Chapitre (S1) 5 Récursivité

Objectifs

- Connaitre la structure typique d'une fonction récursive
- Savoir détecter des situations où une programmation récursive est appropriée.
- Connaitre le vocabulaire propre aux fonctions récursives.

1 GÉNÉRALITÉS

1.1 Introduction : la factorielle

Considérons une suite classique que vous connaissez bien : la factorielle, que l'on note $(u_n) = (n!)$. La suite (u_n) peut être définie en mathématiques de deux manières.

MODE EXPLICITE & PROGRAMMATION IMPÉRATIVE. La première :

$$\forall n \in \mathbb{N}, \quad u_n = 1 \times 2 \times \cdots \times n = \prod_{k=1}^{n} k \quad \text{(convention } u_0 = 1\text{)}.$$

Par exemple $4! = 1 \times 2 \times 3 \times 4 = 24$. Ce mode de définition nous mène directement au code ci-après.

```
def factorielle(n):
    """ renvoie la factorielle de l'entier naturel n. """
    if n  ==  0:
        return 1
    else:
        f = 1
        for k in range (1, n+1):
            f *= k
        return f
```

Notez que le code ci-après est lui aussi correct.

```
def factorielle(n : int) -> int :
    """ renvoie la factorielle de l'entier naturel n. """
    f = 1
    for k in range(1, n+1):
        f *= k
    return f
```

En effet, pour $n = 0$, le `range(1, n+1)` *(on rappelle que la borne de droite est exclue)* est vide, la valeur de f initialisée à 1 n'est donc pas modifiée. Cette fonction est dite de type *itératif*, car elle est construite à partir d'une boucle (ici, `for`).

MODE RÉCURRENT & PROGRAMMATION RÉCURSIVE. Une seconde manière de programmer est d'utiliser la relation de récurrence ci-après :

$$\forall n \in \mathbb{N}^\star, \quad u_n = n \times u_{n-1}, \quad u_0 = 1.$$

Notons que cette définition de la factorielle fait appel à la factorielle elle-même (mais sur une valeur plus petite), ce qui est caractéristique du principe **récursif**. Le calcul de 4! se fait alors ainsi :

- **[Phase d'empilement]**
$$\begin{aligned} 4! &= 4 \times 3! \\ &= 4 \times (3 \times 2!) \\ &= 4 \times (3 \times (2 \times 1!)) \\ &= 4 \times (3 \times (2 \times (1 \times 0!))) \\ &= 4 \times (3 \times (2 \times (1 \times 1))). \end{aligned}$$
- **[Phase de dépilement]** La seconde phase consiste alors à réaliser les différentes multiplications « mises en attentes », en commençant par la dernière

(gestion en pile, comme une pile d'assiettes, selon le principe « dernier arrivé, premier servi »).

$$4 \times (3 \times (2 \times (1 \times 1))) = 4 \times (3 \times (2 \times 1)) = 4 \times (3 \times 2) = 4 \times 6 = 24.$$

Ce mode de programmation est dit *fonctionnel* et la fonction ci-dessous est dite *récursive*.

```
def factorielle(n : int) -> int :
    """ renvoie la factorielle de l'entier naturel n """
    if n == 0:
        return 1
    else :
        return n*factorielle(n-1)
```

Il est probable que, si vous découvrez la notion de récursivité, vous soyez surpris qu'un tel programme fonctionne, sans que vous n'ayez écrit dans son code la moindre boucle. Pour éclairer ce phénomène, il faut comprendre que lorsqu'un appel à une fonction se fait à l'intérieur d'une autre fonction, cette dernière interrompt son exécution (en sauvegardant tout le contexte qui lui permettra de reprendre le calcul là où il en était) pour permettre l'exécution de la fonction appelée. Le mécanisme est identique si la fonction appelée est la même que la fonction appelante. On peut présenter les appels successifs provoqués par `factorielle(3)` de la manière suivante :

- l'appel `factorielle(3)` commence et s'interrompt en déclenchant l'appel de `factorielle(2)` ;
  - ◇ l'appel `factorielle(2)` commence et s'interrompt en déclenchant l'appel de `factorielle(1)` ;
    - — l'appel `factorielle(1)` commence et s'interrompt en déclenchant l'appel de `factorielle(0)` ;
      - ⋆ l'appel `factorielle(0)` va jusqu'à son terme et renvoie la valeur 1 à la fonction `factorielle(1)` qui l'a appelée ;
    - — l'appel `factorielle(1)` reprend, termine son calcul $1 \times 1 = 1$, et renvoie cette valeur à l'appel `factorielle(2)` ;
  - ◇ l'appel `factorielle(2)` reprend, termine son calcul $2 \times 1 = 2$, et renvoie cette valeur à l'appel `factorielle(3)` ;
- l'appel `factorielle(3)` reprend, termine son calcul $3 \times 2 = 6$, et renvoie cette valeur, qui est l'unique valeur de retour de l'appel initial.

