On a vu jusqu'à présent que des fonctions pouvaient appeler d'autres fonctions. Il est également possible de programmer une fonction qui s'appelle elle-même. On parle alors de *fonction récursive*.

La récursivité est une démarche qui fait référence à l'objet même de la démarche à un moment du processus.
En d'autres termes, c'est une démarche dont la description mène à la répétition d'une même règle (Wikipédia).

En informatique, un algorithme qui contient des appels à lui-même est dit *récursif*. La définition de certaines structures de données comme les listes ou les arbres (que nous verrons un peu plus tard dans l'année) est dite récursive.

1 Premiers exemples

1.1 La somme des premiers entiers

Supposons que l'on souhaite calculer la somme $1 + 2 + 3 + \cdots + (n-1) + n$ où n désigne un entier positif.

On peut pour cela procéder de la façon suivante : on utilise une variable d'accumulation (ici on la nomme `S`) dans laquelle on ajoute au fur et à mesure les entiers jusqu'à n (c'est la version *itérative* ci-contre utilisant une boucle « Pour »).

```
def sommeIterative(n):
  S = 0
  for i in range(1,n+1):
     S += i
  return S
```

Mais on peut aussi calculer cette somme en posant $S_n = 1 + 2 + 3 + \cdots + (n-1) + n$ et du coup on constate que :

- $S_1 = 1$
- $S_2 = 1 + 2 = S_1 + 2$
- $S_3 = 1 + 2 + 3 = S_2 + 3$
- ainsi : $S_n = S_{n-1} + n$

La somme des n premiers entiers peut donc être calculée à partir de la somme des $(n-1)$ premiers entiers à laquelle on ajoute l'entier n. C'est une *relation de récurrence* (la dernière valeur est calculée à partir de la ou les valeur(s) précédente) et on donne ci-contre une implémentation de cette version récursive en `Python` du calcul de la somme des premiers entiers.

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

1.2 Exercice : calcul de factorielle n

Sur le même modèle que le calcul de la somme des premiers entiers, écrire un algorithme récursif `fact(n)` permettant de calculer le produit des n premiers entiers (strictement positifs) nommé mathématiquement *factorielle* n.

1.3 Exercice : suite numérique définie par récurrence

La récursivité est naturellement adaptée au calcul des termes d'une suite numérique (liste ordonnée de nombres) définie par récurrence (on donne la ou les premières valeurs de la suite puis la façon de calculer le nombre suivant en fonction du ou des nombres précédents)

Exemple 1.1 $\{1, 3, 5, 7, 9, 11, \cdots\}$ est la suite numérique des nombres impairs de premier terme $u_0 = 1$ avec la relation de récurrence : $u_{n+1} = u_n + 2$ (pour calculer le terme suivant on ajoute 2 au précédent).

On donne la suite numérique (u_n) définie sur l'ensemble des entiers naturels par récurrence par : $u_0 = 12$ (le premier terme de la suite) et la relation de récurrence $u_{n+1} = 3u_n + 1$ (ici pour calculer le terme suivant il faut multiplier le précédent par 3 et lui ajouter 1).
Ecrire une programme itératif puis un programme récursif pour calculer le terme de rang n (n entier positif). On les testera pour $n = 100$.

2 Arbre d'appels et pile d'exécution

2.1 Arbres d'appels : exemple avec l'exécution de `sommeRecursive(3)`

Reprenons l'exemple de la somme des premiers entiers. A chaque fois que la variable entière `n` est strictement supérieure à 0, la fonction s'appelle elle-même avec `sommeRecursive(n-1)`. Cet appel s'appelle *un appel récursif*. Par contre, lorsque `n` est nul, la fonction renvoie `0`. On peut alors représenter l'exécution de `sommeRecursive(3)` de la façon suivante :

```
sommeRecursive(3)  =  return 3  +  sommeRecursive(2)
                                          |
                                   return 2      +  sommeRecursive(1)
                                                           |
                                                    return 1      +  sommeRecursive(0)
                                                                             |
                                                                          return 0
```

