# *Récursivité*

## I./ Notion de pile d'exécution :

Nous avons vu dans les classes précédentes la notion de *fonction* en Python. Une fonction attend un ou plusieurs *arguments* (objets qui sont passés à la fonction) et retourne un ou plusieurs *résultats*.

ⓘ **Point culture :** on parle de *paramètre* (ou de *paramètre formel*) lors de la définition de la fonction, et d'*argument* (ou *paramètre effectif*) lors de l'appel de la fonction :
* Dans « `def fonction(a, b, c):` », a, b et c sont appelés *paramètres*.
* Dans « `d = fonction(45, 6.0, 8)` », 45, 6.0 et 8 sont appelés *arguments*.

Une fonction Python peut très bien appeler une autre fonction Python. Tapez le script Python suivant :

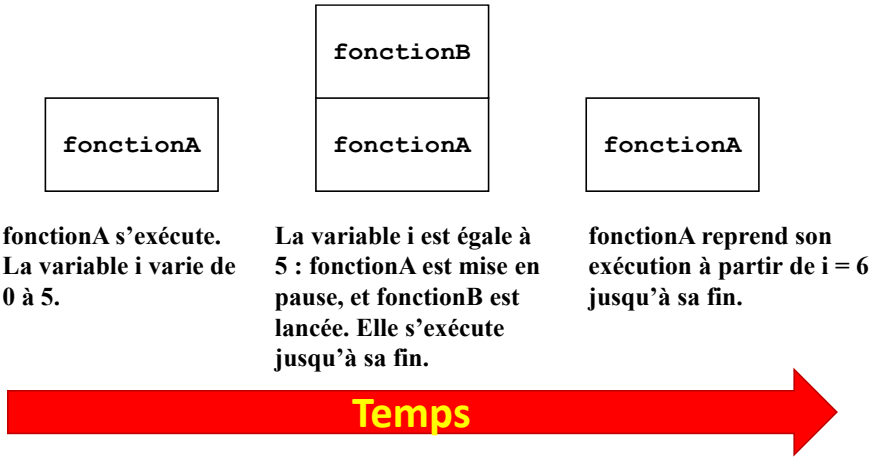
```
def fonctionA():
    print("fonctionA démarre...")
    for i in range(10):
        print("A :", i)
        if i == 5:
            fonctionB()
    print("fonctionA s'arrête...")
    return None

def fonctionB():
    print("fonctionB démarre...")
    for i in range(10):
        print("B :", i)
    print("fonctionB s'arrête...")
    return None
```

ⓘ **Point culture :** certaines fonctions n'ont pas de paramètres, comme les deux fonctions précédentes. Il faut quand même les définir et les appeler en se servant de parenthèses. Certaines fonctions ne retournent aucun résultat, et c'est aussi le cas des deux fonctions précédentes. Pour respecter la définition d'une fonction, on termine ce type de fonctions par « `return None` ». `None` est un objet spécial du langage Python qui représente l'absence de valeur. Cette instruction « `return None` » est facultative : vous pourrez le vérifier en testant le programme avec et sans cette ligne. Il n'est donc pas grave de ne pas la mettre, mais son utilisation est « plus propre ».

Une fois le script tapé, lancez l'instruction : « `fonctionA()` ». Que remarque-ton ?

Lorsque le compteur de boucle « `i` » est égal à 5 dans `fonctionA`, `fonctionB` est lancée. Une fois que `fonctionB` est terminée, `fonctionA` reprend là où elle en était restée. Afin de reprendre au même endroit, il faut que l'interpréteur Python conserve une trace de l'endroit où la précédente fonction s'était arrêtée : c'est le rôle de la *pile d'exécution*.



La fonction au sommet de la pile d'exécution est celle qui est en cours d'exécution. Toutes les fonctions qui sont en-dessous sont en pause, et attendent la fin des fonctions qui se trouvent au-dessus pour redémarrer à partir de l'endroit où elles se sont arrêtées.

