Chapitre 1
La récursivité

Programmation en Python–2ème année–

E-mail
mlahby@gmail.com

23 septembre 2014

Plan

1. Rappel
2. Concepts de base sur la récurrence
   - Récurrence en mathématique
   - Définition d'algorithme récursif
   - Comment écrire une fonction récursive ?
3. Les différents tyde de récursivité
   - La récursivité simple
   - La récursivité multiple
   - La récursivité imbriquée
   - La récursivité mutuelle
4. Exécution d'une fonction récursive : la pile des appels
5. Comparaison de complexité d'un algorithme récursif et itératif
   - Complexité d'une fonction récursive
   - Comparaison de complexité d'un algorithme récursif et itératif
   - Limitation de la récursivité en Python

Rappel

- L'approche efficace pour résoudre un problème complexe consiste souvent à le décomposer en plusieurs sous-problèmes plus simples qui seront étudiés séparément.
- D'autre part, il arrive souvent qu'une même séquence d'instructions sera utilisée à plusieurs reprises dans un programme, et on souhaite évidemment ne pas avoir à la reproduire systématiquement.
- Le concept de la programmation modulaire permet de résoudre les difficultés évoquées ci-dessus en utilisant la notion de fonction (sous programme).
- Toutes les fonctions qu'on a définit jusqu'au maintenant représentent des algorithmes itératifs.
- Une fonction peut appeler une autre fonction. Un cas particulier elle peut appeler elle même, c'est l'objectif de ce cours

Rappel

Syntaxe

La syntaxe Python pour la définition d'une fonction est la suivante :

```
def nomdelafonction (paramètres eventuels) :
    bloc d'instructions
```

Exemples

1. Exemple 1 : Une fonction qui nécessite un paramètre

```
def cube(x) :
    y=x**3
    return(y)
```

En entrant print(cube(2)), on obtiendrait l'affichage du nombre 8

2. Exemple 2 : Une fonction ne nécessite pas forcément de paramètre.

```
def table8() :
    n=1
    while n¡=10 :
        print(n, " x ",8," = ",n*8)
        n=n+1
```

L'appel de la fonction lancerait l'affichage de la table de 8.

Récurrence en mathématique

Nous allons commencer par l'exemple de la suite numérique, (U_n) définie pour n IN par

$$\begin{cases} U_0 = 1 \\ U_n = 2 * U_{n-1} + 3 \end{cases}$$

- U_n est appelé une suite récurrente.
  $U_n = 2 * (2 * (...(2 * U_0 + 3)...) + 3) + 3$
- La conception d'une fonction récursive n'est pas éloignée du principe de démonstration par récurrence.
- Le principe de démonstration par récurrence est le suivant :
  1. On démontre d'une part que la suite U_n satisfait une telle propriété (croissante, décroissante,....) pour le cas de base U_0
  2. D'autre part, on suppose que cette propriété est valide pour U_{n-1} et on démontre que cela implique que la suite U_n satisfait aussi cette propriété pour tout $n > 0$.

Définition d'algorithme récursif

Fonction récursive

Une fonction est dite récursive si elle s'appelle elle-même au cours de son exécution.

Avantages de la récursivité

- La récursivité permet d'exprimer d'une manière élégante la solution de plusieurs problèmes :
  - Récurrences mathématiques classiques
  - Tour d'Hanoï
  - Tri rapide, tri fusion, . . .
  - Recherche dichotomique, . . .

  .
- La récursivité est particulièrement adapté lorsqu'elle est appliquée à une structure récursive.
- Les listes et les arbres peuvent être vu comme des structure récursives.

Comment écrire une fonction récursive ?

Principe

- L'idée de base pour l'écrire d'une fonction récursive consiste à définir tout d'abord le modèle mathématique de la fonction de récurrence.
- Dans ce modèle de mathémtique il faut determiner condition d'arrêt pour assurer la terminaison de l'algorithme.

.

