TP Informatique n° 2

Fonctions et applications

Fonctions

Le but de ce TP est de travailler sur l'écriture des fonctions. La forme classique d'une fonction est :

```
def nom_fonction(variables en entrée):
   bla bla bla
   return(resultats)
```

Les alinéas sont importants !

Q.1 Une première fonction
On considère la fonction :

```
def albert(n,a):
    p = 10
    if n > p :
        return(a/n)
    else :
        return(n)
```

Quels sont les retours des appels `albert(30,6)`, `albert(5,10)` et `albert(10,0)`

Q.2 Une deuxième fonction
On considère la fonction :

```
def de(n,d):
    r = n
    q = 0
    while r > d :
        r = r - d
        q = q + 1
    return(q,r)
```

Que fait cette fonction ?

Q.3 Compter les voyelles
Créer une fonction `voyelles(mot)` qui compte le nombre de voyelles dans une chaine de caractères.

Q.4 Factorielle
Il n'y a pas de factorielle nativement en python ! En créer une.

Complexité

L'étude du temps que met un algorithme pour se réaliser s'appelle la complexité.
On reprends l'idée de construire un tableau contentant les termes d'une suite récurrente.
On considère ici la suite définie par $u_0 = 1$ et $u_{n+1} = \frac{3u_n+4}{2u_n+1}$
On construit deux versions de ce programme :

Q.5 Version A

```
def liste_valeur(n):
   """ construit directement la liste des valeurs de la suite"""
   u = 1
   L =[u]
   for k in range(n):
       u = (3*u+4)/(2*u+1)
       L.append(u)
   return(L)

print(liste_valeur(10))
```

Q.6 Version B

```
def u(n):
   """ Renvoi la valeur u_n de la suite"""
   u = 1
   for k in range(n):
      u = (3*u+4)/(2*u+1)
   return(u)

def liste_suite(n):
   """ construit la liste des valeurs de la suite"""
   L = []
   for k in range(n+1):
       L.append(u(k))
   return L

print(liste_suite(10))
```

Q.7 Qui est mieux ?

On va tester :

```
import time
n=100000
a=time.clock()
liste_suite(n)
b=time.clock()
liste_valeur(n)
c=time.clock()
print('Calcul en',b-a,'secondes')
print('Calcul en',c-b,'secondes')
```

Documentation

Il est possible de documenter une fonction pour expliquer son fonctionnement à l'utilisateur.

Q.8 Exemple

On rajoute du texte au début de la fonction :

```
def division_euclidienne(n,d):
    " Renvoie le couple (quotient,reste) de la division Euclidienne de n par d"
    r = n
    q = 0
    while r > d :
        r = r - d
        q = q + 1
    return(q,r)
```

Faire un test en console, enjoy.

Q.9 Hum

On considère une fonction, comprendre ce qu'elle fait et la documenter.

```
def Hum(n,p):
    resultat = False
    while n!=0 and not (resultat) :
        resultat = (n % 10 == p)
        n = n /10
    return(resultat)
```

Fonctions récursives

On parle de récursivité, lorsque, pour résoudre un problème, on utilise des solutions dans des cas ≪plus petits≫ du même problème. Un algorithme récursif ≪s'appelle donc lui-même≫.

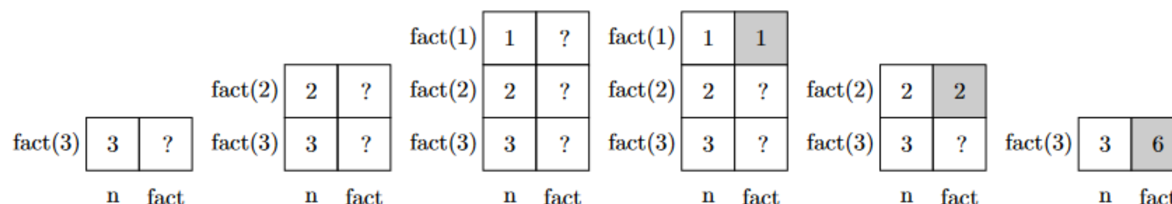
Q.10 Exemple

Tester le programme

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

Tester fact(4), fact(10).

Revenons sur la fonction fact et expliquons comment l'ordinateur parvient à calculer fact(3). L'ordinateur voit qu'on lui demande de calculer fact(3). Il va stocker dans une pile le fait qu'on veut cette valeur, mais qu'on ne pourra la calculer qu'après avoir obtenu la valeur de fact(2). On procède ainsi jusqu'à demander la valeur de fact(1). Il peut alors dépiler les appels, pour de proche en proche calculer la valeur de fact(3).



Le schéma global d'une fonction récursive peut s'écrire :

```
def fonctionRecursive(args) :
   si casDArrêt :
      # on fait ce qu?on a ‘a faire dans le cas d?arr^et
      # Pas d?appel r ?ecursif ici
   sinon :
      # instruction pouvant faire intervenir fonctionRecursive
```

Intêrets :

- la programmation récursive permet en général de résoudre de manière élégante des problèmes compliqués.
- ne pas rentrer dans le détail de l'exécution d'un programme.
- particulièrement adapté pour certaines structures de données, comme les arbres.

Inconvénients :

- problème de débordement de la pile (valeur trop grande).
- implémentation de la gestion de la pile qui peut ralentir considérablement l'exécution d'un programme.
- on maîtrise peu l'efficacité de l'exécution de la fonction.

Q.11 Écrire un algorithme non récursif $puissance(x, n)$ qui permet de calculer x^n pour x et n donnés.

Q.12 En remarquant que $x^n = \begin{cases} (x^2)^{\lfloor \frac{n}{2} \rfloor} & \text{si n pair} \\ x \times (x^2)^{\lfloor \frac{n}{2} \rfloor} & \text{si n impair} \end{cases}$, écrire une fonction récursive $expo(x, n)$ qui calcul x^n.

Q.13 En utilisant la $Q7$, comparer la vitesse d'execution des deux programmes précédents. (essayer pour plusieurs valeurs de x et de n).

Q.14 Tour de Hanoï

On dispose de trois piquets A, B et C sur lesquelles peuvent s'enfiler n disques de diamètres distincts. Initialement les disques sont tous sur le pique A, le plus grand en bas et le plus petit en haut. Le but du jeu est de déplacer tous les disques sur le piquet C en ne bougeant qu'un disque, celui au sommet de la pile, à la fois et en ne posant un disque que sur un disque plus grand.

Implémentation : les piquets seront numérotés 0, 1 et 2. Un déplacement du disque du piquet i sur le piquet j sera représenté par le couple (i, j). Écrire une fonction $resolution(i, j)$ donnant la liste des déplacements des disques pour transporter la pile de disques initialement sur le piquet i vers le piquet j.

Évaluer le nombre D_n de déplacement nécessaire.