Introduction à la Programmation 1 PYTHON

51AE011F

Séance 8 de cours/TD

Université Paris-Diderot

Objectifs:

- Variables locales et globales
- Tester une fonction
- Décrire (spécifier) le comportement d'une fonction.
- Réaliser une spécification.

1 Approfondissements sur les fonctions

Rappels [COURS]

- Les fonctions et procédures peuvent prendre toutes sortes de *paramètres* : des entiers, des booléens, des couples, des listes, mais aussi des fonctions.

```
#La fonction eval evalue f sur inp
def eval (f, inp):
    return f(inp)
```

  En général, on écrit des fonctions qui prennent peu de paramètres, et on évite les paramètres redondants ou inutiles.
- Une fonction permet de *factoriser du code*. Par exemple, si l'on veut remplacer une lettre par une autre dans plusieurs chaînes de caractères, il est utile de créer une fonction que l'on puisse appliquer à chacune des chaînes plutôt que de faire des copier-coller comme dans le programme suivant :

```
s = "abracadabra"
r=""
for i in range(0, len(s), 1):
    if s[i]=='a':
        r = r + 'e'
    else:
        r = r + s[i]
print(r)

s = "magie"
r=""
for i in range(0, len(s), 1):
    if s[i]=='a':
        r = r + 'e'
    else:
        r = r + s[i]
```

```
print(r)
```

— Introduire des fonctions a aussi d'autres buts : en donnant des noms bien choisis aux fonctions et à leurs arguments, et en leur assignant des tâches clairement identifiées, on améliore la lisibilité du code, et sa réutilisabilité dans d'autres programmes.

— Une fonction *doit* renvoyer un résultat à l'aide de l'instruction `return`. Cette instruction stoppe l'exécution d'une *fonction* et termine immédiatement la fonction. Il faut faire attention lorsqu'on l'utilise à ne pas stopper l'exécution trop tôt, notamment en choisissant bien l'*indentation* dans une conditionnelle ou une boucle.

```
def replace_ab(s):
    r=""
    for i in range(0, len(s), 1):
        if (s[i]=='a'):
            r = r+"a"
        else:
            r = r+s[i]
        return r
```

Dans cet exemple, le `return` étant placé dans la boucle, l'exécution de la fonction s'arrête après le premier passage dans la boucle.

Exercice 1 (Bien concevoir ses fonctions, ⋆)

Critiquez la façon dont est conçu ce code et proposez une version améliorée.

```
def f23(t, n):
  z = 0
  for i in range(0, n, 1):
      z = z + t[i]
  print("Somme :"+str(z))

def f24(n, t, msg):
  h = 0
  for i in range(0, n, 1):
      h = h + t[i]
  return (h / n)

t = [1,2,15,30,2]

f23(t, len(t))
print("Moyenne : " + str(f24(t, len(t), "Moyenne")))
```

□

Exercice 2 (Bien utiliser return, ⋆)

1. *Écrivez une fonction* `membership` *qui prend en argument un entier* `n` *et une liste d'entiers* `t`*, et qui renvoie* `True` *si* `n` *apparaît dans* `t`*, autrement dit si* `n=t[i]` *pour au moins un* `i`*. On cherchera à renvoyer le résultat le plus tôt possible.*
2. *En utilisant la fonction* `membership`*, écrivez une fonction* `inclusion` *qui prend en argument deux listes d'entiers* `t` *et* `u` *et qui renvoie* `True` *si l'ensemble des entiers de* `t` *est inclus dans l'ensemble des entiers de* `u`*, autrement dit,* `membership(t,u)` *renvoie* `True` *si pour tout indice* `i` *de* `t`*, l'entier* `t[i]` *apparait dans* `u`*.*

□

Variables locales, variables globales [Cours]

— Une variable définie dans le corps d'une fonction ou d'une procédure est **locale** à cette fonction. Les paramètres sont assimilés à des variables locales.
Une variable définie en dehors de toute fonction ou procédure est dite **globale**.
Par exemple, dans le programme ci-dessous,

```
flag = False
def f(x):
    if x>=3:
        a = 3
    else:
        a = x
    return x
flag = (f(5) == 3)
```

il y a 1 variable *globale* : `flag` et 2 variables *locales* à la fonction `f` : le paramètre `x` et la variable `a`.

