Introduction à la Programmation 1 PYTHON

51AE011F

Séance 9 de cours/TD

Université Paris-Diderot

Objectifs:

— Reconnaître les erreur de programmation

— Débogage "scientifique" d'un programme

1 Erreurs et débogage

Typologies d'erreurs de programmation [COURS]

Dans la pratique de la programmation vous avez déjà rencontré plusieurs types d'erreurs de programmation. Classifions les :

erreurs de syntaxe le code du programme ne respecte pas les règles de syntaxe du langage de programmation ; le programme n'est pas compris par le compilateur (ou l'interpréteur, le cas échéant). Vous n'arrivez pas à obtenir un programme exécutable.

erreurs de typage la syntaxe du programme est correcte, mais l'usage d'une ou plusieurs expressions n'est pas compatible avec leur contexte. Par exemple, l'évaluation de ces expressions donne des valeurs dont le type n'est pas compatible avec le type attendu. Vous n'arrivez pas à obtenir un programme exécutable non plus.

Dans la catégorie plus générale de non-respect de la sémantique du langage de programmation on inclut aussi d'autres erreurs, comme par exemple l'appel d'une fonction jamais définie, l'appel d'une fonction avec un nombre incorrect de paramètres, etc.

non-respect de la spécification le programme est syntaxiquement correct et ne contient aucune erreur de type. Vous arrivez à obtenir un programme qui s'exécute. Dans certain cas, l'exécution du programme donne des résultats (ou un comportement) diffèrent de ce que sa spécification demande. Il s'agit d'erreurs *à l'exécution*.

"échec" un cas particulier du non-respect de la spécification est l'échec à l'exécution, non prévu par la spécification, menant à l'arrêt anticipé de l'exécution d'un programme, par exemple à cause d'une exception. Remarquez qu'une spécification peut *prévoir* l'arrêt d'un programme, même avec une exception. Dans ce cas il ne s'agit pas d'un non-respect de la spécification (mais il peut s'agir d'une spécification mal conçue).

Exercice 1 (Bestiaire d'erreurs, ⋆)
Trouver, classifier, et corriger (si possible) toutes les erreurs dans les fragments de code suivants.

```
1 for 3i in range (0, k, 1) :
2 print (i + "A")
```

code/syntCompErrEx.py

```python
def hello (x) :
    n = 3 + "an integer"
    print ("Hello, ")
    print (x)
    print (" world!" )
    return x

halo (42)
hello (42, 4)
hello ("45)
```

code/typeCompErrEx.py

```python
# spécification (informelle): renverser la liste d'entiers
# [2, 5, -12, 8] et afficher le résultat sur la sortie standard
#

lis = [2, 5, -12, 8]
leng = len (lis)
revlis = [0] * leng
for i in range (0, leng, 1) :
    revlis[i] = lis[leng - i]
for i in range (0, leng, 1) :
    print (lis[i])
```

code/excepRunTErrEx1.py

```python
# proportion des entiers sur la somme d'un tableau
#
# Entree : une liste d'entiers t
# Sortie : une liste d'entiers dont chaque element en position i vaut
# l[i]/sum ou l[i] et l'entier correspondant dans la liste l et sum la
# la somme des éléments de l

def proportion (l) :
    sum = 0
    leng = len (l)
    proportion = [0] * leng
    for i in range (0, leng, 1) :
        sum = sum + l[i]
    for i in range (0, leng, 1) :
        proportion[i] = i / sum
    return proportion

proportion([2,-2])
```

code/excepRunTErrEx2.py

□

Débogage [Cours]

— Le *debugging* (ou *débogage*) est l'approche qu'on suit pour corriger les erreurs de programmation. Notamment, pour corriger les erreurs de non-respect de la spécification.

— Plus précisément, le débogage s'attaque aux *bugs* (ou *bogues*) : défauts d'un programme qui causent un comportement différent du comportement attendu. Pour pouvoir corriger un bug, il faut d'abord le trouver. Le trouver implique l'identification des lignes de code où l'intuition du développeur sur le comportement du programme se sépare du modèle formelle d'exécution

— Bien que pour déboguer vous pouvez toujours vous baser sur des tests empiriques, de la relecture du code, des intuitions, etc, le débogage proprement dit est une application rigoureuse de la *méthode*

scientifique à la programmation.

— Le débogage "scientifique" prévoit donc :

1. *observation* d'une erreur
2. formulation d'une *hypothèse* qui explique l'erreur
3. *prévision*, compatible avec l'hypothèse, d'un comportement non encore observé du programme
4. *expérimentation* pour tester l'hypothèse
   — si l'expérience confirme la prévision (hypothèse confirmée), *raffinement* de l'hypothèse
   — si non (hypothèse réfutée), formulation d'une hypothèse alternative
5. répétition des points 3–4 jusqu'à l'obtention d'une hypothèse valide qui ne peut pas être raffinée ultérieurement

— pour *observer une erreur* il ne suffit pas de l'avoir "vu passer" quelques fois, il faut être capable de la *reproduire* systématiquement. Vous devez identifier une série d'entrées pour votre programme qui, *d'une façon déterministe*, mènent au comportement incorrecte du programme. Cela s'applique aussi à toutes les étapes d'expérimentation prévues par l'"algorithme" de débogage ci-dessus. La reproductibilité est l'essence de la méthode scientifique !