Exemple

Nous pouvons donc définir la fonction factorielle de la manière suivante :

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n * n! & \text{sinon} \end{cases}$$

- n !=n*(n-1) ! représente la relation de récurrence.
- 0 !=1 représente la valeur de la terminaison de l'algorithme.

.

Le code en Python

Méthode itérative

```
def fact_Iter(n) :
  F=1
  for i in range (1,n+1) :
    F=F*i
  return F
```

Méthode récursive

```
def fact_Rec(n) :
  if n==0 :
    return 1
  else :
    return n*fact_Rec(n-1)
```

La récursivité simple

Définition :pour ce type de récursivité on fait un seul appel récursif pour la fonction P dans le corps d'une fonction récursive P.

Exemple 1 : calcul de puissance

$$x^n = \begin{cases} 1 & \text{si } n = 0 \\ n * n! & \text{sinon} \end{cases}$$

.

- Questions :
  1. Ecrire le code de la fonction récursive Puissance qui retourne la valeur de x^n.
  2. Donner la trace d'exécution pour calculer 2^3

Exemple 2 : calcul de puissance (version rapide)

$$x^n = \begin{cases} 1 & \text{si } n = 0 \\ a * a & \text{si n est pair, avec } a = x^{\frac{n}{2}} \\ x * a * a & \text{si n est impair, avec } a = x^{\frac{n}{2}} \end{cases}$$

.

- Questions :
  1. Ecrire le code de cette fonction récursive nommée Puiss_rapide.
  2. Donner la trace d'exécution pour calculer $Puiss_rapide(3, 5)$

La récursivité multiple

Définition : Une récursivité est multiple si il y a plusieurs appels résursifs a une fonction P dans le corps d'une fonction récursive P.

Exemple 3 : suite de Fibonacci

$$F_n = \begin{cases} 1 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ F_{n-1} + F_{n-1} & \text{si } n >= 2 \end{cases}$$

.

- Questions :
  1. Ecrire le code de la fonction récursive *Fibo_Re* qui retourne la valeur de F_n.
  2. Donner la trace d'exécution pour calculer F_4

Exemple 4 : Calcul de combinaison

$$C_n^p = \begin{cases} 1 & \text{si } p = 0 \\ 1 & \text{si } n = 0 \\ C_{n-1}^p + C_{n-1}^{p-1} & \text{sinon} \end{cases}$$

.

- Questions :
  1. Ecrire le code de cette fonction récursive.
  2. Donner la trace d'exécution pour calculer C_6^2

La récursivité imbriquée

Définition : Une récursivité est dite imbriquée si une fonction récursive P contient un appel imbriqué.

Exemple 2 : La fonction d'ACKERMANN

$$Ack(n,p) = \begin{cases} p+1 & \text{si } n = 0 \\ Ack(n-1,1) & \text{si } n > 0 \; et \; p = 0 \\ Ack(n-1, Ack(n,p-1)) & \text{sinon} \end{cases}$$

- Questions :
  1. Calculer a la main Ack(1,0) ; Ack(2,0) et Ack(3,0) :
  2. Ecrire le code de cette fonction récursive.
  3. Donner la trace d'exécution pour calculer $Ack(3,2)$

.

La récursivité mutuelle

Définition : Une récursivité est mutuelle ou croisée quand une fonction P appelle une autre fonction Q qui déclenche un appel récursif à P
Remarque : La situation est obligatoirement symétrique, puisque Q déclenchera un appel de P, qui délenchera à son tour un appel de Q.

