RÉSUMÉ 31 - RÉCURSIVITÉ AVEC PYTHON

Lien vers l'activité : Récursivité avec Python
www.infoforall.fr - Dernière modif. : 08 03 2021

25.1 FONCTION RÉCURSIVE

Une **fonction récursive** est une fonction qui a comme propriété de lancer un appel éventuel à une autre instance de la même fonction.
Plus simplement, on dira que c'est une fonction qui s'appelle elle-même.

25.2 STRUCTURE D'UNE FONCTION RÉCURSIVE

Une fonction récursive peut être structuré de cette façon :

```
1  def fonctionRec(paramV) :
2      if condition_arret :
3          return cas_de_base
4      else :
5          return cas_recursif
```

Je note le paramètre **paramV** pour dire paramètre variable : lors des différents appels, il doit prendre des valeurs différentes et donc varier.

Quelques définitions :

- Le **cas de base** correspond à la réponse non-recursive que peut la fonction. C'est cette réponse qui va faire cesser la succession d'appels.
- Le **cas récursif** correspond à une réponse qui intégre un **nouvel appel à la fonction récursive**. Si on veut que les appels puissent s'arrêter à un moment, il faut bien attendu que cet appel se fasse avec un paramètre **paramV** légèrement différent.
- La **condition d'arrêt** correspond simplement à la condition qui permet d'aboutir au cas de base.

25.3 EXEMPLE DE FONCTION RÉCURSIVE OÙ LE PARAMÈTRE EST UN TYPE SIMPLE

```
1  def factorielle(n) :
2       if n <= 0 :
3          return 0
4      else :
5          return n * factorielle(n-1)
```

Condition d'arrêt : `if n <= 0 :`
Cas de base : `return 0`
Cas récursif : `return n * factorielle(n-1)`

RÉCURSIVITÉ NON-TERMINALE :

Qu'est-ce que la récursivité non-terminale ? Il s'agit des fonctions récursives dans lequel la réponse dans le cas récursif intègre une évaluation à faire en plus de l'évaluation du cas récursif suivant. Dans l'exemple ici, on doit additionner **n** et la réponse de l'appel suivant.

25.4 EXEMPLE DE FONCTION RÉCURSIVE OÙ LE PARAMÈTRE EST UN OBJET

Reprenons le cas de la recherche de la référence de la dernière Cellule d'une liste chaînée, dans le cas de l'implémentation en programmation objet.

```
1  def renvoyerCelluleFinale(cellule) :
2      if cellule.s == None :
3          return cellule
4      else :
5          return renvoyerCelluleFinale(cellule.s)
```

Condition d'arrêt : `if cellule.s == None :`
Cas de base : `return cellule`
Cas récursif : `return renvoyerCelluleFinale(cellule.s)`

RÉCURSIVITÉ TERMINALE :

Qu'est-ce que la récursivité terminale ? Il s'agit des fonctions récursives dans lesquelles la réponse dans le cas récursif ne contient qu'un appel récursif. C'est le cas ici. Certains langages sont capables d'optimiser ce code

et de ne pas produire d'empilement puisqu'il s'agit de faire renvoyer par la première fonction l'appel de la dernière.

25.5 EXEMPLE DE MÉTHODE RÉCURSIVE

On peut trouver des méthodes récursives qui possède juste un paramètre variable comme les cas précédents. Il s'agit des méthodes récursives qui agissent plus fois sur le même objet.
Mais il existe également des méthodes récursives qui changent d'objets à chaque appel. Reprenons le cas précédent mais en version méthode :

```
1       def renvoyerCelluleFinale(self) :
2           if self.s == None :
3               return self
4           else :
5               return self.s.renvoyerCelluleFinale()
```

Rien de neuf pour la condition d'arrêt et le cas de base si ce n'est que la variable qui contient la référence de l'objet se nomme **self**.
Par contre, on voit que l'appel récursif est un peu différent : l'appel est différent car l'objet sur lequel on fait agir la méthode n'est plus le même : il s'agit ici de l'instance de la Cellule suivante.
Si cela ne vous persuade pas, voici comment on peut voir l'appel récursif :

```
code 1          return self.s.renvoyerCelluleFinale()
code 2          return Cellule.renvoyerCelluleFinale(self.s)
```

25.6 SAVOIR CRÉER L'EMPILEMENT

Il suffit de lire progressivement la fonction à partir du premier appel et de suivre le cheminement. On écrit une représentation de cet empilement d'appel.
Le tout est de savoir ce qu'on appelle et à qui on doit répondre une fois atteint le cas final.

```
1   def serieCarre (n) :
2       assert n >= 0
3       if n == 0 :
4           return 0
5       else :
6           return n**2 + serieCarre(n-1)
```

- Appel à `serieCarre(3)` va renvoyer `9 + serieCarre(2)`
  - Appel à `serieCarre(2)` va renvoyer `4 + serieCarre(1)`
    - Appel à `serieCarre(1)` va renvoyer `1 + serieCarre(0)`
      - Appel à `serieCarre(0)` renvoie `0`

25.7 SAVOIR DÉPILER

On part dans l'autre sens : Last In First Out. On commence donc par faire répondre ici la dernière fonction : celle avec le paramètre 0. Et on dépile les appels et les réponses.

- Appel à `serieCarre(3)` renvoie `9 + 5` --> réponse finale de `14`
  - Appel à `serieCarre(2)` renvoie `4 + 1`, soit `5`
    - Appel à `serieCarre(1)` renvoie `1 + 0`, soit `1`
      - Appel à `serieCarre(0)` renvoie `0` à la fonction ci-dessus

24.8 EXEMPLE FINAL VOLONTAIREMENT MAIS INUTILEMENT COMPLIQUÉ

Voici une fonction récursive qui agit sur un string et qui renvoie un tableau crée par concaténation...

```
1   def inutile(texte, position=0) :
2       ok = "ABCDE0123456789"
3       if position >= len(texte) :
4           return []
5       else :
6           if texte[position] in ok :
7               return [texte[position]] + inutile(texte, position + 1)
8           else :
9               return inutile(texte, position + 1)
```

L'empilement provoqué par un appel à `inutile('Ab9dE')` :

- Appel à `inutile('Ab9dE')` va renvoyer `['A'] + inutile('Ab9dE', 1)`
  - Appel à `inutile('Ab9dE', 2)` va renvoyer `inutile('Ab9dE', 3)`
    - Appel à `inutile('Ab9dE', 3)` va renvoyer `['9'] + inutile('Ab9dE', 4)`
      - Appel à `inutile('Ab9dE', 4)` va renvoyer `inutile('Ab9dE', 5)`

        - Appel à `inutile('Ab9dE', 5)` va renvoyer `['E'] + inutile('Ab9dE', 6)`
            - Appel à `inutile('Ab9dE', 6)` renvoie `[]`

Le dépilement donne donc ceci :

- Appel à `inutile('Ab9dE')` va renvoyer `['A'] + ['9', 'E']` -> réponse finale `['A', '9', 'E']`
    - Appel à `inutile('Ab9dE', 2)` va renvoyer `['9', 'E']`
        - Appel à `inutile('Ab9dE', 3)` va renvoyer `['9'] + ['E']`
            - Appel à `inutile('Ab9dE', 4)` renvoie `['E']`
                - Appel à `inutile('Ab9dE', 5)` renvoie `['E'] + []`, soit `['E']`
                    - Appel à `inutile('Ab9dE', 6)` renvoie `[]`