Spécialité NSI en terminale
Récursivité

Introduction

En classe de première, de nombreux programmes sont écrits avec des boucles et des affectations. La notion de fonction permet en particulier d'éviter de réécrire un code similaire à plusieurs endroits d'un programme.

Le style de programmation est :

- impératif, les séquences d'instructions sont exécutées l'une après l'autre ;
- itératif, des instructions sont exécutées dans des boucles `while` ou `for`.

Une fonction qui s'appelle elle-même va permettre de supprimer par exemple une boucle for.

Considérons le code suivant où la variable `i` prend successivement les valeurs entières de 0 à 9 qui sont ensuite affichées :

```
for i in range(10):
    print(i)
```

Notons qu'à la fin de ce code, une variable `i` continue d'exister et a la valeur 9. Considérons alors la fonction `affiche` définie ainsi :

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

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 appelée avec le paramètre 1, puis après l'affichage de 1, la fonction `affiche` est appelée avec le paramètre 2, et ceci continue avec l'affichage de `k` et l'appel qui suit tant que `k < 10`.

La fonction `affiche` est un premier exemple de fonction récursive. Remarquons que dans ce code, nous n'avons ni boucle de type `while` ou `for`, ni instruction d'affectation et aucune variable n'existe en dehors de la fonction.

Nous souhaitons effectuer un compte à rebours, par exemple afficher 5, 4, 3, 2, 1, 0.

Considérons pour cela la fonction `rebours1` qui prend en argument un entier naturel `n`.

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

Cette fonction permet d'afficher les entiers naturels de `n` à 0. Elle est écrite avec une boucle `while` qui pourrait être remplacée par une boucle `for`, comme dans le code suivant :

```
def rebours2(n):
    """n est un entier naturel
       affiche les entiers de n à 0"""
    for i in range(n+1):
        print(n - i)
```

Dans les deux cas, la fonction `print` est appelée avec un paramètre prenant successivement la valeur `n`, `n-1`, ..., 0.

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

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

Examinons le déroulement de l'exécution avec par exemple le paramètre `n` prenant la valeur 3.
Le nombre 3 est positif donc nous obtenons :
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, les fonctions `rebours1` et `rebours2` sont des fonctions itératives.

— De manière générale, un sous-programme est dit récursif s'il s'appelle lui-même.

Dans les exemples précédents nous avons utiliser la fonction `print` pour afficher des valeurs dans la console Python. Nous pouvons aussi dessiner ou effectuer des calculs. Voici un exemple avec le module Turtle :

```
from turtle import *

def dessine(n):
```

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

dessine(200)
```

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 $n > 0$.

Dans l'exemple qui suit, la fonction `dessin` trace deux traits horizontaux de longueur `taille`. La fonction `trace` est récursive, elle appelle la fonction `dessin` et s'appelle elle-même avec un paramètre de plus en plus petit passé à la fonction `dessin`.

```
from turtle import *

def dessin(taille):
    up()
    goto(-taille//2, -taille)
    down()
    forward(taille)
    up()
    goto(-taille//2, taille)
    down()
    forward(taille)

def trace(taille):
    if taille > 0:
        dessin(taille)
        trace(taille-10)

trace(200)
```

Après les exemples de dessin, voici un exemple de calcul avec une multiplication de deux entiers naturels 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)

print(produit(4, 3))
```

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).

Nous pouvons faire quelques remarques sur la fonction `produit` dont nous allons déduire des principes généraux.

— 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.

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.

Nous pouvons remarquer aussi une différence importante entre la fonction `rebours` et la fonction `produit`. L'exemple de la fonction `rebours` nous montre que les codes :

```
Tant que condition:
    instructions
```

et

```
Si condition:
    appel récursif
```

s'exécutent de la même manière.

Mais avec la fonction `produit` ce n'est pas le cas. 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 et pour la machine, ceci est semblable à une boucle `while`.

Donc, si un programme peut être plus facile à implémenter d'une manière récursive plutôt que d'une manière itérative, les coûts en temps et en espace doivent être étudiés afin de connaître son efficacité et par conséquent son utilité.

Une autre remarque : dans le corps de la boucle `while` se trouvent des instructions d'affectation pour modifier les valeurs des variables afin d'assurer l'arrêt de la boucle. Dans la fonction récursive, nous n'avons pas d'affectation. C'est un changement des valeurs des paramètres utilisés dans l'appel récursif qui assure l'arrêt des appels.

