

# TP 10

## Récurtivité

La récursivité n'est pas un nouveau concept du langage Python ni même une nouvelle fonction, mais une technique générale de programmation. Une fonction écrite en Python est dite **récursive** quand, dans le corps de la fonction, elle fait appel à... elle-même. Cela correspond directement à la notion mathématique de récurrence.

### I Exemples simples

Nous avons appris à calculer les puissances de 2 avec une simple boucle `for`. Cependant la définition mathématique la plus naturelle du nombre  $2^n$  est par récurrence :

$$\forall n \in \mathbb{N}, \quad 2^n = \begin{cases} 1 & \text{si } n = 0 \\ 2 \times 2^{n-1} & \text{si } n \geq 1 \end{cases} \quad (1)$$

Il est possible d'écrire un programme Python qui fait exactement cela !

```
def puissance2(n):
    if n == 0:
        return 1
    else:
        return 2 * puissance2(n-1)
```

Pour comprendre bien ce qui se passe dans ce programme, on rajoute des `print()` :

```
def puissance2(n):
    print("Appel avec n =", n)
    if n == 0:
        print("Fini")
        return 1
    else:
        p = 2 * puissance2(n-1)
        print("Retour avec", p)
        return p
```

Tester la fonction avec des petites valeurs de  $n$ .

Ce qu'on observe s'appelle la **pile d'appels**, voir l'annexe IV. La fonction est appelée elle-même successivement avec des valeurs de  $n$  de plus en plus petites, puis les retours se font dans l'ordre inverse des appels, tout comme on récupère les éléments d'une pile sur son bureau dans l'ordre inverse duquel on les a posés.

**Exercice 1.** Une fonction célèbre est écrite ici de façon récursive. Quelle est la fonction `f` ?

```
def f(n):
    if n == 0:
        return 1
    else:
        return n * f(n-1)
```

Dans la suite de ce chapitre, quand on demande « écrire une fonction récursive » il faut que la fonction fasse appel à elle-même au lieu d'utiliser une boucle. Nous verrons qu'il est nécessaire d'avoir d'abord au brouillon une bonne formulation récursive du problème, qu'il est ensuite relativement aisé de traduire en fonction Python.

**Exercice 2.** Soit la suite  $(u_n)_{n \in \mathbb{N}}$  définie par  $u_0 = 1$  et  $\forall n \in \mathbb{N}^*$ ,  $u_n = 3u_{n-1} + 2$ . Écrire une fonction récursive `suite(n)` qui renvoie le terme  $u_n$ .

**Exercice 3.** Soit la somme  $S_n = \sum_{k=1}^n k^3$ .

1. Qu'est-ce que  $S_1$  ? Et éventuellement  $S_0$  ? Quelle est la relation entre  $S_n$  et  $S_{n-1}$  ?
2. Écrire une fonction récursive `somme_cubes(n)` qui prend en argument un nombre entier `n` et qui calcule  $S_n$ .

## II Quelques phénomènes

Retenons que **pour écrire une bonne fonction récursive, il est impératif d'avoir d'abord au brouillon une bonne formulation mathématique du problème**, qui comprenne à la fois une relation de récurrence (exprimer  $f(n)$  en fonction de valeurs plus petites, typiquement  $f(n-1)$  ou bien  $f(n/2)$ ), mais aussi une condition d'initialisation (en général le cas  $n = 0$  ou  $n = 1$ ). La fonction bien écrite au brouillon peut ensuite se traduire facilement en Python.

### Exercice 4. Fibonacci, le retour du retour

On rappelle que la suite de Fibonacci est la suite  $(F_n)_{n \in \mathbb{N}}$  définie par :

$$\forall n \in \mathbb{N}, \quad F_n = \begin{cases} 0 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ F_{n-1} + F_{n-2} & \text{si } n \geq 2 \end{cases} \quad (2)$$

1. Traduire cette définition en une fonction récursive `fibonacci(n)`.
2. Tester la fonction `fibonacci(n)` en prenant des valeurs de `n` de plus en plus grandes, pas à pas, surtout en augmentant doucement à partir d'environ 30 (ne pas tester directement plus de 35). Que se passe-t-il ?
3. Pour comprendre ce qui se passe, insérer au tout début de la fonction la ligne

```
| print("Appel avec n =", n)
```

et ré-essayer, cette fois d'abord avec des tout petits nombres, comme 4, 5, 6. Expliquer (**dessin au tableau**).

### Exercice 5. Exponentiation rapide

Soit  $a \in \mathbb{R}$ . Remarquons que les puissances  $a^n$  vérifient la relation de récurrence suivante, pour  $n \in \mathbb{N}^*$  (on passe sur  $a^0 = 1$ ) :

$$a^n = \begin{cases} a & \text{si } n = 1 \\ (a^{n/2})^2 & \text{si } n \text{ est pair} \\ a \times a^{n-1} & \text{si } n \text{ est impair} \end{cases} \quad (3)$$

Par exemple pour calculer  $a^5$  cela revient à écrire  $a^5 = a \times a^4$  puis  $a^4 = (a^2)^2$ , ainsi  $a^5 = a \times (a^2)^2$  et on peut vérifier que cela nécessite en tout de calculer 3 multiplications, ce qui est mieux que d'écrire  $a^5 = a \times a \times a \times a \times a$  qui en nécessite 4. Pour calculer  $a^6$  cela revient à écrire  $a^6 = (a^3)^2 = (a \times a^2)^2$  et on compte en tout 3 multiplications aussi, au lieu de 5 avec la méthode naïve.

