

10.3

Corriger les erreurs

NSI TERMINALE - JB DUTHOIT

10.3.1 Comment corriger les erreurs

La **mise au point** d'un programme consiste à corriger les erreurs identifiées par les tests.

☛ Certaines erreurs sont faciles à corriger, lorsque le message d'erreur (en cas de « plantage ») ou le test indique clairement l'origine de l'erreur.

☛ D'autres sont plus difficiles à traquer et nécessitent un véritable travail de déTECTIVE pour localiser l'erreur, identifier sa cause et enfin la corriger.

Les erreurs les plus difficiles à corriger sont celles pour lesquelles la cause de l'erreur est très éloignée de l'endroit du programme où elle est détectée !

L'approche classique pour localiser une erreur et identifier sa cause est d'ajouter des traces dans le programme, à l'aide d'instructions `print`.

L'instruction `assert` est également utile pour arrêter l'exécution dès que l'on détecte un résultat incorrect, par exemple un invariant qui n'est plus satisfait. Lorsque l'assertion est fausse, Python affiche la pile des appels de fonctions, ce qui permet de voir la chaîne d'appels qui a conduit à l'erreur.

Une autre approche pour comprendre les causes d'une erreur est d'utiliser un outil de débogage. Un tel outil permet d'exécuter un programme "pas-à-pas", c'est-à-dire instruction par instruction, d'afficher les valeurs des variables à chaque pas et de poser des points d'arrêt, c'est-à-dire des instructions sur lesquelles le programme s'arrêtera. Il permet ainsi de suivre l'exécution du programme "à la trace".

Une fois l'erreur corrigée, il faut s'assurer que l'on n'a pas créé de nouveaux problèmes en exécutant à nouveau tous les tests, pas seulement ceux qui ont causé l'erreur.

10.3.2 Les différents types d'erreurs

Les sources possibles d'erreurs dans un programme sont nombreuses. La suite de cette section décrit les types d'erreurs les plus fréquentes et donne des indications sur la manière de les détecter et de les corriger.

Pour réduire les risques d'erreurs, il est recommandé d'adopter une approche de programmation défensive, c'est-à-dire d'anticiper les erreurs possibles en prenant en compte les indications ci-dessous dès la programmation.

10.3.3 Erreurs de syntaxe

Les erreurs de syntaxe sont dues à des instructions mal-formées, qui ne suivent pas les règles du langage. Elles sont détectées dès le début de l'exécution du programme et sont en général faciles à corriger car le message d'erreur est explicite.

En Python, voici deux erreurs fréquentes :

- Une première erreur fréquente est une mauvaise indentation du code ou le mélange d'espaces et de tabulations.
- Une autre erreur fréquente est l'oubli des deux-points : à la fin des instructions telles que `def`, `if` ou `while`, ou l'utilisation de `=` au lieu de `==` pour un test d'égalité.

10.3.4 Erreurs de type

Contrairement aux langages typés tels que Java, le langage Python ne permet pas de déclarer les types attendus des variables et des paramètres et ne peut donc pas les contrôler avant l'exécution du programme.

Pour se protéger des erreurs de type, il faut utiliser des pré-conditions au début des fonctions pour valider les types des paramètres, notamment avec la fonction `isinstance(val, t)`, qui retourne vrai si `val` est de type `t`. `t` peut être un des types de Python (`int`, `float`, `str`, `tuple`, `list`, `dict`) ou bien une classe définie dans le programme.

10.3.5 Erreurs d'accès

Python vérifie la validité des accès à un élément de tableau ou de tuple, à une clé d'un dictionnaire et à un attribut ou une méthode d'un objet seulement à l'exécution du programme. Pour se protéger de ces erreurs, on peut utiliser les tests suivants :

Expression	Valide si	Test
<code>t[i]</code>	<code>i</code> est un indice du tableau <code>t</code>	<code>i >= 0 and i < len(t)</code>
<code>a, b = t</code>	<code>t</code> est un tuple de 2 éléments	<code>len(t) == 2</code>
<code>d[k]</code>	<code>k</code> est une clé du dictionnaire	<code>k in d</code>
<code>o.a</code>	<code>a</code> est un attribut de l'objet <code>o</code>	<code>hasattr(o, "a")</code>
<code>o.m()</code>	<code>m</code> est une méthode de l'objet <code>o</code>	<code>hasattr(o, "m")</code>

10.3.6 Effets de bords

Un **effet de bord** est une modification d'une variable qui affecte l'état du programme en dehors de la fonction ou de la méthode où elle a lieu.

