La récursivité

La récursivité consiste à définir ou résoudre un problème en le ramenant à une version plus simple de lui-même. Une fonction récursive se caractérise par deux éléments essentiels : un **cas de base**, qui permet d'arrêter le calcul, et un **appel récursif**, qui traite un problème de taille plus petite. Sans cas de base, l'exécution ne peut pas se terminer.

Pour comprendre cette idée, on peut utiliser l'exemple de la **poupée russe**. Une poupée russe est soit la plus petite poupée, qui ne contient rien, soit une poupée qui contient une autre poupée russe. Cette définition fait directement apparaître la structure récursive : chaque poupée contient une version plus petite d'elle-même, jusqu'à atteindre la dernière.

Figure 1. Poupée russe

Ce principe se retrouve dans la définition des **listes en Lisp**. Une liste est soit vide (nil), soit constituée d'un élément et d'une autre liste. En Python, on peut modéliser cette définition à l'aide de **tuples** : la liste vide est représentée par None, et une liste non vide par un couple (valeur, reste). Cette représentation met en évidence la nature récursive des listes et permet d'écrire naturellement des fonctions récursives pour les parcourir ou les traiter.

Étape 1 — Donner la définition abstraite (sans code long)

Commençons par rappeler la définition :

Une liste est soit vide, soit constituée d'un élément et d'une autre liste.

- « autre liste » → **définition récursive**
- « soit vide » → **cas de base**

Étape 2 — Définir la liste vide (nil)

definissons explicitement la fonction :

```
def creer_liste():
    return None
```

- None représente la **liste vide**
- c'est le **cas de base** de la récursivité

Étape 3 — Définir le constructeur cons

```
def cons(x, L):
    return (x, L)
```

- cons **construit une liste**
- il ne parcourt pas la liste
- L doit déjà être une liste (ou None)

➡️ cons () **n'est pas récursif**, mais il **construit une structure récursive**.

Étape 4 — Construire une liste à la main (moment clé)

Il faut **absolument** faire :

L = cons(1, cons(2, cons(3, creer_liste())))

Étape 5 — Décomposer au tableau (très important)

Montrons progressivement :

cons(3, None)

cons(2, (3, None))

cons(1, (2, (3, None)))

Puis la représentation finale :

(1, (2, (3, None)))

Faisons le lien direct avec la poupée russe :

- chaque tuple contient une valeur
- et une autre « poupée » à l'intérieur

Étape 6 — Faire formuler la structure

Question à poser :

Est-ce qu'une liste contient une liste ?

Réponse attendue :

Oui, sauf la liste vide.

➡️ Vous avez gagné la récursivité.

Exercice :

Créer un fonction longueur qui permet de retourner un entier qui représente le nombre d'élèments dans une Liste chainée de deux manières : itérative et récursive.

Version itérative

```
def longueur_iter(L):
    c = 0
    while L is not None:  # tant que la liste n'est pas vide
        c += 1
        L = L[1]          # on passe au reste de la liste
    return c
```

Caractéristiques

- Utilise **une boucle while** pour parcourir la liste
- La variable L **change à chaque tour**
- La variable c **compte les éléments**
- Arrêt de la boucle → **cas de base implicite**

2. Version récursive

```
def longueur(L):
    if L is None:          # cas de base explicite
        return 0
    return 1 + longueur(L[1])  # appel récursif
```

Caractéristiques

- Pas de boucle
- La fonction **s'appelle elle-même** sur le reste de la liste (cdr)
- Le cas de base est explicite (L is None)
- Le « compteur » est **calculé automatiquement par la pile d'appels**

3. Tableau comparatif simple

Point	Itératif	Récursif
Parcours	Boucle while	Appels successifs
Compteur	Variable c	Calcul via 1 + …
Cas de base	Implémenté dans la condition de sortie de la boucle	Condition if L is None explicite
Lisibilité	Peut être plus « mécanique »	Suit la structure naturelle de la liste chaînée
Concept	Modifie l'état	Décompose le problème en sous-problèmes

Lien avec la poupée russe

- **Itératif** : vous ouvrez les poupées une par une, en comptant à chaque étape
- **Récursif** : chaque poupée « sait » qu'elle contient d'autres poupées et retourne 1 + ce que contiennent les poupées suivantes

A retenir

« L'itératif utilise une boucle et un compteur, la récursivité utilise la structure de la liste elle-même et la pile d'appels pour compter les éléments. »

Illustration rapide Pour la fonction récursive :

Pour longueur((1, (2, (3, None)))) :

longueur(L) attend 1 + longueur(L[1])

longueur(L[1]) attend 1 + longueur(L[1][1])

longueur(L[1][1]) attend 1 + longueur(None)

longueur(None) retourne 0

Puis :

0 → 1 → 2 → 3

Phrase clé à faire retenir : Une fonction récursive s'arrête quand elle atteint son cas de base ; avant cela, chaque appel attend le résultat de l'appel suivant.

👉**Toute fonction récursive peut être transformée en une version itérative**, 👉 **mais toute fonction itérative n'est pas naturellement récursive**.

