TP 10
Récursivité

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 :

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

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)
```

Tester la fonctions avec des petites valeurs de n.

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

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

La fonction est appelée elle-même successivement avec des valeurs de n de plus en plus petites, jusqu'à la valeur $n = 0$, puis les retours se font dans l'ordre inverse des appels. Ce qu'on observe s'appelle la **pile d'appels**, voir l'annexe IV : chaque fois que la fonction s'appelle elle-même, elle note dans la mémoire l'endroit exact où elle était afin de pouvoir y revenir quand l'appel est terminé, et elle revient au *dernier* endroit qui a été noté.

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. Cela nécessite d'abord de repenser les programmes !

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 ? 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

À retenir

Pour écrire une fonction récursive, il est impératif d'avoir d'abord au brouillon une bonne formulation mathématique du problème. Similairement au raisonnement par récurrence, une fonction récursive contient toujours :

- Une condition initiale (en général le cas $n = 0$ ou $n = 1$), où la fonction renvoie directement une valeur, toujours avec `return` (sinon ça ne marche pas !),
- Un appel récursif, où la fonction `f(n)` s'appelle elle-même avec des valeurs plus petites, typiquement $n - 1$ ou $n/2$.

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 :

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

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.

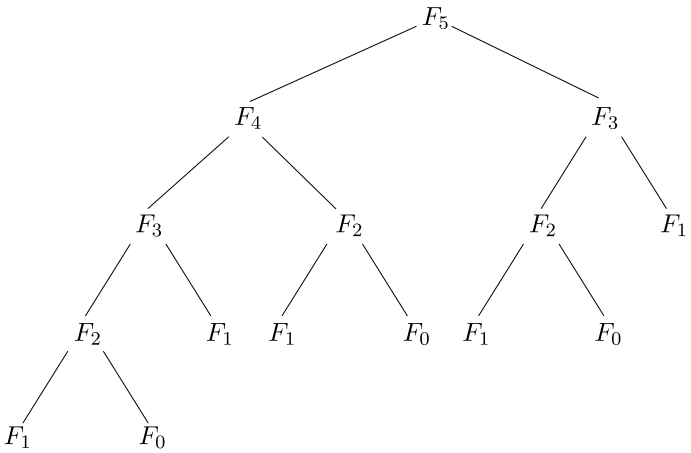

Figure 1. – Arbre des appels pour `fibonacci(5)`.

Exercice 5 *Exponentiation rapide*

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

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

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 ?

$$a^{16} \qquad a^{15} \qquad a^{24}$$

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 `n % 2` pour tester la parité et on utilise 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 $k \in \mathbb{Z}$:

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

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 à 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 $0 \leqslant k \leqslant n$. Par exemple `binome_liste(3)` doit renvoyer `[1, 3, 3, 1]`.

   Écrire cette fonction, de façon récursive.

À retenir

La récursivité est une **méthode de programmation**.
- Elle a l'avantage d'être parfois plus proche du langage mathématique et plus simple à programmer,
- Mais elle peut donner lieu à des programmes moins efficaces, notamment à cause du phénomène d'arbre des appels comme pour `fibonacci`.

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 ou 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 tranche `s[1:]`. Mais attention au mot vide `""` qui apparait naturellement quand on a enlevé plusieurs fois de suite la première lettre !

1. Au brouillon, proposer une formulation récursive du problème, en termes comme ci-dessus de première lettre et de comparaisons 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 $\{1, 2, ..., n\}$ 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 $\{1, 2, ..., n\}$. 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 $\{1, 2, ..., n-1\}$ — et celles contenant n, obtenues à partir des parties de $\{1, 2, ..., n-1\}$ en leur rajoutant n.

Écrire la fonction récursive `parties(n)`.

Par exemple, l'ensemble des parties de $\{1, 2, 3\}$ est

$$\mathcal{P}(\{1,2,3\}) = \Big\{\emptyset, \{1\}, \{2\}, \{1,2\}, \{3\}, \{1,3\}, \{2,3\}, \{1,2,3\}\Big\}$$

et la fonction produit le résultat (l'ordre exact dépend de la façon d'écrire la fonction récursive)

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

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. Reprenons notre exemple

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

Lors d'un appel tel que `factoriel(5)`, alors la fonction démarre avec $n = 5$ et on se retrouve à devoir appeler `factoriel(4)`. Mais avant de passer à `factoriel(4)`, il faut d'abord sauvegarder dans la mémoire l'endroit précis où nous en sommes de `factoriel(5)`, c'est là qu'une fois qu'on aura obtenu le résultat du calcul de `factoriel(4)` on pourra le multiplier par 5 puis le renvoyer. À chaque appel de la fonction à un rang $n-1$, l'ordinateur doit sauvegarder diverses informations sur l'état de la fonction au rang n avant de sauter au rang $n-1$, et ces informations sont organisées selon une **pile** : exactement comme une pile d'assiettes, les nouvelles informations sont posées directement par-dessus les anciennes, et quand on veut les récupérer, on a d'abord accès au dernier élément qui a été empilé. Ainsi quand l'appel `factoriel(4)` — dont les informations étaient sur le dessus de la pile — se termine, on revient directement à `factoriel(5)`.

Cette structure de pile s'observe très bien en comparant les programmes suivants, dont la seule différence est l'ordre des instructions entre le `print` et l'appel récursif : tester (par exemple avec $n = 10$) et comparer

```python
def boum(n):
    if n == 0:
        print("BOUM")
    else:
        print(n)
        boum(n-1)
```

```python
def top(n):
    if n == 0:
        print("TOP")
    else:
        top(n-1)
        print(n)
```

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
```