Le mécanisme consistant à stopper le déroulement d'un appel pour en permettre un autre oblige l'ordinateur à consommer de la mémoire pour stocker son contexte. Afin de contrôler cette dépense en mémoire, le langage Python limite par défaut le nombre d'appels récursifs à environ 1000. Au delà, une erreur est générée :

```
RecursionError: maximum recursion depth exceeded
```

Cette limite est généralement largement assez haute pour permettre l'exécution des exemples que nous rencontrerons, mais sachez qu'il est possible de la modifier par la commande suivante (qui positionne le nombre maximal d'appels ici à 2000) :

```
>>> import sys
>>> sys.getrecursionlimit() # taille par défaut
1000
>>> sys.setrecursionlimit(2000)
>>> sys.getrecursionlimit() # taille nouvelle
2000
```

Remarque 1 (Résolution du problème de taille de pile : récursivité terminale)

- Le problème de taille de pile peut être résolu en évitant justement de faire appel à une telle pile. Par exemple, dans la factorielle, l'utilisation d'une pile d'appels est due au stockage des multiplications.
- Il est possible de réecrire cette fonction factorielle sans multiplication (non détaillé dans ce cours) : on parle alors de *fonction récursive terminale*.

Notons que pour que le calcul ne boucle pas infiniment sur lui-même, il est indispensable qu'au bout d'un certain nombre d'étapes il aboutisse à une valeur de factorielle que l'on sait calculer directement. Une telle situation est appelée un *cas terminal*. Pour notre exemple de la factorielle, c'est la valeur $0! = 1$ qui joue le rôle de cas terminal. À noter aussi que si on lance le calcul sur une valeur strictement négative, celui-ci bouclera sans s'arrêter. Par exemple :

$$(-1)! = (-1) \times (-2)! = (-1) \times ((-2) \times (-3)!) = \dots.$$

La notion de récursivité est généralement assez déstabilisante au départ, mais vous constaterez à l'usage que le principe devient assez rapidement familier, de très nombreux concepts pouvant être exprimés récursivement. Comme vous allez le constater, elle dépasse d'ailleurs très largement le champs des exemples mathématiques par lesquels nous avons commencé.

1.2 Fonction récursive

Définition 1 | Fonction récursive
Une fonction *récursive* est une fonction qui s'appelle une ou plusieurs fois dans son corps.

Exercice 1 [Sol 1]

```
def mystere(n):
    if n == 0:
        return 1 # cas de base
    else:
        return 2*mystere(n-1)
```

Conjecturer ce que renvoie cette fonction et compléter sa signature. *On pourra effectuer (au stylo) le déroulé de cette fonction sur, par exemple, l'appel* `mystere(3)`

Exercice 2 [Sol 2]

```
def mystere(L):
    if len(L) == 0:
        return 0 # cas de base
    else:
        return L[0] + mystere(L[1:])
```

Conjecturer ce que renvoie cette fonction et compléter sa signature. *On pourra effectuer (au stylo) le déroulé de cette fonction sur, par exemple, l'appel* `mystere([1, 4, 3])`

Remarque 2

1. Une fonction récursive est particulièrement adaptée pour des objets mathématiques définis par récurrence.
2. Une fonction récursive est un exemple de la stratégie « diviser pour mieux régner » : on reformule le problème en un simple changement de paramètre d'entrée.

Méthode Concevoir une fonction récursive
Pour concevoir une fonction récursive, il faut :

- Rechercher le ou les cas de bases et leur donner une solution.
- Décomposer le problème général en sous-problèmes identiques au problème de départ et s'assurer que les valeurs des paramètres des sous problèmes tendent vers les valeurs des paramètres du ou des cas de bases.

Exemple 1 (Reste de la division euclidienne) Soient $a, b \in \mathbb{N}$ deux entiers avec $b \neq 0$.

