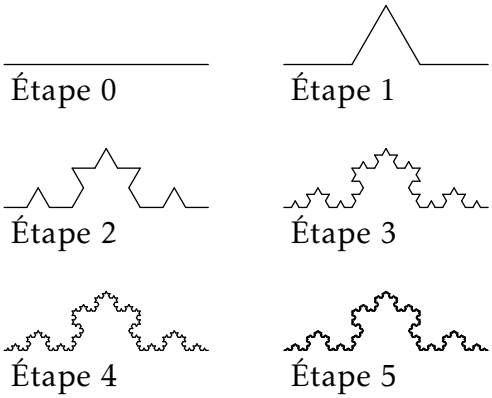
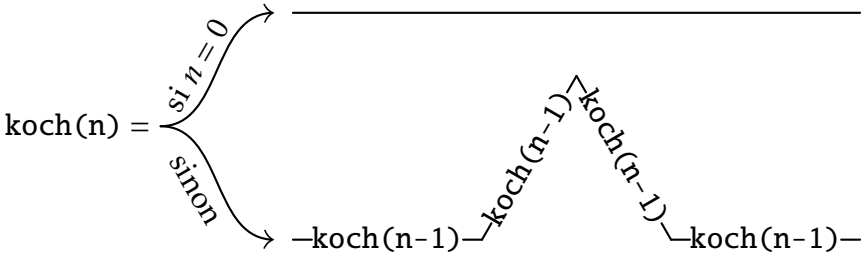
Récursivité

Les fractales

Une **fractale** est une figure géométrique qui se répète indéfiniment, peu importe à quel niveau de zoom on la regarde. La figure ci-contre est une des figures les plus connues, appelée la courbe de Koch, du nom du mathématicien suédois qui l'a inventée en 1904. La fractale est obtenue lorsqu'on répète les étapes à l'infini. Si on zoom sur n'importe quelle branche, on obtient exactement la même figure.
Appelons `koch(n)` la figure obtenue à l'étape `n`. Afin de décrire comment la dessiner, les plus simple est d'utiliser la méthode présentée ci-dessous :





Si `n` vaut 0, on fait un segment. Sinon, on dessine 4 fois `koch(n-1)`, eux-même obtenus en utilisant la même méthode.
On appelle cela une **définition récursive**, c'est-à-dire qu'on définit une valeur initiale et ensuite un processus permettant de construire une nouvelle valeur à partir des précédentes. Ce genre de définition est très utilisé dans la plupart des langages de programmation, comme c'est le cas avec Python.

La récursivité

Considérons la fonction `somme(n)` qui renvoie la somme des `n` premiers nombres entiers. Mathématiquement, on peut décrire cette fonction en utilisant l'égalité :

$$\texttt{somme(n)} = 0 + 1 + \cdots + \texttt{n}$$

```
def somme(n):
    res = 0
    for i in range(n+1):
        res += i
    return res
```

Cette définition mathématique correspond à la définition de la fonction Python ci-dessus. Bien que la nécessité de la variable `i` soit sous-entendue, rien n'indique qu'il faille en plus utiliser une variable pour stocker les résultats intermédiaires. Le passage d'une définition à l'autre n'est donc pas si évidente que cela. Cette fonction peut aussi se définir de façon récursive :

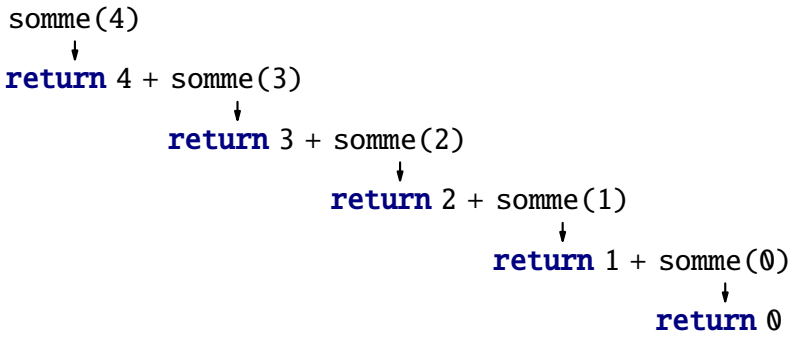
$$\texttt{somme}(n) = \begin{cases} 0 & \text{si } n = 0 \\ n + \texttt{somme}(n-1) & \text{si } n > 0 \end{cases}$$

Cette définition se traduit de façon bien plus directe en Python. On remarque que la fonction s'appelle elle-même. C'est ce que l'on appelle une **fonction récursive**.