Cette façon de représenter l'exécution d'un programme s'appelle *un arbre d'appels*. On constate que pour calculer la valeur renvoyée par `sommeRecursive(3)` il faut d'abord appeler `sommeRecursive(2)` qui va lui même déclencher un appel à `sommeRecursive(1)` qui lui-même appelle `sommeRecursive(0)` qui enfin renvoie `0`. Après ce renvoi de la valeur `0`, l'arbre d'appel devient :

```
sommeRecursive(3)  =  return 3  +  sommeRecursive(2)
                                          |
                                   return 2      +  sommeRecursive(1)
                                                           |
                                                    return 1 + 0
```

L'appel à `sommeRecursive(1)` renvoie alors `1` et l'arbre d'appel devient :

```
sommeRecursive(3)  =  return 3  +  sommeRecursive(2)
                                          |
                                    return 2 + 1
```

L'appel `sommeRecursive(2)` finit alors de s'exécuter pour donner :

```
sommeRecursive(3) = return 3 + 3
```

Finalement le résultat final de `sommeRecursive(3)` est `6` qui est bien ce qui était attendu.

2.2 Pile d'exécution

Exercice 2.1 Ecrire à côté du programme l'affichage obtenu dans la console à l'exécution de ce code.

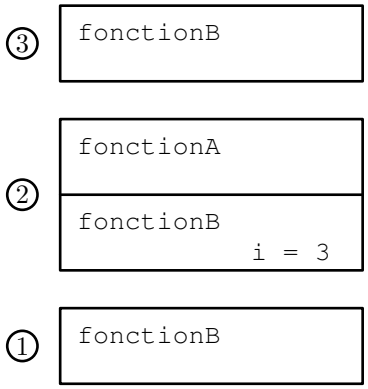
```
def fonctionA() :
    print("Début de la fonction A")
    i = 0
    while i < 5 :
        print("fonctionA", i)
        i = i + 1
    print("Fin fonction A")

def fonctionB() :
    print("Début de la fonction B")
    i = 0
    while i < 5 :
        if i == 3 :
            fonctionA()
            print("Retour à la fonction B")
        print("fonctionB", i)
        i = i + 1
    print("Fin fonction B")

fonctionB()
```

Dans l'exemple ci-dessus, nous avons une fonction (`fonctionB`) qui appelle une autre fonction (`fonctionA`). La principale chose à retenir de cet exemple est que l'exécution de `fonctionB` est interrompue pendant l'exécution de `fonctionA`. Une fois l'exécution de `fonctionA` terminée, l'exécution de `fonctionB` 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 :



Le schéma ci-contre donne les informations suivantes sur le déroulement du programme en indiquant l'état de la pile d'exécution :

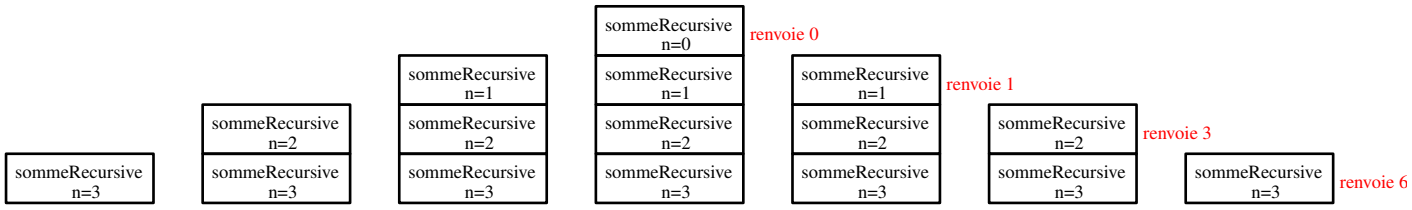
1. appel de la fonction `fonctionB` qui est ajouté à la pile d'exécution du programme,
2. dans la fonction `fonctionB`, la `fonctionA` est appelée, on ajoute cet appel à la pile d'exécution,
3. la fonction `fonctionA` a fini d'être exécutée, elle est donc dépilée de la pile d'exécution, la fonction `fonctionB` termine son exécution.