— Les paramètres et variables locales d'une fonction ou d'une procédure n'appartiennent qu'à cette fonction, c'est la règle de *localité*. En particulier, une fonction (par exemple `g`) ne peut pas faire référence à une variable locale d'une autre fonction (par exemple la variable `a` de `f`). Les variables locales sont détruites après chaque exécution d'une fonction.

```
a = 3
def f(x):
    a = 0
    return a*x
print(f(2))
print(a)

```

Le programme ci-dessus affiche successivement 0 puis 3. Il est équivalent au programme suivant :

```
a = 3
def f(x):
    a_f = 0
    return a_f*x
print(f(2))
print(a)

```

— Au contraire, une variable globale est accessible par toutes les fonctions. Par défaut en Python, une variable globale peut être lue par une fonction ou une procédure. Si on veut pouvoir la modifier, il faut la déclarer `global`[1] au début de la fonction ou de la procédure.

```
count = 0
def show():
    print(count)
def inc():
    global count
    count = count+1
while count <5:
    inc()
show()

```

Dans le programme ci-dessus, la procédure `show` accède en lecture à la variable globale `count`, la procédure `inc` modifie la variable globale `count`.

— Quand on passe une liste en paramètre à une fonction ou à une procédure, ce n'est pas son contenu, mais son adresse dans le tas. Ainsi, on peut écrire des fonctions et procédures qui modifient des listes passées en paramètres.

```
def changeFst(l, a):
    l[0]=a
    return l

t = [1,2,3]
changeFst(t, 0)

```

À la fin de l'exécution du programme ci-dessus, la liste `t` contient la liste `[0,2,3]`. Ceci ne contredit pas la règle de localité. En effet, le paramètre `t` n'a pas été modifié : il pointe toujours vers la même adresse dans le tas. Par contre, ses éléments ont été modifiés.

Exercice 3 (Exécution, ⋆)
Renommez les variables et ajouter le mot-clé `global` *pour lever les ambiguités.*

```
def f (a, b):
    c = 1
    a = a+b
    return a+c

def g (b):
    a = a+b
    b = 1
    c = a
    return f (a, b)
```

□

Exercice 4 (compte en banque, ⋆⋆)
On considère le programme suivant.

```
balance = 0

def deposit(amount):
    global balance
    balance = balance + amount

def withdraw(amount):
    global balance
    newbalance = balance - amount
    if (newbalance>0):
        balance = newbalance
    else:
        amount = balance
        balance = 0
    return amount

print(balance)
deposit(1000)
print(balance)
amount1 = 100
print(withdraw(amount1))
print(balance)
```

```
amount2 = 1000
print(withdraw(amount2))
deposit(1000)
amount = 2000
print(withdraw(amount))
print(amount)
```

1. *Quelles sont les variables globales, les paramètres, et les variables locales ?*
2. *Qu'affiche le programme ?*

□

Exercice 5 (test, ⋆⋆⋆)

1. *Écrire une fonction* `isInMatrix` *qui prend en argument une liste* `M` *de listes d'entiers et un entier* `n` *et qui renvoie* `True` *si* `n` *apparait dans* `M`.

   Contrat:
   Pour $0 \leq n \leq 5$ *et* $6 \leq m \leq 9$.
   `M1 = [[0,1,2],[3,4,5]]` *et* `M2 = [[0,5],[1,4],[2,2],[3, 1]]`.

