2) Récursivité

2.1 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.

Différents styles de programmation seront présentés au chapitre 2.

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 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, ceci continue avec l'affichage de $k\ et\ l'appel\ qui\ suit\ tant\ que\ k < 10$.

La fonction affiche 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 et 0.

Considérons pour cela la fonction rebours 1 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, 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 $n, n-1, \ldots 0$.

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)
```

Examiner le déroulement de l’exécution avec le paramètre $n\ prenat\ la\ valeur\ 3$.

2.2 Fonction récursive

Définitions :

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

Nous parlerons 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 utilisé la 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 naturel 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 de 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))
```

Examiner l'exécution de ce programme pour obtenir(4,3) et commenter.

Quelles remarques pouvez-vous faire sur la fonction produit ?

Principes généraux :

⇨ *Une fonction récursive soit 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 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.

De ce fait, 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)


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))


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

Ecrivons juste la fonction récursive :

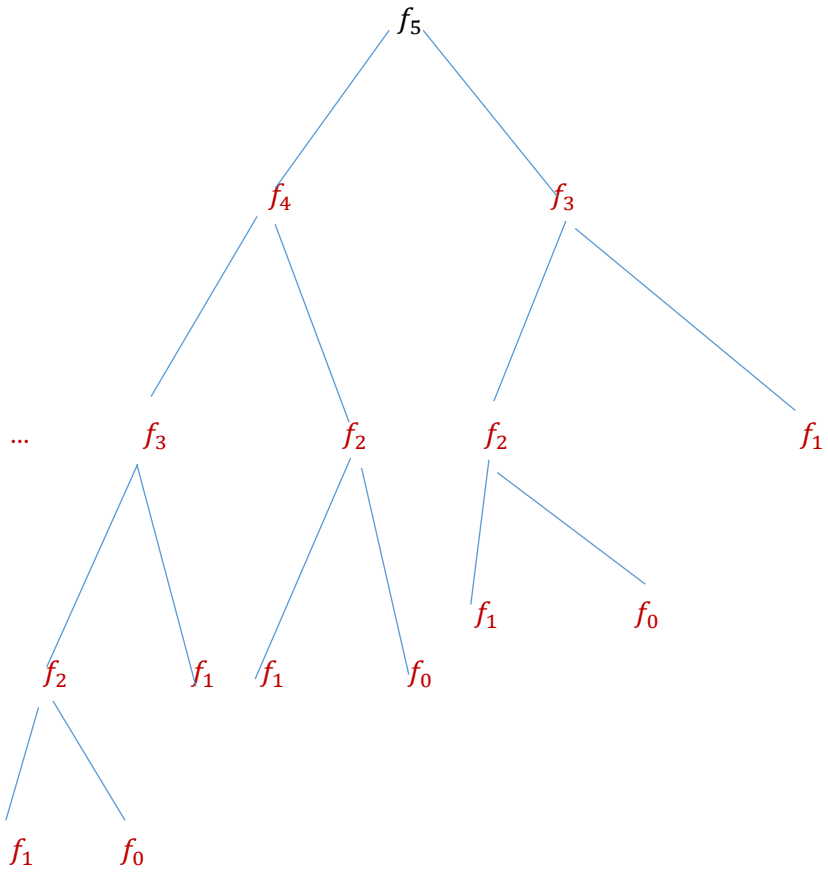
```
def  fibo3(n) :
    if n ==0  or n == 1 :
       return n
    else:
      return fibo3(n-1) + fibon3(n-2)
```

déterminer, le coût ici est un peu difficile. Pour cela, calculer séparément le nombre d'additions, le nombre de tests et le nombre d'appels récursifs.

Si nous notons a_n *le nombre d'additions pour obtenir* f_n *, alors* $a_n = a_{n-1} + a_{n-2}$.

$a_n + 1 = a_{n-1} + a_{n-2} + 1$; *posons* $b_n = b_{n-1} + b_{n-2}$ *avec* $b_0 = a_0 + 1 = 1$ *et* $b_1 = a_1 + 1 = 1$.

Nous constatons alors que $b_n = f_{n+1}$ *et donc* $a_n = f_{n+1} - 1$.



Pour une addition nous avons deux appels récursifs. Le nombre d'appels récursifs pour obtenir

$$f_n \text{ est donc } 2f_{n+1} - 2. \text{Par exemple pour calculer } f_5, \text{nous appelons } fibo3$$

qui procède à 14 appels récursifs ($2\times 8 - 2 = 14$).

Le nombre de tests est égal au nombre d'appels de la fonction fibo3, soit $2f_{n+1} - 1$.

Nous comptons un test pour l'appel initial et un test pour chaque appel récursif.

Pour calculer f_5, *nous procédons donc* à 15 *tests* .

En conclusion, le coût du calcul de f_n $en\ terme\ de\ nombre\ d'additions, de\ tests\ et\ d'appels$

Récursifs est $5f_{n+1} - 4$.

On démontre que le nombre $f_n = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n$: $c'est\ donc\ un\ coût\ exponentielle$.

Lorsqu'un calcul exact n'est pas possible, nous pouvons envisager d'établir un encadrement du coût.

Par exemple encadrons f_5 $A\ vous\ de\ jouer$.

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

La suite des appels pour calculer 4! est :

fact_term(4,1), puis fact_term(3, 4),puis fact_term(2, 12) , puis fact_term(1, 24) et enfin fact_term(0, 24).

Cette fois, les produits au niveau de l'accumulateur sont effectués de la façon suivante :

$(((1\times 4)\times 3)\times 2)\times 1.$

En notation polonaise inverse, l'expression à calculer s'écrit: $1\ 4\times 3\times 2\times 1\times$.