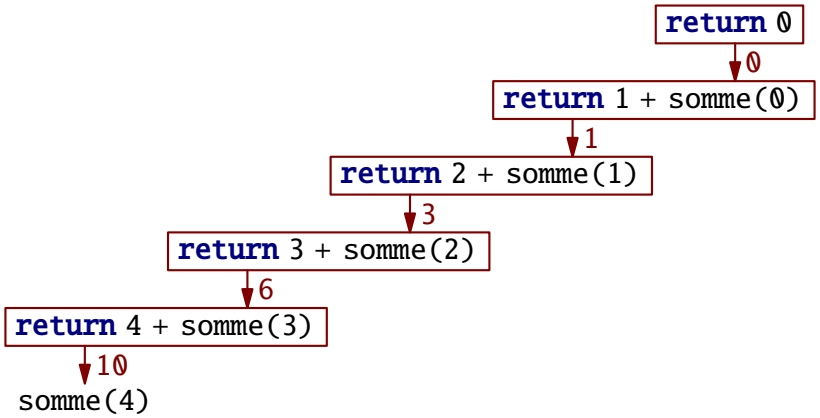
```
def somme(n):
    if n == 0:
        return 0
    else:
        return n + somme(n-1)
```

Arbre d'appel

S'il est assez aisé de dérouler l'exécution de la définition avec une boucle, comment représenter l'exécution de la fonction ? On peut remplacer l'appel de la fonction par le **return** qui sera obtenu, et continuer ainsi jusqu'à arriver au cas 0. Voici l'**arbre d'appel** de `somme(4)` :



Une fois arrivé à un cas de base, on remonte les valeurs obtenues au fur et à mesure.



La variable `res` de la version itérative correspond donc aux résultats successifs renvoyés par la version récursive. D'une certaine manière, la boucle calcule les résultats intermédiaires, en partant de 0. Cela revient à faire directement la remontée des valeurs, comme c'est le cas dans l'exemple ci-contre.

$$\begin{aligned}
&\texttt{somme}(0) = 0\\
&\texttt{somme}(1) = 1 + \texttt{somme}(0) = 1\\
&\texttt{somme}(2) = 2 + \texttt{somme}(1) = 3\\
&\texttt{somme}(3) = 3 + \texttt{somme}(2) = 6\\
&\texttt{somme}(4) = 4 + \texttt{somme}(3) = 10
\end{aligned}$$

Dans un cas d'une fonction où il n'y a qu'un seul appel récurisif à la fois, les deux approches sont très similaires en terme d'efficacité. Par contre, lorsqu'il y a plusieurs appels récurisifs, il se peut que les méthodes ne soient pas équivalentes, à moins d'utiliser des optimisations que nous verrons plus tard.

D'autres fonctions comportent plusieurs appels récursifs, comme pour la suite de Fibonacci, du nom du mathématicien italien l'ayant inventée au 13^{e} siècle. Elle est définie de la manière ci-contre.

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

Exercice 1 :

1) Déterminez de la manière de votre choix la valeur de `fibo`(10).
2) Quel est le problème si on essaie de calculer `fibo`(100) à l'aide d'une fonction récursive ?

Il existe des constructions encore plus élaborée, comme celle de la fonction f_{91} inventée par John McCarthy, détendeur du prix Turing en 1971.

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

Pour calculer une valeur obtenue par cette fonction, il n'est pas possible de calculer toutes les valeurs en partant de 0. Dans ce cas, l'utilisation de la récursivité est inévitable.

