Chap. 1 — Récursivité

1.1 — Problème de la somme des n premiers entiers

Pour définir la somme des n premiers entiers, on utilise généralement la formule $0+1+2+\ldots+n$. Cette formule parait simple mais elle n'est pas évidente à programmer en Python.

ACTIVITÉ

Écrire une fonction `somme(n)` qui renvoie la somme des `n` premiers entiers.

CORRECTION

[1]:
```
# programmation avec tests
# import doctest

def somme(n):
    """
    Calcule la somme des n premiers entiers.
    param : n (int), dernier entier à ajouter

    exemples:
    >>> somme (0)
    0
    >>> somme (5)
    15
    """
    r = 0
    for i in range(n+1):
        r = r + i
    return r

# programmation tests
# doctest.testmod()
```

On remarque que le code Python n'a rien à voir avec sa formulation mathématique.

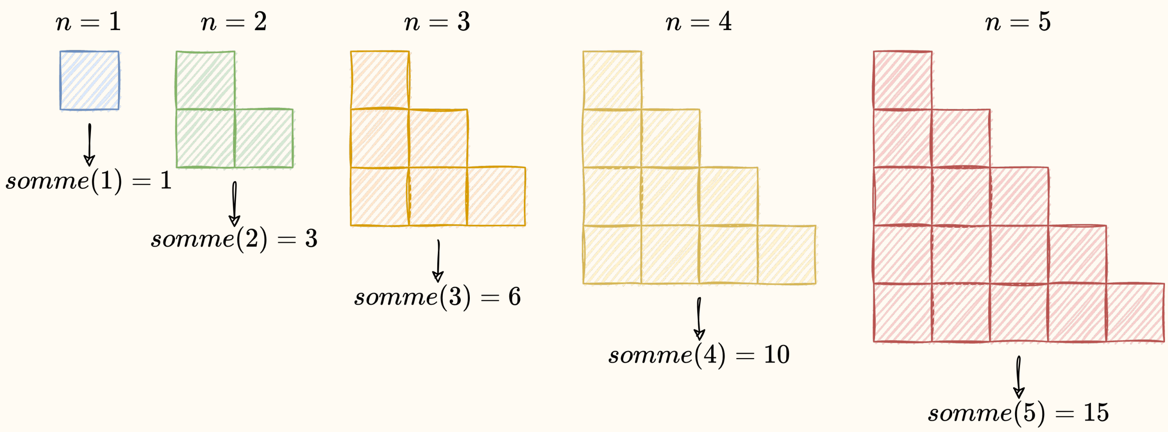
Nouvelle formulation

Il existe une autre manière d'aborder ce problème en définissant une fonction mathématique $somme(n)$.

ACTIVITÉ

Calculer $somme(0)$?

Utilisons maintenant l'illustration ci-dessous pour modéliser quelques exemples de calculs.



En observant ces exemples, **trouver une relation** entre :

- $somme(5)$ et $somme(4)$,
- $somme(4)$ et $somme(3)$.

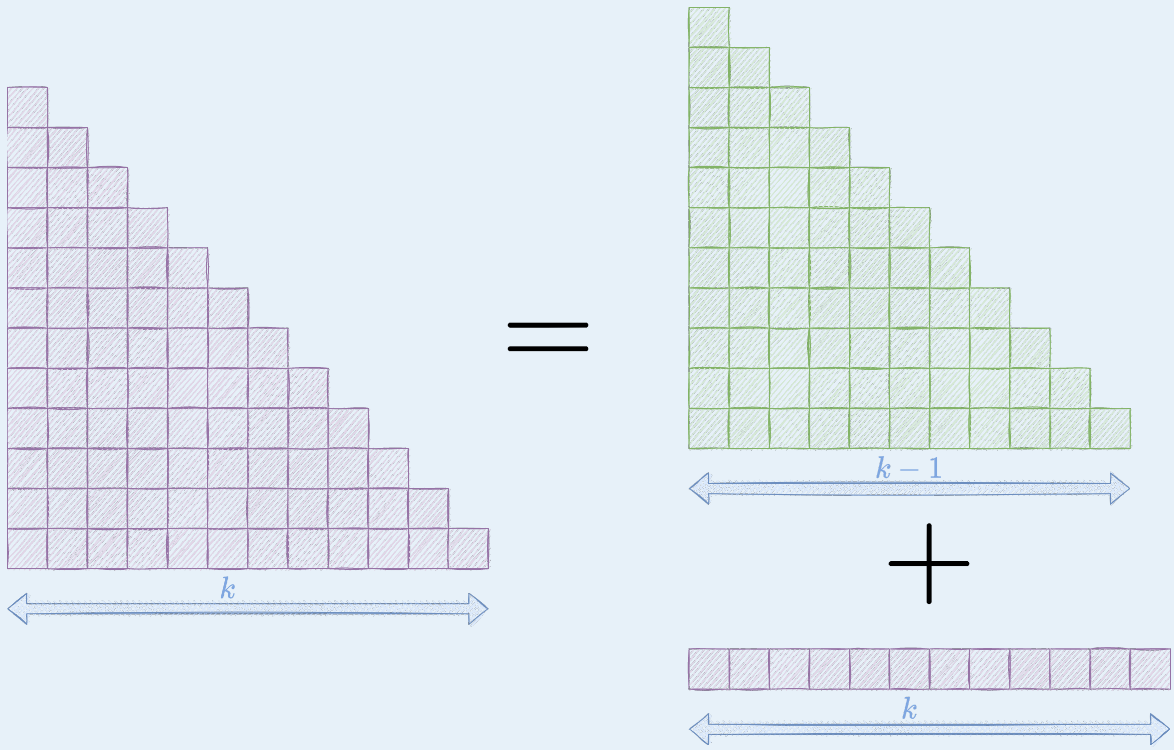
Généraliser la relation entre $somme(n)$ et $somme(n-1)$.

