Chapitre 6 - Récursivité

I - Introduction

La résolution de problèmes en programmation peut être abordée de différentes manières, avec deux approches principales : la récursivité et l'itérativité. La récursivité implique que la fonction s'appelle elle-même pour diviser le problème en des sous-problèmes plus simples, tandis que l'itérativité utilise des boucles pour répéter un ensemble d'instructions jusqu'à ce qu'une condition soit satisfaite. Ces deux principes, bien que distincts, partagent l'objectif fondamental de résoudre des problèmes de manière efficace. La récursivité offre souvent une solution élégante et intuitive, tandis que l'itérativité privilégie la clarté et la performance en évitant les coûts liés aux appels récursifs.
Dans ce cours, nous allons explorer les bases de la récursivité, ses principes et comment l'appliquer en Python.

II - Définition de la récursivité

Définition 1

La récursivité est un concept en informatique où une fonction résout un problème en se divisant elle-même en des instances plus simples du même problème.
Autrement dit, la fonction utilise sa propre définition pour résoudre le problème en le décomposant en des sous-problèmes de même nature. La fonction s'appelle elle-même.

III - Caractéristiques de la récursivité

La récursivité est caractérisée par certaines propriétés et principes qui la distinguent comme une technique particulière de résolution de problèmes. Explorons ces caractéristiques essentielles :

1. **Cas de base (ou état trivial) :**
   Chaque fonction récursive doit avoir un cas de base (un état trivial) qui spécifie la condition où la récursion doit s'arrêter. C'est une condition simple qui n'utilise pas l'appel récursif et permet à la fonction de renvoyer directement une valeur sans poursuivre la récursion. Il est crucial d'avoir un cas de base (un état trivial) pour éviter une boucle infinie.
2. **Appel récursif :**
   La fonction doit s'appeler elle-même avec des arguments modifiés de manière à réduire le problème vers le cas de base.
   L'appel récursif crée une séquence d'appels emboîtés qui converge vers la solution du problème initial en résolvant des versions de plus en plus petites du problème.
3. **Réduction de problème :**
   Chaque appel récursif doit contribuer à résoudre une version plus petite du problème initial. Cela garantit que chaque niveau de récursion travaille sur une portion du problème, contribuant ainsi à la résolution globale.
4. **Retour de valeur :**
   Chaque appel récursif doit renvoyer une valeur qui contribue à la résolution du problème. Ces valeurs contribuent à mesure que la récursion se déroule, et la valeur finale retournée par la fonction récursive est généralement une combinaison des valeurs renvoyées par les différents appels récursifs.

5. **Empilement des appels et convergence :**
   Chaque appel récursif crée une nouvelle instance de la fonction dans la pile d'appels. L'empilement des appels récursifs se produit jusqu'à ce que le cas de base soit atteint. Ensuite, les valeurs sont déployées à mesure que les fonctions se défont de la pile d'appels.

IV - Exemple de fonction récursive

Considérons la fonction récursive qui calcule la somme des N premiers entiers positifs.
Cette fonction s'appelle elle-même avec des arguments modifiés pour réduire le problème à une taille plus petite jusqu'à ce qu'elle atteigne le cas de base.

```
def somme_entiers(n):
        if n == 0:
                return 0  #Cas de base ou etat trivial
        else:
                return n + somme_entiers(n-1)  #appel recursif
```

1) Analyse de la fonction

1. Cas de base (état trivial) :
   Lorsque n devient 0, la fonction retourne directement 0. C'est le cas de base qui arrête la récursion.
2. Appel récursif :
   L'appel récursif est effectué avec *somme_entier(n-1)*, réduisant le problème à la somme des $N-1$ premiers entiers.
3. Réduction de problème :
   À chaque appel récursif, le problème est réduit à une taille plus petite. La somme des N premiers entiers est construite en ajoutant n à la somme des N-1 premiers entiers.
4. Retour de valeur :
   Chaque appel récursif retourne une valeur qui contribue à la somme totale. Ces valeurs sont combinées lors du déroulement de la pile d'appels.

2) Illustration du processus récursif

1. **Appel initial :**

   somme_entiers(5)
   Retourne 5 + somme_entiers(4)
   Retourne 4 + somme_entiers(3)
   Retourne 3 + somme_entiers(2)
   Retourne 2 + somme_entiers(1)
   Retourne 1 + somme_entiers(0)
   Cas de base atteint, retourne 0.

