4. La récursivité

4.1 Fonction récursive

4.1.1 Une définition

La définition d'une **fonction récursive** est assez simple. C'est une fonction qui s'appelle elle-même à l'intérieur de sa propre définition !
Vous avez ci-contre un exemple d'une fonction récursive. On peut aisément deviné ce qu'elle produira.

```
1 def inspecteurJuve():
2     print("Mais ce n'est pas moi !")
3     inspecteurJuve()
```

Mais finalement cet exemple n'est pas un bon exemple puisque *en général*, on préfère que le programme s'arrête.
Par défaut, Python limite le nombre d'appels récursifs à 1000. Au-delà, on obtient le message d'erreur suivant `RecursionError : maximum recursion depth exceed.`
Cette limite peut se modifier. Par exemple, si on souhaite la passer à 2000 :

```
1 import sys
2 sys.setrecursionlimit(2000)
```

4.1.2 Un exemple

Cet exemple est inspiré du livre `Numérique et Sciences Informatiques - Terminale` de Balabonski, Conchon, Filliâtre et Nguyen.

Imaginons que l'on veuille effectuer la somme des n premiers entiers :

$$S = 1+2+3+4+...+(n-1)+n$$

Une première solution consiste à ajouter étape après étape tous les entiers jusqu'à n. C'est ce que nous pourrions facilement faire avec une boucle **POUR**.

```
1 def somme(n):
2     calcul = 0
3     for i in range(1,n+1):
4         calcul = calcul + i
5     return calcul
```

On pourrait aussi imaginer le calcul d'une autre manière :
- *Grand Sage, je dois effectuer le calcul suivant* $1+2+3+4+...+(n-1)+n$. *Pourrais-tu m'aider ?*

- Bien sûr Petit Scarabée, fais moi-d'abord le calcul $1+2+3+4+...+(n-1)$ *je rajouterai ensuite n.*
Autrement dit le calcul pourrait se faire ainsi :

$$S = \underbrace{1+2+3+4+...+(n-1)}_{\text{Pour le Petit Scarabée}} \quad \underbrace{+n}_{\text{Pour le Grand Sage}}$$

Lorsque l'on effectue un programme récursif, on va procéder comme le Grand Sage, on ne va pas s'embarrasser de savoir comment on effectue le calcul $1+2+3+4+...+(n-1)$, on va le laisser au Petit Scarabée. On va juste programmer la dernière étape en considérant que le Petit Scarabée a bien effectué son travail [1].

Notre programme pourrait alors s'écrire ainsi :

```
1 def somme(n):
2     return n + somme(n-1)
```

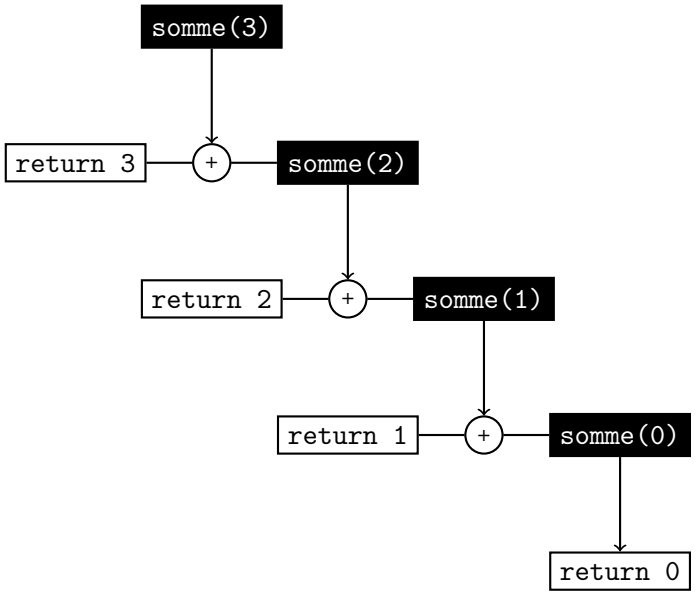
Le problème c'est que ce programme ne va pas donner le résultat attendu !
En effet, si je tape la commande `somme(3)`, le programme va appeler `somme(2)` qui va appeler `somme(1)` qui va lui-même appeler `somme(0)` puis ensuite `somme(-1)` puis encore `somme(-2)`, ...
Les appels récursifs ne vont jamais s'arrêter, comme notre brave fonction `inspecteurJuve()`.
Il nous faut obligatoirement rajouter un cas de base.

