CPGE Oujda

SPE

Récursivité en Langage python

Introduction

La notion de récursivité est avant tout un problème algorithmique plus qu'au niveau du langage lui même. Que ce soit en C, C++, Java, VB, Python, etc.., l'implémentation d'une fonction récursive se fera toujours plus ou moins de la même manière. Ici nous allons traiter de la récursivité avec le Langage python. Mais qu'est-ce que la récursivité ? il s'agit de programmes ou de fonctions d'un programme qui ont la faculté de s'appeler eux-mêmes (on entend également le terme *d'auto-appel* ce qui est logique). La récursivité est une manière simple et élégante de résoudre certains problèmes algorithmiques, notamment en mathématique mais cela ne s'improvise pas, il convient donc de savoir comment ce principe fonctionne.

Fonctions récursives

Les fonctions récursives comme cité plus haut, sont donc des fonctions s'appelant elles-mêmes, elles sont également un moyen rapide pour poser certains problèmes algorithmique ; nous allons voir en détails comment elles fonctionnent.
Prenons un problème simple mais auquel vous n'avez peut-être pas pensé à utiliser la récursivité: le calcul d'une factorielle. Considérons *n!* (qui se lit: factorielle de n) comme étant la factorielle à calculer, nous aurons ceci: *6! = 6x5x4x3x2x1*. Dans cette situation, nous pouvons déjà déterminer notre règle de sortie de notre fonction récursive: la valeur *1* qui symbolise la fin de la récursion !
En effet, il faut une condition de sortie pour la fonction, mais il faut être très vigilant quant au choix de la condition, vous devrez être sûr qu'elle soit validée à un moment ou à un autre sinon c'est comme si vous créez une boucle infinie sans condition de sortie !
La règle de récursion que nous devons définir, est le calcul de la factorielle en elle même soit, si nous considérons notre exemple sur *6!*, cité précédemment, nous pouvons définir notre règle de cette manière: *n! = (n) (n-1) (n-2) ... (1)*. Nous pouvons en déduire que nous allons faire des appels en décrémentant la valeur de *n* à chaque appel de la fonction jusqu'à ce que *n == 1* !
nous avons ce qu'il nous faut pour créer notre fonction récursive, la voici:

Fonction récursive simple

```
def  factoriel ( n)
     if ( n == 0) :
          return 1
      else:
     return n * factoriel (n - 1);
```

Nous pouvons observer ici que le dernier **return** est en fait l'appel récursif et nous soustrayons *1* à chaque appel jusqu'à ce que *n == 1* qui est, comme décrit plus haut, notre condition de sortie.
Non, cela ne s'arrête pas là et c'est ici que nous allons voir le fonctionnement des fonctions récursives. Lorsque la fonction rencontre la condition de sortie, elle remonte dans tous les appels précédents pour

calculer *n* avec la valeur précédemment trouvée !
Les appels des fonctions récursives sont en fait *empilées* (pile qui est une structure de donnée régie selon le mode *LIFO*: Last In First Out, Dernier Entré Premier Sorti). Chaque appel se trouve donc l'un à la suite de l'autre dans la pile du programme. Une fonction de ce type possède donc deux parcours: la *phase de descente* et la *phase de remontée*.

Nous voyons très bien la phase de descente et de remontée dans la pile des appels de la fonction récursive. Ce n'est qu'au moment de la remontée, donc également au moment où la condition de sortie est vraie, que les appels enregistrés sont dépilés au fur et à mesure de la remontée.

Dangers et précautions

Les fonctions récursives étant un moyen assez puissant pour résoudre certains problèmes de façon élégante, elles n'en restent pas moins dangereuses et ce pour plusieurs raisons.

IV-A. Dépassement de capacité

Une des causes assez fréquente quand vous travaillez sur de très grands nombres, est le dépassement de capacité. C'est un phénomène qui se produit lorsque vous essayez de stocker un nombre plus grand que ce que peut contenir le type de votre variable.
Il est d'usage de choisir un type approprié, même si vous êtes certains que le type que vous avez choisi ne sera jamais dépassé, utilisez tant que possible une variable pouvant contenir de plus grandes données. Ceci s'applique à tous types de données.