2. **Déploiement des valeurs :**

   (a) À partir du cas de base, les valeurs sont déployées à chaque niveau de la pile d'appels, **en remontant du cas de base vers l'appel initial**.

   (b) La somme totale est construite en additionnant ces valeurs déployées.

3) Exécution de l'exemple

```
#Test de la fonction somme_entiers
s = somme_entiers(5)
print("Somme des 5 premiers entiers : ",s)
#OUTPUT : Somme des 5 premiers entiers : 15
```

4) Version itérative

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

V - Avantages et inconvénients de la récursivité

La récursivité est une technique puissante, mais elle présente à la fois des avantages et des inconvénients qu'il est important de considérer lors de sa mise en œuvre.

1) Avantages

1. **Élégance et clarté**
   La récursivité peut conduire à un code élégant et lisible, en particulier pour des problèmes complexes qui peuvent être naturellement décomposés en sous-problèmes.
2. **Simplicité conceptuelle**
   Elle simplifie la conception de l'algorithme en se concentrant sur la solution du problème dans son ensemble plutôt que sur les détails d'itérations.
3. **Modularité**
   La récursivité permet la division du problème en sous-problèmes indépendants, favorisant la modularité du code.
4. **Adaptabilité aux structures récursives**
   Elle est bien adaptée pour traiter des structures de données récursives comme les arbres et les listes chaînées.

2) Inconvénients

1. **Coût en termes de performance**
   Les appels récursifs peuvent être moins efficaces en termes de temps et d'espace par rapport aux solutions itératives. Chaque appel récursif ajoute une nouvelle couche à la pile d'appels, ce qui peut entraîner un dépassement de pile.
2. **Difficulté de débogage**
   Les erreurs récursives peuvent être difficiles à déboguer en raison de la nature imbriquée des appels. Il peut être complexe de suivre le flux d'exécution.
3. **Débordement de pile**
   Si la récursion est excessive et la pile d'appels devient trop profonde, cela peut entraîner un dépassement de pile, provoquant une erreur.

4. **Possibilité de répétition de calculs**
   Certains problèmes récursifs peuvent conduire à des calculs répétitifs, ce qui peut être évité avec des approches itératives.

VI - Comparaison itératif vs récursif

La différence principale entre les approches itérative et récursive réside dans la manière dont elles résolvent un problème. Voici une comparaison des caractéristiques fondamentales des approches itérative et récursive :

1. **Définition**
   (a) Itératif : Utilise des boucles pour répéter un ensemble d'instructions jusqu'à ce qu'une condition soit satisfaite.
   (b) Récursif : Une fonction s'appelle elle-même pour résoudre un problème en le divisant en des sous-problèmes plus simples.
2. **Structure de contrôle**
   (a) Itératif : Utilise des structures de contrôle de boucle telles que for et while.
   (b) Récursif : Utilise des appels de fonction récursifs.
3. **Expression du code**
   (a) Itératif : Utilise des instructions répétitives explicites et peut être plus verbeux.
   (b) Récursif : Le code peut être plus court et plus lisible pour certains problèmes.
4. **Gestion de la pile d'appels**
   (a) Itératif : Utilise une seule entrée dans la pile d'appels.
   (b) Récursif : Chaque appel récursif ajoute une nouvelle couche à la pile d'appels.
5. **Cas de base**
   (a) Itératif : Utilise des conditions pour définir quand la boucle doit s'arrêter.
   (b) Récursif : Doit inclure un cas de base qui spécifie quand la récursion doit se terminer.
6. **Performance**
   (a) Itératif : Généralement plus performant en termes de temps et d'espace, en évitant les coûts associés aux appels récursifs.
   (b) Récursif : Peut entraîner des coûts supplémentaires, en particulier pour des problèmes récursifs profonds.
7. **Adaptabilité aux structures de données**
   (a) Itératif : Convient bien pour traiter des structures de données linéaires (tableaux...).
   (b) Récursif : Excellente pour traiter des structures de données récursives telles que les arbres et les listes chaînées.
8. **Compréhension et élégance du code**
   (a) Itératif : Peut être moins intuitif pour certains problèmes complexes, mais offre une lisibilité claire.
   (b) Récursif : Peut offrir une solution plus élégante et intuitive pour des problèmes naturellement récursifs.