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.

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

CORRECTION

[13]:
```
# programmation défensive
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 défensive
doctest.testmod()
```

[13]:
```
TestResults(failed=0, attempted=6)
```

On remarque que le code python n'a rien à voir avec sa formulation mathéma-

tique.

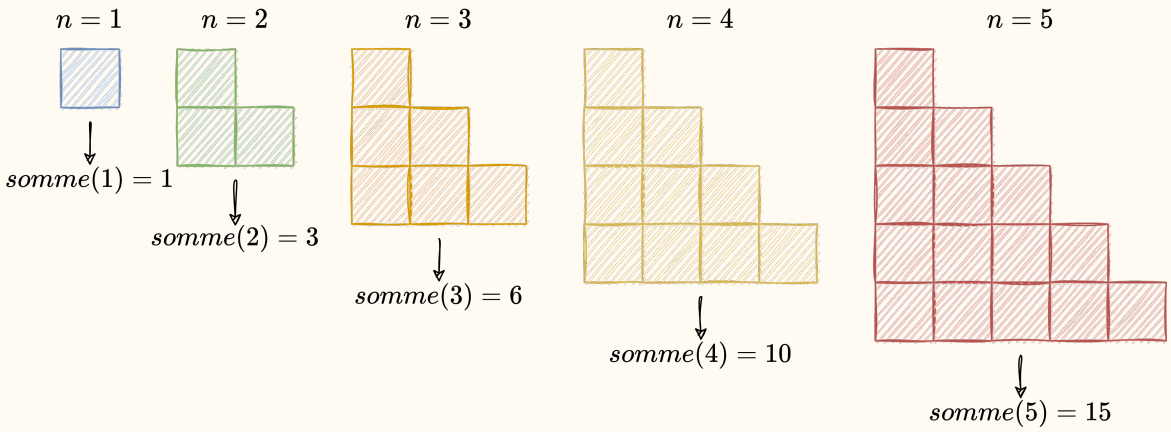
Nouvelle formulation

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 :

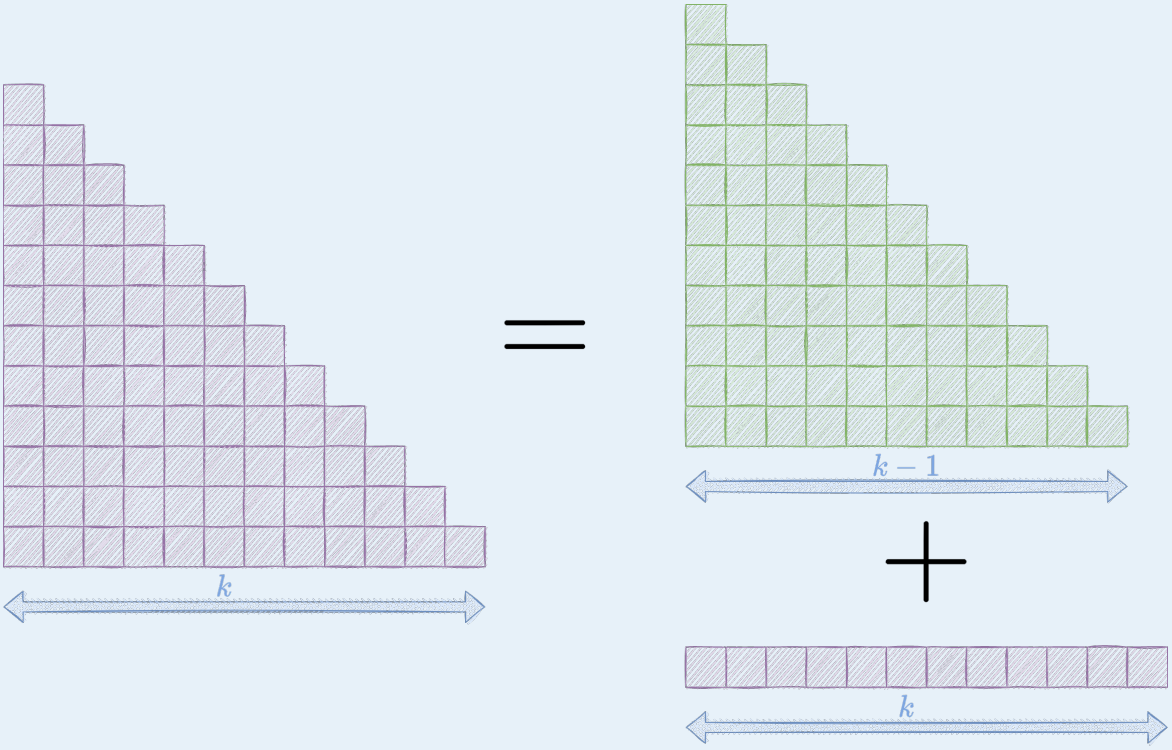
- $somme(5)$ et $somme(4)$,
- $somme(4)$ et $somme(3)$.