CORRECTION

1. $somme(0) = 0$
2. On obtient :
   - $somme(5) = somme(4) + 5$

- $somme(4) = somme(3) + 4$

3. En s'aidant du schéma



on obtient donc :

$$somme(n) = \begin{cases} 0 & \text{si } n = 0 \\ somme(n-1) + n & \text{si } n > 0 \end{cases}$$

Comme on peut le voir, la définition de $somme(n)$ dépend de la valeur de $somme(n-1)$.

$$somme(n) = \begin{cases} 0 & \text{si } n = 0 \\ somme(n-1) + n & \text{si } n > 0 \end{cases}$$

Il s'agit d'une définition **récursive**, c'est-à-dire d'une définition de fonction qui fait appel à elle-même.

L'intérêt de cette définition récursive de la fonction $somme(n)$ est qu'elle est directement *calculable*, c'est-à-dire exécutable par un ordinateur.

ACTIVITÉ

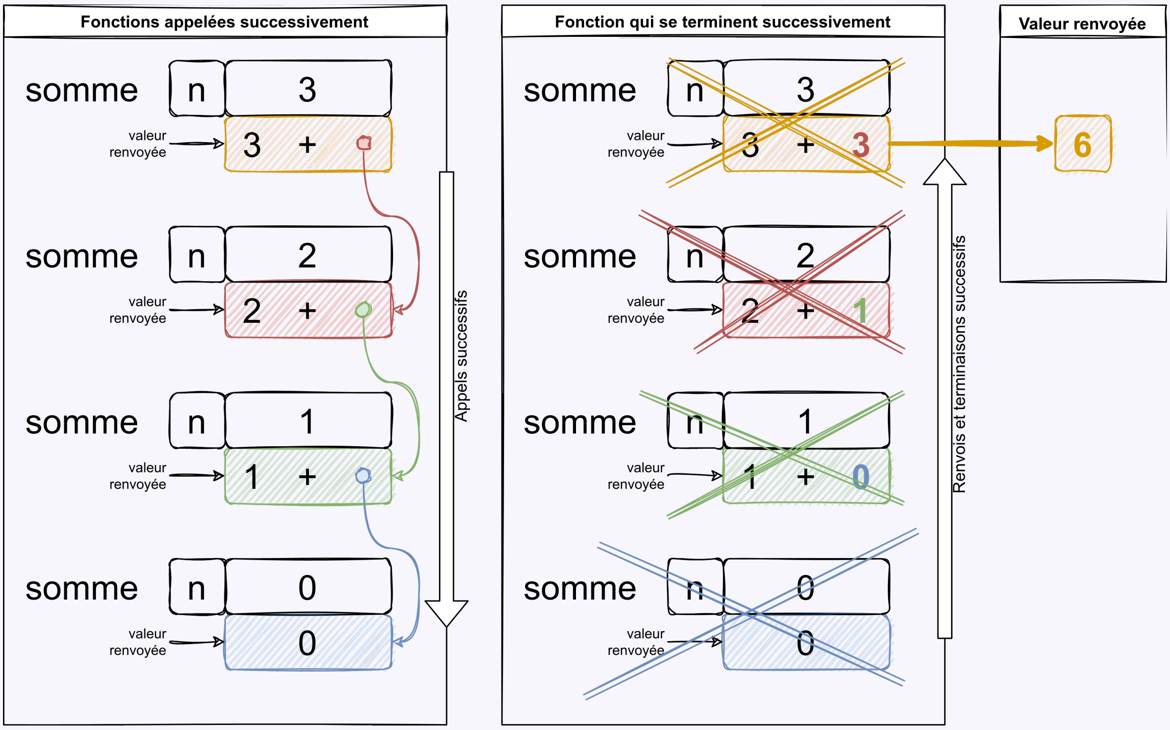
En appliquant exactement la définition récursive de la fonction $somme(n)$, **programmer** une fonction `somme(n)` qui calcule la somme des `n` premiers entiers.

