Notion de fonctions récursive

1. Activité d'introduction : de l'itératif au récursif

Un peu de maths : les suites arithmétiques

On rappelle qu'une suite , de premier terme est dite **arithmétique** si et seulement si

où .

La définition donnée ci-dessus est une définition dite par **récurrence**, c'est-à-dire qu'on définit le terme de rang à partir du terme de rang .

Cette suite peut-être définie par une **formule explicite** :

Exercice

Enoncé

Construire une fonction `maSuiteArithmetique(n)` qui calcule le -ième terme de la suite arithmétique de premier terme `3` et de raison `7`. Quelle formule avez-vous utilisée ?

Une solution

Au vu de l'énoncé, je prends le pari que la majorité d'entre vous avez utilisé la **formule explicite** avec un code de la forme suivante :

```
def maSuiteArithmetique(n) :
    return 3 + n*7
```

C'est évidemment la solution la plus simple. Dans ce cas, tout comme pour les *suites géométrique*, il est inutile de compliquer le code, nous obtenons directement la solution par un simple calcul algébrique.

Encore des maths : les suites arithmético-géométriques

Une suite , de premier terme est dite **arithmético-géométrique** si et seulement si

où .

Encore une fois, la définition donnée ci-dessus est une définition dite par **récurrence**, c'est-à-dire qu'on définit le terme de rang à partir du terme de rang .

Exercice

Enoncé

Construire une fonction `maSuiteAG(n)` qui calcule le -ième terme de la suite arithmético-géométriqueq de premier terme `7` et définie par :

Quelle formule avez-vous utilisée ?

Une solution probable

Ici nous n'avons qu'une formule - sauf pour les petits malins qui seront allé voir sur wikipedia - donc on doit utiliser un processus de répétition des opérations à partir de `7`.

On peut bien sûr appliquer une boucle `pour` dans notre fonction :

```
def maSuiteAG(n) :
    u = 7
    for i in range(1,n+1) : # j'utilise ce range plutôt que range(n) car le i utilisé
correspond au terme du rang calculé.
        u = -2*u + 5
    return u
```

Deux remarques :

- dans ce code, je ne vérifie pas que , et il faudrait... ;
- dans le cas où , la boucle `for` n'est pas effectuée.

Une version plus correcte serait donc celle-ci :

```
def maSuiteAG(n) :
    if type(n)!= int or n<0 :
        raise ValueError("n must be a non negative integer")
    u = 7
    for i in range(1,n+1) : # j'utilise ce range plutôt que range(n) car le i utilisé
correspond au terme du rang calculé.
        u = -2*u + 5
    return u
```

Une telle fonction est dite **itérative**, car elle utilise une boucle de répétitions pour parvenir au résultat souhaité.

C'est vraiment dommage, dans le premier exercice, on utilise simplement **la formule explicite**, alors que dans le deuxième cas, on est obligé de réfléchir à l'algorithme. Ce serait si simple de pouvoir utiliser *directement* la **formule récursive**, comme dans le code ci-dessous :

```
def maSuiteAGR(n) :
```

```
    return -2*maSuiteAGR(n-1) + 5
```

ET si on tentait le coup de tester, en prenant par exemple...

Késako ?

2. Principe de récursivité

Fonction récursive

Une fonction est dite **récursive** quand elle s'appelle elle-même, une ou plusieurs fois.