Récursivité terminale

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 qui est présentée ci-dessous. Son nom est `compte` et nous allons voir que son comportement et le résultat obtenu sont très différents de ceux de la fonction `rebours`.

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

Nous analysons la procédure `compte` comme nous l'avons fait pour la fonction `rebours`, mais cette fois l'affichage ne peut se faire que lorsque la procédure appelée a été exécutée.

Donc l'instruction `compte(4)`, par exemple, donne :

appel de `compte(3)`, et l'affichage de 4 est en attente ;

\- > appel de `compte(2)`, et l'affichage de 3 est en attente ;

\- - - > appel de `compte(1)`, et l'affichage de 2 est en attente ;

\- - - - - > appel de `compte(0)`, et l'affichage de 1 est en attente ;

\- - - - - - - > appel de `compte(-1)`, et l'affichage de 0 est en attente ;

\- - - - - - - > le test n'est plus vérifié, donc on procède aux affichages en attente ;

\- - - - - - - > affichage de 0 ;

\- - - - - > affichage de 1 ;

\- - - > affichage de 2 ;

\- > affichage de 3 ;

affichage de 4.

Les affichages en attente ont été « empilés » et sont ensuite « dépilés » dans l'ordre inverse. Cette notion est développée à la fin du chapitre et particulièrement au chapitre 3.

La fonction `rebours` est dite *récursive terminale* car l'appel récursif est la dernière instruction exécutée. La fonction `compte` n'est pas récursive terminale puisque la fonction `print` est appelée après l'appel récursif.

Exemples classiques

Les deux exemples qui suivent sont issus des mathématiques.

Premier exemple

L'un des exemples les plus classiques est une fonction mathématique, la fonction factorielle. On dit « factorielle n » et on note avec un point d'exclamation $n!$.

Cette fonction est définie « par récurrence » sur les entiers naturels : $0! = 1$ et pour tout entier naturel n non nul, $n! = n \times (n-1)!$

Autrement dit, pour tout n non nul, $n! = n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1$.

Dans l'écriture $0! = 1$ et $n! = n \times (n-1)!$ pour n non nul, nous voyons les deux aspects d'une fonction récursive : un cas simple qui correspond à la condition d'arrêt et un cas complexe, celui de l'appel récursif pour obtenir la valeur de $(n-1)!$ afin de calculer $n!$.

Avec l'écriture sous la forme du produit des entiers de 1 à n, nous pouvons imaginer une boucle pour effectuer les multiplications l'une après l'autre.

Nous avons donc les deux possibilités à notre disposition : soit un programme récursif, soit un programme itératif.

Commençons par un exemple en Python d'une fonction qui calcule $n!$ de manière récursive. L'écriture du code consiste à traduire la définition.

```
def factorielle(n):
    """ n est de type int positif
        renvoie n!"""
    if n > 0:
        return n * factorielle(n-1)
    else:
        return 1
```

La condition `n > 0` au début de l'exécution de la fonction sert à déterminer s'il faut procéder à un appel récursif ou pas. Dès que la valeur de `n` est négative ou nulle, la valeur 1 est renvoyée. A cet effet, les valeurs passées en paramètres lors des appels récursifs constituent une suite décroissante qui en `n` étapes prend une valeur nulle si `n` est un entier positif.

Voici deux exemples en Python de fonctions qui calculent $n!$ de manière itérative.

- Avec une boucle `for` où les produits sont calculés dans l'ordre $1 \times 2 \times 3 \times \ldots \times n$.

```
def factorielle(n):
    f = 1
    for i in range(2, n+1):
        f = f * i
    return f
```

- Avec une boucle `while`, où les produits sont calculés dans l'ordre $n \times (n-1) \times (n-2) \times \ldots \times 1$.

```
def factorielle(n):
    f = 1
    while n > 1:
        f = f * n
        n = n - 1
    return f
```

Il est important de savoir passer d'une fonction récursive à une fonction itérative utilisant une boucle `while` ou une boucle `for`, et réciproquement.

Si nous revenons sur la fonction récursive, nous remarquons que l'algorithme récursif n'est pas terminal. En effet, la dernière instruction est une multiplication par `n` et cette multiplication peut être effectuée seulement après avoir obtenu la valeur renvoyée par l'appel récursif. Les différentes multiplications sont donc mises en attente comme les affichages dans la fonction `compte`.

