TP 10
Récursivité

La récursivité n'est pas un nouveau concept Python ni même une nouvelle fonction, mais c'est 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 par récurrence est :

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

Et bien, on peut effectivement écrire un programme en Python qui fait exactement cela !

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

Testez-le !

```
>>> puissance2(5)
32
>>> puissance2(0)
1
```

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

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

Exercice 1.
1. Tester la fonction avec des petites valeurs de `n` (3, 4) et bien comprendre ce qui se passe pendant l'exécution.
2. Que se passe-t-il si on entre `puissance2(-1)` ?
3. Rajouter la ligne `assert(n >= 0)` au tout début de la fonction et ré-essayer.

Ce qu'on observe s'appelle la **pile d'appels** (voir plus bas) c'est à dire l'ordre logique dans lequel la fonction est appelée ou bien s'arrête lors du déroulement du programme.
Un autre exemple célèbre est la fonction factorielle : nous savons bien entendu le faire avec une boucle et une variable accumulatrice `p` initialisée à `1`, mais la définition mathématique la plus usuelle est

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

ce qui se traduit par le programme Python suivant à tester :

```python
def factoriel(n):
    assert(n >= 0)
    if n == 0:
        return 1
    else:
        return n * factoriel(n-1)
```

Dans la suite de ce chapitre, quand on demande « écrire une fonction qui… » il est sous-entendu qu'il s'agit d'une fonction récursive, qui n'utilise pas de boucle comme précédemment. Pour écrire une bonne fonction récursive, il est nécessaire de commencer par une bonne formulation mathématique du problème sous forme de relation de récurrence.

Exercice 2. Écrire une fonction récursive `puissance(a, n)` qui prend en argument un nombre `a` et un entier `n`, supposé positif, et qui retourne le nombre a^n.

Exercice 3. Écrire une fonction `somme_cubes(n)` qui prend en argument un nombre entier `n`, supposé positif, et qui calcule la somme

$$\sum_{k=0}^{n} k^3 \tag{3}$$

mais de façon récursive !

II Petits problèmes

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 :

$$\forall n \in \mathbb{N}, \quad F_n = \begin{cases} 0 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ F_{n-1} + F_{n-2} & \text{si } n \geqslant 2 \end{cases} \tag{4}$$

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

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

   et ré-essayer, cette fois d'abord avec des tout petits nombres, comme 4, 5. Expliquer.

Nous allons tout de même corriger ce problème.

Exercice 5 (Fibonacci, le retour du retour du retour)**.** Définissions les suites $(a_n)_{n\in\mathbb{N}}$ et $(b_n)_{n\in\mathbb{N}}$ tout simplement par $a_n = F_n$ et $b_n = F_{n+1}$, ainsi le couple de deux termes successifs de la suite de Fibonacci est (a_n, b_n). On rappelle que ce couple vérife la relation de récurrence suivante :

$$\forall n \in \mathbb{N}, \quad (a_n, b_n) = \begin{cases} (0, 1) & \text{si } n = 0 \\ (b_{n-1}, a_{n-1} + b_{n-1}) & \text{si } n \geqslant 1 \end{cases} \tag{5}$$

Écrire une fonction récursive `fibonacci2(n)` qui retourne le *tuple* (F_n, F_{n+1}) de deux termes successifs de la suite de Fibonacci ; la fonction ne doit faire appel qu'une seule fois à `fibonacci2(n-1)`.

Remarque 1. À retenir : la récursivité est une technique parfois bien pratique car elle permet d'écrire des programmes informatique d'une façon beaucoup plus proche de la formulation mathématique du problème. Cependant pour l'ordinateur, ce n'est pas toujours facile à mettre en place et parfois cela consomme beaucoup de temps ou de mémoire.
Lorsque l'ordinateur exécute le contenu d'une fonction `f`, et que le corps de celle-ci appelle une autre fonction `g`, alors il se passe la chose suivante : l'exécution de `f` est mise en pause, les variables qu'elle manipule sont sauvegardées, et la place en mémoire et dans le processeur est libre pour exécuter la fonction `g` ; puis à la fin de `g` la place mémoire qu'elle occupait est libérée et l'ordinateur revient où il en était dans `f`. On parle de **pile d'appel**. Éventuellement la fonction `g` elle-même va appeler d'autres fonctions, et ainsi de suite, et les appels vont s'empiler puis lors des retours tout va se dépiler jusqu'à reprendre à `f`.
C'est un peu comme si… on lit un livre de mathématiques sur les bijections. Mais qu'on ne comprend rien. Alors on ouvre un livre sur les ensembles et la logique. Mais on ne comprend toujours rien. Alors on ouvre un livre de lycée. Là ça va. Bref, on empile les livres ouverts, en mémorisant à chaque fois où on en était. À la fin on referme le livre de lycée pour revenir aux ensembles, puis on continue à lire, puis on ferme le livre sur les ensembles et on revient à celui sur les bijections. C'est cela, la pile d'appels.

Le problème de la récursivité est que cela fait parfois grossir la pile d'appel de façon non contrôlée ; de plus cela consomme beaucoup de mémoire car à chaque appel il faut sauvegarder où en est la fonction… Certains langages de programmation traitent la récursivité de façon plus intelligente et sont capables de la transformer en des boucles, les plus simples à comprendre pour l'ordinateur, sans abuser de la mémoire. C'est le cas du langage OCaml étudié dans d'autres filières.
En résumé, on retiendra que pour éviter les problèmes comme ci-dessus, on écrira toujours nos fonctions récursives avec **un seul** appel à elle-même (dans chaque cas du `if`, au moins), avec un argument qui doit être plus petit (`f(n)` peut appeler `f(n-1)`, ou `f(n-2)`, mais certainement pas `f(n)` ni `f(n+1)`) et on sauvegarde la valeur de retour dès qu'on peut dans une variable.