Tout algorithme récursif peut être réécrit de manière itérative, mais l'inverse n'est pas toujours pertinent ni naturel.

COURS & EXERCICES – LA RÉCURSIVITÉ EN PYTHON

PAGE 1 — ÉNONCÉS DES EXERCICES (SANS CORRECTION)

Activité 2 – La factorielle

La factorielle est un exemple classique permettant de comprendre la récursivité, car sa définition mathématique est naturellement récursive.

En mathématiques, la factorielle d'un entier naturel n, notée $n!$, est définie comme le produit de tous les entiers strictement positifs inférieurs ou égaux à n.

Exemples :

- $3! = 3 \times 2 \times 1 = 6$
- $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$

On admet la définition suivante :

- $0! = 1$(par convention)
- $n! = n \times (n-1)!$si $n > 0$

Question 1 – Cas de base et cas récursif

Compléter la définition suivante en identifiant clairement :

- le **cas de base**
- le **cas récursif**

factorielle(n) =

… si n = 0

… si n > 0

Question 2 – Approche itérative

Écrire une fonction Python fact_iter(n) qui calcule la factorielle de n **sans utiliser la récursivité**, uniquement à l'aide d'une boucle.

Question 3 – Approche récursive

Écrire une fonction récursive factorielle(n) qui calcule la factorielle de n en utilisant la définition mathématique précédente.

Question 4 – Déroulement d'un appel récursif

Dérouler l'exécution de factorielle(4) en montrant :

- les appels successifs de la fonction,
- le moment où le cas de base est atteint,
- les valeurs retournées à chaque étape.

Question 5 – Limites de la récursivité

Calculer la factorielle de 3125 :

- avec la version récursive,
- puis avec la version itérative.

Que constatez-vous ?
Expliquez pourquoi ce comportement apparaît en Python.

Activité 3 – Pair ou impair ?

On souhaite écrire une fonction est_pair(n) qui renvoie :

- True si l'entier naturel n est pair,
- False sinon.

On rappelle les propriétés suivantes :

- 0 est un nombre pair.
- Un entier naturel $n > 0$ est pair si et seulement si $n - 1$ ne l'est pas.

Question 1 – Définition récursive

À partir des propriétés précédentes, proposer :

- le cas de base,
- le cas récursif

pour la fonction est_pair(n).

Question 2 – Écriture de la fonction

Écrire une fonction récursive est_pair(n) en Python.

Question 3 – Déroulement

Dérouler l'exécution de est_pair(3) en montrant les appels récursifs successifs.

Activité 4 – Suite de Fibonacci

La suite de Fibonacci est une suite de nombres dans laquelle chaque terme est la somme des deux précédents.

Elle est définie par :

- $F_0 = 1$
- $F_1 = 1$
- $F_n = F_{n-1} + F_{n-2}$pour $n \geq 2$

Question 1 – Fonction récursive

Écrire une fonction récursive fibo(n) qui renvoie le terme de rang n de la suite de Fibonacci.

Attention : cette fonction comporte **deux cas de base**.

Question 2 – Déroulement

Dérouler l'exécution de fibo(4) en détaillant tous les appels récursifs générés.

Activité 5 – Somme des éléments d'un tableau

On souhaite calculer la somme des éléments d'un tableau d'entiers.

Question 1 – Approche récursive

Écrire une fonction récursive somme(T) qui renvoie la somme des éléments du tableau T.

Indication :

- Le cas de base correspond à un tableau vide.
- Le cas récursif consiste à additionner le premier élément avec la somme du reste du tableau.

PAGE 2 — CORRECTIONS

Activité 2 – Correction

Question 1

factorielle(n) =

1 si n = 0

n × factorielle(n − 1) si n > 0

Question 2 – Version itérative

```
def fact_iter(n):
    resultat = 1
    for i in range(1, n + 1):
        resultat *= i
    return resultat
```

Question 3 – Version récursive

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

Question 4 – Déroulement de factorielle(4)

factorielle(4)

→ 4 × factorielle(3)

→ 4 × (3 × factorielle(2))

→ 4 × (3 × (2 × factorielle(1)))

→ 4 × (3 × (2 × (1 × factorielle(0))))

→ 4 × 3 × 2 × 1 × 1

→ 24

Question 5 – Limites

La version récursive provoque une erreur de type RecursionError, car Python limite le nombre d'appels récursifs imbriqués.
La version itérative ne rencontre pas cette limite.

Activité 3 – Correction

```
def est_pair(n):
    if n == 0:
        return True
    return not est_pair(n - 1)
```

Activité 4 – Correction

```
def fibo(n):
    if n == 0 or n == 1:
        return 1
    return fibo(n - 1) + fibo(n - 2)
```

Activité 5 – Correction

```
def somme(T):
    if T == []:
        return 0
    return T[0] + somme(T[1:])
```

Conclusion pédagogique

Une solution itérative repose sur des boucles.
Une solution récursive repose sur la définition du problème.
La récursivité est élégante mais doit toujours comporter un **cas de base clair**.