NSI Terminale - Programmation

Récursivité - résumé

qkzk

2020/06/22

Récursivité

Définition

La récursivité est une démarche qui fait référence à l'objet même de la démarche à un moment du processus. En d'autres termes, c'est une démarche dont la description mène à la répétition d'une même règle. Par exemple :

- *écrire un algorithme qui s'invoque lui-même ;*
- *définir une structure à partir de l'une au moins de ses sous-structures.*

Algorithmes récursifs

Exemple

Ne faisons pas dans l'originalité :

$$n! = n \times (n-1)! \text{ si } n \geq 1 \text{ et } 0! = 1$$

Principe :

```
5! = 5 * 4!
   = 5 * 4 * 3!
   = 5 * 4 * 3 * 2!
   = 5 * 4 * 3 * 2 * 1!
   = 5 * 4 * 3 * 2 * 1 * 0!
   = 5 * 4 * 3 * 2 * 1 * 1
   = 120
```

Programmation

```
def fact(n):
    if n <= 1:
        return 1
    else:
        return n * fact(n-1)
```

```
>>> fact(5)
120
```

Déroulé d'un programme récursif

Chaque appel récursif s'ajoute à la pile des appels successifs de la fonction. Chaque appel possède son propre environnement, donc ses propres variables. La pile est nécessaire pour mémoriser les valeurs propre à chaque appel.

Attention : En Python la taille de la pile des appels récursifs est limitée. Si la récursivité est trop profonde et que l'on atteint cette limite, on déclenche une `RecursionError`.

Eléments Caractéristiques

1. **il faut au moins une situation qui ne consiste pas en un appel récursif**

```
if n <= 1:
   return 1
```

Cette situation est appelée **situation de terminaison** ou **situation d'arrêt** ou **cas d'arrêt** ou **cas de base**.

2. **chaque appel récursif doit se faire avec des données qui permettent de se rapprocher d'une situation de terminaison**

```
   return n * fact(n-1)
```

Il faut s'assurer que la situation de terminaison est atteinte après un nombre fini d'appels récursifs.

La preuve de terminaison d'un algorithme récursif se fait en identifiant la construction d'une suite strictement décroissante d'entiers positifs ou nuls.

Dans le cas de factorielle, il s'agit simplement de la suite des valeurs du paramètre.

Récursivité terminale

Un algorithme récursif simple est terminal lorsque l'appel récursif est le dernier calcul effectué pour obtenir le résultat. Il n'y a pas de "calcul en attente". L'avantage est qu'il n'y a rien à mémoriser dans la pile.

Exemple d'algorithme récursif terminal

Prédicat de présence d'un caractère dans une chaîne :

Un caractère c **est présent** dans une chaîne s non vide, s'il est le premier caractère de s ou s'il **est présent** dans les autres caractères de s. Il n'est pas présent dans la chaîne vide.

```
def est_present(c, s):
    if s == '':
        return False
    elif c == s[0]:
        return True
    else:
        return est_present(c, s[1:])
```

Rendre terminal un algorithme récursif

On utilise un **accumulateur**, passé en paramètre, pour calculer le résultat au fur et à mesure des appels récursifs.

La valeur de retour du cas de base devient la valeur initiale de l'accumulateur et lors d'un appel récursif, le "calcul en attente" sert à calculer la valeur suivante de l'accumulateur.

Ainsi on obtient :

```
def fact_term(n, acc = 1):
    if n <= 1:
        return acc
    else:
        return fact_term(n-1, acc * n)
```

et

```
def occurrences_term(c,s, acc = 0):
    if s == "":
       return acc
    elif c == s[0]:
         return occurrences_term(c, s[1:], acc + 1)
    else:
         return occurrences_term(c, s[1:], acc)
```

Récursivité multiple

Un algorithme récursif est multiple si l'un des cas qu'il distingue se résout avec plusieurs appels récursifs.

C'était le cas de l'algorithme de dérivation.

Autre exemple avec le tri rapide

Retour sur les coefficients binomiaux

Vous connaissez aussi la relation de récurrence (Triangle de Pascal) :

- $C(n,p) = 1$ si $n = p$ ou $p = 0$
- $C(n,p) = C(n-1,p) + C(n-1,p-1)$ pour $n > p > 0$

Ce qui donnerait en Python

```
def C(n, p):
    if p == 0:
        return 1
    elif n == p:
        return 1
    else:
        return C(n-1, p-1) + C(n-1, p)
```

On peut observer l'explosion combinatoire du nombre d'appels récursifs et les redondances des calculs.

Structures récursives

Les listes de Python sont des **tableaux dynamiques**, c'est-à-dire des tableaux dont *la taille peut varier.*

De manière plus formelle, les listes sont des structures de données dynamiques et intrinsèquement récursives.

Elles se définissent ainsi :

Une liste d'éléments de type **T** est

- soit la liste vide
- soit un couple *(t,R)* où t est de type **T** et R est une liste d'éléments de type **T**

Dans le cas d'une liste non vide *(t,R)* :

- t est le premier élément de la liste aussi appelée **tête** de la liste
- R est la liste des éléments qui suivent t, également appelée **reste** de la liste

On parle pour de telles structures de **listes chaînées**, qui se distingue donc des listes par tableaux.

Listes chainées : plus efficaces que les tableaux pour supprimer un élément de la liste, ou en insérer un.

Avec cette définition des listes, la définition d'une constante pour la liste vide (`[]`) et de primitives permettant

- de construire un couple *(x, R)* (`[x]+R`)
- d'accéder à la tête d'une liste non vide (`l[0]`)
- d'accéder au reste d'une liste non vide (`l[1:]`)

suffit pour définir tous les traitements sur les listes.

L'écriture de ces traitements se fait alors à l'aide de fonctions récursives.

Exemple d'algorithme récursif sur les listes

Longueur d'une liste :

```
def length(liste):
    if liste == [] :
        return 0
    else:
        return 1 + length(liste[1:])
```