Informatique PCSI
Prérequis TP 5 : fonctions récursives

Notion de récursivité

Considérons une fonction qui dessine des carrés dont les dimensions des côtés sont contenus dans une liste. Nous disposons d'une fonction qui dessine un carré et d'une fonction qui l'appelle de manière répétitive à l'aide d'une boucle. Nous utilisons le module Turtle.

```
from turtle import *
def carre(n):
    for i in range(4):
        forward(n)
        left(90)

def dessine1(liste, angle):
    for d in liste:
        left(angle)
        carre(d)
```

Pour dessiner n carrés, on dessine un carré, puis on dessine $n-1$ carrés. Ainsi, la fonction qui dessine n carrés peut être la même que celle qui dessine 1 carré ou $n-1$ carrés. C'est juste la valeur de son paramètre qui change. Après avoir dessiner un premier carré, elle s'appelle elle-même pour dessiner $n-1$ carrés. C'est le principe de récursivité.

```
def dessine2(liste, i, angle):
    if i >= 0 and i < len(liste):
        left(angle)
        carre(liste[i])
        dessine2(liste, i+1, angle)
```

On écrit les instructions `lst = [100, 90, 80, 70, 60]`, `a = 20`, puis `dessine1(lst, a)` pour obtenir un premier dessin ou `dessine2(lst, 0, a)` pour obtenir un dessin identique.

Considérons la fonction `affiche` définie ainsi :

```
def affiche(k):
    if k < 10:
        print(k)
        affiche(k+1)
```

Si nous écrivons l'instruction `affiche(0)`, la fonction `affiche` est appelée avec le paramètre 0 qui vérifie le test `k < 10`. Après l'affichage de 0, la fonction `affiche` est à nouveau appelée mais avec le paramètre 1, puis après l'affichage de 1, la fonction `affiche` est à nouveau appelée mais avec le paramètre 2, et ceci continue avec l'affichage de `k` et l'appel qui suit tant que `k < 10`.

Dans ce code, nous n'avons ni boucle de type `while` ou `for`, ni instruction d'affectation.

Une fonction qui s'appelle elle-même permet de remplacer une boucle. On parle alors de programme récursif par opposition à itératif.

Considérons la fonction `rebours1` qui prend en argument un entier naturel `n` et effectue un compte à rebours, par exemple affiche 5, 4, 3, 2, 1, 0.

```
def rebours1(n):
    """n est un entier naturel
       affiche les entiers de n à 0"""
    while n >= 0:
        print(n)
        n = n - 1
```

Sur le modèle de la fonction `affiche`, nous pouvons écrire une fonction `rebours` qui produit le même résultat que la fonction `rebours1`.

```
def rebours(n):
    if n >= 0:
        print(n)
        rebours(n-1)
```

Quel est le déroulement de l'exécution avec par exemple le paramètre `n` prenant la valeur 3 ?
Si le paramètre de la fonction `rebours` est positif, le test est vérifié et l'exécution produit :
affichage de 3, puis appel `rebours(2)`,
`-` > affichage de 2, puis appel `rebours(1)`,
`- - -` > affichage de 1, puis appel `rebours(0)`
`- - - - -` > affichage de 0, puis appel `rebours(-1)`,
Le nombre -1 est strictement négatif, donc la condition `n >= 0` n'est plus vérifiée et l'exécution du programme est terminée. La même fonction `rebours` a été appelée cinq fois, et cette fonction a appelé la fonction `print` quatre fois pour les affichages des nombres de 3 jusqu'à 0.

Fonction récursive

Définitions

- Une fonction est dite *récursive* si elle s'appelle elle-même.

  Nous parlons alors d'*appel récursif*. La fonction `rebours` est récursive.
- Par opposition, la fonction `rebours1` est *itérative*.

Voici deux autres exemples avec le module Turtle :

```
from turtle import *

def dessine(n):
    if n > 0:
        forward(n)
        right(90)
        dessine(n-5)
```

Si `n` est un entier strictement positif, la tortue trace un segment de longueur `n` pixels puis tourne à droite de 90 degrés. Dans l'appel récursif le nouveau paramètre est `n-5`, donc la tortue trace un segment de longueur `n-5` pixels puis tourne à droite de 90 degrés, et ainsi de suite. Ce processus continue tant que $n > 0$. Tester par exemple avec l'instruction `dessine(200)`.

