Récursivité

Table des matières

I) Introduction

La **récursivité** est une méthode de programmation très efficace ayant des utilisations dans tous les domaines possibles. Le principe de ce cours est de saisir l'idée qui se cache derrière l'utilisation de la récursivité. Les applications seront faites en majorité en relation avec d'autres chapitres comme les arbres binaires, les tris par insertion, sélection, et par fusion.

II) Application

A) principe

Une fonction dite **récursive** est une fonction qui **s'appelle elle-même** un certain nombre de fois. En voici un exemple :

```
def recursif(n) :
    return recursif(n-1)
```

Cette fonction lorsqu'appellée produit une erreur. En effet, il n'y a rien qui permet d'arrêter la fonction, elle s'appellera donc elle-même à l'**infini**. Pour remédier à cela, on utilise un **cas de base**. (Ou **condition initiale**). Le principe du cas de base est qu'on va fixer une condition qui va arrêter les appels récursifs. (Les appels de la fonction à elle-même)

Exemple :

```
def recursif(n,i=0) :
    if n==0 :
        return (n,i)
    return recursif(n-1,i+1)
```

On repère ici qu'à chaque appel de la fonction, la valeur de n baisse de 1. De plus, la valeur de i augmente de 1 dans l'appel suivant si on entre i dans l'appel récursif. (i ne prendra pas la valeur 0 à chaque fois). En clair, la valeur de i indiquera le **nombre d'appels récursifs qui ont été nécessaires** pour que la fonction s'arrête.
Attention : ce genre de conditions sont à prendre avec des pincettes : Si on entre un nombre à virgule ou un négatif en tant que paramètres, n ne prendra jamais la valeur 0 et la fonction fera une erreur.

B) Exemple

La **factorielle** d'un nombre entier naturel (entier positif) est le **produit de tous les entiers qui le précèdent** . On note la factorielle de n : **n!**

En clair, n!=1 x 2 x 3 x 4 x … x (n-1) x (n)

Par exemple, 8!=1 x 2 x 3 x 4 x 5 x 6 x 7 x 8

On peut faire un programme le calculant sans récursivité :

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

ou **avec** la récursivité :

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

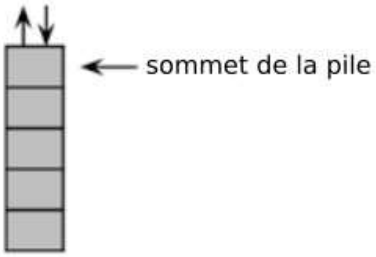
Ici l'algorithme n'est plus court que d'une ligne mais il peut dans certains cas sauver des **dizaines** de lignes de code.

C) approfondissement

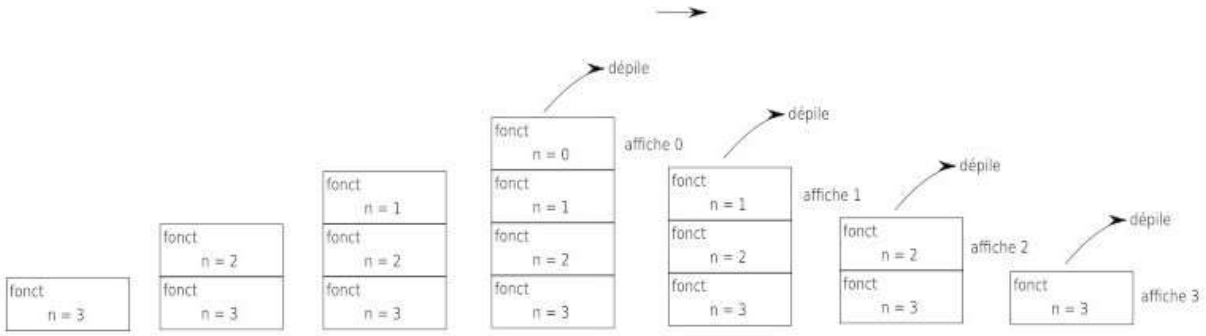
L'utilisation de la récursivité se base sur l'usage d'une **pile.** Une **pile d'exécution** précisément.
Voici la schématisation de la somme des entiers naturels jusqu'à n :

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

Une pile se base sur le **LIFO (last in first out)**, le dernier élément entré est le premier à sortir. On schématise :



On modélise maintenant somme_recu(3) :



Donc ici chaque à chaque dépile on ajoute la valeur qui a été retournée. Donc ici 0+1+2+3=6

L'utilisation d'une fonction récurive peut être pratique car elle peut éviter l'utilisation abusive de boucles **for** ou **while**.

D) Exercices

Facile : écrire une fonction récursive **compte_a_rebours(n)** qui donne les entiers de **n à 0**, dans cet ordre.

Moyen : écrire une fonction récursive **inverser_chaine(chaine)** qui **inverse** une chaine de cractères entrée en paramètre.

Difficile : écrire une fonction **est_palindrome(chaine)** qui retourne **True** si la chaine de caractères est un **palindrome**, **False** sinon.

E) Corrections

Facile :

```
compte_a_rebours(n) :
    if n==0 :
        return 0
    print(n)
    return compte_a_rebours(n)
```

Il est aisé de comprendre ce que fait cette fonction.

Moyen :

pour coder cette fonction, il est nécessaire de savoir d'où l'on part et où l'on va. Il est facile de constater que le but est de retourner le dernier élément de la chaîne à chaque appel récursif et de les retourner un à un ce qui les retournera à l'envers. On utilisera une chaine auxiliaire

```
def inverser_chaine(chaine,chaine_fin=''):
        if chaine=='':
                return chaine_fin
        a=chaine[-1]   ##prend pour valeur de dernier élément de chaine et l'enlève de chaine
        chaine_fin+=a  ## équivalent à : chaine_fin=chaine_fin + a
        return inverser_chaine(chaine[:-1],chaine_fin) ##[-1] est le dernier élément d'une liste
```

ici en clair, on parcourt les éléments du dernier au premier et les met dans la chaîne de caractères auxiliaire, tout en enlevant les valeurs une à une. Lorsque « chaine » est vide, le mot est retourné et on arrête la fonction.

Difficile :

Méthode 1 :

On peut réutilise la fonction qu'on vient de faire en y effectuant des changements :

```
def inverser_chaine(chaine,chaine_fin='',i=0):
    if len(chaine)==len(chaine_fin):   ##S'arrête si on a inséré sans problème toutes les lettres
        return True
    a=chaine[-i-1]   ##prend pour valeur de dernier élément de chaine
    chaine_fin=chaine_fin+a
    if a!=chaine[i]:    ##on compare la lettre à son miroir par rapport au mot
        return False
    return inverser_chaine(chaine,chaine_fin,i+1)
```

Vous l'aurez compris on utilise un indice i qui nous servira à comparer la lettre qu'on ajoute à chaine_fin avec son symétrique par rapport au milieu du mot.

Méthode sans chaîne auxiliaire :

```
def est_palindrome(chaine, i=0):
    if i >= len(chaine) // 2:
        return True
    if chaine[i] != chaine[-(i + 1)]:
        return False
    return est_palindrome(chaine, i + 1)
```

Donc à la place de comparer une nouvelle chaîne de caractères on effectue la récursion sur l'indice. On vérifie que la moitié du mot est symétrique avec l'autre moitié, d'où « if i>=len(chaine)//2:». Car si une moitié est parallèle avec l'autre alors la réciproque est vraie.