1. Avec la même méthode, combien de multiplications sont nécessaires pour calculer les nombres suivants ?

$$a^{16} \qquad a^{15} \qquad a^{24} \quad (4)$$

2. Écrire une fonction récursive `puissance_rapide(a, n)` qui prend en argument un entier `a` et un entier `n` (celui-ci est supposé strictement positif) et qui calcule  $a^n$  selon ce procédé.

On rappelle que, comme les exposants sont entiers, on utilise le reste dans la division euclidienne par 2 `n % 2` pour tester la parité et le quotient `n // 2`.

**Exercice 6.** On rappelle la formule de Pascal pour les coefficients binomiaux, ré-écrite en fonction de  $n-1$  et  $k-1$  et valable pour tout  $k \in \mathbb{Z}$  avec la définition étendue des coefficients binomiaux :

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} \quad (5)$$

1. Écrire une fonction récursive `binome(n, k)` qui renvoie le coefficient binomial  $\binom{n}{k}$ , pour tous  $n \in \mathbb{N}$  et  $k \in \mathbb{Z}$ .

Attention au cas d'initialisation !

2. Tester la fonction ci-dessus avec des valeurs de  $n$  et de  $k$  de plus en plus grandes en augmentant lentement, par exemple  $n$  autour de 25 à 30 et  $k$  petit ou à la moitié de  $n$ . Encore une fois qu'observe-t-on ?

3. Pour corriger ce problème, une méthode est d'écrire une fonction `binome_liste(n)` qui renvoie une liste de longueur  $n + 1$  correspondant à toute la ligne du triangle de Pascal des coefficients  $\binom{n}{k}$  pour  $0 \leq k \leq n$ . Écrire cette fonction, de façon récursive.

### Avertissement

Ce qu'il faut retenir, c'est que la récursivité est une méthode de programmation qui a l'avantage d'être parfois plus proche du langage mathématique et plus simple à programmer ; mais qui en même temps peut donner lieu à des programmes moins efficaces et à une structure plus compliquée de la pile d'appels.

Dans les sujets écrits, il peut être précisé « écrire une fonction récursive qui... » ou bien « écrire une fonction itérative (c'est à dire, non-récursive) qui... ». Si rien n'est précisé, alors vous êtes libre de choisir la méthode avec laquelle vous vous sentez le plus à l'aise.

## III Problèmes

### Exercice 7. Ordre lexicographique

On souhaite écrire une fonction qui compare deux mots selon l'ordre du dictionnaire, appelé **ordre lexicographique**. Cela signifie que le mot  $s$  vient avant le mot  $t$  si :

- La première lettre de  $s$  vient avant la première lettre de  $t$ ,
- Ou bien les premières lettres sont les mêmes, et la deuxième lettre de  $s$  vient avant la deuxième lettre de  $t$ ,
- etc,
- Éventuellement on arrive à la situation où les deux mots sont égaux ; ou bien ou l'un est plus court que l'autre et ils sont égaux sur le plus court, c'est à dire que l'un est un préfixe de l'autre, et c'est le plus court des deux qui vient avant dans le dictionnaire (par exemple « TRAVAIL » vient avant « TRAVAILLER »).

La fonction que l'on souhaite écrire s'appellera `compare(s, t)` et renverra 1 si  $s$  vient avant  $t$  dans le dictionnaire,  $-1$  si  $s$  vient après  $t$ , et 0 si les deux mots sont égaux. On se limitera à des caractères parmi les 26 lettres de l'alphabet en minuscule ; sinon cela nécessite d'abord de décider comment comparer les majuscules, les lettres accentuées, et cela complique nettement les choses.

Pour l'écrire de façon récursive, on pourra décomposer un mot  $s$  en sa première lettre `s[0]` et le reste du mot obtenu avec la sélection de tranche `s[1:]`. Mais attention au mot vide "" qui apparaît naturellement quand on a enlevé plusieurs fois de suite la première lettre. Il faut donc qu'il apparaisse dans la partie initialisation de la récursivité !

1. Au brouillon, proposer une formulation récursive du problème, en termes comme ci-dessus de première lettre et de comparaison du reste du mot.
2. On donne `alphabet = "abcdefghijklmnopqrstuvwxyz"`. Écrire une fonction (non récursive) `numero(x)` qui prend en argument un caractère seul  $x$  et renvoie sa position en tant que lettre de l'alphabet (de 0 pour `a`, à 25 pour `z`).
3. Écrire la fonction `compare(s, t)`.

