Chapitre 1

Récursivité

Notions introduites

- définitions récursives
- programmation avec fonctions récursives
- arbre d'appels
- modèle d'exécution et pile d'appels

On aborde dans ce chapitre la programmation à l'aide de *fonctions récursives*. Il s'agit à la fois d'un style de programmation mais également d'une technique pour définir des concepts et résoudre certains problèmes qu'il n'est parfois pas facile de traiter en programmant uniquement avec des boucles.

1.1 Problème : la somme des n premiers entiers

Pour définir la somme des n premiers entiers, on a l'habitude d'écrire la formule suivante [1] :

$$0 + 1 + 2 + \cdots + n \tag{1.1}$$

Bien que cette formule nous paraisse simple et intuitive, il n'est pas si évident de l'utiliser pour écrire en Python une fonction `somme(n)` qui renvoie la somme des `n` premiers entiers. L'une des difficultés est de trouver un moyen de programmer la répétition des calculs qui est représentée par la notation $+ \cdots +$.

Une solution pour calculer cette somme consiste à utiliser une boucle **for** pour parcourir tous les entiers `i` entre `0` et `n`, en s'aidant d'une variable locale `r` pour accumuler la somme des entiers de `0` à `i`. On obtient par exemple le code Python suivant :

1. Cet exemple est inspiré du cours *Programmation récursive* de Christian Queinnec.

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

S'il n'est pas difficile de se convaincre que la fonction `somme(n)` ci-dessus calcule bien la somme des n premiers entiers, on peut néanmoins remarquer que ce code Python n'est pas *directement* lié à la formule (1.1).

En effet, il n'y a rien dans cette formule qui puisse laisser deviner qu'une variable intermédiaire `r` est nécessaire pour calculer cette somme. Certains peuvent y voir l'art subtil de la programmation, d'autres peuvent se demander s'il ne serait pas possible de donner une définition mathématique plus précise à cette somme, à partir de laquelle il serait plus « simple » d'écrire un programme Python.

Il existe en effet une autre manière d'aborder ce problème. Il s'agit de définir une fonction mathématique $somme(n)$ qui, pour tout entier naturel n, donne la somme des n premiers entiers de la manière suivante :

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

Cette définition nous indique ce que vaut $somme(n)$ pour un entier n quelconque, selon que n soit égal à 0 ou strictement positif. Ainsi, pour $n = 0$, la valeur de $somme(0)$ est simplement 0. Dans le cas où n est strictement positif, la valeur de $somme(n)$ est $n + somme(n-1)$.

Par exemple, voici ci-dessous les valeurs de $somme(n)$, pour n valant 0, 1, 2 et 3.

$$\begin{array}{lcllcllcl} somme(0) & = & 0 & & & & & & \\ somme(1) & = & 1 + somme(0) & = & 1 + 0 & = & 1 & & \\ somme(2) & = & 2 + somme(1) & = & 2 + 1 & = & 3 & & \\ somme(3) & = & 3 + somme(2) & = & 3 + 3 & = & 6 & & \end{array}$$

Comme on peut le voir, la définition de $somme(n)$ dépend de la valeur de $somme(n-1)$. Il s'agit là d'une définition *récursive*, c'est-à-dire d'une définition de fonction qui fait appel à elle-même. Ainsi, pour connaître la valeur de $somme(n)$, il faut connaître la valeur de $somme(n-1)$, donc connaître la valeur de $somme(n-2)$, etc. Ceci jusqu'à la valeur de $somme(0)$ qui ne dépend de rien et vaut 0. La valeur de $somme(n)$ s'obtient en ajoutant toutes ces valeurs.

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. En particulier, cette définition est directement programmable en Python, comme le montre le code ci-dessous.

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

L'analyse par cas de la définition récursive est ici réalisée par une instruction conditionnelle pour tester si l'argument `n` est égal à `0`. Si c'est le cas, la fonction renvoie la valeur `0`, sinon elle renvoie la somme `n + somme(n - 1)`. Cet appel à `somme(n - 1)` dans le corps de la fonction est un *appel récursif*, c'est-à-dire un appel qui fait référence à la fonction que l'on est en train de définir. On dit de toute fonction qui contient un appel récursif que c'est une *fonction récursive*.

Par exemple, l'évaluation de l'appel à `somme(3)` peut se représenter de la manière suivante

```
somme(3) = return 3 + somme(2)
                        |
                   return 2 + somme(1)
                                |
                           return 1 + somme(0)
                                         |
                                      return 0
```

où on indique uniquement pour chaque appel à `somme(n)` l'instruction qui est exécutée après le test `n == 0` de la conditionnelle. Cette manière de représenter l'exécution d'un programme en indiquant les différents appels effectués est appelée un *arbre d'appels*.