Exercice 2 (Déboguer, ⋆⋆)

Tester, déboguer "scientifiquement", et corriger le programme suivant :

```
def reverseList (li) :
    leng = len (li)
    for i in range (0, leng, 1) :
        li[leng - i - 1] = li [i]
    return li
```

code/exoRevListBuggy.py

□

Tri par insertion [Cours]

— Un algorithme de tri est un algorithme qui permet de réordonner une collection d'éléments (par exemple une liste d'entiers) selon un ordre déterminé (p.ex. en ordre croissant). Le tri par insertion est un des algorithmes de tri les plus simples.

— Dans le tri par insertion on maintient une collection d'éléments déjà triés. Cette collection est initialement vide ; à la fin de l'exécution elle contiendra la totalité des éléments, dans le bon ordre.

— À chaque itération, le tri par insertion prend un des éléments encore à trier et l'insère à la bonne position parmi les éléments déjà triés. Le nombre d'éléments déjà triés augmente donc de 1 à chaque itération.

Exercice 3 (Déboguer, ⋆⋆⋆)

Tester, déboguer "scientifiquement", et corriger le programme suivant :

```
# Insère un élément dans un tableau partiellement trié
#
# Entree : li une liste d'entiers partiellement triée, li est
# triée par ordre croissant jusqu'à l'indice (last - 1).
# Sortie : A la fin de la procedure, li est triée jusqu'à l'indice last.

def insert (x, li, last) :
    i = len (li) - 1
    while (i > 0 and li[i] >= x) :
        li[i] = li[i-1]
        i = i - 1
    li [i] = x

# Trie la liste d'entiers donnée en entrée
```

```
#
# Entree : li une liste d'entiers
# Sortie : A la fin de la procedure, la liste li est triée.
def sort (li) :
    for i in range (1, len(li), 1) :
        insert (li[i], li, i)
```

code/exoInsertionSortBuggy.py

□

2 DIY

Exercice 4 (Médiane d'une liste, ⋆)
On s'intéresse ici à des listes de nombres entiers à une seule dimension, possédant un nombre impair d'éléments et dont tous les éléments sont différents. On appelle médiane d'une telle liste `L` *l'élément* `m` *de* `L` *tel que* `L` *contienne autant d'éléments strictement inférieurs à m que d'éléments strictement supérieurs à m.*

1. *Écrire une fonction* `nbInf` *qui, étant donné une liste d'entiers* `L` *et un entier* `v`*, renvoie le nombre d'éléments de la liste* `L` *strictement inférieurs à* `v`.
2. *Écrire une fonction* `mediane` *qui, étant donné une liste* `L` *satisfaisant les conditions énoncées, renvoie la position de la médiane dans la liste* `L`.
3. *Écrire une fonction* `verifArray` *qui, étant donné une liste d'entiers* `L` *, renvoie la valeur booléenne* `True` *si la liste* `L` *satisfait effectivement les conditions énoncées et la valeur* `False` *si ce n'est pas le cas.*
4. *Écrire une fonction* `listInf` *qui, étant donné une liste* `L`*, renvoie une liste de taille 0 si* `L` *ne satisfait pas les conditions énoncées et une liste contenant tous les éléments de* `L` *inférieurs à la médiane de* `L` *sinon.*

□

Exercice 5 (Sudoku, ⋆)
On rappelle qu'une grille de Sudoku est une grille de 9 cases sur 9 cases contenant des chiffres de 1 à 9 de telle sorte que sur chaque ligne et sur chaque colonne, chaque chiffre apparaît exactement une fois. De plus, on peut voir cette grille comme composée de 9 carrés de 3 sur 3 cases et dans chaque carré, chaque chiffre doit aussi apparaître exactement une fois. Dans ce qui suit nous encoderons une grille de Sudoku par une liste de listes d'entiers `sud` *telle que* `sud[i][j]` *pour* `i` *et* `j` *allant de* 0 *à* 8 *représente le contenu de la case se trouvant à l'intersection de la* $i+1$*-ème ligne et de la* $j+1$*-ème colonne.*