En python le nombre d'appels récursifs en attente est limité. Par défaut, cette limite est de l'ordre du millier d'appels. S'il y a un dépassement, un message d'erreur est affiché, et le programme ne termine pas : '' RuntimeError : maximum recursion depth exceeded in comparison '' .

⇨Les types de structures linéaires courantes (seront abordées plus tard) : les files et les piles.

Il suffit de penser à une file d'attente. Dans une boulangerie, la première personne dans la file d'attente commande son pain et ressort ; premier entré, premier sorti !!!

Pour une pile, on peut penser à des livres que l'on a déposés un par un sur une table. Ils forment une pile.

On peut prendre un livre sur la pile, (nous dirons ≪ $dépiler$ ≫). Les livres ne sont manipulés qu'un par un :dernier entré, premier sorti.

Reprenons une définition récursive de la fonction factorielle :

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

La récursivité est-elle terminale ? Justifier votre réponse.

Observez comment est utilisé une pile pour exécuter ces calculs. Empilez les instructions.

Pour n=4, n=3 , n=2 , n=1 et n = 0 ;

Considérons la fonction récursive terminale suivante :

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

Exécutez ce programme et commentez - le.

```
def affiche(n) :
    if n > 0 :
        affiche(n -1)
    else:
        print("terminé")
affiche(978)
```

affiche(979)

Qu' affiche la machine?

On peut se demander pourquoi avec la fonction fibo3 qui nécessite un grand nombre d'appels récursifs par exemple pour calculer $f_{30}, le\ programme\ a\ \ un\ temps\ d'exécution\ relativement$

Long mais ne provoque pas d'erreur. Examinons le cas de fibo3(4).

Nous empilons les instructions en attente.

n vaut 4 : empilement de return fibo3 (n -1) + fibo3 (n-2), ($f_4 = f_3 + f_2$).

n vaut 3 : empilement de return fibo3 (n-1) + fibo3 (n-2), ($f_3 = f_2 + f_1$).

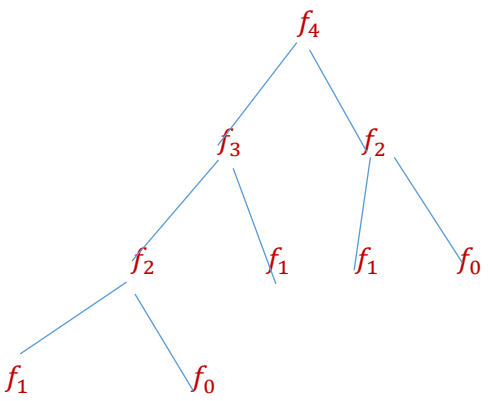
n vaut 2 : empilement de return fibo3 (n-1) + fibo3 (n-2), ($f_2 = f_1 + f_0$).

Ensuite n vaut 1 donc renvoi de fibo3(1) :return 1 .

Puis n vaut 0 donc renvoi de fibo3(0) : return 0.

Le dépilement de return fibo3 (n-1) + fibo3 (n-2) et la somme sont exécutés et donnent le résultat pour f_2.

Etc…



Exercices sur la récursivité à faire et refaire

Exercice 1 : 1. Version non terminale

```
def puissance(x , n) :
    if n  == 0 :
        return 1
    else :
        return x * puissance(x, n − 1)
```

2. Pour une version terminale, on ajoute un accumulateur en paramètre:

```
def puissance(x, n, acc):
    if n == 0 :
        return acc
    else :
        return puissance(x, n − 1 , x ∗ acc)
```

Exécuter puissance(2, 5 , 3) et indiquée les résultats renvoyés et faire visualiser les appels récursifs .

```
def puissance(x, n, ch = " "):
    if n == 0 :
        return 1
    else :
        ch = ch + " – "
        param = str(x) + '' , '' + str(n -1)
        print(ch + "> appel de puissance( " + param + " ) " )
        y = puissance(x, n − 1, ch)
        Print( ch + "> résultat de puissance(" + param + ") : " + str(y)
        return x ∗ y
```

Exécuter puissance(2 , 4)

<u>Exercice 2</u> :

```
def puissance(x , n) :
    if n == 0 :
        return 1
    else :
        p = puissance( x , n//2)
        if n % 2 == 0 :
            return p * p
        else :
            return x * p * p
```

Exercice 3 :

#récursif

```
def  puis(a, n):
      if n == 0 :
            return  1
      elif n % 2:
          return a * puis(a*a, n // 2)
      else :
          return  puis(a*a, n//2)
```

#récursif trminal

```
def  puis_t(a, n,  acc=1) :
     if n == 0 :
          return acc
      elif  n % 2 :
           return  puis_t(a*a, n//2, a * acc)
      else :
           return  puis_t(a*a, n//2, acc)
```

#itératif

```
def  puis_it(a, n):
     acc = 1
      while n != 0:
           if  n % 2 :
                acc = a * acc
            a, n = a * a, n//2
       return  acc
```

Exercice 4 : Trois exemples d'écriture d'une fonction somme .

```
def somme1(liste) :
    if liste == [] :
        return  0
    else :
        return  liste [0]  + somme1(liste[1:])


def somme2(liste):
    if liste  == [] :
        return 0
    else :
        return somme2(liste[:-1]) + liste[-1]
def somme3(liste):
    if liste == [] :
        return  0
    else :
        x  = liste.pop() # la liste est modofiée et vide à la fin !!!
        return somme3(liste) + x
```

Exécuter la première fonction avec le paramètre [4, 7 , 2]