Leçon d'informatique : Assertions et jeu de tests

S. Benlhajlahsen - PCSI$_1$

Sommaire

I Erreurs et exceptions **1**

II Assertions **2**

III Jeu de tests **2**

Extrait du programme :

Notions	Commentaires
Assertion.	L'utilisation d'assertions est encouragée par exemple pour valider des entrées. La levée d'une assertion entraîne l'arrêt du programme. Ni la définition ni le rattrapage des exceptions ne sont au programme.
Jeu de tests associé à un programme.	Il n'est pas attendu de connaissances sur la génération automatique de jeux de tests ; un étudiant doit savoir écrire un jeu de tests à la main, donnant à la fois des entrées et les sorties correspondantes attendues. On sensibilise, par des exemples, à la notion de partitionnement des domaines d'entrée et au test des limites.

I Erreurs et exceptions

Erreurs et exceptions dans un programme : On a plusieurs fois rencontré des messages d'erreurs dans le *shell*. Généralement, ce sont des erreurs de syntaxes ou des **exceptions**.

I.A Erreurs de syntaxe

Dans l'exemple de la figure 1, le programme détecte l'oubli des deux points (`:`). Celui s'arrête et n'exécute pas le reste du programme. Le shell renvoie alors le message d'erreur `SyntaxError`.

```
>>> if 3>2
        print("hello")
SyntaxError: invalid syntax
```

FIGURE 1

Remarque : Les erreurs de syntaxe sont des erreurs qui doivent être corrigées dans la phase de *debug*.

I.B exceptions

Même si une instruction ou une expression est syntaxiquement correcte, elle peut générer une erreur lors de son exécution. Les erreurs détectées durant l'exécution sont appelées des `exceptions` et ne sont pas toujours fatales. Elle peuvent être gérer par les instructions `assert` ou `raise`.

Quelques exemples d'exceptions :

- La division par zéro :

```
>>> 10 * (1/0)
ZeroDivisionError: division by zero
```

- Variable qui n'a pas été préalablement définie :

```
>>> 3+x*2
NameError: name 'x' is not defined
```

- Erreur de type, opération entre des variables non-compatibles

```
>>> "2"+3
TypeError: can only concatenate str (not "int") to str
```


- Erreur de type, opération entre des variables non-compatibles

```
>>> "2"+3
TypeError: can only concatenate str (not "int") to str
```


- On tente d'importer un module qui n'existe pas :

```
>>> import maths
ModuleNotFoundError: No module named 'maths'
```

- On souhaite extraire un élément d'une liste avec un mauvais indice :

```
>>> L = [0, 1, 2]
>>> print(L[3])
IndexError: list index out of range
```

Remarque : Ces exceptions sont les principales que vous rencontrerez mais il en existe beaucoup d'autres.

Remarque : Une exception peut être laissée si on sait d'avance que cette situation ne peut pas apparaître dans l'execution.

II Assertions

Idée : Dans une phase de debug, il peut être intéressant d'arrêter l'execution du programme si une erreur de syntaxe ou une exception est rencontrée. C'est ce qui se passe en figure 2 où la division par zéro fait arrêter le programme. On a utiliser l'instruction `assert` et on a fait une assertion :

- si la condition après `assert` est vérifiée, le programme passe à la suite ;
- le programme s'arrête si la condition n'est pas vérifiée. La suite du programme (`print("bonjour")`) est ignorée.

```
>>> def f(x):
        assert x!=0
        return 1/x
print(f(0))
print("bonjour")
------
AssertionError
```

FIGURE 2

Remarque hors-programme - lever une exception : Dans le cadre du programme, on se limitera à `assert`. On notera que la commande `assert` ne permet pas de cibler un type d'erreur précis. Prenons l'exemple de la figure 3. On souhaite toujours calculer l'inverse de `x`. Le programme va essayer de calculer `1/x` (`try`) puis

- si on demande `f(0)`, il rencontre la première exception, on lui dit de passer à la suite le programme (`pass`).
- si on demande `f("a")`, il rencontre la deuxième exception, on lui dit de s'arrêter et de soulever l'exception (`raise`).

Cette méthode permet de gérer des erreurs sans arrêter le programme.

III Jeu de tests

Maintenant qu'on connaît les principales erreurs, on peut essayer de construire un jeu de test qui permet de les détecter. On se limitera à l'utilisation de `assert`.

```python
def f(x):
    try:
        return 1/x
    except ZeroDivisionError:
        pass
    except TypeError:
        raise Exception("f prend en argument un nombre")
```

FIGURE 3

Exemple 1 - partitionnement du domaine d'entrée Supposons qu'on calcule le reste de la division euclidienne de a par b. On se limite à des entiers avec $a \geqslant 0$ et $b > 0$. On écrit donc une fonction `reste` et on rajoute un *docstring* pour la commenter. On voit un jeu de test apparaître (voir figure 4) :

- vérifier que les arguments en entrée sont des entiers ;
- vérifier qu'ils sont dans le bon intervalle.

On dit qu'on a vérifier le **domaine des entrées**.

```python
def reste(a, b):
    "entrée : deux entiers positifs a et b avec a>= b"
    "sortie : reste de la division de a par b"
    assert type(a) == int          # a est un entier
    assert type(b) == int          # b est un entier
    assert a>=0
    assert b>0
    assert a>b
    return a%b
```

FIGURE 4

Exemple 2 - test des limites : On souhaite découper une liste de taille $n \geqslant 2$ en deux sous-listes non-vide, des `i`-premiers termes et des `n-i`-derniers termes avec $1 \leqslant i \leqslant n-1$. La fonction renverra la liste des sous-listes. Dans cet exemple, on voit bien qu'il faut que l'indice soit dans le bon intervalle. Les assertions doit permettre de tester les limites de `i`.

```python
def separer(L, i):
    """
    entrées : une liste de taille n>=2 et un entier i tq 1<=i<=n-1
    sorties : liste des deux sous-listes
    """
    n = len(L)
    assert type(L)==list
    assert n>=2
    assert i>=1
    assert i<=n-1
    return [ L[:i] , L[i:] ]
```

FIGURE 5

Conclusion : Le jeu de test par les assertions permettent d'éliminer beaucoup d'erreurs dans les programmes. Cette pratique doit être accompagnée de l'utilisation de commentaire ou de `docstring`.