   ```
   isInMatrix([[]], n)  →  False
     isInMatrix(M1, n)  →  True
     isInMatrix(M1, m)  →  False
    isInMatrix(M2 , n)  →  True
     isInMatrix(M2, m)  →  False
   ```

2. *Écrire les tests correspondants au contrat ci-dessus.*
3. *Afin de factoriser le code et de ne pas répéter* `x` *fois les tests, proposer un encodage pour les tests et une boucle permettant d'afficher le résultat des tests. Afficher les tests pour M1 et les entiers compris entre 0 et 9 inclus.*
4. *Afin de factoriser le code et de ne pas recoder systématiquement la boucle de test, on propose d'écrire une fonction.*
   Écrire une fonction test qui prend en argument une fonction et une liste de tests et qui renvoie une liste de résultats. Chaque test représente l'entrée sur laquelle tester la fonction et la sortie attendue. La fonction renverra une liste de résultats représentant l'entrée, la sortie attendue et la sortie obtenue.

   Contrat:
   Par exemple, on suppose donnée une fonction `f` *qui devrait renvoyer le double de son argument, mais qui n'a pas le comportement attendu...*

   ```
   test(f, [(0,0),(1, 2),(5,10)]  →  [(0,0,1), (1,2,2), (5,10,10)]
   ```

   Quels sont les paramètres et variables locales à la fonction `test` *?*
5. *Tester la fonction test sur la fonction* `isInMatrix` *et les paramètres définis dans les contrats.*
6. *Afin de factoriser l'affichage des tests, écrire une procédure* `Show` *qui affiche les entrées, les sorties attendues et les sorties obtenues pour chacun des échecs.*
7. *Ajouter deux variables globales* `n_test` *et* `n_failure` *qui permettent d'ajouter le nombre de tests effectués et le nombre de tests ratés.*
8. *Modifier la procédure* `show` *afin qu'elle affiche le nombre total de tests et de ratés depuis le début du programme.*

□

Documentation et Spécification [Cours]

Un programme peu ou mal documenté est difficile à corriger, modifier et réutiliser. Il est donc important de documenter et de commenter systématiquement son code.

Pour chaque **fonction**, avant l'en-tête, on place quelques lignes de commentaires, au format suivant :

- une ligne pour chaque paramètre d'entrée, indiquant son type, ce que ce paramètre représente et une propriété attendue par la fontion sur l'entrée, que l'on appelle **précondition** ;
- une ligne pour la valeur renvoyée, son type, ce qu'elle représente et la propriété promise par la fonction sur cette sortie, que l'on appelle **postcondition**.

Des commentaires supplémentaires liés à l'implémentation de la fonction seront placés dans son corps, comme dans l'exemple suivant :

```
# Verification de la correction d'une date.
#
# Entree : d un entier designant le jour.
# Entree : m un entier designant le mois.
# Entree : y un entier designant l'annee.
# Sortie : True si la date est correcte, False sinon.

def checkDate (d, m, y):
    if (m >= 1 and m <= 12 and d >= 1 and d <= 31):
        if (m == 4 or m == 6 or m == 9 or m == 11):
            return (d <= 30)
        else:
            if (m == 2): # Est-ce une annee bissextile?
                if ((y%4 == 0 and y%100 !=0 ) or y%400 == 0):
                    return (d <= 29)
                else:
                    return (d <= 28)
            else:
                return true
    else:
        return false
```

Pour les **procédures**, on indiquera **l'effet qui sera observé** (affichage, modification d'un tableau, modification de variables globales, lecture d'un fichier,...) après l'exécution de la procédure à la place de la **postcondition**, qui n'a pas lieu d'être puisqu'aucune valeur n'est renvoyée par une procédure.

```
#
# Procedure LastFirst
#
# Entree: t un tableau d'entier
#
# Effet1: mise a 0 du dernier element de t
# Effet2: affichage du premier element de t
#

def LastFirst(t):
    t[-1] = 0
    print(t[0])
```

La spécification des fonctions et procédures ainsi fournie doit permettre de les utiliser sans avoir à connaître leur code.

