TP- RECURSIVITE

L'objectif de ce TP est de mieux appréhender le principe de la récursivité.

1/ Une fonction qui appelle une autre fonction :

Analysez puis testez le programme suivant :

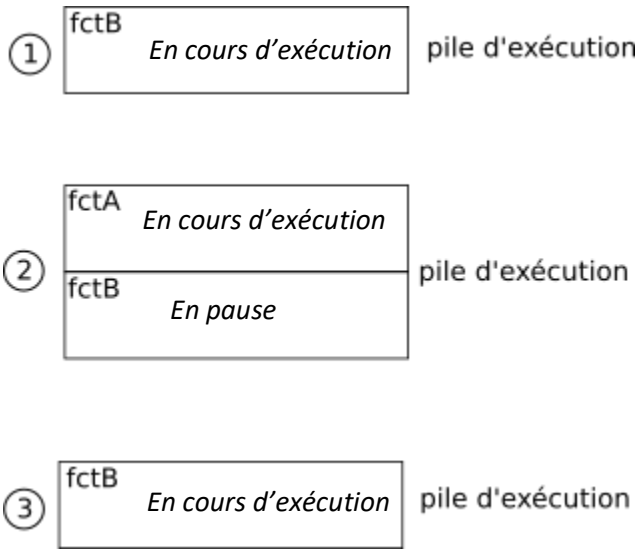
```
def fctB():
   print ("Début fonction fctB")
   for i in range (0,5):
      if i==3:
         fctA()
         print("    Retour à la fonction fctB")
      print("    fctB avec son i="+ str(i))
   print ("Fin fonction fctB")

def fctA():
   print ("    Début fonction fctA")
   for i in range(0,5):
      print( "       fctA avec son i=" + str(i))
   print ("    Fin fonction fctA")

fctB()
```

Vous devriez obtenir ceci :

```
Début fonction fctB
   fctB avec son i=0
   fctB avec son i=1
   fctB avec son i=2
   Début fonction fctA
      fctA avec son i=0
      fctA avec son i=1
      fctA avec son i=2
      fctA avec son i=3
      fctA avec son i=4
   Fin fonction fctA
   Retour à la fonction fctB
   fctB avec son i=3
   fctB avec son i=4
Fin fonction fctB
```

Dans l'exemple ci-dessus, nous avons **une fonction (fctB) qui appelle une autre fonction (fctA).** La principale chose à retenir de cet exemple est que **l'exécution de fctB est interrompue pendant l'exécution de fctA**. Une fois l'exécution de fctA terminée, l'exécution de **fctB reprendra là où elle avait été interrompue.**

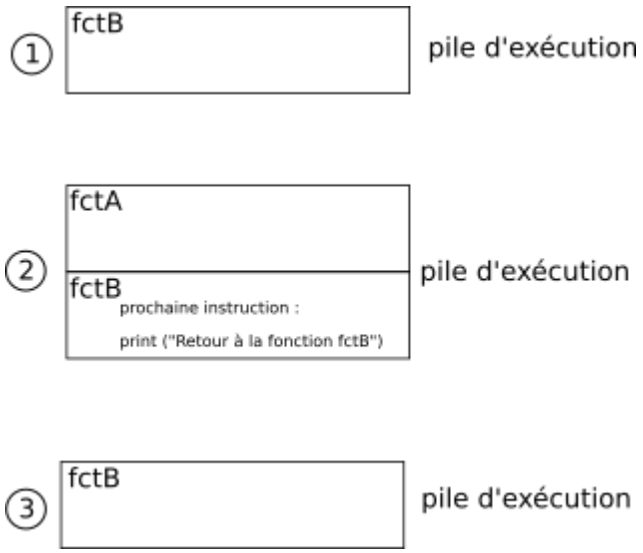
Pour gérer ces fonctions qui appellent d'autres fonctions, le système utilise une "pile d'exécution". Une pile d'exécution permet d'enregistrer des informations sur les fonctions en cours d'exécution dans un programme. On parle de pile, car les exécutions successives "s'empilent" les unes sur les autres. Si nous nous intéressons à la pile d'exécution du programme étudié ci-dessus, nous obtenons le schéma suivant :



