Fonctions récursives

Définition

Définition

Une fonction récursive est une fonction qui s'appelle elle-même dans sa définition.

Exemple : La fonction factorielle

```
def factorielle(n):
    assert n>=0 and int(n)==n
    if n==0:
        return 1
    else:
        return n*factorielle(n-1)
```

- ⊕ certains algorithmes s'écrivent naturellement avec une fonction récursive.
- ⊖ Le nombre d'appels récursifs est limité avec Python

- Rappeler une définition possible non récursive de la fonction `factorielle`.

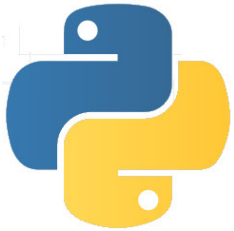
Concevoir une fonction récursive

La conception d'une fonction récursive est proche du principe de récurrence. On définit une fonction `f` prenant en argument un entier naturel `n`. On commence par donner une instruction dans le cas où $n = 0$ puis on exprime `f(n)` en fonction de `f(n-1)`.

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

Remarques :

- Il est possible de définir des fonctions récursives `f(n)` qui font appel à des valeurs de rangs strictement inférieurs à $n-1$.
- Lorsqu'une fonction `f` fait appel dans sa définition à une fonction `g` et réciproquement, on dit que `f` et `g` sont "mutuellement récursives".
- Comme pour les fonctions non récursives, on peut analyser l'algorithme mis en œuvre dans cette fonction et étudier, en particulier : la terminaison, la correction et la complexité de l'algorithme récursif.



Terminaison

Rappel : Pour vérifier la terminaison d'une structure itérative **while**, on identifie un variant de boucle qui :

- prend des valeurs entières positives ;
- diminue strictement à chaque passage dans la boucle.

Pour démontrer (de manière naïve) la terminaison d'une fonction récursive, on identifie de la même manière un entier naturel qui décroit à chaque appel récursif

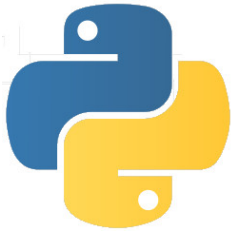
Exemple pour la fonction `factorielle(n)` :

Correction

Rappel : Pour vérifier la terminaison d'une structure itérative, on cherche un invariant de boucle. Un invariant de boucle est une propriété qui, si elle vraie juste avant un tour de boucle, reste vraie après son exécution. Cette propriété se démontre généralement par récurrence.

Pour démontrer la correction d'une fonction récursive, on procèdera aussi généralement par récurrence.

Exemple pour la fonction `factorielle(n)` :



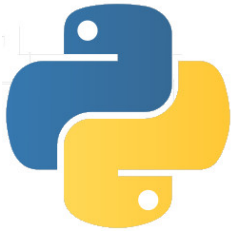
Complexité

Pour déterminer la complexité d'une fonction récursive simple, on peut compter le nombre d'opérations effectuées au rang 0 et au rang n. On obtient alors une suite.

Exemple pour la fonction `factorielle(n)` :

Exercice : Fonction puissance

1. Définir une fonction `puissance` récursive qui prend comme argument un nombre positif a et une puissance entière positive p et renvoie a^p.
2. Vérifier la terminaison et la correction de l'algorithme.
3. Analyser la complexité de l'algorithme.
4. Proposer une version non récursive de cet algorithme.
5. Vérifier la terminaison et la correction de l'algorithme non récursif proposé, puis étudier sa complexité.



Exercice : Suite de Fibonacci

La suite de Fibonacci est une suite d'entiers dans laquelle chaque terme de la suite est la somme des deux termes qui le précède :

$$\begin{cases} F_0 = 0 \\ F_1 = 1 \\ F_n = F_{n-1} + F_{n-2} \quad \forall n \geq 2 \end{cases}$$

Elle doit son nom au mathématicien italien Leonardo Fibonacci qui, dans un problème récréatif posé en 1202, décrit la croissance d'une population de lapins : "Un homme met un couple de lapins dans un lieu isolé fermé. Combien de couples obtient-on en un an si chaque couple engendre tous les mois un nouveau couple à compter du troisième mois de son existence ?"

1. Écrire une fonction récursive qui prend en argument n et qui renvoie F_n.
2. Répondre à la question posée par Leonardo Fibonacci.

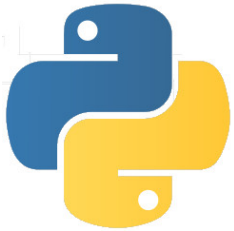
Exercice : Approximation de $\sqrt{a}$

On considère la suite suivante :

$$\begin{cases} u_0 = 1 \\ u_n = \frac{1}{2}\left(u_{n-1} + \frac{a}{u_{n-1}}\right) \end{cases} \quad \forall n \in \mathbb{N}^*$$

1. Définir une fonction récursive `racine` qui prend comme argument une valeur de a et une valeur entière n et renvoie u_n.
2. Analyser la complexité de l'algorithme.

Pour définir la fonction `racine`, un élève propose la définition suivante :




```
def racine(a,n):
        assert a>=0
        if n==0:
                return 1
        else:
        x=racine(a,n-1)
                return 0.5*(x+a/x)
```

5. Quel est l'intérêt de la variable x ?
6. Proposer une version non récursive de cette fonction et analyser sa complexité.
7. Pour $a = 3$, proposer les commandes Python permettant de déterminer la valeur minimale de n pour laquelle `racine(3,n)` approche $\sqrt{3}$ à 10^{-8} près.