Fonctions récursives

I. Introduction

Dans le chapitre sur les fonctions, nous avons vu comment définir et utiliser des fonctions qui effectuent des calculs ou des traitements. Une fonction peut appeler d'autres fonctions pour réaliser son travail. Une situation particulière se produit lorsqu'une fonction s'appelle **elle-même** : on parle alors de récursivité.

Définition 1 : *Fonction récursive*

Une fonction récursive est une fonction qui s'appelle elle-même lors de son exécution. La récursivité est une technique de programmation qui permet de résoudre certains problèmes de manière élégante en les décomposant en sous-problèmes plus petits de même nature.

Remarque : La récursivité est un concept fondamental en informatique. De nombreuses structures de données (arbres, listes chaînées) et de nombreux algorithmes se prêtent naturellement à une formulation récursive. De plus, les suites définies par récurrence en mathématiques correspondent souvent à des calculs récursifs.

II. Principe de la récursivité

1) Structure d'une fonction récursive

Pour qu'une fonction récursive fonctionne correctement et termine son exécution, elle doit obéir à une structure en deux parties :

Structure d'une fonction récursive

Une fonction récursive doit contenir deux types de cas :

- ⋆ Le cas de base (ou cas d'arrêt) : c'est une situation simple pour laquelle on connaît directement le résultat, sans avoir besoin d'appeler à nouveau la fonction. Ce cas permet **d'arrêter** la récursion.
- ⋆ Le cas récursif : c'est la situation générale où le problème est décomposé en un ou plusieurs sous-problèmes plus petits. La fonction s'appelle alors elle-même sur ces sous-problèmes.

Important : le cas de base doit être atteignable après un nombre fini d'appels récursifs. Sinon, la fonction ne terminera jamais : on parle de récursion infinie.

Exemple : Considérons le calcul de $n!$ (factorielle de n) pour un entier naturel n. Par définition :

$$n! = n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1$$

avec la convention $0! = 1$.

On peut remarquer que $n! = n \times (n-1)!$. Ainsi, pour calculer $n!$, on peut :

- ⋆ **Cas de base :** si $n = 0$, alors $0! = 1$ (on connaît le résultat directement).
- ⋆ **Cas récursif :** si $n > 0$, alors $n! = n \times (n-1)!$ (on appelle la fonction sur $n-1$).

2) Schéma de déroulement d'un appel récursif

Lors d'un appel récursif, on peut distinguer deux phases :

- ⋆ La phase de descente : les appels successifs jusqu'à atteindre le cas de base.
- ⋆ La phase de remontée : après le cas de base, chaque appel en attente reprend son calcul et renvoie son résultat.

Remarque : Python (et la plupart des langages de programmation) utilise une pile d'appels pour gérer les appels de fonctions. Chaque appel récursif empile une nouvelle entrée ; le cas de base déclenche le dépilement.

III. Exemples

Cette section regroupe plusieurs exemples classiques de fonctions récursives. Chaque sous-partie illustre une situation où la récursivité s'applique naturellement.

1) Factorielle

```python
def factorielle(n):
    if n == 0:                      # Cas de base
        return 1
    else:                           # Cas récursif
        return n * factorielle(n - 1)
```

Exemple :

- ⋆ **Utilisation :**

```
>>> factorielle(0)
1
>>> factorielle(5)
120
>>> factorielle(10)
3628800
```

- ⋆ **Déroulement pour `factorielle(4)` :**
  - `factorielle(4)` appelle `factorielle(3)` car $4 \neq 0$
  - `factorielle(3)` appelle `factorielle(2)` car $3 \neq 0$
  - `factorielle(2)` appelle `factorielle(1)` car $2 \neq 0$
  - `factorielle(1)` appelle `factorielle(0)` car $1 \neq 0$
  - `factorielle(0)` renvoie `1` (cas de base)
  - Puis remontée : $1 \times 1 = 1$, $2 \times 1 = 2$, $3 \times 2 = 6$, $4 \times 6 = 24$

  Le résultat final est 24.

2) Somme des premiers entiers

Calculons la somme $S_n = 1 + 2 + 3 + \cdots + n$ pour un entier $n \geqslant 0$.

