

La Récursivité

1. Principes et première approche

Une **fonction récursive** est une fonction qui s'appelle elle-même lors de son exécution. Contrairement à la programmation itérative qui utilise des boucles (`for`, `while`), la récursivité résout un problème en le décomposant en sous-problèmes identiques mais de taille plus réduite.

Exemple d'introduction : Décompte récursif

```
def f(n):
    if n == 0:
        print("Bye-bye !")
    else:
        print("Salut !")
        f(n - 1) # Appel récursif avec n - 1

f(5)
```

Déroulement :

1. Appel `f(5)` : condition `5 == 0` fausse → affiche "Salut !" et appelle `f(4)`.
2. Le processus se répète pour $n = 4, 3, 2, 1$.
3. Appel `f(0)` : condition `0 == 0` vraie → affiche "Bye-bye !". La chaîne d'appels prend fin.

2. Conception d'un algorithme récursif

Pour illustrer la différence entre itération et récursion, calculons la somme des entiers de 1 à N : $S(N) = 1 + 2 + 3 + \dots + N$.

Approche itérative

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

Utilise une variable d'accumulation et une boucle pour incrémenter le résultat étape par étape.

Approche récursive

```
def somme_recurs(n):
    if n == 0:
        return 0 # Cas de base
    else:
        return n + somme_recurs(n - 1)
```

Repose sur la relation de récurrence mathématique : $S(n) = n + S(n - 1)$.

3. Structure fondamentale d'une fonction récursive

Toute fonction récursive valide doit respecter **3 règles d'or** :

- **Le Cas de base (Condition d'arrêt)** : Un cas simple traité directement sans appel récursif. Il permet de stopper la récursion.
- **Le Cas général (Appel récursif)** : La fonction s'appelle elle-même en modifiant ses arguments pour se rapprocher du cas de base.
- **La Garantie de terminaison** : Les paramètres doivent obligatoirement évoluer vers la condition d'arrêt (sinon, boucle infinie).

4. La Pile d'Exécution (Call Stack)

Lors de l'exécution, l'ordinateur mémorise l'état de chaque fonction en attente dans la **pile d'exécution**. Chaque appel empile un "cadre d'évaluation" contenant l'adresse de retour et les variables locales.

Trace d'exécution pour `somme_rekurs(4)` :

1. `somme_rekurs(4)` → calcule $4 + \text{somme_rekurs}(3)$ [En attente]
2. `somme_rekurs(3)` → calcule $3 + \text{somme_rekurs}(2)$ [En attente]
3. `somme_rekurs(2)` → calcule $2 + \text{somme_rekurs}(1)$ [En attente]
4. `somme_rekurs(1)` → calcule $1 + \text{somme_rekurs}(0)$ [En attente]
5. `somme_rekurs(0)` → Atteint le cas de base ! Renvoie 0
6. Remontée : $\text{somme_rekurs}(1) = 1 + 0 = 1$
7. Remontée : $\text{somme_rekurs}(2) = 2 + 1 = 3$
8. Remontée : $\text{somme_rekurs}(3) = 3 + 3 = 6$
9. Remontée : $\text{somme_rekurs}(4) = 4 + 6 = 10$ → RÉSULTAT FINAL : 10

Attention au Débordement de Pile (Stack Overflow)

Chaque appel récursif consomme de la mémoire. En Python, la profondeur maximale de récursion est limitée par défaut à **1 000 appels**. Si la limite est dépassée (absence de cas de base ou mauvais paramètre), Python lève l'exception `RecursionError`.

5. Bilan : Avantages vs Inconvénients

Avantages	Inconvénients / Limites
Lisibilité & Élégance : Code concis, très proche des définitions mathématiques.	Surcoût mémoire : Nécessite de l'espace dans la pile pour chaque appel intermédiaire.
Structures complexes : Idéal pour manipuler des structures arborescentes (arbres, graphes, fichiers).	Risque de plantage : Dépassement de la pile (<code>RecursionError</code>) si la profondeur est trop grande.
Preuves facilitées : La correction de l'algorithme se prouve naturellement par récurrence.	Performances : Les appels de fonctions sont légèrement plus lents qu'une simple boucle itérative.

6. Exercice d'application

Soit l'algorithme récursif ci-dessous :

```
def f(a, n):  
    if n == 0:  
        return 0  
    else:  
        return a + f(a, n - 1)
```

Questions :

1. Identifier le **cas de base** et le **cas général** de la fonction `f`.
2. Tracer l'arbre des appels récursifs et donner la valeur retournée par l'appel `f(4, 3)`.
3. En déduire le rôle mathématique accompli par la fonction `f(a, n)` pour $n \geq 0$.