Exercice 6 (Exponentiation rapide)**.** Soit $a \in \mathbb{R}$. Remarquons que le nombre a^n vérifie la relation de récurrence suivante :

$$a^n = \begin{cases} 1 & \text{si } n = 0 \\ (a^{n/2})^2 & \text{si } n \text{ est pair} \\ a \times a^{n-1} & \text{si } n \text{ est impair} \end{cases} \tag{6}$$

1. Écrire une fonction récursive `puissance_rapide(a, n)` qui prend en argument un entier `a` et un entier `n` (celui-ci est supposé positif) et qui calcule a^n selon ce procédé.
2. Supposons que n est une puissance de 2, disons $n = 2^p$. Combien de multiplications effectue cette fonction, comparé à la fonction `puissance(a, n)` de la première partie ? Et si n est proche d'une puissance de 2 ? Expliquer le nom.

Exercice 7 (*(*)* Fibonacci ultime)**.** On peut démontrer les relations suivantes dans la suite de Fibonacci : pour tout $k \in \mathbb{N}$,

$$F_{2k} = F_k(2F_{k+1} - F_k) \tag{7}$$
$$F_{2k+1} = (F_{k+1})^2 + (F_k)^2 \tag{8}$$

En s'inspirant à la fois de la suite de Fibonacci pour les tuples, et de l'algorithme d'exponentiation rapide, écrire une fonction `fibonacci_rapide(n)` qui renvoie le tuple de deux termes consécutifs (F_n, F_{n+1}) en utilisant les relations précédentes et en distingant selon la parité de n.

III Avec de la combinatoire et des listes

Exercice 8. On rappelle la formule de Pascal pour les coefficients binomiaux :

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} \tag{9}$$

démontrée par le calcul sous la condition $1 \leqslant k \leqslant n-1$ (sinon, l'un des deux termes voire les deux sont nuls, et il est facile de vérifier que la relation est quand même valable).
Écrire une fonction récursive `binome(n)` qui renvoie pour tout $n \geqslant 0$ la *liste* des $n+1$ coefficients binomiaux $\binom{n}{k}$ pour $0 \leqslant k \leqslant n$. Autrement dit c'est la fonction qui calcule d'un coup toute une ligne du triangle de Pascal en fonction uniquement de la ligne précédente. Attention aux bords du triangle !

Test : on doit avoir

```
>>> binome(4)
[1, 4, 6, 4, 1]
```

Rappelons maintenant que pour les listes comme sur les chaines nous avons vu l'opération de concaténation `L + M` et les sélections de tranches `L[a:b]`. Cela permet par exemple de séparer une liste en un élément seul et le reste de la liste, et de recoller les morceaux. Par exemple le premier élément est `L[0]` et le reste de la liste est `L[1:]`, ou aussi le dernier élément est `L[-1]` et le début de la liste sauf le dernier élément est `L[:-1]`.

Exercice 9. On rappelle que sur les chaines de caractères, l'ordre lexicographique correspond à l'ordre du dictionnaire, et que pour comparer deux caractères individuels `a`, `b` on compare leurs codes asociés `ord(a)` et `ord(b)`. Écrire une fonction `compare(s, t)` qui teste si la chaine `s` est inférieur (au sens de l'inégalité large) à la chaine `t`, de façon récursive. Traiter soigneusement tous les cas on réfléchissant :

1. Ou bien l'une des deux est la chaine vide `""`, alors laquelle est la plus petite ?
2. Ou bien le premier caractère de `s` vient strictement avant le premier caractère de `t` : c'est donc que `s` vient avant `t`.
3. Ou bien l'inverse.
4. Sinon les deux premiers caractères sont égaux et il faut comparer le reste de la chaine. C'est ici que la récursivité est intéressante.

Exercice 10. On représente un ensemble par une liste, la liste de ses éléments (on suppose que les éléments de la liste sont distincts).

1. Supposons par exemple qu'on découpe une liste `L` de taille n en une liste `M` de taille $n-1$ et un dernier élément `x`. Comment exprimer récursivement l'ensemble des parties de `L` ?
2. Écrire une fonction `parties(L)` qui retourne la liste de toutes les parties de `L`.
3. Écrire une fonction `parties_vide(n)` qui renvoie l'ensemble des parties itérées n fois de l'ensemble vide : $\mathcal{P}(\mathcal{P}(\cdots(\mathcal{P}(\emptyset))\cdots))$. On rappelle que l'ensemble des parties de vide est l'ensemble singleton ensenmble vide, et que l'ensemble des parties de celui-ci a deux éléments qui sont lui-même et l'ensemble vide. Ne pas la tester pour $n \geqslant 5$ (pourquoi ?)

Test : on doit avoir

```
>>> parties([1,2,3])
[[], [1], [2], [1, 2], [3], [1, 3], [2, 3], [1, 2, 3]]
```

Exercice 11. On continue à représenter un ensemble par une liste.

1. Supposons que dans une liste `L` de longueur n on puisse retirer, pour tout $0 \leqslant i < n$, l'élément `x` d'indice i ; la liste restante s'appelle `M`. Comment exprimer l'ensemble des permutations de `L` qui commencent par `x` à partir de l'ensemble des permutations de `M` ?
2. Alors comment exprimer récursivement l'ensemble de *toutes* les permutations de `L` ?
3. Écrire une fonction `permutations(L)` qui retourne la liste de toutes les permutations possibles de `L`.

Si cela marche bien, on a par exemple :

```
>>> permutations([1, 2, 3])
[[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]
```

Mais cela fonctionne quelque soit le type des éléments de `permutations(L)` (entiers, chaines...)