Tests

juillet 2020

Remerciements aux collègues de l'université d'Aix Marseille.

Le principe de ce TP n'est pas d'écrire soi-même des programmes qui réalisent une fonctionnalité demandée, mais de concevoir des tests permettant de mettre en évidence des bugs dans des programmes donnés. Certains des programmes fournis sont corrects, d'autres non : votre objectif est de détecter tous les programmes faux.

1 Trouver le(s) bon(s) programme(s)

On s'intéresse ici au calcul de la factorielle d'un entier : $n! = 1 \times 2 \times 3 \times \cdots \times (n-1) \times n$
(avec pour convention $0! = 1$).

Exercice 1
Écrire, **sur papier**, *un jeu de tests bien choisis pour un programme censé calculer la factorielle de n'importe quel entier $n \geq 0$.*

Le fichier fourni `facto.py` contient 11 implémentations du calcul de la factorielle, nommées respectivement `factorielle1, factorielle2 ... factorielle11`. Certaines d'entre elles sont correctes, d'autres non.

Il vous est demandé de ne pas regarder le contenu de ce fichier. Deux méthodes sont possibles pour faire l'exercice 2 :

1. sans modifier le fichier `facto.py` et sans utiliser `doctest` :
   — créez un fichier `test_facto.py` et, au début de votre code, importez le fichier fourni :
   ```
   import facto
   ```
   — vous accédez ensuite à la fonction `factorielle`n en écrivant `facto.factorielle`n
   — ou plus simplement (mais pas recommandé dans le cas plus général d'un module quelconque), commencez par
   ```
   from facto import *
   ```
   puis utilisez directement `factorielle`n. Par exemple :
   ```
   print("factorielle1(0) = ",factorielle1(0))
   ```
2. en modifiant le fichier `facto.py` et en utilisant `doctest` : ajoutez, au début de chaque fonction `factorielle`n, dans la documentation, le jeu de tests correspondant (cf diaporama).

Exercice 2 *Utilisez votre jeu de tests pour déterminer lesquelles des 11 fonctions proposées sont :*
— ***incorrectes*** *dès lors qu'au moins un des tests est falsifié,*
— *ou* ***probablement correctes*** *dès lors que tous les tests sont validés.*

2 Trouver le bug

Après ces petites mises en bouche, passons au plat principal. Il s'agit d'un bug rencontré par les lecteurs MP3 Microsoft Zune, commercialisés à partir du 14 novembre 2006. Un beau matin, les utilisateurs voulant allumer leur appareil ont eu la désagréable surprise de découvrir l'écran de démarrage restant gelé. Impossible pour eux d'utiliser leur lecteur. Le plus « drôle » dans l'histoire est que rien ne pouvait être fait pour régler ce problème, mais que dès le lendemain matin, tout s'est remis à fonctionner comme si rien ne s'était jamais passé. Une mise à jour du logiciel a évidemment ensuite été mise en ligne pour que le bug ne se reproduise pas par la suite.

Mais quel est donc ce bug étrange ? Comme on peut le deviner du fait du redémarrage sans souci le lendemain, le bug se situe dans la partie du code en charge des fonctions calendaires du lecteur. Microsoft a rapidement trouvé l'origine du bug qu'ils ont depuis publié : la faute provient de 10 lignes de code écrites originellement en C, traduites en Python pour l'occasion et fournies dans le fichier `zune.py`.

Ces quelques lignes prennent en argument un paramètre `days` qui est initialisé, lors du démarrage du lecteur, au nombre de jours écoulés depuis le 1er janvier 1980. Le principe du code est assez simple, et utilise un prédicat `is_leap_year` qui renvoie `True` si l'année est bissextile et `False` sinon.

Exercice 3

Commencer par écrire ce prédicat `is_leap_year` en se rappelant qu'un numéro d'année est bissextile si elle est divisible par 4 sans être divisible par 100, ou si elle est divisible par 400.

On peut donc maintenant tester le code du Zune, en attribuant préalablement une valeur à la variable `days`.

Exercice 4

Trouver, à l'aide d'un jeu de tests bien choisis, une valeur de `days` pour laquelle le programme a un comportement indésirable. Décrire le comportement du programme dans ce cas et déterminer une date possible à laquelle le lecteur MP3 a refusé de fonctionner. Expliquer également pourquoi tout rentre dans l'ordre le lendemain.

Ne trichez pas ! *Si vous ne trouvez pas le bug avec votre jeu de tests, cherchez à compléter celui-ci plutôt qu'à trouver le bug par lecture du code.*

Exercice 5

Proposer finalement une correction du bug de Zune (au fait, pourquoi faut-il un correctif si tout rentre dans l'ordre le lendemain ?). Tester à nouveau votre programme pour s'assurer qu'il ne présente a priori *plus d'erreur.*

Morale de l'histoire : tester (à défaut de prouver la correction de) son code est crucial, même pour des petits programmes aussi indolores que le calcul de la date !

3 Validation d'un tri

En préparation de la séance sur les tris de demain :

Exercice 6 (Validation d'un programme de tri)

Écrire une fonction `oracle_tri` qui prend en argument :

- *une fonction de tri (eh oui, on peut donner des fonctions en argument d'autres fonctions !)*
- *une liste*

qui exécute la fonction de tri sur cette liste, et vérifie que le résultat obtenu est conforme (qu'il s'agit bien des mêmes éléments en ordre croissant).

Exercice 7 (Génération de tests)

Écrire différentes fonctions générant des listes à trier :

- *déjà en ordre croissant, ou presque*
- *en ordre décroissant*
- *en ordre aléatoire*
- *avec ou sans doublons*
- *...*

Chacune de ces fonctions pourra être paramétrée par la taille de la liste à générer.

Exercice 8

Testez un des algorithmes de tri que vous avez implémenté (ou à défaut, la méthode `sort` de Python) à l'aide de votre oracle et de vos procédures de génération.

Vous pouvez même écrire une fonction qui automatise tout le processus : génération de listes de différentes formes et taille, puis test systématique avec l'oracle et affichage des éventuels cas problématiques.