

Fonctions

1 Fonctions et programmation

La programmation d'un problème est une tâche complexe difficile à maîtriser globalement.

Pour limiter les erreurs et la taille des programmes (donc du code objet), on va chercher à identifier des sous-tâches qui correspondent à des calculs bien déterminés qu'on est amené à répéter plusieurs fois dans le programme.

Cela permet de raisonner de manière plus abstraite sur l'ensemble du programme et de se concentrer sur la programmation de parties plus simples et plus courtes. L'expérience montre qu'un programmeur standard ne peut maîtriser qu'un programme de 2-3 pages, soit 200-300 lignes, ce qui impose de découper les programmes plus complexes en parties plus simples à écrire. Pour les applications de taille importante -par exemple le noyau Unix- on utilisera en plus des techniques et des outils de génie logiciel.

Exemple 1 : calculer le minimum de deux entiers

Dans les calculs numériques, on a souvent besoin de prendre le maximum de deux nombres. Plutôt que de répéter les instructions qui permettent de calculer ce max, on va utiliser une fonction :

```
fonction min (a: reel,b:reel):reel
  si a > b alors retourner b
  sinon retourner a fsi
```

En fait, vous avez déjà utilisé des fonctions des bibliothèques : *print*, *input* sont des fonctions programmées dans les bibliothèques du langage (et leur code est suffisamment compliqué pour essayer à tout prix de ne pas le réécrire!).

Exercice 1 Le programme de résolution d'une équation de degré 2 vu au premier cours contient des calculs complexes effectués avec des fonctions de la bibliothèque python. Identifier ces fonctions.

Exemple 2 : calculer la factorielle.

```
fonction fact (n: entier):entier
  si n <=0 alors retourner 1
  sinon retourner (n*fact(n-1)) fsi
```

La fonction ici est *réursive* : la définition de la fonction contient un appel à la fonction elle-même : la partie *fact(n-1)* de l'expression de la dernière ligne.

Exercice 2 Donner (sous forme algorithmique) une fonction **réursive** calculant la somme des n premiers entiers.

2 Principes

2.1 Un exemple

Le programme python contenu dans le fichier *puissance.py* contient la définition de la fonction *puis*(x, n) qui calcule la fonction $x, n \rightarrow x^n$ (x, n entiers) et un script d'utilisation de cette fonction.

```
#la definition de la fonction
def puis(x, n):      #x,n paramètres formels
    "calcule x**n"   #chaîne de commentaire pour la doc python
    #ici commence le corps de la fonction
    p=1              #p,i variables locales
    i=1
    while (i<=n):
        p=p*x
        i=i+1
    return p         #renvoi de la valeur calculée

#un script
i=0
while (i<10):
    x=puis(2,i)     #appel de la fonction avec les paramètres effectifs
                  #qui peuvent être des constantes, des variables ou des
                  #expressions
    print 'la puissance',i,'ème de 2 est', x
    i=i+1
```

Nous passons en revue les caractéristiques permettant de définir et utiliser une fonction.

2.2 Paramètres

En programmation, les arguments d'une fonction s'appellent les paramètres. Il y a deux types de paramètres :

- Les paramètres formels sont les identificateurs qui sont utilisés pour définir la fonction. Il n'ont pas de valeur déterminée et correspondent à des variables mathématiques utilisées pour la définition comme x ou n dans *puis* : $x, n \rightarrow x^n$
- Les paramètres effectifs sont les valeurs utilisées lors de l'appel de la fonction qui calcule alors la valeur de la fonction après substitution des paramètres formels correspondants. Ainsi 2 est le paramètre effectif de l'appel *puis*(2, 1) qui calcule *puis* après substitution de x par 2 et n par 1 dans la définition de *puis* .

Un paramètre formel est toujours un nom de variable, un paramètre effectif peut être une expression. Par exemple *puis*($y + 2, z$) est autorisé (à condition que le programme ait affecté des valeurs à y et z avant cet appel).