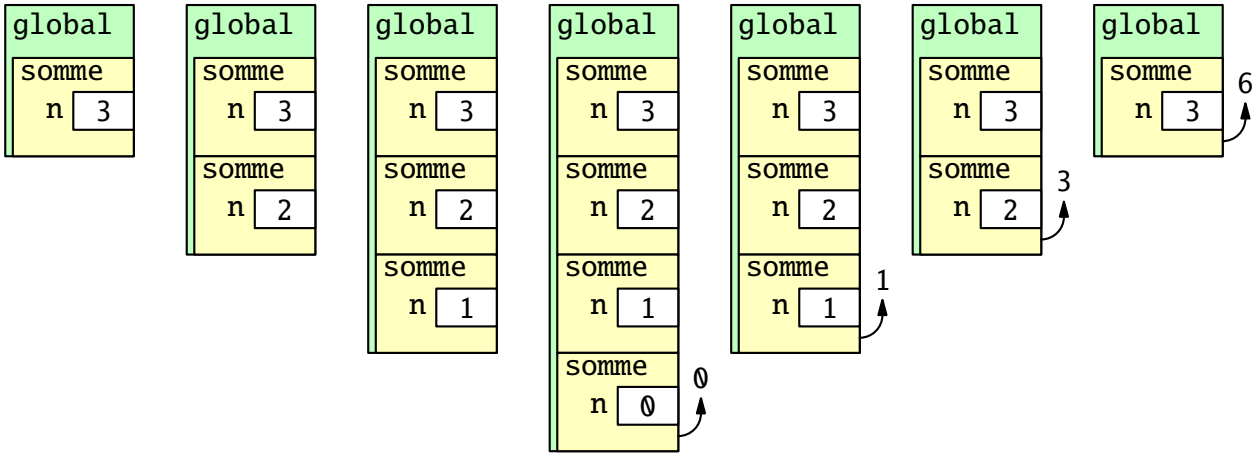
$$
\begin{aligned}
f_{91}(99) &= f_{91}(f_{91}(110)) && \text{puisque } 99 \leq 100\\
&= f_{91}(100) && \text{puisque } 110 > 100\\
&= f_{91}(f_{91}(111)) && \text{puisque } 100 \leq 100\\
&= f_{91}(101) && \text{puisque } 111 > 100\\
&= 91 && \text{puisque } 101 > 100
\end{aligned}
$$

Exercice 2 :

1) Calculer $f_{91}(98)$ et $f_{91}(97)$.
2) Conjecturer la valeur de $f_{91}(n)$ pour tout $n \leq 100$.

Exécution d'une fonction récursive

Comme nous l'avons déjà vu l'année dernière, lorsqu'on appelle une fonction avec Python, une partie de la mémoire est utilisée pour stocker les valeurs des paramètres et variables locales de cette fonction. Lors d'un appel récursif, un nouvel appel de la fonction est réalisé avant que le premier ne soit terminé. Cela se traduit par une nouvelle instance en mémoire avec ses propres variables locales. Dès qu'un appel est terminé, cette instance est détruite et l'instance précédente poursuit son exécution, jusqu'à ce que la première instance ait également fini son exécution.
Le schéma ci-dessous correspond à l'utilisation de la mémoire lors de l'appel de `somme(3)`.



On peut voir que lorsque la valeur de `n` augmente, la taille en mémoire va également augmenter. C'est un des inconvénients de l'approche récursive. La simplicité du code est contrebalancée par une utilisation de la mémoire bien plus grande. Certains langages de programmation spécialisés, comme les langages fonctionnels, sont capables de réduire le nombre d'appels nécessaires dans certains cas. Python ne possède pas de tels mécanismes. Par contre afin d'éviter que l'exécution d'un programme n'utilise toute la mémoire de l'ordinateur, le nombre d'appels récursif est limité à 1000. Au delà, l'exécution est arrêtée avec un message d'erreur. Il est possible d'augmenter cette limite, mais pour la plupart des fonctions que nous considérerons, nous ne devrions pas avoir à nous préoccuper de cette limite.
Comme pour les boucles non bornées, la terminaison d'un appel à une fonction récursive est très importante, mais n'est pas garantie. Pour les fonctions `somme` ou `fibo`, il n'est pas difficile de se convaincre qu'elles se terminent puisqu'elle peuvent être traduite par des boucles **`for`**. Pour la fonction f_{91}, cette terminaison est plus difficile à prouver. Nous ne ferons pas cette preuve, mais nous pouvons étudier les fonctions suivantes pour essayer de comprendre pourquoi elles ne se terminent pas dans certain cas. On parle alors de **définition incorrecte.**