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).
En réalité la pile d'exécution ne contient pas la fonction à exécuter mais l'adresse mémoire de la prochaine instruction machine à exécuter. Le schéma simpliste ci-dessous ne représente que 3 principales étapes alors qu'en réalité il y en a beaucoup plus.

Il faut également retenir que les variables sont associées à la fonction dans lesquelles elles sont définies. Ici nous avons une variable `i` dans la fonction `fonctionA` et une variable `i` dans la fonction `fonctionB`. La pile d'exécution garde en mémoire la valeur des variables internes à chaque fonction c'est ce qui permet de conserver la valeur de `i` référencée dans `fonctionB` quand `fonctionA` est exécutée : ainsi on reprend bien l'exécution de `fonctionB` avec l'affichage `"fonction B 3"`.

Remarque 2.1 Le langage `Python` affiche la trace d'appels correspondant à la pile d'exécution quand il y a des erreurs au moment de l'exécution d'un programme. Voir `https://fr.wikipedia.org/wiki/Trace_d%27appels` pour un exemple.

Exemple 2.2 Retour à la récursivité. L'instruction `sommeRecursive(3)` produit les étapes suivantes d'empilement et dépilement dans la pile d'exécution :



Exercice 2.2 Quelle est la taille maximale la pile d'exécution quand on saisit l'instruction `fact(5)` (fonction de l'exercice 1.2)? Justifier la réponse en dressant un arbre d'appels.

2.3 Erreur à l'exécution

Exercice 2.3 Exécuter le code ci-contre, que se produit-il ?

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

Dans le cas d'une fonction récursive (qui s'appelle elle-même) une pile d'exécution est créée. Ici la fonction produit des appels *récursifs* sans jamais s'arrêter : on empile à la pile d'exécution des appels à `affiche()` sans jamais dépiler et donc l'exécution ne se termine pas. Le langage `Python` a prévu de limiter le nombre d'appels récursifs à 1 000.

Remarque 2.3 Il est possible de modifier le nombre d'appels récursifs autorisé par `Python` en utilisant le module `sys` et la fonction `setrecursionlimit`.

3 Synthèse : comment définir une fonction récursive bien formée

Un *algorithme récursif* est un algorithme qui résout un problème en calculant des solutions d'instances plus petites du même problème. On parle de fonction récursive **bien définie** quand :

- ⋆ la fonction s'appelle elle-même sur des cas plus simples,
- ⋆ **et** la fonction s'arrête.

En pratique, pour programmer une telle fonction, il faut donc respecter certaines règles pour s'assurer que la fonction finira par s'arrêter, c'est-à-dire ne bouclera pas indéfiniment :

- prévoir un cas de base (ou cas d'arrêt) qui ne fait pas d'appel récursif ,
- s'assurer que les appels récursifs se font sur des cas plus simples que le cas initial (liste moins longue, nombre plus petit, ...).

Exemple 3.1 Si son reprend l'exemple de la somme des premiers entiers le cas de base est le cas où n est nul et, à chaque appel récursif, le nombre utilisé diminue ce qui nous assure que l'algorithme s'arrête (ce nombre finira par être nul).

Exercice 3.1 Justifier que la fonction `fact` définie dans l'exercice 1.2 est bien formée.

Remarque 3.2 En `Python` la programmation récursive nécessite de limiter le nombre d'appels récursifs. Il existe des algorithmes récursifs utilisant des accumulateurs pour en améliorer l'efficacité (voir par exemple l'exercice sur le calcul rapide d'une puissance). En plus de vérifier que la fonction récursive est bien formée, il faudra veiller à choisir une définition limitant le nombre d'appels récursifs.

La programmation récursive est utilisée, par exemple, pour réaliser des fractales (triangles de Sierpinski entre autres), concevoir un solveur de Sudoku, programmer le jeu du Démineur, parcourir des arbres, etc.
Le principal avantage de la récursivité est de pouvoir concevoir, dans certains cas, des programmes plus simples, plus courts et donc plus faciles à comprendre.