1. *Écrire une méthode* `oneOcc` *qui prend en paramètre une liste d'entiers et renvoie* `True` *si dans la liste tous les entiers sont compris entre 1 et 9 inclus et si chaque entier n'apparaît qu'une seule fois.*
2. *Écrire une méthode* `columnToList` *qui prend en argument une grille de sudoku* `sud`*, un numéro de colonnes |j| comrpris entre 1 et 9 et renvoie la liste des éléments situés sur cette colonne dans la grille.*
3. *Écrire une méthode* `squareToList` *qui prend en argument une grille de sudokus* `sud`*, un numéro de carrées |j| comrpris entre 1 et 9 et renvoie la liste des éléments situés dans ce carré de 3 par 3 dans la grille. (On suppose que les carrés de 3 par 3 sont numérotés de 1 à 9 en partant en haut à gauche et en finissant en bas à droite),*
4. *Écrire une méthode* `isSudoku` *qui prend en arguments une liste de listes et renvoie* `True` *si elle correspond à une grille de sudoku et* `False` *sinon (n'oubliez de vérifier toutes les conditions y compris la taille de la grille).*

□

Exercice 6 (Recherche dichotomique dans une liste triée, ⋆⋆)
On souhaite définir une fonction pour rechercher efficacement un élément dans une liste d'entiers. La fonction prendra en paramètres une liste d'entiers `li` *et un entier* `x` *et renverra un indice* `i` *tel que* `li[i]` *soit égal à* `x`*, si un tel indice existe, et* `-1` *sinon.*

1. *Définir une fonction* `searchA` *qui vérifie la spécification ci-dessus et qui renvoie un résultat* **dès qu'***un indice qui satisfait la spécification a été trouvé.*
2. *Dans quelles situations la fonction* `searchA` *effectue-t-elle le plus d'accès à la liste ? Dans ce cas, combien de fois accède-t-on à la liste ?*
3. *On suppose maintenant, et dans la suite de l'exercice, que les listes d'entiers que reçoit notre fonction sont toujours triées par ordre croissant (c'est-à-dire que* `li[i]` $\leq$ `li[i+1]` *tant qu'on est dans les bornes de la liste).*
   Définir une nouvelle fonction `searchB` *qui assure que si la valeur recherchée est hors des valeurs extrêmes de la liste, on ne fait pas plus de deux accès à la liste avant de renvoyer* `-1`.
4. *On conserve la même hypothèse que ci-dessus (les listes passées à la fonction sont triées) et on demande de définir une fonction* `searchC` *qui tire partie du fait que la liste est triée à partir de l'observation suivante :*
   Si la valeur recherchée est présente, on la trouvera dans la première moitié ou la seconde moitié de la liste ; la comparaison de cette valeur avec une seule case de la liste suffit à savoir dans quelle moitié rechercher.
   Pour cela, on déclarera dans `searchC` *deux variables auxiliaires* `imin` *et* `imax` *(de type* `int`*0 qui serviront à stocker les indices des extrémités de la portion de la liste dans laquelle on doit chercher la valeur. Initialement ces indices devront couvrir la liste entière et on poursuit la recherche jusqu'à ce qu'on ait trouvé la valeur recherchée ou que la portion de la liste dans laquelle recherchée soit de longueur 1.*
5. *(⋆⋆⋆) Dans quelle situation* `searchC` *accède-t-elle le plus de fois aux valeurs de la liste ? Dans ce cas, combien de fois la fonction accède-t-elle aux valeur de la liste ?*
6. *Pour aller plus loin : une fonction peut appeler une autre fonction définie dans le programme. Elle peut en fait s'appeler elle-même. (Est-ce que cela change quelque chose à l'exécution d'appels imbriqués de fonctions tel qu'on l'a vu dans cette séance ?)*
   On peut utiliser cela pour mieux organiser le code de la fonction de recherche. On définit `searchD` *qui repose sur une fonction auxiliaire comme suit :*

```
def searchD (li, x) :
    return searchAux(t, x, 0, len(t) - 1)
```

   où `searchAux(t, x, imin, imax)` *recherche* `x` *entre les indices* `imin` *et* `imax` *de la liste* `li`. *Pour cela, searchAux utilisera la même observation que plus haut, mais en vue de réappeler la fonction searchAux avec des bornes de recherche mises à jour, par exemple :*
   `searchAux(li, x, imin, (imin+imax)/2)`.

□

Exercice 7 (Spécification, ⋆⋆⋆)

Écrire un programme `Backdoor` *qui, en boucle infinie, lit une ligne de texte de l'utilisateur à la fois, et l'affiche sur l'écran après l'avoir converti en majuscule les caractères "a", "b", "c", "d", "e", et "f" ; les autres caractères ne sont pas modifiés. Si, par contre, la ligne entrée est le mot secret* `"3XzRWo"` *(une "backdoor" insérée par le développeur), le programme doit terminer avec une exception après avoir affiché sur l'écran l'ASCII art suivant :*

```
        (__)
        (oo)
  /------\/
 / |    ||
*  /\---/\
   ~~   ~~
```

Vous avez à disposition : la fonction `input()` *pour lire une ligne de texte de l'utilisateur.*

À noter : *dans ce cas terminer avec une exception (dans un cas très spécifique) fait partie de la spécification du programme. Il ne s'agit donc pas d'une erreur de programmation.*

□