe se resserre sur un problème de plus en plus petit, jusqu'au cas de base. ■

Plus simplement : l'appel récursif doit **modifier l'argument** vers le cas d'arrêt.

```
return n * factorielle(n - 1)
```

Ici, l'argument passe de n à $n-1$: on se rapproche du cas d'arrêt $n \leqslant 1$.

■ **Exemple 4.13.**
Une fonction récursive qui affiche un compte à rebours : le paramètre n décroît de 1 à chaque appel jusqu'à atteindre le cas d'arrêt $n = 0$. ■

```
def compte_a_rebours(n):
    if n == 0:
        print("Decollage !")
    else:
        print(n)
        compte_a_rebours(n - 1)

compte_a_rebours(3)
```

Ce programme affiche : `3`, `2`, `1`, puis `Decollage !`.

Il faut s'assurer que la situation de terminaison est atteinte après un **nombre fini** d'appels récursifs.

■ **Exemple 4.14.**
Dans le cas de la factorielle, la suite des valeurs du paramètre est $n, n-1, n-2, \ldots, 1$:

c'est une suite **strictement décroissante** d'entiers positifs, qui atteint forcément le cas d'arrêt $n \leqslant 1$ après un nombre fini d'étapes. La fonction se termine donc toujours. ∎

> **R** La preuve de terminaison d'un algorithme récursif se fait en identifiant la construction d'une suite **strictement décroissante d'entiers positifs ou nuls**. Dans le cas de la factorielle, il s'agit simplement de la suite des valeurs du paramètre. Si l'on arrive à exhiber une telle suite, l'algorithme se termine ; si l'on n'y arrive pas, il risque de ne jamais s'arrêter.

Pour résumer

Le tableau suivant récapitule les deux ingrédients indispensables d'une fonction récursive.

Ingrédient	Rôle	Exemple avec `factori`
Cas d'arrêt	Situation qui ne fait pas d'appel récursif	`if n <= 1: return 1`
Appel récursif	Appel de la fonction sur un paramètre modifié	`return n * factorie`
Terminaison	Suite strictement décroissante d'entiers vers le cas d'arrêt	$n, n-1, n-2, \ldots, 1$

IV. Exercices

Les exercices suivants reprennent et complètent les notions du chapitre. Pensez à bien identifier, pour chaque fonction récursive que vous écrivez, le cas d'arrêt et l'appel récursif, et à vérifier que l'appel récursif modifie l'argument vers le cas d'arrêt.

Exercices d'automatismes

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

■ Exercice 4.1.

Dire si les affirmations suivantes sont vraies ou fausses, en justifiant brièvement.

1. Une fonction récursive est une fonction qui s'appelle elle-même.
2. Une fonction récursive peut ne pas contenir de cas d'arrêt : elle s'arrête toute seule.
3. Le cas d'arrêt est aussi appelé cas de base ou situation de terminaison.
4. Pour se terminer, une fonction récursive doit modifier son paramètre pour se rapprocher du cas d'arrêt.
5. En Python, la profondeur de récursivité est illimitée.

■

■ Exercice 4.2.

On considère la fonction suivante :

```
def factorielle(n):
    if n <= 1:
        return 1
    else:
        return n * factorielle(n - 1)
```

1. Quel est le cas d'arrêt de cette fonction ?
2. Quel est l'appel récursif ?
3. Que renvoie `factorielle(0)` ? `factorielle(1)` ? `factorielle(4)` ?

■

■ **Exercice 4.3.**
Que va afficher le programme suivant ?

```
def mystere(n):
    if n == 0:
        print("Fin")
    else:
        print(n)
        mystere(n - 1)

mystere(3)
```

■

■ **Exercice 4.4.**
Rappel de Première : écrire une fonction **itérative** `somme_liste(liste)` qui renvoie la somme des éléments d'une liste de nombres, à l'aide d'une boucle `for` et d'un accumulateur.

```
def somme_liste(liste):
    ...
```

■

■ **Exercice 4.5.**
Compléter la fonction récursive `puissance` qui calcule n^k en utilisant la relation $n^k = n \times n^{k-1}$ et le fait que $n^0 = 1$.

```
def puissance(n, k):
    if k == 0:
        return ...
    else:
        return ...
```

■

■ **Exercice 4.6.**
On exécute le programme suivant :

```
def boucle_infinie(n):
    return boucle_infinie(n)
```

```
boucle_infinie(3)
```

Que se passe-t-il ? Expliquer ce qui se produit, et nommer l'erreur Python déclenchée.

