Cours – Mise au point des programmes

Objectifs :

- ⇨ Apprendre à tester son code et à le rendre plus robuste
- ⇨ Voir comment prouver la correction d'un algorithme

I – Mise au point

Pour illustrer cette partie du cours, nous utiliserons le programme suivant :

```
def est_croissant(t):
    i = len(t) - 1
    while i >= 0:
        if t[i] <= t[i + 1]:
            return True
        else:
            return False
        i -= 1
```

1) Tests

Dès qu'une fonction est écrite (même les plus simples), il est important de la tester pour vérifier son bon fonctionnement. Pour cela, on utilise des jeux de tests, c'est-à-dire un ensemble de valeurs d'entrées pour la fonction dont on connait la valeur de sortie théorique et on compare cette valeur à celle données par la fonction.
Exemple : on programme une fonction `float racine(float x)` qui calcule la racine carrée d'un nombre flottant. On testera que `racine(4.0)` renvoie bien `2.0`, que `racine(49)` renvoie bien `7.0` et que `racine(2)` renvoie bien `1.414213562373095`
Il faut bien penser à tester les valeurs particulières comme `racine(1.0)` et `racine(0.0)`.
De tels tests (qui n'opèrent que sur une valeur bien précise ou quelques valeurs particulières choisies) sont appelés

..... .

On peut également écrire un petit programme de test qui fabrique un jeu de tests aléatoires pour vérifier le fonctionnement. Pour ce faire on aura besoin d'une fonction pour créer des paramètres d'entrée de manière aléatoire et d'une autre pour valider le résultat de la fonction à tester.

Exemple :

```
def test_fonction_tri(f, nb_tests:int, taille_max:int):
    nb_erreurs = 0
    for taille in range(taille_max):
        for _ in range(nb_tests):
            t = tableau_aleatoire(taille)
            tableau_original = t.copy()
            f(t)
            if not est_trie(t):
                print("Erreur pour le tableau", tableau_original)
                print("trié en", t)
                nb_erreurs += 1
                if nb_erreurs > 20:
                    print("Trop d'erreurs, abandon du test...")
                    return
    print("Fin de la série de test.", nb_erreurs, "erreurs détectées")
```

Application 1 :
Pour notre fonction `est_croissant`, écrire une docstring, puis réaliser un test simple sur un tableau croissant. Qu'obtient-on ?

2) Débogage

a. Exceptions et traceback

Lorsque l'interpréteur détecte une erreur dans le programme, il **lève une exception** qui arrête le traitement normal du flux du programme. Dans le cas d'une erreur de syntaxe, l'exception est même levée avant que la moindre ligne de code ne soit exécutée.
Pour les autres types d'erreurs, l'interpréteur affiche le ..... qui indique le type d'erreur ainsi que la pile des appels : ce sont les **informations de contexte**.
La pile des appels est la suite de fonctions qui ont été appelées successivement au moment de l'exécution de la ligne qui a causé l'erreur.

Les informations du traceback doivent être analysées avec soin car elles mènent souvent rapidement au diagnostic de l'erreur.

Plusieurs instructions permettent également de gérer les exceptions en python :

```
try:
    code à exécuter qui risque de provoquer une exception
except type_d_exception as e:
    bloc de traitement de l'erreur
[finally]:
    Bloc exécuté dans tous les cas (qu'il y ai eu exception ou pas)
```

On peut également provoquer la levée d'une exception avec l'instruction `raise` *`type_d_exception`*.

Application 2 :
A partir de l'assertion manquée vue précédemment, corriger la fonction `est_croissant` pour qu'elle ne génère plus d'exception.

b. Suivi d'exécution et utilisation du débogueur

Application 3 :
Effectuer d'autre tests pour voir si la fonction est correcte.

Lorsque le comportement du programme n'est pas correct (échec aux jeux de tests notamment) mais qu'aucune exception n'est levée, il y a souvent nécessité de déboguer le programme. Pour cela il est utile de suivre l'exécution du programme au ralenti en surveillant l'évolution des variables afin de voir à quel moment (et pourquoi) celle-ci s'écarte des valeurs attendues.