CORRECTION

[2]:
```python
# programmation avec tests
# import doctest

def somme(n):
    """
    Calcule la somme des n premiers entiers.
    params: n (int), dernier entier à ajouter

    exemples:
    >>> somme (0)
    0
    >>> somme(10)
    55
    """
    if n==0:
        return 0
    else:
        return n + somme(n-1)

# programmation avec tests
# doctest.testmod()
```

Exemple

Voici par exemple comment on peut représenter l'évaluation de l'appel à `somme(3)`



Pour calculer la valeur renvoyée par `somme(3)`, il faut d'abord appeler `somme(2)`. Cet appel va lui même déclencher un appel à `somme(1)`, qui a son tour nécessite un appel à `somme(0)`.

Ce dernier se termine directement en renvoyant la valeur 0. `somme(1)` peut alors se terminer et renvoyer le résultat de`1+0`. Enfin, l'appel à `somme(2)` peut lui même se terminer et renvoyer la valeur `2+1`.

Ce qui permet à `somme(3)` de se terminer en renvoyant le résultat `3+3`.

Ainsi on obtient bien la valeur 6 attendue !

1.2 Formulation récursive

Une formulation récursive est constituée par :

- un ou des **cas de base** (on n'a pas besoin d'appeler la fonction)
- des **cas récursifs** (on a besoin d'appeler la fonction)

Les cas de bases sont habituellement les cas de valeurs particulières pour lesquelles il est facile de déterminer le résultat.

Deuxième exemple

ACTIVITÉ

On rappelle que la fonction *puissance* est définie en mathématique par :
$x^n = \underbrace{x \times x \times \ldots \times x}_{n \text{ fois}}$

Déterminer pour la fonction *puissance* :

- un cas de base
- le cas récursif

CORRECTION

Écriture mathématique :

$$x^n = \begin{cases} 1 & \text{si } n = 0 \\ x \times x^{n-1} & \text{si } n > 0 \end{cases}$$

Écriture fonctionnelle :

$$puissance(x, n) = \begin{cases} 1 & \text{si } n = 0 \\ x \times puissance(x, n-1) & \text{si } n > 0 \end{cases}$$

ACTIVITÉ

Implémenter une fonction récursive `puissance(x,n)` de la fonction *puissance*.

CORRECTION

```
[3]: def puissance(x,n):
         """Renvoie x à la puissance x, c'est à dire
         x × x × ... × x (avec n facteurs)

         Args:
             x (int): nombre à multiplier (base)
             n (int): exposant de la puissance

         Returns:
             [int]: x à la puissance n

         Example:
         >>> puissance(2,10)
         1024
         """
         if n == 0:
             return 1
         else:
             return x * puissance(x,n-1)
```

Double cas de base et double récursion

Il peut y avoir plusieurs cas de bases. Il peut aussi y avoir plusieurs récursions, c'est-à-dire plusieurs appels récursif à la fonction.

Exemple

La fonction $fibonacci(n)$ est définie récursivement, pour tout entier n, par :

$$fibonacci(n) = \begin{cases} 0 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ fibonacci(n-2) + fibonacci(n-1) & \text{si } n > 1 \end{cases}$$

Cette formulation récursive possède deux cas de base (pour $n = 0$ et $n = 1$) et une double récursion.

ACTIVITÉ

Déterminer la valeur des 6 premiers termes de la suite de Fibonacci.

Implémenter la fonction récursive `fibonacci(n)` qui renvoie le nième terme de la suite de Fibonacci.