Nous pouvons "découper" l'exécution de ce programme en 3 parties :

1. la fonction fctB s'exécute jusqu'à l'appel de la fonction fctA
2. l'exécution de la fctB est mise en "pause" pendant l'exécution de la fonction fctA
3. une fois que l'exécution de fctA est terminée, on termine l'exécution de la fonction fctB

Il est important de bien comprendre que **la fonction située au sommet de la pile d'exécution est en cours d'exécution**.
Toutes les fonctions situées "en dessous" sont mises en pause jusqu'au moment où elles se retrouveront au sommet de la pile. **Quand une fonction termine son exécution, elle est automatiquement retirée du sommet de la pile (on dit que la fonction est dépilée).**
La pile d'exécution permet de retenir la prochaine instruction à exécuter au moment où une fonction sera sortie de son ""état de pause" (qu'elle se retrouvera au sommet de la pile d'exécution).



Évidemment l'explication donnée ci-dessus est quelque peu simpliste : c'est l'adresse mémoire de la prochaine instruction machine à exécuter qui est conservée dans la pile d'exécution

Dans l'exemple ci-dessus, on retrouve une variable i dans les deux fonctions : fctA et fctB. **La variable i présente dans la fonction fctA n'a rien à voir avec la variable i présente dans la fonction fctB** (elles portent le même nom, mais elles représentent 2 adresses mémoires différentes). Il est très important de bien comprendre que les variables créées dans une fonction ne "sortent" pas de la fonction : chaque fonction possède sa propre liste de variable, comme déjà dit ci-dessus la variable i de la fonction fctB est différente de la variable i de la fonction fctA. On parle de **portée** des variables.
La pile d'exécution conserve une "trace" des valeurs des variables lorsqu'une autre fonction est exécutée. Par exemple la valeur de i (fctB) est conservée au moment de l'exécution de fctA. Quand l'exécution de fctA se termine est que l'exécution de fctB "reprend", la valeur référencée par i (fctB) a été "conservée" (voilà pourquoi on reprend l'exécution de fctB avec un "fctB 3").

Une fonction peut s'appeler elle-même, on parle alors de fonction récursive.

2/ une fonction qui s'appelle elle-même :

Analysez puis testez le programme suivant :

```
def fct ():
  print ("Hello")
  fct ()

fct ()
```

Comme vous pouvez le constater, nous avons une erreur dans la console Python. Dans le cas où une fonction s'appelle elle-même (fonction récursive), on retrouve le même système de pile d'exécution. Dans l'exemple traité ci-dessus, les appels s'enchainent sans rien pour mettre un terme à cet enchainement, la taille de la pile d'exécution augmente sans cesse (aucune fonction ne termine son exécution, nous n'avons pas de "dépilement" juste des "empilements"). **Le système interrompt le programme en générant une erreur quand la pile d'exécution dépasse une certaine taille.**

Quand on écrit une fonction récursive, il est donc nécessaire de bien penser à mettre en place une structure qui à un moment ou à un autre mettra fin à ces appels récursifs.

Exemple pour notre fct :

```
def fct (a):
   a=a-1
   if a>0 :
      print ("Hello a=" + str(a))
      fct (a)
   else:
      return 0

fct (12)
```

La fonction reçoit un argument « a » qui se décrémente à chaque appel de la fonction.

autre exemple :