Ainsi, pour calculer la valeur renvoyée par `somme(3)`, il faut tout d'abord appeler `somme(2)`. Cet appel va lui-même déclencher un appel à `somme(1)`, qui à son tour nécessite un appel à `somme(0)`. Ce dernier appel se termine directement en renvoyant la valeur `0`. Le calcul de `somme(3)` se fait donc « à rebours ». Une fois que l'appel à `somme(0)` est terminé, c'est-à-dire que la valeur `0` a été renvoyée, l'arbre d'appels a la forme suivante, où l'appel à `somme(0)` a été remplacé par `0` dans l'expression `return 1 + somme(0)`.

```
somme(3) = return 3 + somme(2)
                        |
                   return 2 + somme(1)
                                |
                           return 1 + 0
```

À cet instant, l'appel à `somme(1)` peut alors se terminer et renvoyer le résultat de la somme `1 + 0`. L'arbre d'appels est alors le suivant.

```
somme(3) = return 3 + somme(2)
                        |
                   return 2 + 1
```

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 de `3 + 3`.

`somme(3)` = `return 3 + 3`

On obtient bien au final la valeur `6` attendue.

Une notation ambiguë. La formule (1.1) est non seulement éloignée du programme qui la calcule, elle est également ambiguë quant à la spécification de son résultat. À lire cette formule à la lettre, comment savoir si la somme des premiers entiers pour $n = 2$ est $0 + 1 + 2$ ou $0 + 1 + 2 + 2$? Si la réponse à cette question peut sembler évidente, elle ne l'est que parce que nous avons l'habitude de la signification de $+ \cdots +$ et savons que les entiers (0, 1 et 2) dans cette formule sont déjà des instances de n.
De manière générale, la programmation imposant la précision, nous sommes souvent amenés comme ici à considérer avec une rigueur nouvelle certaines choses « évidentes ».

1.2 Formulations récursives

Une formulation récursive d'une fonction est toujours constituée de plusieurs cas, parmi lesquels on distingue des *cas de base* et des *cas récursifs* du calcul.

Les cas récursifs sont ceux qui renvoient à la fonction en train d'être définie ($somme(n) = n + somme(n-1)$ si $n > 0$). Les cas de base de la définition sont à l'inverse ceux pour lesquels on peut obtenir le résultat sans avoir recours à la fonction définie elle-même ($somme(n) = 0$ si $n = 0$). Ces cas de base sont habituellement les cas de valeurs particulières pour lesquelles il est facile de déterminer le résultat.

Prenons comme deuxième exemple l'opération de puissance n-ième d'un nombre x, c'est-à-dire la multiplication répétée n fois de x avec lui-même, que l'on écrit habituellement de la manière suivante

$$x^n = \underbrace{x \times \cdots \times x}_{n \text{ fois}}$$

avec, par convention, que la puissance de x pour $n = 0$ vaut 1.

Pour écrire une version récursive de x^n, on va définir une fonction $puissance(x, n)$ en commençant par chercher les cas de base à cette opération. Ici, le cas de base évident est celui pour $n = 0$. On écrira donc la définition (partielle) suivante :

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

Pour définir la valeur de $puissance(x,n)$ pour un entier n strictement positif, on suppose que l'on connaît le résultat de x à la puissance $n-1$, c'est-à-dire la valeur de $puissance(x,n-1)$. Dans ce cas, $puissance(x,n)$ peut simplement être définie par $x \times puissance(x,n-1)$. Au final, on obtient donc la définition suivante :

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

Faire confiance à la récursion. Pour faciliter l'écriture des cas récursifs, il est très important de supposer que les appels récursifs donnent les bons résultats pour les valeurs sur lesquelles ils opèrent, sans chercher à « construire » dans sa tête l'arbre des appels pour se convaincre du bien fondé de la définition.

Définitions récursives plus riches

Toute formulation récursive d'une fonction possède au moins un cas de base et un cas récursif. Ceci étant posé, une grande variété de formes est possible.

Cas de base multiples. La définition de la fonction $puissance(x,n)$ n'est pas unique. On peut par exemple identifier deux cas de base « faciles », celui pour $n = 0$ mais également celui pour $n = 1$ avec $puissance(x,1) = x$. Ce deuxième cas de base a l'avantage d'éviter de faire la multiplication (inutile) $x \times 1$ de la définition précédente. Ainsi, on obtient la définition suivante avec deux cas de base :

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

Bien sûr, on pourrait continuer à ajouter des cas de base pour $n = 2$, $n = 3$, etc., mais cela n'apporterait rien à la définition. En particulier, cela ne réduirait pas le nombre de multiplications à effectuer.