CORRECTION

$$\begin{aligned}
fibonacci(0) &= 0 \\
fibonacci(1) &= 1 \\
fibonacci(2) &= fibonacci(0) + fibonacci(1) = 0+1 = 1 \\
fibonacci(3) &= fibonacci(1) + fibonacci(2) = 1+1 = 2 \\
fibonacci(4) &= fibonacci(2) + fibonacci(3) = 1+2 = 3 \\
fibonacci(5) &= fibonacci(3) + fibonacci(4) = 2+3 = 5
\end{aligned}$$

CORRECTION

[4]:
```
def fibonacci(n):
    """nième terme de la suite de Fibonacci

    Exemples:
    >>> fibonacci(1)
    1
    >>> fibonacci(5)
    5
    """
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n-2) + fibonacci(n-1)
```

1.3 — Activités

ACTIVITÉ

Écrire une fonction récursive `boucle(i,k)` qui affiche les entiers compris entre `i` et `k` inclus. Par exemple, `boucle(0,3)` doit afficher les entiers, `0 1 2 3`.

CORRECTION

[5]:
```
def boucle(i,k):
    """
    Affiche les nombres entiers
    compris entre i et k inclus

    Exemple :
    >>> boucle (0,3)
    0
    1
    2
    3
    """
    if i == k :
        print (k)
    else:
        print (i)
        boucle(i+1,k)
```

ACTIVITÉ

Donner une définition récursive qui correspond au calcul de la fonction factorielle $n!$ définie par :

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ 1 \times 2 \times \ldots \times n & \text{si } n > 0 \end{cases}$$

Donner une fonction `fact(n)` qui implémente cette définition.

CORRECTION

La fonction mathématique est :

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \times (n-1)! & \text{si } n > 0 \end{cases}$$

CORRECTION

[6]:
```python
def fact(n):
    """
    Calcule le n factoriel, c'est-à-dire :
    n × (n-1) × ... × 2 × 1

    exemple:
    >>> fact(0)
    1
    >>> fact(5)
    120
    """
    if n==0:
        return 1
    else:
        return n * fact(n-1)
```

1.4 Définitions bien formées

Il est important de respecter quelques règles élémentaires lorsqu'on écrit une définition récursive.

- vérifier que la récursion **se termine** (grâce au(x) cas de base)
- vérifier que les valeurs utilisées respectent les domaines de définition de la fonction
- vérifier qu'il y a une définition pour toutes les valeurs du domaine

ACTIVITÉ

Relever les problèmes concernant les trois définitions suivantes :

$$f(n) = \begin{cases} 1 & \text{si } n = 0 \\ n + f(n+1) & \text{si } n > 0 \end{cases}$$

$$g(n) = \begin{cases} 1 & \text{si } n = 0 \\ n + g(n-2) & \text{si } n > 0 \end{cases}$$

$$h(n) = \begin{cases} 1 & \text{si } n = 0 \\ n + h(n-1) & \text{si } n > 1 \end{cases}$$

CORRECTION

La définition de f est incorrecte car la valeur $f(n)$, pour tout n strictement positif, ne permet pas d'atteindre le cas de base ($n = 0$). Par exemple $f(1) = 1 + f(2) = 1 + 2 + f(3) = \ldots$

La définition de g s'applique aux *entiers natures*. Mais par exemple la valeur $g(1) = 1 + g(-1)$ et le terme $g(-1)$ n'a aucun sens pour cette définition !

Il manque une valeur de l'ensemble de définition : le nombre 1 n'a pas d'image par la fonction h !

REMARQUE

Les définitions récursives s'appliquent à toute une variété d'objets (et pas uniquement à la définition de fonctions). Nous verrons dans l'année des

définitions récursives de structures de données.

1.5 Programmer avec des fonctions récursives

Quand on programme avec des fonctions récursives, il y a **deux points** importants à vérifier :

- le choix d'une définition récursive plutôt qu'une autre aura une influence sur l'efficacité d'exécution. Jusqu'au dernier appel récursif, la pile d'exécution contient les environnements d'exécutions de **tous** les appels à la fonction récursive. Python limite explicitement le nombre d'appels récursifs dans une fonction. Après 1000 appels, l'exception (= *erreur*) `RecursionError` est levée. Pour passer cette limite à 2000 appels maximums, on exécutera le code Python suivant :

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

- le domaine de définition *mathématique* n'est pas toujours le même que l'ensemble des valeurs du type Python avec lesquelles la fonction Python sera appelée;

Exemple

La fonction *mathématique* $somme(n)$ est définie sur l'ensemble des **entiers naturels**.

Mais comment empêcher d'appeler la fonction Python `somme(n)` avec autre chose qu'un entier naturel? Ainsi, *comment empêcher un appel comme* `somme(-1)` *?*

Pour cela, on utilise les principes de la **programmation défensive** vue en première : on utilise l'instruction `assert n >= 0`. Ainsi, une erreur **sera déclenchée** pour tout appel avec `n < 0`.