- Rappelons qu'il existe un unique couple $(q_{a,b}, r_{a,b})$ d'entiers tels que :
  $$a = bq_{a,b} + r_{a,b}, \quad \text{et :} \quad 0 \leqslant r_{a,b} < b.$$
  L'entier $r_{a,b}$ est appelé *reste de la division euclidienne de a par b*. Notons que si $a < b$, alors $r_{a,b} = a$ puisque $a = 0 \times b + a$ et que $0 \leqslant a < b$.
- Cherchons une fonction récursive qui renvoie $r_{a,b}$.
  - ◇ **[Cas de base]** Si $a < b$, il faut renvoyer a.
  - ◇ **[Récursivité]** On remarque que $r_{a,b} = r_{a-b,b}$, car :
    $$a = bq_{a,b} + r_{a,b} \iff a - b = b(q_{a,b} - 1) + \underbrace{r_{a,b}}_{=r_{a-b,b}}.$$
    Cela mène à la fonction récursive ci-après.

```
def reste(a:int,b:int)->int:
    if a < b:
        return a
    else:
        return reste(a-b, b)
```

On voit que le premier paramètre est alors abaissé, car changé en `a-b` < `a`. On finira donc par tomber sur le cas de base. On peut tester :

```
>>> reste(10, 3)
1
>>> reste(3, 10)
3
```

Exercice 3 [Sol 3]

Écrire une fonction récursive `pgcd(a:int, b:int)->int` qui renvoie le plus grand diviseur commun de 2 entiers. *On rappelle que $pgcd(a, b) = pgcd(b, r)$ où r est le reste de la division euclidienne de a par b. On pourra utiliser la fonction* `reste` *codée précédemment*

Exercice 4 [Sol 4]

Écrire une fonction récursive `produit(L:list)->float` qui renvoie le produit des éléments d'une liste.

2 TYPES DE RÉCURSIVITÉ

2.1 Simple & Multiple

Définition 2

- Une fonction récursive est dite *simple* si elle ne contient au plus qu'un seul appel récursif (éventuellement un par sous-cas qu'elle traite).
- Dans le cas contraire, elle est dite *multiple*.

Exemple 2

1. Tous les exemples/exercices précédents sont des fonctions récursives simples.
2. On considère la suite de FIBONNACI définie par :

$$u_0 = a, \quad u_1 = b, \quad \forall n \in \mathbb{N}, \quad u_{n+2} = u_{n+1} + u_n.$$

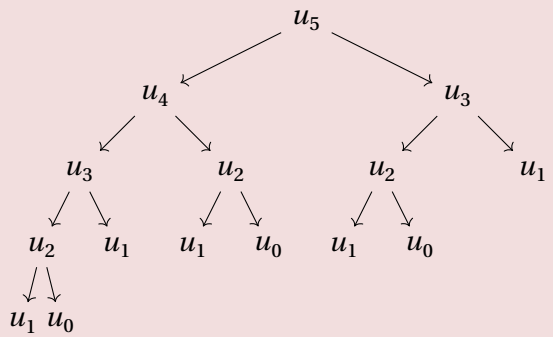
On remarque que :

- Il y a 2 cas de bases.
- Le cas général se divise en deux sous cas donnés par la formule de récurrence.

```
def fibonacci(a:int, b:int, n : int) -> int:
    """ renvoie le terme d'indice n de la suite de \
    ↪ fibonacci """
    if n == 0:
        return a
    elif n == 1:
        return b
    else :
        return fibonacci(a, b, n-1) + fibonacci(a, b, n-2)
```

Attention aux récursivités multiples

- Si à l'usage vous trouverez les fonctions récursives de plus en plus naturelles et faciles à écrire, une approche trop naïve peut conduire à des programme extrêmement peu efficaces. L'exemple le plus classique est la fonction `fibonacci` précédente.
  Si ce code renvoie bien la valeur de u_n, il le fait de manière extrêmement maladroite puisque les deux appels récursifs générés à chaque étape conduisent à recalculer plusieurs fois de nombreux termes de la suite, comme le montre l'exemple de l'appel `fibonacci(5)` (on dispose à chaque fois sur la ligne d'en-dessous les appels générés par ceux de l'étape précédente, et pour plus de concision u_n est écrit au lieu de `fibonacci(n)`) :



Vous constatez que u_1 par exemple a été évalué 5 fois !

- Notez qu'il est possible de construire une version (toujours récursive) plus efficace de la fonction `fibonacci` contournant ce problème en utilisant un principe dit de *programmation dynamique* que vous verrez en 2ème année. Il est aussi possible de ruser en changeant les conditions initiales lors d'un des deux appels (voir le Chapitre (S2) 2 du second semestre).

2.2 Mutuelle ou Croisée

Définition 3 | Mutuelle

On dit que deux fonctions récursives sont *mutuellement récursives* si l'une fait appel à l'autre et vice versa. On parle alors de *récursivité croisée*.

Exercice 5 [Sol 5] On considère les deux fonctions :

```
def pair(n:int)->int:
    if n == 0:
        return True
    else:
        return impair(n-1)
```

```
def impair(n:int)->int:
    if n == 0:
        return False
    else:
        return pair(n-1)
```

Au stylo, déterminer le résultat de l'appel `pair(3)`, et `pair(2)`.