Notre programme devient alors :

```
1 def somme(n):
2     if n == 0 :
3         #cas de base
4         return 0
5     else :
6         return n + somme(n-1)
```

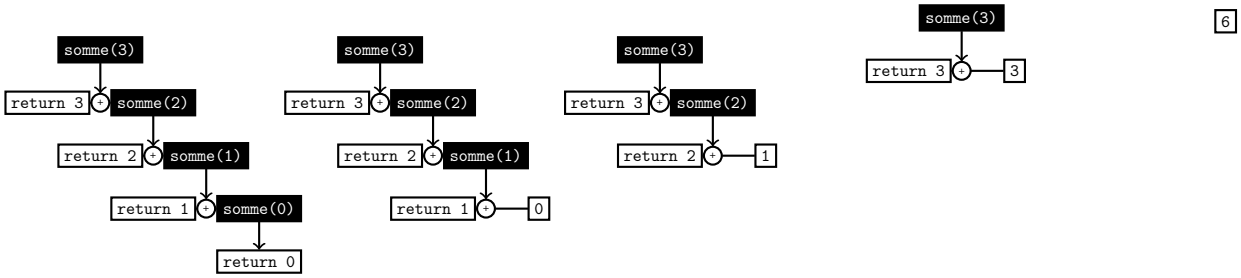
Que se passe-t-il si on tape la commande `somme(3)` ?
Pour calculer la valeur renvoyé par `somme(3)`, il faut d'abord appeler `somme(2)`. Cet appel va lui-même déclencher un appel à `somme(1)` qui à son tour va effectuer un appel à `somme(0)`.
Ce dernier appel se termine directement en renvoyant la valeur 0.

Une fois que la valeur 0 a été renvoyé, l'appel à `somme(0)` est remplacé par 0 dans l'expression `return 1 + somme(0)`.
A cet instant, l'appel à `somme(1)` peut alors se terminer et renvoyer le résultat sous la forme `1+0`.
Enfin, l'appel à `somme(2)` peut lui-même renvoyer la valeur `2+1` comme résultat, ce qui permet à `somme(3)` de se terminer en renvoyant le résultat `3+3=6`.



On parle alors de **pile d'appels**.

1. Je me demande si ce n'est pas un peu trop risqué d'avoir autant confiance dans le travail de notre Petit Scarabée...



On peut retrouver ce phénomène, en ligne, sur Pythontutor : http://pythontutor.com

4.1.3 Définitions récursives bien formées

Il est important de respecter les quelques règles élémentaires suivantes lorsque l'on écrit une fonction récursive :

- s'assurer que les valeurs utilisées pour appeler la fonction soient **strictement décroissantes** au fur et à mesure des appels.
- s'assurer que l'on a bien défini un **cas de base**.
- s'assurer que les valeurs utilisées pour appeler la fonction restent dans le **domaine de définition** de la fonction.

Voici quelques exemples problématiques.

■ **Example 4.1 — 1.** Ligne 5, les valeurs utilisées ne sont pas décroissantes. Le cas de base ne sera alors jamais atteint.
En théorie, on obtient une pile d'appels infinie. ■

```
1 def somme(n):
2     if n==0:
3         return 0
4     else :
5         return n + somme(n+1)
```

■ **Example 4.2 — 2.** Les valeurs utilisées sont bien strictement décroissantes et nous avons bien un cas de base.
Malheureusement, ce cas de base n'est pas toujours atteint. En effet pour les nombres impairs nous avons un problème. Par exemple, pour `somme(1)`, on va faire un appel à `somme(-1)` puis à `somme(-3)`, ...
En théorie, on obtient à nouveau une pile d'appels infinie. ■

```
1 def somme(n):
2     if n==0:
3         return 1
4     else :
5         return n + somme(n-2)
```

■ **Example 4.3 — 3.** La fonction récursive est bien définie mais l'appel de la fonction ne respecte pas le domaine de définition de la fonction. Il n'y aura pas d'arrêts des appels car on passera à côté du cas de base.
■

```
1 def somme(n):
2     if n==0:
3         return 0
4     else :
5         return n + somme(n-1)
6 somme(3.5)
```

Revenons sur ce dernier exemple. Une solution serait de s'assurer que l'on utilise bien la fonction `somme()` avec un entier positif. On pense donc naturellement à utiliser un `assert`.
Cette solution est parfaitement correcte, mais elle pose un petit souci. Une fois le premier appel vérifié, les autres appels se feront bien évidemment sur des nombres entiers positifs. Il n'est plus nécessaire d'effectuer à chaque fois ces tests.

```
def somme(n):
    assert isinstance(n,int)
    assert n>=0
    if n==0:
        return 0
    else :
        return n + somme(n-1)
```

Une solution consiste à effectuer l'assertion dans une fonction principale `somme()` qui lancera, si le teste est réussi, la fonction récursive `somme_rec()`.