Exemple 1 : La parité d'un entier naturel n

$$pair(n) = \begin{cases} vrai & \text{si } n = 0 \\ impair(n-1) & \text{sinon} \end{cases}$$

$$impair(n) = \begin{cases} faux & \text{si } n = 0 \\ pair(n-1) & \text{sinon} \end{cases}$$

- Questions :
  1. Ecrire le code de la fonction récursive pair(n) ;
  2. Ecrire le code de la fonction récursive impair(n) ;
  3. Donner la trace d'exécution pour calculer $pair(6)$
  4. Donner la trace d'exécution pour calculer $impair(8)$

Trace d'exécution d'une fonction récursive

Principe de fonctionnement

- L'exécution d'une fonction récursive est basée sur une pile de la mémoire vive.
- La manipulation de cette zone mémoire appelé pile est similaire à la structure données pile traitée cours de la première année.
- On empile les appels successifs dans une pile. (LIFO : Last In First Out)

.

Attention il faut être sur que la fonction se termine. Considérons par exemple cette fonction proche de la factorielle :

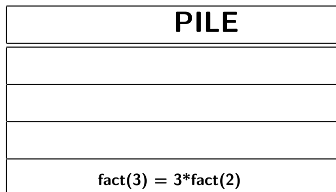
Une fonction

$$f(n) = \begin{cases} 1 & \text{si } n = 0 \\ n * f(n+1) & \text{si } n > 0 \end{cases}$$

Que vaut f(4) ?

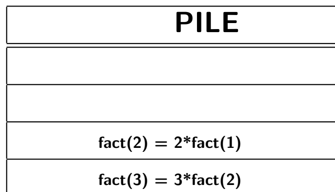
Trace d'exécution pour calculer 3 !

- Phase 1 : Empiler les apples



- Appel de fact(3) : retourne 3*fact(2)

.



- Appel de fact(3) : retourne 3*fact(2)
- Appel de fact(2) retourne 2*fact(1)=2

.

Trace d'exécution pour calculer 3 !

- Phase 1 : Empiler les apples

PILE
fact(1) = 1*fact(0)
fact(2) = 2*fact(1)
fact(3) = 3*fact(2)

- Appel de fact(3) : retourne 3*fact(2)
- Appel de fact(2) : retourne 2*fact(1)
- Appel de fact(1) : retourne 1*fact(0)

.

PILE
fact(0) = 1
fact(1) = 1*fact(0)
fact(2) = 2*fact(1)
fact(3) = 3*fact(2)

- Appel de fact(3) : retourne 3*fact(2)
- Appel de fact(2) retourne 2*fact(1)=2
- Appel de fact(1) : retourne 1*fact(0)
- Appel de fact(0) retourne 1

.

Trace d'exécution pour calculer 3 !

- Phase 2 : Dépiler les apples

PILE
fact(1) = 1*fact(0)=1*1=1
fact(2) = 2*fact(1)
fact(3) = 3*fact(2)

- Appel de fact(3) : retourne 3*fact(2)
- Appel de fact(2) : retourne 2*fact(1)
- Appel de fact(1) : retourne 1*fact(0)= 1
- Appel de fact(0) : retourne 1

.

Trace d'exécution pour calculer 3 !

- Phase 2 : Dépiler les apples

PILE
fact(2) = 2*fact(1)=2*1=2
fact(3) = 3*fact(2)

- Appel de fact(3) : retourne 3*fact(2)
- Appel de fact(2) : retourne 2*fact(1)=2*1=2
- Appel de fact(1) : retourne 1*fact(0)= 1
- Appel de fact(0) : retourne 1

.

Trace d'exécution pour calculer 3 !

- Phase 2 : Dépiler les apples

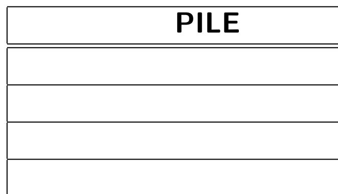
PILE
fact(3) = 3*fact(2)=3*2=6

- Appel de fact(3) : retourne 3*fact(2)=3*2=6
- Appel de fact(2) : retourne 2*fact(1)=2*1=2
- Appel de fact(1) : retourne 1*fact(0)= 1
- Appel de fact(0) : retourne 1

.

Trace d'exécution pour calculer 3 !