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

À retenir

Un calcul peut être décrit à l'aide d'une **définition récursive**. L'écriture d'une **fonction récursive** nécessite de distinguer les **cas de base** (pour lesquels on peut donner un résultat facilement) et les **cas récursifs** (qui font appel à la définition en cours).

Il faut veiller à ce que la fonction Python ne s'applique que sur le **domaine** de la fonction mathématique (utiliser par exemple l'instruction `assert`). Enfin, il faut comprendre le modèle d'exécution des fonctions récursives pour choisir la définition qui **limite** le nombre d'appels récursifs.

1.5 Applications

ACTIVITÉ

Écrire une fonction `nombre_de_chiffre(n)` qui renvoie le nombre de chiffre du nombre entier positif `n`. Par exemple, `nombre_de_chiffre(314159)` devra renvoyer `6`.

CORRECTION

[7]:
```python
def nombre_de_chiffre(n):
    """Nombre de chiffre d'un nombre entier

    Args:
        n (int): nombre à évaluer

    Returns:
        int: nombre de chiffre de n

    Example:
    >>> nombre_de_chiffre(314159)
    6
    """
    if n <= 9:
        return 1
    else:
        return 1 + nombre_de_chiffre(n//10)
```

ACTIVITÉ

Soit u_n la suite d'entiers définie par :

$$u_{n+1} = \begin{cases} \frac{u_n}{2} & \text{si } u_n \text{ est pair,} \\ 3 \times u_n + 1 & \text{sinon.} \end{cases}$$

avec u_0 un entier plus grand que 1.

Écrire une fonction récursive `syracuse(u_n)` qui affiche les valeurs successives de la suite u_n **tant que u_n est plus grand que 1**.

CORRECTION

```
[8]: def syracuse(u_n):
         """
         Affiche les termes de la suite de Syracuse.

         exemple :
         >>> syracuse(5)
         5
         16
         8
         4
         2
         1
         """
         print(u_n)
         if u_n > 1:
             if u_n % 2 == 0:
                 syracuse(u_n//2)
             else:
                 syracuse(3*u_n+1)
```

REMARQUE

La conjecture de Syracuse affirme que, quelle que soit la valeur de u_0, il existe toujours un indice n dans la suite tel que $u_n = 1$. Cette conjecture défie toujours les mathématiciens.

ACTIVITÉ

En appelant $carre(x)$ la fonction qui a x associe $x \times x$, on peut utiliser une autre définition de la fonction mathématique $puissance(x, n)$:

$$puissance(x,n) = \begin{cases} 1 & \text{si } n = 0 \\ x & \text{si } n = 1 \\ carre\left(puissance(x, \frac{n}{2})\right) & \text{si } n > 1 \text{ et } n \text{ est pair} \\ x \times carre\left(puissance(x, \frac{n-1}{2})\right) & \text{si } x > 1 \text{ et } n \text{ est impair} \end{cases}$$

Combien d'appels récursifs engendre l'appel `puissance(7,28)` ? **Comparer** à la fonction `puissance(x,n)` vue dans le cours.

Implémenter la fonction `carre(n)` puis, en suivant cette définition, la fonction `puissance(x,n)`.

Rappel : le test de parité est réalisé par un test à zéro du reste de la division entière par 2 (soit `r % 2 == 0`).

Correction

`puissance(7,28)` $\rightarrow$ `puissance(7,14)` $\rightarrow$ `puissance(7,7)` $\rightarrow$ `puissance(7,3)` $\rightarrow$ `puissance(7,1)` $\rightarrow$ `return 7`

Il faut donc 1 appel initial et 4 appels récursifs. Pour la fonction $puissance(x,n)$ initiale, il faudrait 1 appel initial et 27 appels récursifs.

De manière générale le nombre d'appel récursif est lié au nombre $\log_2(n)$ où $\log_2$ est la fonction logarithme de base 2.

Ici, il faut $1 + \lfloor \log_2(n) \rfloor$ appels. Ainsi, le calcul de `puissance(x,1000)` ne nécessite que $1 + \lfloor \log_2(1000) \rfloor = 10$ appels !

Écrire une version récursive d'une fonction qui renvoie le nombre de bits égaux à 1 d'un entier strictement positif (par ex. `nombre_de_bit(255)` doit renvoyer `5`).

John McCarthy a inventé la fonction $f_{91}(n)$ définie par :

$$f_{91}(n) = \begin{cases} n - 10 & \text{si } n > 100, \\ f_{91}(f_{91}(n+11)) & \text{si } n \leq 100. \end{cases}$$

Implémenter cette fonction et **donner** un tableau de valeurs de $f_{91}(n)$ pour $n \in [\![0..100]\!]$.