**Exercice 8.** On représente une partie de l'ensemble  $\{1, 2, \dots, n\}$  par une liste contenant les éléments, rangés par ordre croissant de la partie. On souhaite écrire une fonction `parties(n)` qui donne la liste de toutes les parties de  $\{1, 2, \dots, n\}$ . On peut démarrer avec  $n = 0$ , il n'y a que la partie vide. En général il y a deux types de parties : celles ne contenant pas  $n$  — ce sont donc des parties de  $\{1, 2, \dots, n - 1\}$  — et celles contenant  $n$ , obtenues à partir des parties de  $\{1, 2, \dots, n - 1\}$  en leur rajoutant  $n$ .

Écrire la fonction récursive `parties(n)`.

## IV Annexe : la pile d'appels

**Le mécanisme d'appel** Pour comprendre la récursivité, il est important de comprendre plus en profondeur le processus d'appel de fonction. Supposons qu'on soit en train d'exécuter une fonction  $f$ , et qu'on tombe sur une ligne  $y = g(x)$  provoquant l'appel d'une autre fonction  $g$  en lui passant un paramètre  $x$ , et en récupérant la valeur de retour dans une variable  $y$ .

```
def f():
    x = ...
    ...
    y = g(x)
    ...

def g(x):
    ...
    return ... x ...
```

L'ordinateur réalise les opérations suivantes :

1. Il va falloir mettre en pause la fonction **f**, pour sauter dans le code de la fonction **g**,
2. Avant cela il faut sauvegarder la position précise où on en était dans **f**, pour pouvoir y revenir,
3. Il faut aussi copier la valeur de **x** pour que la fonction **g** puisse l'utiliser,
4. Puis la fonction **g** s'exécute normalement, avec ses propres variables et sa propre zone de mémoire. À la fin il faut copier la valeur renvoyée dans une variable **y** de **f** puis re-sauter dans le code de **f** au point précis qu'on avait sauvegardé, et enfin on peut libérer la mémoire utilisée par la fonction **g**.

Souvent, une fonction **f** appelle une fonction **g** qui elle-même fait appel à une fonction **h** et ainsi de suite. À chaque appel il faut sauvegarder les variables locales de la fonction et le point exact où il faudra revenir, puis sauter dans le code de la fonction appelée et utiliser de la mémoire pour la nouvelle fonction. Ces sauvegardes sont entassées les unes à la suite des autres dans la mémoire, on parle de **pile d'appels**, et sont libérées dans l'ordre inverse duquel elles ont été empilées.

En emboitant des appels à une fonction elle-même, la récursivité peut faire grossir la pile d'appels de façon incontrôlée et occuper beaucoup plus de mémoire que souhaité : les zones mémoires vont se retrouver empilées les unes à la suite des autres pour **f(n)** puis pour **f(n-1)** puis... jusqu'à **f(0)**, et vont se libérer en sens inverse. Cela occupe nettement plus de place qu'une simple boucle telle que `for i in range(n)` qui utilise une seule variable et dont l'occupation mémoire n'augmente pas avec **n**. C'est cela le principal défaut de la récursivité.

**Une analogie** Une analogie est la suivante. Imaginons qu'on lise un livre de mathématiques au chapitre sur les bijections. Mais qu'on ne comprenne pas. Alors on laisse le livre ouvert mais par dessus on ouvre un autre livre sur les ensembles et la logique. Mais on ne comprend toujours pas. Alors on laisse celui-ci ouvert et on ouvre un livre de lycée. Là, on lit et on comprend. À la fin on referme le livre de lycée et on retombe sur celui d'ensembles et de logique, qu'on peut continuer à lire. Et quand on a fini on le referme et on retombe là où on en était sur les bijections. Les livres se sont empilés sur le bureau, chaque lecture étant mise en pause à un moment pour ouvrir un autre livre par-dessus, et lorsqu'il est refermé on reprend exactement la lecture au point où on était.

**Dans d'autres langages** Certains langages de programmation ont été conçus pour encourager la récursivité, avec un point de vue plus proche des mathématiques. C'est le cas du langage OCaml, enseigné notamment en MPSI et MP2I, réputé pour ses algorithmes qui transforment par derrière un programme récursif en une version non-récursive tout aussi efficace. Dans ce langage, même les listes sont définies de façon récursive : soit c'est la liste vide, soit c'est un élément (*tête*) attaché au reste de la liste (*queue*). Une fonction aussi simple que la longueur d'une liste est alors récursive : c'est zéro pour la liste vide, et sinon c'est un de plus que la longueur de la queue. On parle de **type récursif**. Cela est particulièrement intéressant pour traiter des structures en arbres, graphes, et de divers problèmes de combinatoire, qui seraient nettement plus compliqués à écrire avec de simples listes et boucles Python. Un exemple d'OCaml qui se lit aussi facilement que la description mathématique :

```
type liste =
  | Vide
  | Attache of int * liste

let rec longueur l = match l with
  | Vide -> 0
  | Attache(tete, queue) -> 1 + longueur queue
```