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 $somme(n-1)$.

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.

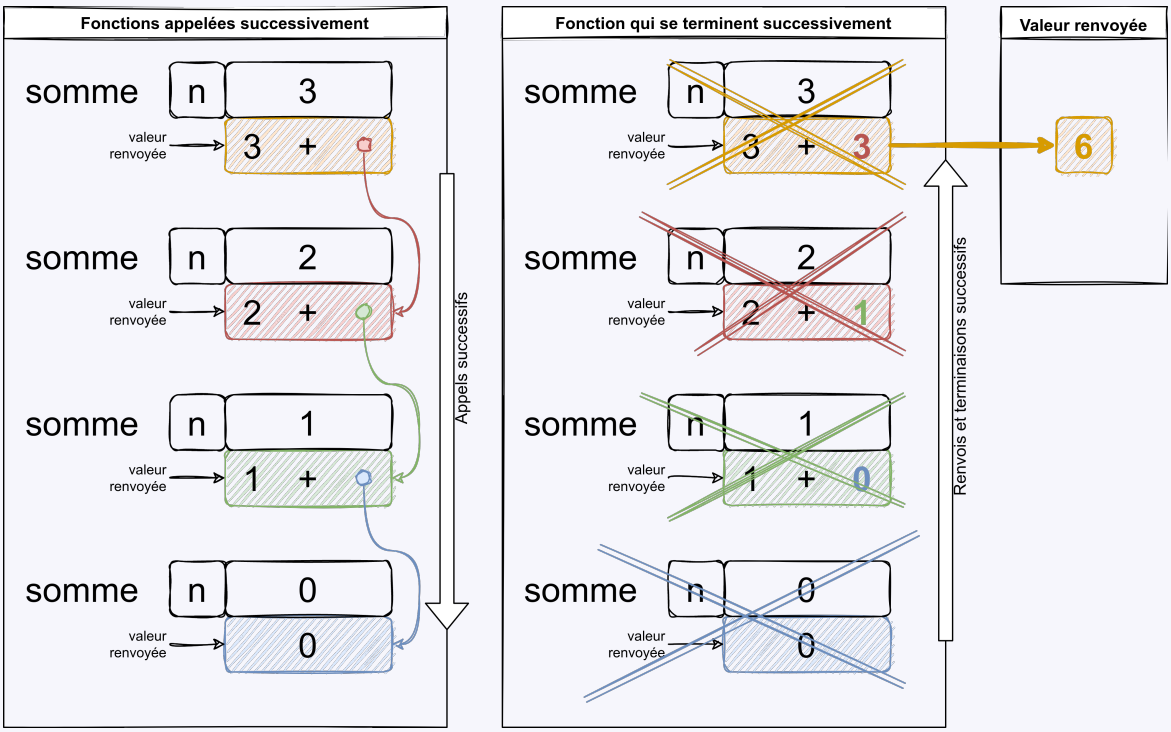
CORRECTION

[14]:
```
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 défensive
doctest.testmod()
```

[14]:
```
TestResults(failed=0, attempted=6)
```

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*.

[22]:
```
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)

doctest.testmod()
```

2]: `TestResults(failed=0, attempted=8)`

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

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

[26]:
```python
def fibonacci(n):
    """nième terme de la suite de Fibonacci

    Args:
        n (int): rang du terme à calculer

    Returns:
        int: nième terme

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

```python
doctest.testmod()
```

6]: TestResults(failed=0, attempted=10)

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

[15]:
```
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)

# programmation défensive
doctest.testmod()
```

[15]: TestResults(failed=0, attempted=6)

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

[17]:
```
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)

# programmation défensive
doctest.testmod()
```

[17]:
```
TestResults(failed=0, attempted=6)
```

1.4 Définitions bien formées

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

#TODO

1.5 Applications

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.

[18]:
```
def nombre_de_chiffre(n):
    """Renvoie le 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)

# programmation défensive
doctest.testmod()
```

8]: `TestResults(failed=0, attempted=6)`

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**.

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.

CORRECTION

[19]:
```
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)

# programmation défensive
doctest.testmod()
```

[19]:
```
TestResults(failed=0, attempted=7)
```