IV-B. Débordement de pile (Stack Overflow)

Ceci est sans doute une des causes les plus souvent rencontrés dans le plantage de programmes avec des fonctions récursives. Nous savons que les appels récursifs de fonctions sont placés dans la pile du programme, pile qui est d'une taille assez limité car elle est fixée une fois pour toutes lors de la compilation du programme.

Dans la pile sont non seulement stockés les valeurs des variables de retour mais aussi les adresses des fonctions entre autres choses, les données sont nombreuses et un débordement de la pile peut très vite arriver ce qui provoque sans conteste une sortie anormale du programme.
Une autre méthode existe cependant ! Si vous êtes presque sûr de dépasser ce genre de limites, préférez alors une approche itérative plutôt qu'une approche récursive du problème. Une approche récursive demande beaucoup de moyens en ressources car énormément de données doivent êtres stockées dans la pile d'exécution alors qu'en revanche, une approche itérative telle une boucle **for** est bien moins coûteuse en terme de ressources et est bien plus sûre, sauf dans le cas d'un dépassement de capacité bien sûr !

IV-C. Forme itérative

Voyons en vitesse une possibilité de forme itérative pour notre calcul de *6!* :

Forme itérative

```
def factoriel_iterative ( n):
    ret = 1;
    i = 1;
   for  i in range(1,n+1):
      ret *= i;
   return ret;
```

Exercices :

Ecrire les fonctions récursives suivantes :

1) calcul de s=0+2+4+….+N (N est un entier positif)
2) somme (M,N) (M et N sont des entiers positifs)
3) calcul de X^N (X reel et N entier)
4) Calcul du PGCD(A,B)
5) Conversion d'un nombre décimal en binaire
6) Lecture d'un tableau
7) Recherche dichotomique dans une liste L
8) Tri rapide (quick sort) (voir la question dans la partie cours du tri rapide)
9) **Problème de Hanoï :**

   Le jeu est constitué d'une plaquette de bois où sont plantées trois tiges numérotées 1,2 et 3. Sur ces tiges sont empilés des disques de diamètres tous différents. Les seules règles du jeu sont que l'on ne peut déplacer qu'un seul disque à la fois, et qu'il est interdit de poser un disque sur un disque plus petit.
   Au début, tous les disques sont sur la tige 1 (celle de gauche), et à la fin ils doivent être sur celle de droite.
10) écrire une fonction r_ecursive estDans(t,x, k) qui retourne True si x apparait dans le tableau t _a partir de l'indice k, False sinon.

Solutions

5)
```
def bin (n):
    if n > 1 :
          bin (n//2)
          print n%2
```

6)
```
def saisie ( t, n):
    if ( n>0) :
          saisie ( t , n-1 )
```

```
        x=int(input("x=?"))
        t.append(x)

t=[]
saisie(t,4)
print(t)
```

6)

```
def rechercheDichotomique( t, x):
    g = 0
    d = N-1
      while(g <= d):
       m = (g+d)/2
       if(t[m] == x):
          return m;
       elif(t[m] < x):
          d = m-1;
       else:
          g = m+1;
    return -1;
```

Notons que l'on peut écrire cette fonction sous forme récursive :

```
// recherche de x dans t[g..d[
def dichoRec( t, x, g, d) :
    if(g >= d): // l'intervalle est vide
       return -1;
    m = (g+d)/2;
    if(t[m] == x):
       return m;
    elif(t[m] > x):
       return dichoRec(t, x, g, m)
    else:
       return dichoRec(t, x, m+1, d)


def int rechercheDicho( t, x):
    return dichoRec(t, x, 0, len(t));
}
```

8) **Algorithme :**

```
Procédure   Hanoi(n, départ, intermédiaire, destination)
        Si n > 0 alors
                Hanoi(n-1,départ, destination, intermédiaire)
                Déplacer un disque de départ vers destination
                Hanoi(n-1,intermédiaire,départ,destination)
```

Fin Si

Fin

10)

```
def estDans ( t , x , k ) :
        i f k > l en ( t ) □ 1 :
                return Fal s e
        i f t [ k ] == x :
                return True
        return estDans ( t , x , k+1)
```