3.3 La récursivité en Python

La récursivité est un notion importante en programmation ; elle n'est pas réservée qu'au langage Python. Nous l'étudions donc ici avec le langage Python

3.3.1 Définition de fonction récursive

Une fonction est dite récursive si elle s'appelle elle même

3.3.2 Etude d'un exemple : la factorielle

Un premier exemple, très classique, et facile est la définition de factorielle qui est donnée par :

$$\begin{cases} 0! = 1 \\ 1! = 1 \\ n! = n \times (n-1)! \end{cases}$$

L'algorithme donne :

```
#-*-coding:Latin-1 -*


#algo Fonction factorielle avec recursivité


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

Et nous avons comme résultats :

```
>>> factoriel(0)
1
>>> factoriel(1)
1
>>> factoriel(10)
3628800
>>> factoriel(30)
265252859812191058636308480000000
>>>
```

Dans le script, nous aurions aussi pu écrire :

```
return n*factoriel(n-1)
```

Plutôt que

```
f= n*factoriel(n-1)
return f
```

Remarque 10 :

1. La récursivité est une technique de programmation très puissante : elle permet de trouver des solutions élégantes à des problèmes compliqués. Certains domaines comme les mathématiques sont plus propices à des solutions récursives simples. La difficulté est de penser à cette technique de programmation pour imaginer un algorithme.
2. Deux conditions sont nécessaires pour être en mesure d'utiliser la récursivité :

⇒ Il faut pouvoir exprimer un algorithme sous forme d'une fonction de telle manière que sa valeur à un certain rang ne dépende que de sa valeur aux rangs inférieurs
⇒ On doit aussi connaître sa valeur aux rangs initiaux.

3. **Technique pour écrire une fonction récursive :** il suffit d'utiliser la fonction qui n'a pas encore été écrite, en supposant qu'elle donne déjà un résultat
4. **Fonctionnement d'une fonction récursive**
   La récursivité, c'est une utilisation dynamique de la mémoire ; chaque appel à la fonction provoque l'obtention d'un nouvel espace mémoire

3.3.3 Suite de Fibonnacci

La suite de Fibonacci $(u_n)_{n\in\mathbb{N}}$ est une suite définie par ses deux premiers termes et le terme d'ordre n est défini par la somme des deux termes qui le précèdent ; autrement dit :

$$\begin{cases} u_0 \in \mathbb{R} \\ u_1 \in \mathbb{R} \\ u_n = u_{n-1} + u_{n-2} \end{cases}$$

La résolution algorithmique est celle ci :

```
#-*-coding:Latin-1 -*

#algo Fonction Suite de Fibonacci récursive


def Fibonacci(a,b,n):
    if  (n==0):
        return a
    elif (n==1):
        return b
    else:
        return Fibonacci(a,b,n-1) + Fibonacci(a,b,n-2)
```

Le résultat de l'exécution du script donne :

```
>>> Fibonacci(1,1,3)
3
>>> Fibonacci(1,1,5)
8
>>> Fibonacci(2,6,30)
6020698
>>>
```

Remarque 11 :

La version récursive effectue **un nombre exponentiel** de multiplication, car elle ne mémorise pas les calculs déjà effectués. La solution récursive n'est pas toujours la meilleure ; elle est même, parfois, très très lente.

3.3.4 Exercices

Exercice 7 :

Ecrire une fonction `Python` récursive $U(n)$ qui retourne U_n avec n un entier positif passé en paramètre :

$$\begin{cases} U_0 = 5 \\ U_n = \sqrt{1 + U_{n-1}} \end{cases}$$

Exercice 8 :

En supposant que la fonction puissance ne soit pas native, coder un programme, utilisant une fonction récursive pour calculer la fonction puissance entière. Algorithme `calcul_de_puissance`

Exercice 9 :

Ecrire une fonction récursive `Binomial(n,p)` permettant de calculer le coefficient binomial C_n^p, où n et p sont des entiers naturels passés en paramètres.
On rappelle que :

$$C_n^p = \frac{n!}{p!\,(n-p)!} \text{ et que } C_n^p = C_{n-1}^p + C_{n-1}^{p-1}$$

Exercice 10 :

Ecrire une fonction récursive `sommation(n)` qui calcule la somme des inverses des carrés des n premiers entiers naturels non nuls, n est passé en paramètre : $S = \sum_{k=1}^{n} \frac{1}{k^2}$