Le deuxième exemple produit la courbe du dragon (ou « fractale du dragon » ou « courbe de Heighway » ou « dragon de Heighway ») qui a été étudiée pour la première fois par les physiciens de la NASA John Heighway, Bruce Banks, et William Harter.

```
def dragon(n, s):
    if n == 0:
        forward(5)
    else:
        dragon(n-1, 1)
        right(s * 90)
        dragon(n-1, -1)

speed('fastest')
ht()
dragon(10, 1)
```

Deux exemples avec la fonction `print` :

```
T = 8 # exécuter les fonctions avec pour paramètre n = T

def affiche1(n):
    if n > 0:
        print(n*' ' + '/' + 2*(T-n)*" " + '\\')
        affiche1(n-1)

def affiche2(n):
    if n > 0:
        print(n*' ' + '/' + 2*(T-n)*" " + '\\')
        affiche2(n-1)
        print(n*' ' + '\\' + 2*(T-n)*" " + '/')
```

Un exemple de calcul avec la multiplication de deux entiers naturels qui est effectuée uniquement à l'aide d'additions :

```
def produit(n, p):
    """n et p sont deux entiers naturels
       renvoie le produit de n par p"""
    if p == 0:
        return 0
    else:
        return n + produit(n, p-1)
```

Examinons l'exécution du programme pour obtenir `produit(4, 3)` :

le calcul de `4 + produit(4, 2)` est en attente du résultat de `produit(4, 2)` ;
- > le calcul de `4 + produit(4, 1)` est en attente du résultat de `produit(4, 1)` ;
- - - > le calcul de `4 + produit(4, 0)` est en attente du résultat de `produit(4, 0)` ;
- - - - - > la valeur de `produit(4, 0)` est obtenue, c'est 0 ;
- - - > l'addition `4 + produit(4, 0)` est effectuée, le résultat est 4, $(4 + 0)$;
- > l'addition `4 + produit(4, 1)` est effectuée, le résultat est 8, $(4 + 4)$;
l'addition `4 + produit(4, 2)` est effectuée, le résultat est 12, $(4 + 8)$;
la valeur renvoyée est 12 (le produit de 4 par 3).

Deux remarques importantes déduites du programme précédent doivent être généralisées.

- Le test `p == 0` est une condition d'arrêt. Il peut y avoir plusieurs conditions. Au moins une condition d'arrêt est obligatoire afin que le programme ne boucle pas indéfiniment.
- Les valeurs passées en paramètres dans les appels récursifs constituent une suite de nombres qui décroit vers la valeur d'arrêt. Dans l'exemple, la valeur n -1 passée en paramètre permet de faire décroître la valeur du paramètre. Pour que le programme ne boucle pas indéfiniment, il est impératif que le ou les paramètres atteignent une valeur d'arrêt après un nombre fini d'appels.

0.1 Principes généraux

- Une fonction récursive doit contenir une ou des conditions d'arrêt. Sinon le programme boucle indéfiniment.
- Les valeurs passées en paramètres dans les appels récursifs doivent être différentes. Sinon la fonction s'exécute à chaque appel de manière identique et continue donc de s'exécuter indéfiniment.
- Après un nombre fini d'appels, la ou les valeurs passées en paramètres doivent permettre de satisfaire la condition d'arrêt.

Dans le corps d'une boucle `while`, des instructions d'affectation permettent de modifier les valeurs des variables afin d'assurer l'arrêt de la boucle. Dans une fonction récursive, c'est un changement des valeurs des paramètres utilisés dans l'appel récursif qui assure l'arrêt des appels.

Un point essentiel est à noter concernant l'exécution des fonction `rebours` et `produit`. L'exemple de la fonction `rebours` nous montre que le code :

```
Si condition:
    appel récursif
```

s'exécute de la même manière que le code :

```
Tant que condition:
    instructions
```

Mais avec la fonction `produit`, des calculs ne peuvent pas être effectués avant d'obtenir les valeurs renvoyées par les appels récursifs et la mémoire de la machine est donc sollicitée pour stocker des instructions en attente. On parle dans ce cas de récursivité profonde. Dans le cas de la fonction `rebours`, on parle de récursivité terminale (ce qui est semblable à une boucle `while`).

Dans le code de la fonction `rebours`, l'affichage avec la fonction `print` est suivi de l'appel récursif qui est la dernière instruction à être exécutée. Si nous permutons ces deux lignes de code, nous obtenons la fonction `compte` écrite ci-dessous. Son comportement est similaire à celui de la fonction `produit`.

```
def compte(n):
    if n >= 0:
        compte(n-1)
        print(n)
```