Exercice 3 : Expliquer pourquoi les définitions suivantes sont incorrectes :

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

Lorsqu'on programme une fonction récursive, il faut penser que les valeurs possibles des variables en Python sont parfois plus larges que celles de la définition mathématique. Ainsi, la notation n désigne généralement un entier naturel, mais en Python, rien n'empêche cette valeur d'être négative. Il peut donc être utile de rajouter un `assert` en début de fonction ou de remplacer des tests du type `n == 0` par `n <= 0`.

Exponentiation rapide

Il n'existe pas toujours qu'une seule définition récursive pour une fonction. Parfois, on peut proposer plusieurs approches, dont certaines peuvent se montrer bien plus intéressantes et efficaces une fois programmées. C'est le cas du calcul de la puissance d'un nombre.

La définition récursive "naturelle" est : $\texttt{puiss}(x,n) = \begin{cases} 1 & \text{si } n = 0 \\ x \times \texttt{puiss}(x,n-1) & \text{si } n > 0 \end{cases}$

Cela revient à faire k mutliplications, ce qui n'est pas vraiment optimal. Au contraire, on peut utiliser les propriétés suivantes $x^{2k} = \left(x^2\right)^k$ et $x^{k+1} = x \times x^k$.

Cela permet d'écrire la définition suivante : $\texttt{puiss}(x,n) = \begin{cases} 1 & \text{si } n = 0 \\ \texttt{puiss}(x^2,n/2) & \text{si } n \text{ est pair} \\ x \times \texttt{puiss}(x,n-1) & \text{si } n \text{ est impair} \end{cases}$

Le nombre de multiplications nécessaires pour calculer la puissance est alors de l'ordre de $\log_2(n)$, ce qui est nettement plus rapide que la méthode naturelle. Cette méthode est très utilisée en cryptographie moderne où la plupart des calculs se font sur des puissances.

Cet exemple montre également que la récursion ne se fait pas uniquement selon un unique paramètre mais peut en utiliser plusieurs.

Fonctions mutuellement récursives

Des fonctions peuvent être mutuellement récursives, en s'appelant l'une l'autre. C'est le cas des fonctions a et b inventées par Douglas Hofstadter un livre qui lui a valu le prix Pulitzer en 1980. Les définitions sont les suivantes :

$$a(n) = \begin{cases} 1 & \text{si } n = 0 \\ n - b(a(n-1)) & \text{si } n > 0 \end{cases} \qquad b(n) = \begin{cases} 0 & \text{si } n = 0 \\ n - a(b(n-1)) & \text{si } n > 0 \end{cases}$$

Les premières valeurs obtenues sont données ci-contre. On peut remarquer que les valeurs sont presques toutes identiques.

n	0	1	2	3	4	5	6	7	8	9	…
$a(n)$	1	1	2	2	3	3	4	5	5	6	…
$b(n)$	0	0	1	2	2	3	4	4	5	6	…

En faite $a(n) \neq b(n)$ si et seulement si $n+1$ apparaît dans la suite de Fibonacci.

En conclusion

La récursivité permet d'écrire des fonctions de façon plus simple en se rapprochant de leur définition mathématique et les rendant ainsi plus lisibles. Cela permet généralement de ne pas avoir à rajouter de variables intermédiaires, au prix d'une utilisation de la mémoire plus importante.

Lorsqu'on écrit une fonction récursive, il faut bien s'assurer que tous les cas de base ont été traités afin de garantir la terminaison de la récursion.

Dans certains cas, l'utilisation de la récursivité se fait au détriment de l'efficacité et au contraire, dans d'autres cas, elle est nécessaire pour pouvoir programmer certaines fonctions pour lesquelles une approche itérative serait bien plus compliquée.