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

def somme(n):
    assert isinstance(n,int)
    assert n>=0
    return somme_rec(n)
```

4.2 Exercices

exercice 4.1 — Le critère de divisibilité par 7. Le critère de divisibilité par 7 a été (re) découvert par Chika Ofili un jeune nigérian de 12 ans. On peut parfois le trouver sous le nom de de **critère de Chika**.
« Je lui avais donné un livre intitulé *First Steps for Problem Solvers* à consulter pendant les vacances et à l'intérieur du livre se trouvait une liste des tests de divisibilité [...] Sauf qu'il n'y avait pas de test pour vérifier la divisibilité par 7, car il n'y a pas de test facile ou mémorable pour diviser par 7, c'est du moins ce que je pensais », raconte Mary Ellis, chef du département de Mathématiques de Westminster Under School.
Dans un moment d'ennui, Chika s'est penché sur le problème et l'a résolu.
https://www.agenceecofin.com/communication/1211-71031-un-jeune-nigerian-de-12-ans-decouvre-un
On peut trouver plusieurs vidéos explicatives dont celle-ci :

https://youtu.be/I2Gt4QhbAWU

Par exemple, prenons le nombre 861.
Son chiffre des unités est **1**. On effectue alors l'opération : $86+\mathbf{1}\times 5=91$.
Pour le résultat obtenu, le nouveau chiffre des unités est **1**. On effectue alors l'opération : $9+\mathbf{1}\times 5=14$.
Pour le résultat obtenu, le nouveau chiffre des unités est **4**. On effectue alors l'opération : $1+\mathbf{4}\times 5=21$.
Pour le résultat obtenu, le nouveau chiffre des unités est **1**. On effectue alors l'opération : $2+\mathbf{1}\times 5=7$.
Comme le résultat final est 7, on en déduit que le nombre 861 est bien un multiple de 7.
(On pouvait en réalité s'arrêter dès le nombre 14, car c'était un multiple de 7.)

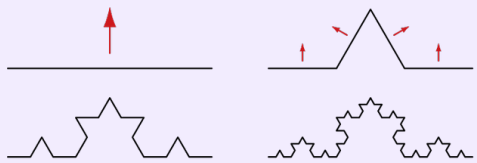
Reprenons avec le nombre 3 191.
Son chiffre des unités est **1**. On effectue alors l'opération : $319+\mathbf{1}\times 5=324$.
Pour le résultat obtenu, le nouveau chiffre des unités est **4**. On effectue alors l'opération : $32+\mathbf{4}\times 5=52$.
Pour le résultat obtenu, le nouveau chiffre des unités est **2**. On effectue alors l'opération : $5+\mathbf{2}\times 5=15$.
Pour le résultat obtenu, le nouveau chiffre des unités est **5**. On effectue alors l'opération : $1+\mathbf{5}\times 5=26$.
Pour le résultat obtenu, le nouveau chiffre des unités est **6**. On effectue alors l'opération : $2+\mathbf{6}\times 5=32$.
Pour le résultat obtenu, le nouveau chiffre des unités est **2**. On effectue alors l'opération : $3+\mathbf{2}\times 5=13$.
Pour le résultat obtenu, le nouveau chiffre des unités est **3**. On effectue alors l'opération : $3+\mathbf{3}\times 5=16$.
Pour le résultat obtenu, le nouveau chiffre des unités est **6**. On effectue alors l'opération : $1+\mathbf{6}\times 5=31$.
Pour le résultat obtenu, le nouveau chiffre des unités est **1**. On effectue alors l'opération : $3+\mathbf{1}\times 5=8$.
Pour le résultat obtenu, le nouveau chiffre des unités est **8**. On effectue alors l'opération : $0+\mathbf{8}\times 5=40$.
Pour le résultat obtenu, le nouveau chiffre des unités est **0**. On effectue alors l'opération : $4+\mathbf{0}\times 5=4$.
Comme 4 est plus petit strictement que 7, on en déduit que 3 191 n'est pas un multiple de 7.
(On pouvait en réalité s'arrêter dès le résultat 15, car ce n'est pas un multiple de 7.)

1. Écrire un algorithme récursif qui permet de vérifier si un nombre est un multiple de 7 par la méthode de Chika.
2. Améliorer votre algorithme pour conclure dès qu'un résultat intermédiaire est inférieur strictement à 100.

■

exercice 4.2 — Critère de divisibilité par 3. Écrire une fonction récursive qui permet de savoir si un nombre entier est divisible par 3. ■

exercice 4.3 Programmer à l'aide du module `turtle` la courbe de Koch[a].

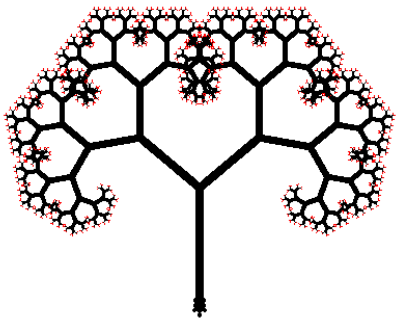


■

a. https://fr.wikipedia.org/wiki/Flocon_de_Koch

Vous pouvez trouvez quelques variantes de la courbe de Koch sur le lien suivant :
https://natureletchaud.github.io/recursivite/.

exercice 4.4 Programmer à l'aide du module `turtle` un arbre récursif. ■



exercice 4.5 — Travail à la main. On considère la fonction ci-contre. Construire la pile d'appels correspondant à la commande `puis(3,5)` puis calculer le résultat obtenu.
Quelle est l'utilité de cette fonction ? ■