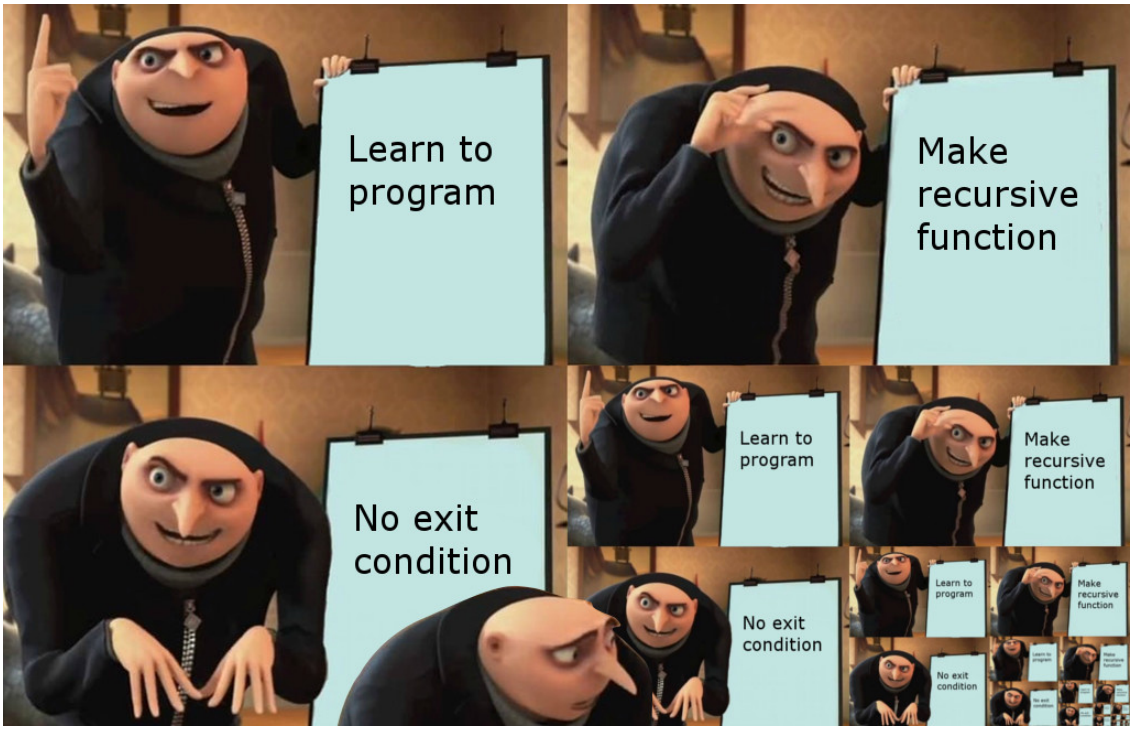
Des problèmes

Décomposons l'instruction l'appel à `maSuiteAGR(3)` :

- `maSuiteAGR(3)` doit calculer `-2*maSuiteAGR(2) +5`, et donc doit calculer :
  - `maSuiteAGR(2)`, qui doit calculer `-2*maSuiteAGR(1) +5`, et donc doit calculer :
    - `maSuiteAGR(1)`, qui doit calculer `-2*maSuiteAGR(0) +5`, et donc doit calculer :
      - `maSuiteAGR(0)`, qui doit calculer `-2*maSuiteAGR(-1) +5`, et donc doit calculer :
        - `maSuiteAGR(-1)`, qui doit calculer `-2*maSuiteAGR(-2) +5`, et donc doit calculer :
          - ...

"HELP ! Mais ça s'arrête quand !" me direz-vous !

Et bien jamais, en théorie, car nous n'avons pas **précisé de condition d'arrêt**.



Mais en réalité cette instruction s'arrêtera quand python aura levé une erreur de type `RecursionError`, qui signifie qu'une limite aura été atteinte (nous en parlerons plus tard pour lever toute ambiguité).

Supprimer le problème : le cas d'arrêt ou cas de base

Pour supprimer le problème précédent, revenons aux maths : dans une définition par récurrence de suite, on signale **toujours** la valeur du premier terme (qui peut être , ou , ou même selon le problème et la définition de l'indice). Or dans notre fonction `maSuiteAGR`, jamais nous ne précisons ce cas, c'est-à-dire que quand , alors la suite vaut . Rajoutons-donc cette condition dans la fonction :

```
def maSuiteAGR(n) :
    if n== 0 : # Cas de base
        return 7
    else : # Cas récursif
        return -2* maSuiteAGR(n-1) + 5
```

Et testons de nouveau `maSuiteAGR(3)` :

- `maSuiteAGR(3)` doit calculer `-2*maSuiteAGR(2) +5`, et donc doit calculer :
  - `maSuiteAGR(2)`, qui doit calculer `-2*maSuiteAGR(1) +5`, et donc doit calculer :
    - `maSuiteAGR(1)`, qui doit calculer `-2*maSuiteAGR(0) +5`, et donc doit calculer :
      - `maSuiteAGR(0)`, qui **maintenant renvoie 7** !
    - donc `maSuiteAGR(1)` renvoie `-2*7+5` soit `-9` ;
  - donc `maSuiteAGR(2)` renvoie `-2*(-9)+5` soit `23` ;
- donc `maSuiteAGR(3)` renvoie `-2*23+5` soit `-41`.

Non seulement la fonction s'arrête, mais en plus elle renvoie la bonne valeur, c'est-à-dire .