Exercice 6 (Inclusion de tableaux, $\star$)

Écrire une fonction qui implémente la spécification suivante :

```
#
# Inclusion d'un tableau dans un autre.
#
# Entree : un tableau d'entiers a
# Entree : un tableau d'entiers b
# Sortie : true si et seulement si tous les elements du tableau a
# sont contenus dans le tableau b en prenant en compte
# les repetitions.
#
```

□

Exercice 7 (Séquences dans un tableau, ⋆)
Écrire une fonction qui implémente la spécification suivante :

```
#
# Entree : un tableau d'entiers a.
# Sortie : True si et seulement si a contient deux
# sequences consecutives identiques. Par exemple, sur
# 1,2,3,2,3,7 la valeur Ã  renvoyer est True.
#
```

```
/*
    Entrée : un tableau d'entiers a.
    Sortie : true si et seulement si a contient deux
             séquences consécutives identiques. Par exemple, sur
             {1,2,3,2,3,7} la valeur à renvoyer est true.
*/
```

□

2 DIY

Dans les exercices suivants, vous vous appliquerez à bien factoriser votre code.

Exercice 8 (Réaliser des spécifications, ⋆)
Écrire une fonction qui implémente la spécification suivante :

```
# Somme des entiers d'un intervalle.
#
# Entree: low un entier designant le debut de l'intervalle.
# Entree: high un entier designant la fin de l'intervalle.
# Sortie: la somme des entiers compris au sens large entre
# low et high.
```

□

Exercice 9 (Utiliser des spécifications, ⋆⋆)

La conjecture de Goldbach *dit que tout nombre pair plus grand que 2 peut être écrit comme la somme de deux nombres premiers.*

1. *Écrire une fonction* `boolean isPrime(int p)` *qui implémente la spécification suivante :*

```
# Teste si un entier est premier.
#
# Entree : un entier p plus grand que 1.
# Sortie : true si et seulement si n est premier, false sinon.
```

2. *Écrire la spécification de la fonction* `isGoldbach(n)` *qui renvoie* `True` *si et seulement si la conjecture est vérifiée pour l'entier* `n`. *(Si* `n` *n'est pas pair ou est strictement plus petit que 2 la fonction renvoie* `True`*.)*
3. *Donner le code de la fonction* `isGoldbach(n)`. *(On utilisera la fonction précédemment définie.)*
4. *Écrire une fonction* `goldbach(p)` *qui implémente la spécification suivante. (On utilisera la fonction précédemment définie.) :*

```
# Verifie la conjecture de Goldbach jusqu'a un certain rang.
#
# Entree : un entier n.
# Sortie : True si et seulement si tout entier inferieur ou
# egal a p verifie la conjecture de Goldbach.
#
```

□

Exercice 10 (Chaîne correspondante à un tableau, ⋆)
Écrire une fonction qui implémente la spécification suivante :

```
# Entree : un tableau de tableaux d'entiers t.
# Sortie : une chaine de caracteres le representant.
#
# Par exemple, si t vaut 1,2,3,2,3,7, la chaine de caracteres
# sera "1,2,3,2,3,7".
#
```

□

Exercice 11 (Tri d'un tableau, $\star\star$)

1. *Écrire une procédure* `swap` *qui implémente la spécification suivante :*

```
#
# Echange le contenu de deux cases d'un tableau.
#
# Entree : un tableau d'entiers t.
# Entree : un entier i compris dans les bornes du tableau.
# Entree : un entier j compris dans les bornes du tableau.
#
# Apres avoir execute cette procedure les contenus des
# cases i et j du tableau t sont echanges.
#
```