2.3 Calcul de la valeur retournée

L'exécution de l'instruction **return exp**

1. évalue exp,

2. stoppe l'évaluation de la fonction,
3. retourne la valeur de `exp` comme résultat de l'appel courant à la fonction appelante.

Une fonction peut contenir plusieurs instructions *return* dont une seule sera évaluée à l'exécution.

Certaines fonctions n'ont pas d'instruction *return* : on les appelle aussi *procédures* et elle ne renvoient pas de valeur (en fait elles renvoient l'objet `None`). Ces fonctions agissent par *effet de bord* en modifiant des variables globales ou l'environnement (affichage par exemple).

```
def echo (char c):  
    print c
```

L'intérêt d'une telle fonction est les effets de bords qu'elle produit (c'est à dire la modification de l'environnement produit par l'exécution de la fonction). Ici cet effet de bord est l'impression d'un caractère sur la sortie standard.

PIEGE : en général une fonction qui agit par effet de bord modifie des variables du programme (les variables *globales*, voir la définition plus loin). Si ce type de fonction est mal conçu et utilisé cela est la source d'erreurs de programmation difficile à identifier.

3 Portée, Variables Globales et Locales

La *portée* d'une variable est la partie de programme dans laquelle on peut consulter la valeur de la variable (et éventuellement modifier).

Une variable définie à l'intérieur d'une fonction est une variable *locale*.

La portée d'une variable débute à sa première apparition et se termine à la fin de la fonction (ou du script) qui contient cette définition.

Une variable déclarée en dehors de toute fonction est une variable *globale*. Elle est visible et utilisable dans toute fonction. Par contre elle n'est pas modifiable sauf si la fonction la déclare modifiable avec l'instruction *global*.

Quand plusieurs variables ont le même nom et ont des portées imbriquées, ce nom désigne la variable la plus imbriquée.

Identifier variables locales, globales paramètres formels et effectifs et portées dans le programme suivant :

```
def foo(x):  
    global MAX  
    x=0  
    i=1  
    while i<MAX:  
        x=x+i  
        MAX=i  
        i=i+1  
    return x
```

```
MAX=3  
i=1  
j=1  
while (j<MAX):  
    j=foo(i)  
    print j  
MAX=0  
print MAX
```

3.1 Passage de paramètres

Passage de paramètres en python : pour les paramètres de type simple, le passage de paramètres se fait par valeur. A l'appel de la fonction, les paramètres formels prennent la valeur des paramètres effectifs et le corps de la fonction est effectué avec ces valeurs. Les valeurs des variables locales de la fonctions sont perdues à la fin de l'appel. Pour les paramètres de type complexe (objets) la fonction reçoit en paramètre la référence de l'objet passé effectivement en paramètre. Cette référence ne peut pas être modifiée mais l'objet correspondant à cette référence peut être modifié. Décrire ce qu'affichent les programmes :

```
def foo( x):  
    x=0
```

```
x=1  
foo(x)  
print x
```

foo a été appelée avec le paramètre effectif *x* (qui vaut 1) et la fonction *foo* a été exécutée sur une variable temporaire dont la valeur a été initialisée à celle de *x* à l'appel, mais cette variable n'a aucun lien avec *x*. Tous les calculs effectués par *foo* concernent cette variable temporaire et non pas *x*. Après l'appel de *foo*, la variable *x* a donc 1 pour valeur, celle qu'elle avait avant l'appel.

```
def bar(l):  
    i=1  
    n=len(l)  
    while i<n:  
        l[i]=0  
        i=i+1
```

```
l=[1,2,3]  
foo(l)  
print l
```

4 Exemples

Exemple 1 : Calcul efficace de la fonction puissance.

La fonction donnée pour le calcul de x^n effectue n multiplications. Tant que les nombres restent dans le même type *int* (ou *long* si on sait qu'on a de grand entiers) on peut considérer que le coût est constant quel que soit la taille du nombre. Donc il est intéressant de diminuer le nombre de multiplications et on va donner maintenant un principe fondamental qui va permettre de réduire ce nombre à (à peu près) $\log(n)$ (log en base 2).