■

Exercices d'application

Les exercices d'application demandent d'écrire des fonctions itératives et récursives complètes, en identifiant systématiquement le cas d'arrêt et l'appel récursif.

■ **Exercice 4.7.**

On souhaite additionner tous les termes d'une liste de nombres.

1. Écrire une fonction **itérative** `addition_list(liste)` qui additionne tous les termes d'une liste.
2. Écrire une fonction **récursive** `addition_list(liste)` qui additionne tous les termes d'une liste. On précisera le cas d'arrêt (la liste vide) et l'appel récursif.
3. Tester avec `addition_list([1, 2, 3, 4, 5])`.

■

■ **Exercice 4.8.**

On souhaite renvoyer le maximum d'une liste de nombres.

1. Écrire une fonction itérative `max_list(liste)`.
2. Écrire une fonction récursive `max_list(liste)`. On pourra comparer le premier élément avec le maximum du reste de la liste.
3. Tester avec `max_list([1, 2, 5, 4, 3])`.

■

■ **Exercice 4.9.**

On souhaite savoir si une valeur est présente dans une liste.

1. Écrire une fonction itérative `est_dans_list(valeur, liste)` qui renvoie `True` si la valeur est dans la liste, `False` sinon.
2. Écrire une fonction récursive `est_dans_list(valeur, liste)`. On précisera les deux cas d'arrêt (liste vide ; premier élément égal à la valeur cherchée).
3. Tester avec `est_dans_list(4, [1, 2, 5, 4, 3])`.

■

■ **Exercice 4.10.**
On souhaite mettre toutes les lettres d'une chaîne de caractères en majuscules.
1. Écrire une fonction itérative `majuscule(chaine)` (on pourra utiliser la méthode `upper` sur un caractère).
2. Écrire une fonction récursive `majuscule(chaine)`.
3. Tester avec `majuscule("abcde")`, qui doit renvoyer `"ABCDE"`.

■

■ **Exercice 4.11.**
On souhaite compter tous les chiffres présents dans une chaîne de caractères.
1. Écrire une fonction itérative `nombre_dans_string(chaine)` (on pourra utiliser la méthode `isdigit`).
2. Écrire une fonction récursive `nombre_dans_string(chaine)`.
3. Tester avec `nombre_dans_string("ab1cd5e4")`, qui doit renvoyer 3.

■

■ **Exercice 4.12.**
La fonction de Fibonacci est définie par :
— $F(0) = 1$;
— $F(1) = 1$;
— pour tout entier positif n, $F(n) = F(n-1) + F(n-2)$.
1. Implémenter itérativement et récursivement cette fonction de Fibonacci. Dans le cas récursif, donner la condition d'arrêt, le cas trivial et l'appel récursif.
2. Quel est l'inconvénient du calcul récursif ? On le démontrera par une visualisation des appels effectués par la fonction récursive et une visualisation du calcul itératif (on pourra s'appuyer sur l'arbre des appels du cours et sur le nombre d'appels déclenchés par `fibonacci(6)`, qui vaut 25).

```
def fibonacci(n):
    if n <= 1:
        return 1
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)
```

■

■ **Exercice 4.13.**
On rappelle que la division euclidienne de a par b ($a \geqslant 0, b > 0$) s'écrit $a = b \times q + r$

avec $0 \leqslant r < b$: q est le quotient, r le reste.

1. Écrire une fonction récursive `quotient(a, b)` qui calcule le quotient de la division entière de deux nombres strictement positifs, en soustrayant b à a tant que $a \geqslant b$.
2. Écrire une fonction récursive `reste(a, b)` qui calcule le reste de la division entière de deux nombres strictement positifs.

■

■ **Exercice 4.14.**

Un palindrome est une chaîne de caractères qui se lit de gauche à droite ou de droite à gauche, comme « radar », « elle », « été », « esope reste ici et se repose », si l'on fait abstraction des blancs. Écrire une fonction récursive `est_palindrome(chaine)` qui vérifie si une chaîne de caractères est un palindrome ou non. On précisera les cas d'arrêt (une chaîne vide ou de longueur 1, et deux extrémités différentes). ■

■ **Exercice 4.15.**

La fonction miroir donne l'inverse d'une chaîne de caractères, c'est-à-dire la chaîne formée des mêmes caractères mais dans l'ordre inverse ; par exemple `miroir("seminaire")` renvoie `"erianimes"`.