## II./ Fonction récursive :

Nous venons de voir qu'une fonction peut appeler une autre fonction. Mais une fonction peut s'appeler elle-même : c'est ce que l'on appelle une *fonction récursive*.

Tapez le script Python suivant, et lancez-le à l'aide de la commande « **`fonction_recursive()`** »

```
def fonction_recursive():
    print("La fonction démarre !")
    fonction_recursive()
    return None
```

Que remarque-t-on ?

Le message « **`maximum recursion depth exceeded`** » indique que la capacité de la pile d'exécution a été dépassée. Cela vient du fait que la fonction ne cesse de s'appeler elle-même inconditionnellement. Afin d'éviter cette erreur, il faut que les appels récursifs se fassent lorsqu'une condition est remplie.

*Exemple 1 :* calcul de la factorielle d'un nombre entier.

Soit un nombre entier n. La factorielle de n, notée n!, est égale à :

- 1 si n = 0 ou si n = 1
- $n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1$ si n > 1.

La factorielle de n est donc le produit de tous les nombres entiers compris entre 1 et n.

Tapez le script Python suivant, et vérifiez que la fonction **`factorielle(n)`** calcule bien la factorielle du nombre passé en argument.

```
def factorielle(n):
    if n <= 1:
        return 1
    resultat = 1
    for i in range(2, n+1):
        resultat = resultat * i
    return resultat

# Programme principal
# => affiche les nombres de 0 à 10 ainsi que la valeur de leurs factorielles
for i in range(11):
    print("La factorielle de", i, "est égale à", factorielle(i))
```

Une autre façon de définir la factorielle d'un nombre n est la suivante (on parle de *relation de récurrence*) :

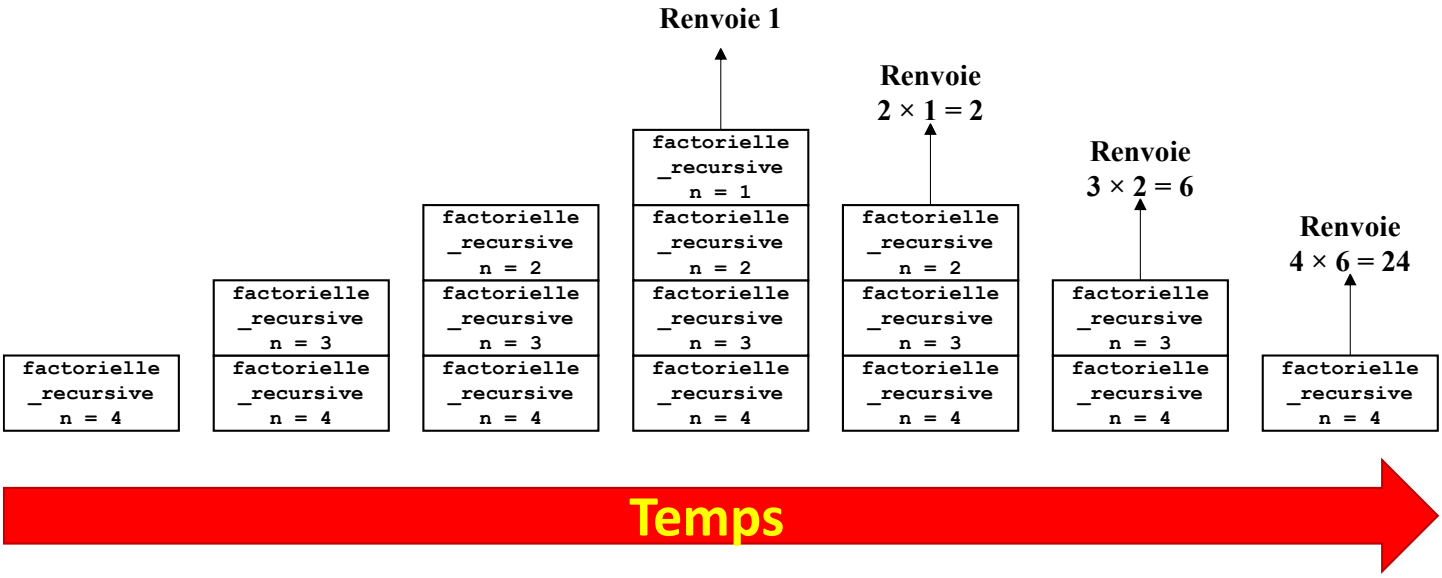
- 0! = 1 et 1! = 1
- Pour tout n > 1, n! = n × (n – 1)!