Exercice 3 Essayer de trouver ce principe.

L'idée est d'utiliser la dichotomie et la récursion : on va transformer le problème en deux sous-problèmes semblables au premier mais de taille moitié et donner la méthode permettant de résoudre le problème initial à partir des solutions des deux sous-problèmes.

Principe :

- si n est pair $n = 2p : x^{2p} = x^p * x^p$
- si n est impair $n = 2p + 1 : x^{2p+1} = x^p * x^p * x$

Les facteurs x^p vont correspondre à des appels récursifs.

```
def puis(x, n):
    "calcul de x**n recursif dichotomique"
    if (n==0):
        return 1;
    elif (n==1):
        return x
    else:
        y=puis(x,n/2)
        if (n % 2==0):
            return y*y
        else :
            return y*y*x
```

```
#script d'utilisation
i=0
while (i<10):
    print puis(2,i)
    # appel de la fonction avec les
    # parametres effectifs 2 et i
    i=i+1
```

Cette fonction fait de l'ordre de $\log(n)$ multiplications : un appel fait au plus 2 multiplications (cas impair) et il n'y a que $\log(n)$ appels récursifs possibles (à chaque fois le deuxième argument est divisé par 2).

Exercice 4 Est-ce que cette fonction est récursive?

Par contre on faire une objection sérieuse sur cet algorithme et le calcul de complexité.

Exercice 5 Laquelle?

Il y a deux divisions (opération qui coûte asymptotiquement aussi cher qu'une multiplication et qui est plus compliquée). Mais on est sauvé car ce sont des divisions par deux : on peut les implémenter par un test sur le dernier bit pour la parité et un décalage pour la division proprement dite.

Une deuxième objection est que chaque appel récursif coûte du temps (création des variables temporaires, gestion sur la pile d'exécutions des adresses de retour,...). Cela conduit à chercher une version qui fait le même calcul que la fonction récursive mais en remplaçant la récursion par

une itération : on peut trouver à la main ou utiliser des techniques de dérécursivation (que certains compilateurs savent effectuer tout seul). Ce sujet est abordé dans un cours d'algorithmique avancé.

Exercice 6 Ecrire une fonction non récursive qui calcule x^n en $\log(n)$ opérations.

Exemple 2 :

Ecrire une fonction qui calcule la valeur d'une opération binaire qui est dénotée par un caractère '+', '-', '*', '/' sur ses deux opérandes $e1, e2$ qui sont de type float.

```
def eval2(c, e1, e2):
    if c=='+':
        return e1+e2
    elif c=='-':
        return e1-e2
    elif c=='*':
        return e1*e2
    elif c=='/':
        try:
            return e1/e2
        except:
            print "division par zero"
    else:
        print "fonction inconnue"
```

On peut noter l'utilisation de la construction *try ...except ...* qui en cas d'erreur à l'exécution dans les instruction du try arrête l'évaluation sans déclencher d'erreur mais passe la main à la partie except qui est exécutée. La construction complète est *try ...except ...else ...* la troisième partie (*else*) permet de poursuivre le programme quand la partie *try* ne déclenche pas d'erreur.

Que retenir ?

Voici des questions auxquelles il faut savoir répondre sans hésiter et qui portent sur les points importants à comprendre et retenir.

Une réponse syntaxiquement correcte ne suffit pas, il faut aussi comprendre la question et savoir expliquer sa réponse.

- A quoi sert une fonction ?
- Définir paramètres formels, variables locales et globales.
- Peut-on avoir une fonction sans paramètres ?
- Expliquer ce qu'est un effet de bord d'une fonction ?
- Si une fonction ne renvoie rien et n'a aucun effet de bord, à quoi peut-elle servir ?
- Peut-on avoir plusieurs instructions return dans une fonction ?
- Définir ce qu'est une fonction récursive (donner un exemple).
- Quelle est la différence entre un paramètre formel et une variable locale ?