1. Écrire une fonction récursive `miroir(chaine)` qui réalise cette fonction.
2. Proposer une version itérative équivalente.

■

■ **Exercice 4.16.**

Soit une fonction récursive `puissance` qui prend en paramètre un réel n et un entier k et renvoie le calcul de n^k.

1. Écrire la fonction récursive `puissance` qui calcule n^k de façon naïve en considérant $n^k = n \times n^{k-1}$ et $n^0 = 1$.
2. Il existe une autre méthode de calcul de n^k plus efficace : la méthode par **dichotomie**. Elle repose sur le fait que si k est pair alors $n^k = (n \times n)^{k/2}$ et $n^k = n \times n^{k-1}$ sinon. Écrire la fonction récursive `puissance_dic` qui calcule n^k par dichotomie.

■

■ **Exercice 4.17.**
Écrire une fonction récursive `liste_aleatoire(n)` qui prend en entrée un nombre n et qui renvoie en sortie une liste de n nombres aléatoires pris entre 0 et 10. On utilisera `random.randint(0, 10)` après `import random`.

```
import random

def liste_aleatoire(n):
    ...
```

■

■ **Exercice 4.18.**
Écrire une fonction récursive `compter_dans_dictionnaire(liste)` qui, à partir d'une liste de nombres, renvoie un dictionnaire dont les clés sont les valeurs de la liste et les valeurs associées le nombre d'occurrences de chacune.

```
liste = [5, 3, 5, 1, 8, 9, 8]
compter_dans_dictionnaire(liste)
```

■

■ **Exercice 4.19.**
Type Bac. On souhaite créer une fonction qui donne l'écriture binaire d'un nombre décimal.

1. Écrire une telle fonction de façon itérative (rappel de Première : l'écriture binaire s'obtient par divisions euclidiennes successives par 2).
2. Écrire une telle fonction de façon récursive. Souligner la condition d'arrêt, le cas trivial et l'appel récursif.

■

Exercices de réflexion

Les exercices de réflexion demandent de concevoir des fonctions récursives plus complexes, de combiner récursivité et structures de données, et d'analyser des situations limites.

■ **Exercice 4.20.**
On souhaite trier une liste par ordre croissant en utilisant la méthode « diviser pour régner » : c'est le **tri fusion**.

1. Écrire une fonction récursive `fusion_liste(liste1, liste2)` qui, à partir de deux listes triées par ordre croissant, renvoie la liste fusionnée des

éléments triés par ordre croissant.

2. Écrire une fonction récursive `trier_liste(liste)` qui trie une liste par ordre croissant : on coupe la liste en deux, on trie récursivement chaque moitié, puis on les fusionne.

```
def fusion_liste(liste1, liste2):
    ...

def trier_liste(liste):
    ...
```

■

■ **Exercice 4.21.**
On considère la classe `Pile` suivante (structure LIFO, vue en Première) :

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

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

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

    def est_vide(self):
        return len(self.data) == 0
```

Écrire une fonction récursive `vider_pile(pile)` qui vide une pile (elle renvoie la pile vidée). ■

■ **Exercice 4.22.**
On reprend la classe `Pile` de l'exercice précédent.

1. Créer une pile et lui empiler les nombres de 1 à 10 (le nombre 1 sera le dernier à être dépilé).
2. Écrire une fonction itérative `retourner_pile(pile, pile_inversee)` qui retourne la pile : elle dépile tous les éléments de `pile` et les empile dans `pile_inversee`.
3. Écrire une fonction récursive qui retourne la pile.

■

■ **Exercice 4.23.**

On considère la classe `File` suivante (structure FIFO, vue en Première) :

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

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

    def defiler(self):
        return self.data.pop(0)

    def est_vide(self):
        return len(self.data) == 0
```

1. Créer une instance de `File` avec les nombres de 1 à 10.
2. Écrire une fonction itérative qui vide la file.
3. Écrire une fonction récursive qui vide la file.

■

■ **Exercice 4.24.**

Classique : les tours de Hanoï.

1. Rappeler (ou chercher sur internet) les règles des tours de Hanoï.
2. Implémenter la fonction récursive calculant la solution des déplacements de n anneaux dans les tours de Hanoï, sachant qu'on ne peut pas poser un anneau plus grand sur un anneau plus petit.

■