```
def puiss(x,n) :
    if n == 0 :
        return 1
    elif n%2 == 0 :
        return puiss(x,n//2)**2
    else :
     return x*puiss(x,(n-1)//2)**2
```

exercice 4.6 — Travail à la main. On considère la fonction ci-contre. Construire la pile d'appels correspondant à la commande `fib(5)` puis calculer le résultat obtenu. ■

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

exercice 4.7 Écrire une fonction qui inverse l'ordre des caractères dans un texte.

1. Version 1, en utilisant une pile.
2. Version 2, en utilisant uniquement une fonction récursive.

■

exercice 4.8 Écrire une fonction récursive qui vérifie qu'un mot ou un texte est bien un palindrome.
On pourra tester le programme avec les palindromes suivants :
https://www.topito.com/top-10-des-palyndromes-bien-nazes-que-tout-le-monde-connait-mais-qui-f
On pourra aussi vérifier le palindrome écrit par Georges Perec en 1969, « Au moulin d'Andé » comportant 1247 mots (c'est le record !) :
https://jeretiens.net/palindrome-de-georges-perec-au-moulin-dande/ ■

exercice 4.9 Écrire une fonction récursive qui indique tous les chemins accessible depuis un répertoire [a].

- On utilisera le module `os` avec la commande classique `import os`.
- `os.path.isdir(chemin)` permet de vérifier si `chemin` est un répertoire valide ou non, autrement dit si c'est un répertoire ou un fichier. La réponse est un booléen.
- `os.listdir(chemin)` donne la liste de tous les sous-répertoires et fichiers contenus dans `chemin`.

■

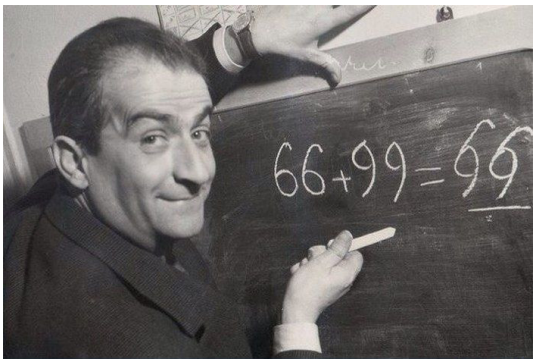
a. Sur une idée de Guillmaume Connan, professeur et formateur de l'académie de Nantes

exercice 4.10 Écrire une fonction récursive qui inverse l'ordre d'une pile. ■

4.3 Mini-projet : le compte est bon

Le **compte est bon** est un jeu proposé dans l'émission « Des chiffres et des lettres ». Le but est le suivant. On choisit au hasard un nombre entre 101 et 999. En parallèles on tire 6 nombres parmi la liste suivante :
`[1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 6, 7, 7, 8, 8, 9, 9, 10, 10, 25, 50, 75, 100]`.
Le but est d'effectuer des opérations élémentaires [2] sur ces 6 nombres pour retrouver le nombre entre 101 et 999 tiré en premier.
Dans certains cas il n'existe aucune solution.

Par exemple, on peut obtenir le nombre 666 avec les nombres 3, 75, 2, 4, 1, 1.
$75 - 1 = 74$
$74 \times 3 = 222$
$4 - 1 = 3$
$222 \times 3 = 666$



Votre travail consiste à programmer un solveur sous la forme d'une fonction récursive qui renverra toutes les solutions possibles.

2. Les opérations élémentaires sont l'addition, la soustraction, la multiplication et la division uniquement si le résultat est entier.