Récapitulons

Pour utiliser une **fonction récursive** correctement, il faudra distinguer :

- le ou les **cas d'arrêts** (ou **cas de base**), c'est-à-dire des cas particuliers pour lesquels la valeur (ou l'objet) renvoyé par la fonction est connu ;
- le **cas récursif**, pour lequel la fonction s'appelle elle-même, une ou plusieurs fois.

Exemple commenté

La somme des premiers entiers est la somme :

Comment faire pour construire une fonction récursive `sommeR(n)` qui effectue la somme des premiers entiers, avec passé en argument.

- Quel est le **cas récursif** ?
  On a ,

  donc le cas récursif est `sommeR(n) = sommeR(n-1) + n`
- Quel est le cas de base ?
  Il y a plusieurs possibilités, soit en partant de l'indice 0 car `sommeR(0)=0`, soit en partant de l'indice 1, car `sommeR(1) = 1`.

Une implémentation récursive possible est alors :

```
def sommeR(n) :
    if n== 0 : # Cas de base
        return 0
    else : # Cas récursif
        return sommeR(n-1) + n
```

3. Applications directes

Exercice : factorielle

Enoncé

On rappelle que la factorielle d'un entier naturel est donné par :

1. Écrire une fonction **itérative** `factorielle(n)` qui renvoie la factorielle d'un entier naturel donné, et lève une `ValueError` si n'est pas entier ou est négatif.
2. Écrire une fonction **récursive** `factorielleR(n)` qui renvoie la factorielle d'un entier naturel donné, et lève une `ValueError` si n'est pas entier ou est négatif.

Solution Itérative

```
def factorielle(n) :
    if type(n) != int or n<0 :
        raise valueError("n must be a positive integer")
    produit = 1
    for i in range(1,n+1) :# On peut effectivement partir de 2, et gagner un tour de
boucle.
        produit =produit*i
    return produit
```

Solution récursive

```
def factorielleR(n) :
    if type(n) != int or n<0 :
        raise valueError("n must be a positive integer")
    if n==0 or n==1 :
        return 1
    else :
        return factorielleR(n-1)*n
```

Exercice : étoiles

Enoncé

1. Implémenter une procédure[1] **itérative** `etoile(n)` qui écrit dans le Shell Python un triangle formé de caractères `*` tels que dans l'exemple suivant :

```
>>> etoileR(5)
*
**
***
****
*****
```

2. Implémenter une procédure **récursive** `etoileR(n)` qui effectue le même travail.

Solution Itérative

```
1  def etoile(n) :
2      if type(n)!= int or n<=0 :
3          raise valueError("n must be a positiv integer")
4      for i in range(1, n+1) :
5          print("*"*i)
```

Solution récursive

```
1  def etoileR(n) :
2      if type(n)!= int or n<=0 :
3          raise valueError("n must be a non null positiv integer")
4      if n == 1 :
5          print("*")
6      else :
7          etoileR(n-1)
8          print("*"*n)
```

Coefficients binomiaux et triangle de Pascal

1. Soient et deux réels quelconques.

   Développez les expressions suivantes :

   a.

   b.

   c.

   d.

   e.

   Solution

2. Complétez deux lignes supplémentaires du tableau suivant, nommé **Triangle de Pascal**:

   Solution

3. On appelle coefficient binomial de rang et de degré le nombre du **Triangle de Pascal** correspondant à la -ième ligne et à la _ième colonne. Ce nombre est noté .

   Comment exprimer récursivemment ce coefficient ?

   Solution

4. Implémenter une fonction `binomeR(n, p)` qui renvoie la valeur du coefficient binomial du triangle de Pascal.

   Solution

5. *Facultatif :* Implémenter une fonction `developpe(n)` qui renvoie la chaîne de caractères correspondant au développement de .

Solution

Exemple de code utilisant la récursivité

Lorsqu'on veut lister tous les fichiers situés dans un répertoire et dans tous les sous-répertoires de celui-ci, on peut utiliser une méthode récursive telle que présentée dans le code suivant :

```
import os

def explorer_repertoire(repertoire):
    for item in os.listdir(repertoire):
        chemin = os.path.join(repertoire, item)
        if os.path.isdir(chemin):
            explorer_repertoire(chemin)
        else:
            print(chemin)

explorer_repertoire("/mon_repertoire")
```

1. une procédure est une fonction sans valeur de retour, c'est-à-dire une fonction renvoyant toujours `None`. ↩