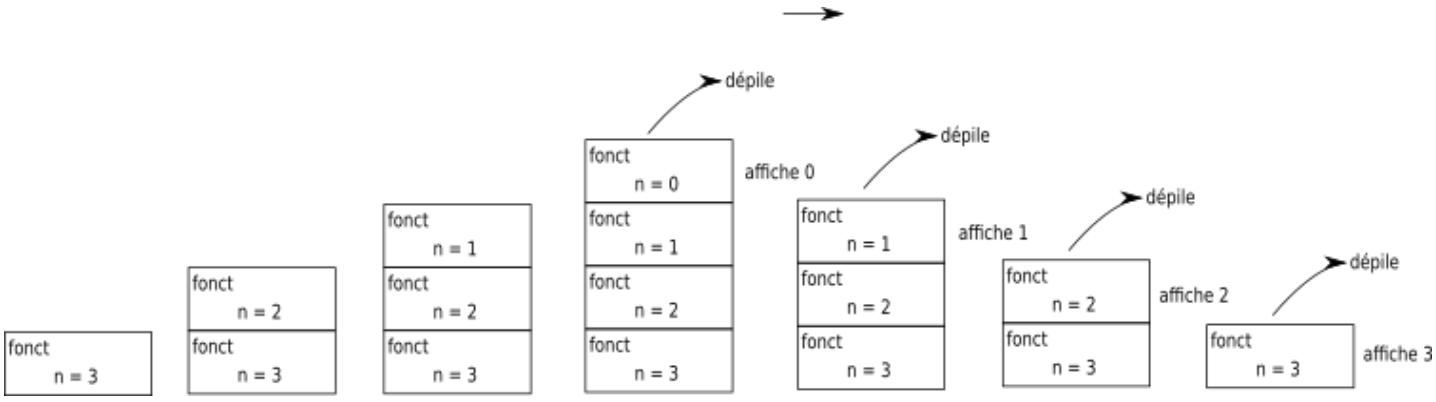
Essayez de prévoir le résultat de l'exécution du programme ci-contre. Vérifiez votre hypothèse en exécutant le programme.

```
def fonct(n):
   if n>0:
      fonct(n-1)
   print(n)

fonct(3)
```

Essayons de comprendre en détail ce qui se passe dans le programme ci-dessus :

1er appel de la fonction *fonct* avec le paramètre n = 3 ; n > 0 donc appel de la fonction *fonct* avec le paramètre n = 2
2e appel de la fonction *fonct* avec le paramètre n = 2 ; n > 0 donc appel de la fonction *fonct* avec le paramètre n = 1
3e appel de la fonction *fonct* avec le paramètre n = 1 ; n > 0 donc appel de la fonction *fonct* avec le paramètre n = 0
4e appel de la fonction *fonct* avec le paramètre n = 0 ; n = 0 donc on exécute l'instruction *print(n)* => affichage : 0
on "dépile" (3e appel, n = 1) : on exécute l'instruction *print(n)* => affichage : 1
on "dépile" (2e appel, n = 2) : on exécute l'instruction *print(n)* => affichage : 2
on "dépile" (1er appel, n = 3) : on exécute l'instruction *print(n)* => affichage : 3

Voici un schéma expliquant le processus en termes de pile d'exécution :



Il ne faut jamais perdre de vu qu'à chaque nouvel appel de la fonction *fonct,* le paramètre n est différent.

3/ faisons de beaux dessins en utilisant la récursivité :

Étudiez le Wikibook consacré au module Turtle (wikibook Turtle) afin d'acquérir les bases de ce module.
(https://fr.wikibooks.org/wiki/Programmation_Python/Turtle)

Essayez de prévoir le résultat de l'exécution du programme ci-dessus. Vérifiez votre hypothèse en exécutant le programme.

```
import turtle as t
t.forward(100)
t.left(120)
t.forward(100)
t.left(120)
t.forward(100)
```

Nous allons maintenant étudier le flocon de Koch :

Visionnez la vidéo consacrée au flocon de Koch :

https://youtu.be/PW_Pka9iBko

Après avoir testé le programme ci-dessous, vous l'étudierez attentivement. Vous vous concentrerez notamment sur le rôle des paramètres taille et étape de la fonction flocon.

```
import turtle as t

def koch(longueur, n):
    if n == 0:
        t.forward(longueur)
    else:
        koch(longueur/3, n-1)
        t.left(60)
        koch(longueur/3, n-1)
        t.right(120)
        koch(longueur/3, n-1)
        t.left(60)
        koch(longueur/3, n-1)

def flocon(taille, etape):
    koch(taille, etape)
    t.right(120)
    koch(taille, etape)
    t.right(120)
    koch(taille, etape)

flocon(100, 3)
```