- Phase 2 : Dépiler les apples



- On dépile le dernier appel (fact(3))
- Le résultat retourné est 6

.

Complexité d'une fonction récursive

- La complexité d'une fonction récursive représente le côut de cette fonction, à savoir le nombre d'opérations élémentaires qu'elle effectue ou son occupation mémoire totale.
- La notion de complexité sera présentée plus en détail dans le prochain chapitre.
- On considère la suite (u_n) suivante, qui calcule une approximation de $\sqrt{3}$

$$\begin{cases} U_0 = 2 \\ U_n = \frac{1}{2}(U_{n-1} + \frac{3}{U_{n-1}}) \end{cases}$$

- En Python, on peut écrire la fonction récursive u qui retourne la valeur U_n

```
def u(n) :
    if n==0 :
        return 2
    else :
        r=u(n-1)
        return 0.5*(r+3/r)
```

- On note $C(n)$ le nombre d'opérations arithmétiques (addition, multiplication et division) effectue par la fonction $u(n)$ (avec n :l'argument de la fonction u).

Complexité d'une fonction récursive

- En suivant la définition de la fonction u, on obtient les deux équations suivantes :

$$C(0) = 0$$
$$C(n) = C(n-1) + 3$$

- En effet, dans le cas n $= 0$, on ne fait aucune opération arithmétique.
- Et dans le cas $n > 0$, on fait d'une part un appel récursif sur la valeur $n-1$, d'où $C(n-1)$ opérations, puis trois opérations arithmétiques (une multiplication, une addition et une division).
- Il s'agit d'une suite arithmétique de raison 3, dont le terme général est :

$$C(n) = 3n$$

- Donc, on conclut que la complexité de la fonction récursive u est $O(n)$

.

Complexité d'une fonction récursive

- Si en revanche on avait écrit la fonction u plus naïvement, avec deux appels récursifs u(n-1), c'est-à-dire :

```
def u(n) :
    if n==0 :
        return 2
    else :
        return 0.5*(u(n-1)+3/u(n-1))
```

- alors les équations définissant $C(n)$ seraient les suivantes : :

$$C(0) = 0$$
$$C(n) = C(n-1) + C(n-1) + 3$$

- En effet, il convient de prendre en compte le coût $C(n-1)$ des deux appels à $u(n-1)$
- Il s'agit d'une suite arithmético-géométrique, dont le terme général est :

$$C(n) = 3(2^n - 1)$$

- Donc, la complexité de la fonction récursive u devient $O(2^n)$

.

Comparaison de complexité

La comparaison de complexité entre un algorithme récursif et un algorithme itératif repose sur deux aspects de l'évaluation :

1. Compelexité spatiale : on mesure l'espace mémoire occupée par les variables.
2. complexité temporelle : on mesure le nombre d'opérations.

Complexité temporelle (CT)

- La taille mémoire occupée par une fonction itérative est inférieure à celle utilise par une fonction récursive, en raison des appels successifs dans une cette dernière.
- Exemple : $CS(fact_Iter) = O(1) < CS(fact_Rec) = O(n)$

.

Complexité temporelle

- La même remaque pour la complexité temporelle, en effet le coût d'execution d'une fonction itérative est généralement inferieure au temps d'une exécution d'une fonction récursive pour le même problème.
- Exemple : $CT(Fibo_Iter) = O(n) < CT(Fibo_Rec) = O(2^n)$

.

Limitation de la récursivité en Python

Limitation de la récursivité en Python

- Le langage Python limite, arbitrairement, le nombre d'appels imbriqués à 1000.
- Une fonction qui fait plus de 1000 appels récursifs provoque une erreur
- Exemple :

```
>>>def   fact(n) :
        if n==0 :
           return 1
        else :
           return n*fact(n-1)
>>>fact(1005)
RuntimeError : maximum recursion depth exceeded
```

- Il existe de nombreuses situations où l'on sait que le nombre d'appels sera bien inférieur à 1000.

.