Pour ce faire on peut rajouter des instructions dans le programme pour afficher des informations dans la console python au fur et à mesure de l'exécution du programme.

Application 4 :
Modifier la fonction précédente en rajoutant les lignes suivantes puis exécutez-là sur l'exemple qui la faisait échouer afin de comprendre le problème. Corriger le programme en conséquence.

```
def est_croissant(t):
    """Renvoie True si les éléments de t sont rangés en ordre croissant."""
    print("Test avec le tableau", t)
    i = len(t) - 1
    while i >= 0:
        print("Nouveau tour avec i =", i)
        if t[i-1] <= t[i]:
            return True
        else:
            return False
        i -= 1
```

Une fois le programme corrigé, on retire les lignes qui affichaient les informations utiles à la correction du programme en essayant de ne pas en oublier (ou on les met en commentaire). Ceci peut être fastidieux. De même

il est parfois difficile de retrouver l'information utile lorsqu'il y en a beaucoup d'affichées. Pour ces raisons, on utilise souvent préférentiellement un ..... .

Un débogueur est un programme permettant d'exécuter le programme à déboguer en suivant le flux du programme et la valeur des variables.
Le débogueur propose principalement 2 modes d'exécutions :

- Le mode ..... où l'utilisateur avance manuellement d'un ou plusieurs « pas » élémentaires dans le programme en rentrant ou pas dans les sous-fonctions. Dans Thonny, cela se fait avec la commande « Déboguer le script courant (Ctrl-F5) » puis les touches F6 ou F7 pour avancer dans l'exécution. A tout instant, on peut voir l'état des variables (dans la fenêtre variables)
- Le mode ..... où l'utilisateur pose des « points d'arrêt » dans le programme (Avec Thonny, cela se fait en double-cliquant sur le numéro de ligne). Lorsqu'on lance le débogage, l'exécution se déroule alors normalement jusqu'à ce que l'interpréteur atteigne une ligne où se trouve un point d'arrêt. A ce moment-là, l'exécution est stoppée, l'affichage des variables réactualisé, et on bascule en mode pas à pas. Ce mode est particulièrement utile lorsqu'on souhaite déboguer un bout de code qui n'est exécuté qu'assez tard dans l'exécution normale du programme.

NB : Dans les débogueurs plus évolués, on peut également poser des points d'arrêt conditionnels. Ce sont des lignes du programme sur lesquels l'exécution ne s'arrêtera que si une expression qu'on aura précisée est vrai (par exemple arrêt si la variable i a une valeur supérieure ou égale à 100). On peut également déclarer des « expression espionnes » qui seront évaluées à chaque arrêt de l'exécution.

3) Preuve de la correction d'un algorithme : l'invariant de boucle

Quand les programmes contiennent des boucles, il peut être difficile d'être sûr que celle-ci est écrite correctement, que les indices et variables de boucles sont corrects au début, pendant et à la fin de la boucle. Pour répondre à cette problématique (« Ma boucle est-elle correctement programmée ? »), on peut utiliser un **invariant de boucle** : c'est une propriété qui est vraie initialement, pendant chaque tour de boucle et donc nécessairement vraie à la fin de la boucle.

Voyons ceci à travers un exemple :

```
def div_euclidienne(a, b):
    q = 0
    r = a
    while r >= b:
        q += 1
        r -= b
    return q, r
```

Grâce à l'invariant de boucle, on a pu PROUVER que notre fonction réalisait bien la division euclidienne. C'est évidemment quelque chose de très fort en programmation puisqu'on est alors sûr que si les préconditions sont satisfaites, le programme donnera une réponse juste (à condition de se terminer ce qui est assez simple à démontrer).

Références :

Documentation du code python : https://www.codeflow.site/fr/article/documenting-python-code
PEP8 (guide d'écriture) : https://www.python.org/dev/peps/pep-0008/