Exercice 6 [Sol 6] On considère les suites (u_n) et (v_n) définies par :

$$u_0 = a, \quad v_0 = b, \quad \begin{cases} u_{n+1} = u_n - v_n \\ v_{n+1} = u_n + v_n. \end{cases}$$

On suppose deux variables `a, b` fixées en début de programme.

1. Écrire deux fonctions récursives croisées `u(n:int)->int` et `v(n:int)->int` qui permettent de calculer respectivement u_n et v_n.
2. Écrire une fonction `uv(n:int)->(int, int)`, impérative cette fois, renvoyant la valeur de (u_n, v_n).

Remarque 3 Ce type de récursivité intervient aussi dans des exemples beaucoup plus complexes ; par exemple, l'algorithme Min/Max en théorie des jeux (programme de 2ème année).

SOLUTIONS DES EXERCICES

Solution 1 Dans l'appel `mystere(3)` :

- puisque $3 \neq 0$, on calcule `2*mystere(2)`,
- puisque $2 \neq 0$, on calcule `2*2*mystere(1)`,
- puisque $1 \neq 0$, on calcule `2*2*2*mystere(0)`,
- comme $0 = 0$, on est sur un cas de base, on calcule `2*2*2*1`.
- Phase de dépilement : les multiplications sont réalisées, la fonction renvoie $\boxed{8}$.

Voici la fonction complétée.

```
def mystere(n:int)->int:
    """renvoie la valeur de 2**n"""
    if n == 0:
        return 1 # cas de base
    else:
        return 2*mystere(n-1)
>>> mystere(3)
8
```

Solution 2 Dans l'appel `mystere([1, 4, 3])` :

- puisque `[1, 4, 3]` est non vide, on calcule `1 + mystere([4, 3])`,
- puisque `[4, 3]` est non vide, on calcule `1 + 4 + mystere([3])`,
- puisque `[3]` est non vide, on calcule `1 + 4 + 3 + mystere([])`,
- puisque `[]` est vide, on est sur un cas de base, on calcule `1 + 4 + 3 + 0`,
- Phase de dépilement : les additions sont réalisées, la fonction renvoie $\boxed{8}$.

De façon générale, on peut conjecturer que cette fonction renverra la somme des éléments d'une liste si elle est non vide, et 0 dans le cas contraire. Voici la fonction complétée.

```
def mystere(L:list)->float:
    """ renvoie la somme des éléments de L """
    if len(L) == 0:
        return 0 # cas de base
    else:
        return L[0] + mystere(L[1:])
>>> mystere([1, 4, 3])
8
```

Solution 3

```
def pgcd(a:int, b:int)->int:
    if b == 0 :
        return a
    return pgcd(b, reste(a, b))
```

Ce programme semble correct dans le cas $a \geqslant b$. Si $a \leqslant b$, alors :

- soit b est nul, donc a aussi et la fonction renvoie bien le même résultat.
- Soit b est non nul, et la fonction renvoie `pgcd(b, a)`. Les paramètres se retrouvent alors dans le bon ordre.

```
>>> pgcd(3, 6)
3
>>> pgcd(6, 3)
3
>>> pgcd(3, 1)
1
```

Solution 4 En s'inspirant d'une fonction mystère précédente (somme des éléments d'une liste), on débouche directement sur le code ci-dessous.

```
def produit(L:list)->float:
    """ renvoie la somme des éléments de L """
    if len(L) == 0:
        return 1 # cas de base
    else:
        return L[0] * produit(L[1:])
>>> produit([1, 4, 3])
12
```

Solution 5 Pour le premier appel :

- comme $3 \neq 0$, on calcule `impair(2)`,
- comme $2 \neq 0$, on calcule `pair(1)`,
- comme $1 \neq 0$, on calcule `impair(0)`,
- C'est un cas terminal, la fonction renverra **`False`**.

Pour le second :

- comme $2 \neq 0$, on calcule `impair(1)`,
- comme $1 \neq 0$, on calcule `pair(0)`.
- C'est un cas terminal, la fonction renverra **`True`**.

Solution 6

1. Pour calculer les termes, on peut utiliser les fonctions :

```
def u(n:int)->int:
    if n == 0:
        return a
    else :
        return u(n-1)-v(n-1)
```

```
def v(n:int)->int:
    if n == 0:
        return b
    else:
        return u(n-1)+v(n-1)
```

2. On utilise ici des boucles classiques.

```
def uv(n:int)->(int, int):
    u, v = a, b
    for _ in range(1, n+1):
        u, v = u-v, u+v
    return u, v
>>> a, b = 2, 3
>>> u(5), v(5)
(4, -20)
>>> uv(5)
(4, -20)
```