Modifiez le script Python précédent en rajoutant la fonction **`factorielle_recursive()`** :

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

Modifiez ensuite le programme principal pour qu'il affiche la valeur de **`factorielle(n)`** et **`factorielle_recursive(n)`** pour les nombres compris entre 0 et 10. Vérifiez que les deux fonctions renvoient toutes les deux la valeur de la factorielle de n.

Que se passe-t-il lorsqu'on tape « **`factorielle_recursive(4)`** » dans la console Python ? Le schéma suivant montre l'évolution de la pile d'exécution en fonction du temps :




L'argument de la fonction **`factorielle_recursive`** est égal à 4. Comme 4 > 1, on appelle à nouveau la fonction **`factorielle_recursive`** avec un argument égal à 4 – 1 = 3. La nouvelle instance de la fonction est placée en haut de la pile d'exécution pendant que l'instance précédente est mise en attente. Comme 3 > 1, on appelle à nouveau la fonction avec un argument égal à 3 – 1 = 2. La nouvelle instance de la fonction est placée en haut de la pile d'exécution pendant que l'instance précédente est mise en attente. Comme 2 > 1, on appelle à nouveau la fonction avec un argument égal à 2 – 1 = 1. La nouvelle instance de la fonction est placée en haut de la pile d'exécution pendant que l'instance précédente est mise en attente. Comme 1 ≤ 1 est une condition vraie, la fonction retourne la valeur 1. On dépile alors la fonction qui est terminée. L'instance précédente de la fonction **`factorielle_recursive`** va donc récupérer cette valeur de retour et retourner à son tour la valeur 2 × 1 = 2. On dépile alors la fonction qui est terminée. L'instance précédente retourne alors la valeur 3 × 2 = 6. On dépile alors la fonction qui est terminée. L'instance précédente (qui est la toute première, celle que l'on a lancée dans la console Python) retourne alors la valeur 4 × 6 = 24.

Le test « **`if n <= 1`** » constitue *la condition de sortie*, nécessaire pour que le programme puisse s'arrêter.

L'intérêt de la version récursive du programme est sa plus grande simplicité. L'inconvénient majeur est que l'exécution du programme récursif va occuper plus de place en mémoire, à cause de la pile d'exécution, et qu'on risque le dépassement de pile pour de très grandes valeurs de n.

*Exemple 2 :* calcul des termes de la suite de Fibonacci (https://fr.wikipedia.org/wiki/Suite_de_Fibonacci).

La suite de Fibonacci est définie par :

- $F_0 = 0$
- $F_1 = 1$
- A partir de $F_2$, chaque terme est égal à la somme des deux termes : $F_2 = F_0 + F_1 = 0 + 1 = 1$
- $F_3 = F_1 + F_2 = 1 + 1 = 2$
- $F_4 = F_2 + F_3 = 1 + 2 = 3$
- $F_5 = F_3 + F_4 = 2 + 3 = 5$
- Etc…

Voici une version non récursive d'une fonction calculant le terme numéro « n » de la suite de Fibonacci :

```
def fibo(n):
    if n <= 1:
        return n
    F0 = 0
    F1 = 1
    for i in range(2, n+1):
        terme_suivant = F0 + F1
        F0 = F1
        F1 = terme_suivant
    return terme_suivant
```

Tapez ce script et vérifiez que la fonction **`fibo(n)`** donne bien les valeurs correctes des termes de la suite de Fibonacci.

Créez une version récursive, que vous appellerez « **`fibo_recursif(n)`** » de cette fonction.