2. *En utilisant la procédure précédente, écrire une procédure* `findmin` *de spécification suivante :*

```
#
# Echange le contenu de la case i avec la case j du tableau t qui
# contient le plus petit entier des cases d'indice compris entre
# i et l'indice de fin du tableau.
#
# Entree : un tableau d'entiers t.
# Entree : un entier i compris dans les bornes du tableau.
#
#
```

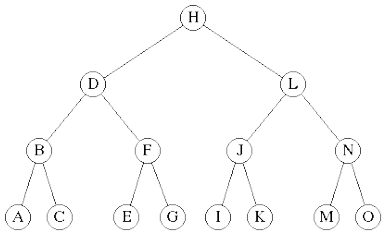
3. *En utilisant la procédure précédente, écrire une procédure* `sort` *qui implémente la spécification suivante :*

```
#
# Tri le tableau.
#
# Entree : un tableau d'entiers t.
#
```

□

Exercice 12 (Tournoi, $\star\star\star$)

Un tournoi est un arbre binaire (les nœuds de l'arbre contiennent toujours deux fils) dont chaque noeud interne correspond au gagnant du match entre ces deux fils. Par exemple, dans le tournoi



on a $B = \max(A, C)$, $D = \max(B, F)$, etc. Les participants au tournoi sont les entiers apparaissant sur les feuilles de l'arbre ; dans l'exemple ci-dessus, les participants sont donc les entiers A, C, E, G, I, K, M, O. Pour simplifier le problème, le nombre de participants est supposé être une puissance de 2.

Un arbre est représenté par un tableau en décidant que :

- *la racine de l'arbre est à la position 0 ;*
- *le fils gauche d'un nœud situé dans la case i est situé dans la case $2 * i + 1$;*

— *le fils droit d'un nœud situé dans la case i est situé dans la case $2 * i + 2$.*

Ainsi, l'arbre précédent peut être représenté par le tableau de taille 15 suivant :

```
{ H, D, L, B, F, J, N, A, C, E, G, I, K, M, O }
```

Pour représenter un tournoi avec 2^n participants, il faut $2^{n+1} - 1$ noeuds dans l'arbre. (Montrez-le par récurrence pour vous en convaincre.) Il faut donc créer un tableau de taille $2^{n+1} - 1$ et on l'initialise avec la valeur -1 pour indiquer qu'aucun match n'a encore eu lieu.

Pour commencer le tournoi, on place les participants sur les feuilles de l'arbre. Tant qu'il ne reste pas qu'un seul participant, on fait un tour supplémentaire de jeu. Un tour de jeu consiste à faire jouer tous les matchs du niveau le plus bas de l'arbre dont les matchs ne sont pas encore joués. On fait alors "remonter" le gagnant de chaque match joué pour ce tour au niveau suivant, ce qui a pour effet de préparer le tour suivant, joué au niveau du dessus dans l'arbre.

1. *Après avoir spécifié et implémenté des fonctions utiles pour :*
   — *construire l'arbre nécessaire à la représentation d'un tournoi entre 2^n participants représentés par un tableau ;*
   — *représenter l'indice du père d'un nœud dans l'arbre ;*
   — *faire jouer deux participants en faisant remonter le gagnant dans l'arbre ;*
   Écrivez une fonction `tournament` *de spécification suivante :*

```
#
#
# Entree : un entier n.
# Entree : un tableau d'entiers t de taille 2 puissance n.
# Sortie : un tableau representant le tournoi des participants
# de t.
#
#
```

2. *Comment généraliser la fonction précédente à un nombre de participants qui n'est pas une puissance de 2 ?*
3. *Après avoir spécifié et implémenté une fonction qui calcule l'ensemble des participants qui ont perdu contre le vainqueur d'un tournoi, écrivez une fonction* `kbest` *de spécification suivante :*

```
#
# Entree : un tableau d'entiers t.
# Entree : un entier k compris entre 1 et la taille du tableau.
# Sortie : un tableau representant les k meilleurs participants
# de t.
#
```

□