- ⋆ **Cas de base :** si $n = 0$, alors $S_0 = 0$.
- ⋆ **Cas récursif :** si $n > 0$, alors $S_n = n + S_{n-1}$.

```
def somme(n):
    if n == 0:
        return 0
    else:
        return n + somme(n - 1)
```

Exemple :

```
>>> somme(0)
0
>>> somme(5)
15
>>> somme(100)
5050
```

3) Suites récurrentes

Les suites définies par récurrence correspondent naturellement à des calculs récursifs. On distingue les suites récurrentes simples ($u_{n+1} = f(u_n)$ ou $u_{n+1} = f(u_n, n)$) et les suites récurrentes doubles ($u_{n+2} = f(u_{n+1}, u_n)$).

a) Suite simple sans n dans la formule de récurrence

Soit (u_n) définie par $u_0 = 2$ et $\forall n \in \mathbb{N}$, $u_{n+1} = 2u_n + 1$.

```
def suite_u(n):
    if n == 0:
        return 2
    else:
        return 2 * suite_u(n - 1) + 1
```

b) Suite simple avec n dans la formule de récurrence

Soit (v_n) définie par $v_0 = 1$ et $\forall n \in \mathbb{N}$, $v_{n+1} = v_n + n$ (la formule fait intervenir n en plus de v_n).

```
def suite_v(n):
    if n == 0:
        return 1
    else:
        return suite_v(n - 1) + (n - 1)
```

Remarque : Pour $v_{n+1} = v_n + n$, on a $v_n = v_{n-1} + (n-1)$. Lors de l'appel récursif sur $n-1$, le paramètre passé représente donc le « n » du terme précédent, d'où l'utilisation de `(n - 1)` dans la formule.

c) Suite récurrente double : Fibonacci

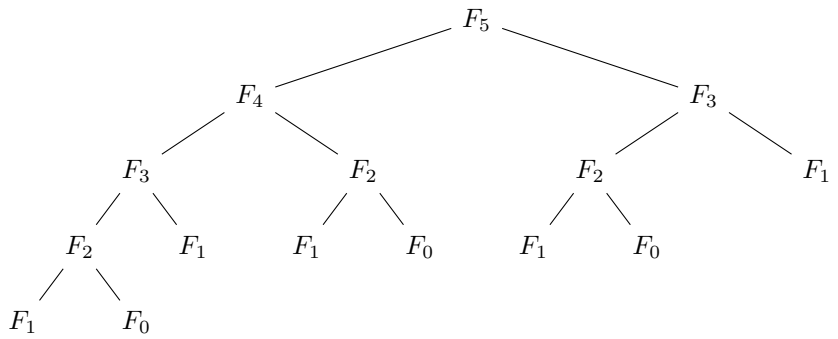
La suite de Fibonacci est définie par :

$$F_0 = 0, \quad F_1 = 1 \quad \text{et} \quad \forall n \geqslant 2, \quad F_n = F_{n-1} + F_{n-2}$$

```python
def fibonacci(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)
```

Cette implémentation est **très inefficace**. Le problème vient du fait qu'il y a **deux appels récursifs** dans le même cas. Pour calculer `fibonacci(n)`, il faut calculer `fibonacci(n-1)` et `fibonacci(n-2)`. Or le calcul de `fibonacci(n-1)` lui-même nécessite `fibonacci(n-2)` et `fibonacci(n-3)` : `fibonacci(n-2)` est donc recalculé deux fois. Et ainsi de suite : chaque valeur intermédiaire est recalculée un grand nombre de fois.

L'arbre des appels pour `fibonacci(5)` illustre ce phénomène :



On voit que F_3 est calculé 2 fois, F_2 est calculé 3 fois, F_1 et F_0 encore plus souvent. Le nombre total d'appels croît exponentiellement avec n.

Règle importante

Faire deux appels récursifs (ou plus) dans le même cas conduit systématiquement à ce problème : les mêmes sous-problèmes sont résolus plusieurs fois. Pour éviter cela, il faut soit utiliser une approche itérative (boucle avec variables), soit utiliser l'approche de la **programmation dynamique** qui permet de stocker les résultats intermédiaires pour éviter les recalculs.