Nous pouvons transformer cette fonction pour la rendre récursive terminale. Le principe d'une telle transformation est d'utiliser un paramètre supplémentaire, un accumulateur, par lequel des informations sont passées. C'est cet accumulateur qui est renvoyé si la condition d'arrêt des appels récursifs est satisfaite. Il doit donc contenir le résultat.

Avec la fonction factorielle, cela donne le code suivant :

```
def factorielle(n):
    if n > 0:
        return n * factorielle(n-1)
    else:
        return 1

def factorielle_terminale(n, acc):
    if n > 0:
        return factorielle_terminale(n-1, acc*n)
    else:
        return acc
```

La fonction `factorielle_terminale` est récursive terminale. Cette fonction renvoie la valeur de `acc` $\times n!$. Donc `factorielle_terminale(n, 1)` a pour valeur $n!$.

Une fonction récursive terminale peut s'interpréter plus facilement comme une fonction itérative avec une boucle `while`. Il n'y a aucune opération en attente hormis les appels récursifs.

Deuxième exemple

Dans l'exemple de la fonction factorielle ainsi que dans tous les cas présentés auparavant, le corps de la fonction récursive ne contient qu'un seul appel récursif. Mais il y a des cas où le corps de la fonction contient deux appels récursifs ou plus.

Ce deuxième exemple est celui de la suite de Fibonacci. Il s'agit d'une suite d'entiers naturels définie par récurrence.

Les deux premiers termes sont 0 et 1, puis un terme est la somme des deux termes précédents. On obtient ainsi la suite de nombres 0, 1, 1, 2, 3, 5, 8, 13, 21, ...

La définition mathématique est : $f_0 = 0$, $f_1 = 1$ et $f_n = f_{n-1} + f_{n-2}$ pour $n > 1$.

Le n-ième terme peut se calculer avec $n - 2$ additions. On peut donc écrire une fonction, utilisant une boucle `while` ou une boucle `for`, qui prend en paramètre un entier `n` et renvoie le terme f_n.

```
# avec une boucle while
def fibo1(n):
    u, v, cpt = 0, 1, 0
    while cpt < n:
        u, v, cpt = v, u + v, cpt + 1
    return u
print('fibo1(400) =', fibo1(400))

# avec une boucle for
def fibo2(n):
    u, v = 0, 1
    for i in range(n):
        u, v = v, u + v
    return u
print('fibo2(400) =', fibo2(400))
```

La définition mathématique nous incite à écrire une fonction récursive. Il suffit de « copier » la formule de calcul.

```
def fibo3(n):
    if n == 0 or n == 1:
        return n
    else:
        return fibo3(n-1) + fibo3(n-2)
```

Ce n'est pas vraiment une bonne idée et c'est même un bon exemple de ce qu'il ne faut pas faire. En effet, cette fonction récursive est non terminale. Les additions ne peuvent être effectuées qu'après le retour des appels récursifs. Or, le nombre d'appels est presque multiplié par deux chaque fois qu'on augmente `n` d'une unité. Il suffit de constater que $2^{30} \simeq 10^9$, pour conclure qu'avec des valeurs supérieures à 30, le calcul devient lent, et sûrement trop long pour des valeurs supérieures à 40.

Il est intéressant d'effectuer des tests dans l'interpréteur Python avec différentes valeurs à partir de 30. On peut arrêter l'exécution avec « Ctrl+C » si l'attente est trop longue !

```
for i in range(40):
    print(i, fibo3(i))
```

On procède comme indiqué auparavant pour écrire une fonction récursive terminale. Il convient d'ajouter deux accumulateurs dans les paramètres. Avec des valeurs par défaut, l'appel initial de la fonction est plus simple.

```
def fibo4(n, a=0, b=1):
    if n == 0:
        return a
    else:
        return fibo4(n-1, b, a+b)

print(fibo4(300))
```

On peut obtenir une écriture moins simple mais peut-être plus intuitive avec un compteur `i` :

```
def fibo5(n, i=0, a=0, b=1): # i joue le rôle du compteur cpt
    if i == n:
        return a
    else:
        return fibo5(n, i+1, b, a+b)

print(fibo5(300))
```