Cas récursifs multiples. Il est également possible de définir une fonction avec plusieurs cas récursifs. Par exemple, on peut donner une autre définition pour $puissance(x, n)$ en distinguant deux cas récursifs selon la parité de n. En effet, si n est pair, on a alors $x^n = (x^{n/2})^2$. De même, si n est impair, on a alors $x^n = x \times (x^{(n-1)/2})^2$, où l'opération de division est supposée ici être la division entière. Ceci nous amène à définir la fonction $puissance(x, n)$ de la manière suivante, en supposant que l'on dispose d'une fonction $carre(x) = x \times x$.

$$puissance(x, n) = \begin{cases} 1 & \text{si } n = 0, \\ carre(puissance(x, n/2)) & \text{si } n \geq 1 \text{ et } n \text{ est pair}, \\ x \times carre(puissance(x, (n-1)/2)) & \text{si } n \geq 1 \text{ et } n \text{ est impair}. \end{cases}$$

Pour des raisons dont nous discuterons un peu plus loin, cette définition va nous permettre d'implémenter en Python une version plus efficace de la fonction *puissance*.

Double récursion. Les expressions qui définissent une fonction peuvent aussi dépendre de *plusieurs* appels à la fonction en cours de définition. Par exemple, la fonction $fibonacci(n)$, qui doit son nom au mathématicien Leonardo Fibonacci, est définie récursivement, pour tout entier naturel n, de la manière suivante :

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

Voici par exemple les premières valeurs de cette fonction.

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

Récursion imbriquée. Les occurrences de la fonction en cours de définition peuvent également être *imbriquées*. Par exemple, la fonction $f_{91}(n)$ ci-dessous, que l'on doit à John McCarthy (informaticien et lauréat du prix Turing en 1971), est définie avec deux occurrences imbriquées, de la manière suivante :

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

Voici par exemple la valeur de $f_{91}(99)$:

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

Le fait que $f_{91}(99)$ renvoie 91 n'est pas un hasard ! On peut en effet démontrer que $f_{91}(n) = 91$, pour tout entier naturel n inférieur ou égal à 101.

Récursion mutuelle. Il est également possible, et parfois nécessaire, de définir plusieurs fonctions récursives en *même temps*, quand ces fonctions font référence les unes aux autres. On parle alors de définitions *récursives mutuelles*.

Par exemple, les fonctions $a(n)$ et $b(n)$ ci-dessous, inventées par Douglas Hofstadter (il y fait référence dans son ouvrage *Gödel, Escher, Bach : Les Brins d'une Guirlande Éternelle*, pour lequel il a obtenu le prix Pulitzer en 1980), sont définies par récursion mutuelle de la manière suivante :

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

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

On peut vérifier que les premières valeurs de ces deux suites sont bien

n :	$0, 1, 2, 3, 4, 5, 6, 7, 8, 9, \ldots$
$a(n)$:	$1, 1, 2, 2, 3, 3, 4, 5, 5, 6, \ldots$
$b(n)$:	$0, 0, 1, 2, 2, 3, 4, 4, 5, 6, \ldots$

D'une manière amusante, on peut montrer que les deux séquences diffèrent à un indice n si et seulement si $n+1$ est un nombre de Fibonacci, c'est-à-dire s'il existe un entier k tel que $fibonacci(k) = n + 1$.

Définitions récursives bien formées

Il est important de respecter quelques règles élémentaires lorsqu'on écrit une définition récursive. Tout d'abord, il faut s'assurer que la récursion va bien se terminer, c'est-à-dire que l'on va finir par « retomber » sur un cas de base de la définition. Ensuite, il faut que les valeurs utilisées pour appeler la fonction soient toujours dans le domaine de la fonction. Enfin, il convient de vérifier qu'il y a bien une définition pour toutes les valeurs du domaine.

Prenons comme premier exemple la fonction $f(n)$ ci-dessous définie de la manière suivante :

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

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

$$f(1) = 1 + f(2) = 1 + 2 + f(3) = \ldots$$

Notre deuxième exemple est celui d'une définition récursive pour une fonction $g(n)$ qui s'applique à des entiers naturels.

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

Cette définition n'est pas correcte car, par exemple, la valeur de $g(1)$ vaut

$$g(1) \quad = \quad 1 + g(-1)$$

mais $g(-1)$ n'a pas de sens puisque cette fonction ne s'applique qu'à des entiers naturels. Comme nous le verrons dans la section suivante, cette définition peut conduire à une exécution infinie ou à une erreur, selon la manière dont elle est écrite en Python.

Enfin, notre dernier exemple est celui d'une fonction $h(n)$ qui s'applique également à des entiers naturels et qui est définie de la manière suivante :

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

Cette définition est incorrecte, puisqu'une valeur est oubliée. L'avez-vous repérée ? [2]

Définition récursive de structures de données. Les techniques de définition récursive peuvent s'appliquer à toute une variété d'objet, et pas seulement à la définition de fonctions. Nous verrons en particulier aux chapitres 6 et 8 des structures de données définies récursivement.

2. C'est la valeur de $h(1)$ qui n'est pas définie.