4) Puissance et exponentiation rapide

a) Fonction puissance récursive basique

Une première approche naïve : $a^n = a \times a^{n-1}$ avec $a^0 = 1$.

```python
def puissance(a, n):
    if n == 0:
        return 1
    else:
        return a * puissance(a, n - 1)
```

Cette fonction ne fait qu'**un seul** appel récursif, donc pas de recalcul : elle est correcte et efficace (complexité $O(n)$).

b) Exponentiation rapide

L'exponentiation rapide (ou exponentiation par carrés) exploite :

- ⋆ Si n est pair : $a^n = (a^{n/2})^2$
- ⋆ Si n est impair : $a^n = a \times a^{n-1}$

À chaque appel, l'exposant est environ divisé par 2. Le nombre d'opérations passe de $O(n)$ à $O(\log n)$.

```
def exp_rapide(a, n):
    if n == 0:
        return 1
    elif n % 2 == 0:
        t = exp_rapide(a, n // 2)
        return t * t
    else:
        return a * exp_rapide(a, n - 1)
```

Dans le cas n pair, il est **essentiel** de stocker le résultat de `exp_rapide(a, n // 2)` dans une variable `t` avant de faire `return t * t`.
En effet si l'on écrivait `return exp_rapide(a, n // 2) * exp_rapide(a, n // 2)`, on ferait **deux appels récursifs** identiques. Comme pour Fibonacci, cela conduirait à recalculer les mêmes valeurs plusieurs fois et l'algorithme perdrait tout son intérêt. En stockant le résultat dans `t`, on n'effectue qu'un seul appel et on réutilise le résultat.

Exemple : Comparaison pour 2^{10} :

```
>>> puissance(2, 10)
1024
>>> exp_rapide(2, 10)
1024
```

Les deux fonctions renvoient le même résultat, mais `exp_rapide` effectue environ 4 multiplications contre 10 pour `puissance`.

5) Dessin de fractales

Une fractale est une figure géométrique qui se répète à différentes échelles : chaque partie de la figure ressemble à l'ensemble. La récursivité est idéale pour générer de telles figures. Le module `turtle` de Python permet de dessiner à l'écran ; il est préinstallé avec Python.

a) Le flocon de Koch

La courbe de Koch est construite récursivement : à chaque niveau, chaque segment droit est remplacé par quatre segments formant une pointe. Le cas de base (niveau 0) est un segment droit ; au niveau n, on trace quatre courbes de Koch de niveau $n-1$ avec des rotations de 60.

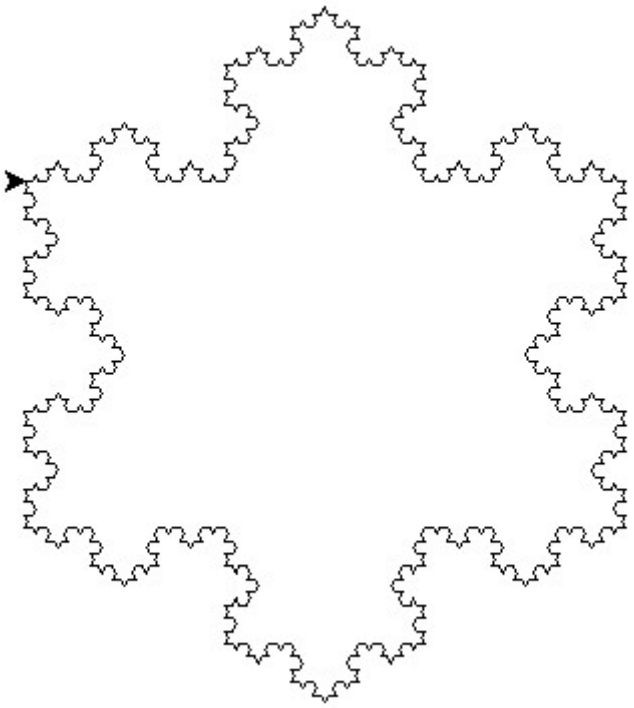
```python
import turtle

def koch(tortue, longueur, niveau):
    """
    Trace une courbe de Koch de niveau 'niveau' et de longueur de base 'longueur'.
    Entrées : tortue (objet turtle), longueur (float), niveau (int)
    """
    if niveau == 0:
        tortue.forward(longueur)
    else:
        koch(tortue, longueur / 3, niveau - 1)
        tortue.left(60)
        koch(tortue, longueur / 3, niveau - 1)
        tortue.right(120)
        koch(tortue, longueur / 3, niveau - 1)
        tortue.left(60)
        koch(tortue, longueur / 3, niveau - 1)

def flocon_koch(tortue, longueur, niveau):
    """ Trace un flocon de Koch (triangle formé de 3 courbes de Koch). """
    for _ in range(3):
        koch(tortue, longueur, niveau)
        tortue.right(120)
```

