TD 9 : Introduction à la récursivité

L'exemple classique que l'on prend pour illustrer la récursivité est le calcul de $n!$.
En mathématiques, cette suite est définie par récurrence : $0! = 1$ et $(n+1)! = n! \times (n+1)$
On peut définir une fonction classique suivante :

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

Il est possible de programmer ce calcul par récursivité :

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

La fonction, après avoir été mise en mémoire, est capable de s'appeler autant de fois que nécessaire afin d'arriver au résultat direct qu'elle connaît, puis de remonter au résultat attendu.
On parle de fonction récursive.
Le principe d'exécution est le suivant, par exemple pour : $4!$:
Je demande $4!$:

- 4 est différent de 0 - Je cherche $4 * 3!$ – je dois calculer $3!$
  - 3 est différent de 0 - Je cherche $3 * 2!$ – je dois calculer $2!$
    - 2 est différent de 0 - Je cherche $2 * 1!$ – je dois calculer $1!$
      - 1 est différent de 0 - Je cherche $1 * 0!$ – je dois calculer $0!$
        0 est égal à 0 - Je retourne 1
      - Je calcule $1 * 0! = 1 * 1 = 1$, je retourne 1
    - Je calcule $2 * 1! = 2 * 1 = 2$, je retourne 2
  - Je calcule $3 * 2! = 3 * 2 = 6$, je retourne 6
- Je calcule $4 * 3! = 4 * 6 = 12$, je retourne 12

(on pourra utiliser *python-tutor* en cas de besoin).
D'une manière générale, les suites définies par récurrence se prêtent bien à l'utilisation de la récursivité. Voici l'exemple de la suite de Fibonacci :

```
def fibonacci(n):
    if n==0:
        return 0
    elif n==1:
        return 1
    else:
        return fibonacci(n-1)+fibonacci(n-2)
```

On remarquera que l'on peut appeler plusieurs fois la même fonction en récursivité. On verra toutefois que cette manière de programmer est très inefficace.

Définition : Une fonction est dite *récursive* si elle s'appelle elle même. On parle alors d'appels récursifs. On leur oppose les fonctions dites *itératives*.

Les trois principes généraux :

1. Une fonction récursive doit contenir une ou plusieurs conditions d'arrêts, sinon elle boucle indéfiniment.
2. Les valeurs en paramètres de la fonction lors des différents appels récursifs doivent être différents car sinon la fonction s'exécute de manière identique à chaque appel et donc boucle indéfiniment.
3. Après un nombre fini d'appels, la ou les valeurs mises en paramètres doivent permettre de valider la condition d'arrêt.

Remarques : Dans le cas de la fonction calculant $n!$, les calculs ne peuvent pas être effectués avant d'obtenir les valeurs renvoyées par les appels récursifs, ce qui sollicite la mémoire de la machine pour stocker les instructions en attente. On parle de récursivité profonde.
Ce n'est pas le cas pour la fonction qui suit : on parle alors de récursivité terminale.

```
def affiche(k):
    if k<10:
        print(k)
        affiche(k+1)
```

Enfin, on applique souvent un algorithme récursif dans le cas d'une stratégie "diviser pour régner" : Traiter les cas de base - Diviser le problème à traiter en sous problèmes plus simples - Puis on applique récursivement l'algorithme à chaque sous problème - Enfin on trouve la solution du problème initial en combinant les différents résultats intermédiaires.

Exercice 1 (Échauffement)

On considère la fonction suivante :

```
def pair(n):
    while n>0:
        n=n-2
    return n==0
```

(1) Que fait cette fonction ?
(2) Écrire une version récursive de cette fonction.
(3) Écrire une fonction itérative qui prend en argument un entier n et renvoie le compte à rebours de n à 0. Faire sa version récursive.

Exercice 2 (La multiplication)

(4) Écrire une fonction itérative puis récursive qui prend en entrée un flottant x et un entier n et renvoie x^n en utilisant la définition : $x^0 = 1$ et $\forall n \in \mathbb{N}$, $x^{n+1} = x \times x^n$ et en n'utilisant pas les puissances bien entendu !
(5) Faire de même en utilisant la définition : $x^0 = 1$ et $\forall n \in \mathbb{N}$, si n est pair, avec $n = 2k$ alors $x^n = (x^k)^2$ et si n est impair, avec $n = 2k + 1$ alors $x^n = x(x^k)^2$
(6) Faire de même en utilisant la définition : $x^0 = 1$ et $\forall n \in \mathbb{N}$, si n est pair, avec $n = 2k$ alors $x^n = (x^2)^k$ et si n est impair, avec $n = 2k + 1$ alors $x^n = x(x^2)^k$
(7) Comparer le nombre d'appels lorsque $n = 64$ par exemple.

Exercice 3 (La suite de Syracuse)

La suite de Syracuse est définie par son premier terme u_0 et par la relation de récurrence : pour tout $n \geqslant 1$,

$$u_{n+1} = \begin{cases} \dfrac{u_n}{2} & si \quad u_n \ est \ pair \\ 3u_n + 1 & sinon \end{cases}$$

(8) Écrire une fonction récursive qui prend en paramètres deux entiers n et $u0$ et renvoie le terme de rang n de la suite de Syracuse définie par le premier terme $u_0 = u0$.

(9) La conjecture de Collatz (1937) affirme que la suite finit toujours par donner un terme égal à 1. Elle devient alors périodique avec les valeurs 4, 2 puis 1. Écrire une fonction récursive `collatz` prenant en paramètre un entier naturel $u0$ et qui retourne le rang du premier terme de la suite qui est égal à 1 dans le cas où $u_0 = u0$.