Cours sur la récursivité en Python

Plan du Cours :

1. **Introduction à la récursivité (20 min)**
    - Qu'est-ce que la récursivité ?
    - Structure d'une fonction récursive
    - Cas de base et cas récursif
    - Avantages et inconvénients de la récursivité
2. **Exemple pratique 1 : Approche inspirée des jeux vidéo (40 min)**
    - Résolution d'un labyrinthe avec la récursivité
    - Implémentation d'une fonction pour explorer un labyrinthe
    - Utilisation de la récursivité pour trouver la sortie
3. **Exemple pratique 2 : Approche inspirée de la vie réelle (40 min)**
    - Modélisation de l'exploration d'une structure arborescente (arbre généalogique)
    - Calcul de la profondeur d'une structure
    - Utilisation de la récursivité pour résoudre un problème réel
4. **Conclusion et récapitulation (20 min)**

1. Introduction à la récursivité (20 min)

Qu'est-ce que la récursivité ?

La **récursivité** est une méthode en programmation où une **fonction s'appelle elle-même** pour résoudre un problème. Cette technique permet de diviser un problème complexe en **sous-problèmes plus petits** et plus simples à résoudre. La récursivité est particulièrement utile pour traiter des structures de données hiérarchiques (comme les arbres ou les graphes) ou des problèmes qui peuvent être décomposés en sous-parties similaires.

Concept clé : Décomposer un problème en sous-problèmes

La récursivité est basée sur un principe mathématique appelé **récurrence**, qui stipule que si vous pouvez résoudre un petit cas particulier d'un problème, et que vous savez comment réduire un problème à une version plus simple de lui-même, alors vous pouvez utiliser cette idée pour résoudre le problème entier.

Structure d'une fonction récursive

La fonction récursive suit généralement deux parties essentielles :

1. **Cas de base (condition d'arrêt) :** C'est un cas particulier où la fonction ne s'appelle plus elle-même. Cela met fin à la récursivité. Un bon exemple de cas de base serait une situation où le problème est suffisamment simple pour être résolu directement sans besoin de décomposer davantage.

2. **Cas récursif :** Le cas récursif est l'endroit où la fonction s'appelle elle-même, mais avec un **problème plus simple** ou **plus petit** qu'elle traite à chaque étape. Cette simplification continue jusqu'à ce qu'elle atteigne le cas de base, où elle arrête de s'appeler.

Exemple simple : Calcul de la factorielle

La factorielle d'un nombre `n` est le produit de tous les entiers positifs inférieurs ou égaux à `n`. Mathématiquement, la factorielle de `n` (notée `n!`) est définie comme :

- `n! = n * (n - 1) * (n - 2) * ... * 1`
- Et par convention, `0! = 1`

La récursivité est particulièrement bien adaptée pour le calcul de la factorielle car la définition de la factorielle est elle-même récursive : la factorielle d'un nombre `n` est `n` multiplié par la factorielle de `n - 1`.

Fonction récursive pour calculer la factorielle :

Voici comment vous pourriez écrire une fonction récursive en Python pour calculer la factorielle d'un nombre `n` :

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

Explication détaillée du code :

1. **Cas de base :**

   ```
   if n == 0:
       return 1
   ```

   Lorsque `n` est égal à 0, la fonction retourne directement 1. Ceci est notre **cas de base**, car nous connaissons la valeur exacte de `0!`, et à partir de là, nous n'avons plus besoin d'appeler la fonction récursivement.

2. **Cas récursif :**

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

   Si `n` n'est pas égal à 0, nous appelons la fonction `factorielle` avec `n - 1`. Cela continue de réduire la valeur de `n` jusqu'à atteindre le cas de base. Chaque appel de fonction attend que l'appel suivant renvoie sa valeur avant de multiplier `n` par le résultat.

Illustration du processus de récursion :

Prenons l'exemple du calcul de `factorielle(3)` :

- **Premier appel :** `factorielle(3)`
    - Cas récursif : `3 * factorielle(2)`
- **Deuxième appel :** `factorielle(2)`
    - Cas récursif : `2 * factorielle(1)`
- **Troisième appel :** `factorielle(1)`
    - Cas récursif : `1 * factorielle(0)`
- **Quatrième appel :** `factorielle(0)`
    - Cas de base : retourne 1

Maintenant, la récursivité remonte en cascade en multipliant les résultats à chaque étape :

- `factorielle(1)` retourne `1 * 1 = 1`
- `factorielle(2)` retourne `2 * 1 = 2`
- `factorielle(3)` retourne `3 * 2 = 6`

Donc, `factorielle(3)` retourne **6**.

Schéma visuel du processus de récursion :

```
factorielle(3)
↳ 3 * factorielle(2)
    ↳ 2 * factorielle(1)
        ↳ 1 * factorielle(0)
            ↳ 1
```

Chaque appel récursif empile les calculs jusqu'à ce que le **cas de base** soit atteint, puis les résultats sont retournés dans l'ordre inverse, terminant ainsi la récursion.

Avantages et inconvénients de la récursivité

Avantages :

- **Simplicité et lisibilité :** La récursivité permet souvent de résoudre des problèmes complexes avec des solutions simples et élégantes.
- **Approprié pour certaines structures de données :** Les problèmes qui ont une structure hiérarchique, comme les arbres ou les graphes, sont naturellement adaptés à une solution récursive.

Inconvénients :

- **Problèmes de performances :** Chaque appel de fonction récursive ajoute une nouvelle entrée dans la pile d'exécution, ce qui peut entraîner un dépassement de la pile (stack overflow) pour des problèmes de grande taille.
- **Répétition de calculs :** Si la récursivité n'est pas optimisée (par exemple, avec une technique de mémoïsation), elle peut effectuer des calculs redondants, ce qui est inefficace.

Récursivité vs Boucle :

Prenons l'exemple du calcul de la factorielle. Voici la même logique, mais implémentée en utilisant une boucle :

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

Cette solution non récursive utilise une **boucle** pour effectuer le même calcul. Le choix entre récursion et boucle dépend du contexte du problème à résoudre.

Prochaine étape : Résolution pratique avec la récursivité

Dans les deux approches pratiques qui suivent, nous allons :

1. Explorer l'utilisation de la récursivité pour résoudre des problèmes **dans un contexte de jeux vidéo**, comme explorer un labyrinthe.
2. Appliquer la récursivité à un problème **inspiré de la vie réelle**, tel que l'exploration d'un arbre généalogique.