Exemple : Pour dessiner un flocon de Koch de niveau 3 :

```python
tortue = turtle.Turtle()
tortue.speed(0)
flocon_koch(tortue, 300, 4)
turtle.done()
```



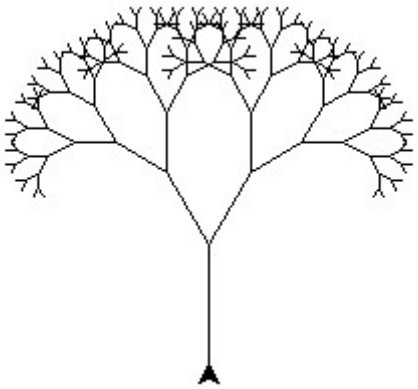
b) Arbre récursif

Un arbre fractal simple : un tronc avec deux branches qui sont chacune des arbres plus petits, inclinés par rapport au tronc.

```python
def arbre(tortue, longueur, angle, niveau):
    """
    Trace un arbre fractal récursif.
    Entrées : tortue, longueur du segment, angle entre les branches, niveau de récursion
    """
    if niveau == 0:
        return
    tortue.forward(longueur)
    tortue.left(angle)
    arbre(tortue, longueur * 0.7, angle, niveau - 1)
    tortue.right(2 * angle)
    arbre(tortue, longueur * 0.7, angle, niveau - 1)
    tortue.left(angle)
    tortue.backward(longueur)
```

Exemple : Pour dessiner un arbre fractal de niveau 6 avec un angle de 30 :

```python
tortue = turtle.Turtle()
tortue.left(90)
tortue.speed(0)
arbre(tortue, 60, 30, 6)
turtle.done()
```



Remarque : Pour le dessin de fractales, le **cas de base** correspond au niveau 0 : soit on ne dessine rien (arbre), soit on trace un segment simple (Koch). Le **cas récursif** réduit la taille et le niveau pour obtenir une figure auto-similaire.

IV. Erreurs à éviter

1) Récursion infinie

Si le cas de base n'est jamais atteint, la fonction s'appelle indéfiniment. Python interrompt alors l'exécution avec une erreur du type `RecursionError` ou `Maximum recursion depth exceeded`.

Exemple :

⋆ **Oubli du cas de base :**

```
def factorielle_erreur(n):
    return n * factorielle_erreur(n - 1)   # Pas de cas n == 0 !
```

⋆ **Cas de base inaccessible :**

```
def mauvaise(n):
    if n < 0:
        return 0
    return 1 + mauvaise(n + 1)    # n+1 ne se rapproche jamais de n < 0
```

2) Cas de base mal placé

Le cas de base doit être testé **avant** l'appel récursif. Sinon, l'appel récursif a lieu dans tous les cas et le programme ne peut pas terminer.

Bonne pratique

Dans une fonction récursive, on teste **toujours** le cas de base en premier, généralement avec une instruction `if` au début du corps de la fonction.

V. Récursif ou itératif ?

Une boucle (`for` ou `while`) permet de réaliser les mêmes calculs que beaucoup de fonctions récursives. On parle alors d'approche itérative.

Exemple : Comparaison pour la factorielle :

Récursif :

```
def facto_rec(n):
    if n == 0:
        return 1
    return n * facto_rec(n - 1)
```

Itératif :

```
def facto_iter(n):
    p = 1
    for k in range(1, n + 1):
        p = p * k
    return p
```

Quand utiliser la récursivité ?

⋆ **Avantages de la récursivité :** le code est souvent plus court et plus lisible lorsque le problème se décompose naturellement (suites récurrentes, structures arborescentes). De plus, on peut utiliser le principe « **diviser pour régner** »comme dans l'algorithme d'exponentiation rapide ou dans l'algorithme de tri-rapide.

⋆ **Inconvénients :** chaque appel récursif consomme de la mémoire (pile d'appels) ; pour des profondeurs importantes, l'approche itérative est préférable.