En Python, cela arrive principalement de deux manières :

- par les **variables globales**. Une variable globale est une variable déclarée en dehors de toute fonction ou méthode. En python, elle peut être utilisée partout dans le programme. En revanche, elle ne peut être modifiée dans une fonction ou une méthode que si on l'a déclarée par l'instruction `global x` dans cette fonction ou méthode. Par conséquent, il est rare de créer un effet de bord ainsi par erreur.
- par **l'aliassage** : L'aliassage survient lorsque deux variables réfèrent à la même donnée, ce qui peut arriver avec les types structurés tableau et dictionnaire et avec les objets, notamment lorsqu'on les passe en paramètre.

Exemple

Voici un exemple d'aliassage :

```
t1 = ['a']
t2 = t1
...
t2[0] = 'b'
print(t1) #['b'] (normal)
def f(t):
    t[0] = 'z'
f(t1)
print(t1) #['z']
```

```
print(t2) #[ 'z' ]
```

Les effets de bord indésirables sont particulièrement difficiles à détecter car, en général, ils ne provoquent pas d'erreur d'exécution et ils se manifestent en d'autres points du code que leur origine.

⚠ Pour limiter les risques, il est important de bien documenter chaque fonction. Si une fonction modifie un paramètre contenant une valeur mutable, il faut l'indiquer explicitement dans la spécification ou bien éviter l'effet de bord en effectuant une copie de la valeur.

Exercice 10.8

On considère le programme suivant :

```
t=[1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 , 10]
def zero(n):
    for i in range(n):
        t[i] = 0
    return t
zero(4)
```

1. Ce programme produit-il une erreur ?
2. Sinon, produit-il un effet de bord sur le tableau **t** ?
3. Quelle est la valeur de **t** à la fin du programme ?

10.3.7 Erreur variable locale

Le code suivant fonctionne :

```
x = 10
def f():
    print(x)
```

mais le code ci-après non !!!

```
x = 10
def f():
    print(x)
x = x + 1
```

⚠ On obtient l'erreur `UnboundLocalError: local variable 'x' referenced before assignment` qui signifie qu'à la ligne 4, on a tenté de faire une affectation. Python considère que **x** est une nouvelle variable...

On peut corriger cela avec `global x` mais c'est une pratique à utiliser avec parcimonie ;-)

10.3.8 Conditionnelles

Les instructions conditionnelles (`if`) comportent des expressions booléennes qui déterminent le flot d'exécution du programme. Les principaux types d'erreurs sont les suivants :

- oubli de certains cas lorsque l'on évalue plusieurs alternatives avec `elif`

- mauvaises conditions aux limites lorsque l'on compare une valeur à une borne, par exemple en utilisant `<` ou lieu de `<=` ou l'inverse
- mauvais usage des opérateurs booléens et en particulier :
`i < n and t[i] == 0` est différent de `t[i] == 0 and i < n`, qui provoque une erreur si `i` est en dehors des bornes de `t`.

Pour éviter ces erreurs, il faut faire des tests !

10.3.9 Boucles

Les boucles bornées (`for`) et non bornées (`while`) contrôlent également le flot d'exécution du programme.

Les boucles non bornées sont contrôlées par une expression booléenne, aussi les mêmes erreurs de mauvaises conditions aux limites et de mauvais usage des opérateurs booléens que ci-dessus peuvent se produire. Ces erreurs peuvent conduire à des boucles infinies, mais la raison la plus courante d'une boucle infinie est de ne pas modifier correctement dans le corps de la boucle les valeurs qui sont utilisées dans la condition booléenne de la boucle.

Il est important d'identifier le variant de boucle, l'expression qui doit changer à chaque itération de la boucle de telle sorte que la condition deviendra fausse.

Les boucles bornées ne peuvent pas produire de boucle infinie. L'erreur la plus fréquente est la mauvaise évaluation des bornes, en particulier avec les expressions `range(n)` et `range(i, j)` qui énumèrent respectivement les valeurs de 0 à `n - 1` (et non pas de 1 à `n`) et de `i` à `j - 1` (et non pas de `i` à `j`).

Comme pour les conditionnelles, il est important de tester son code en prenant soin de couvrir tous les cas limites. Les assertions peuvent également aider à la mise au point, par exemple pour tester un variant de boucle.

10.3.10 Calculs en virgule flottante

Les calculs en virgule flottante sont source d'erreurs car un ordinateur ne peut pas représenter de manière exacte tous les nombres réels. Par exemple, en Python, `0.1 + 0.2` vaut `0.30000000000000004` !

Un problème similaire se manifeste lorsque l'on ajoute ou soustrait un nombre très grand et un nombre très petit : `10e20 - 1`, `10e20` et `10e20 + 1` sont égaux pour Python.

Il en résulte qu'il ne faut jamais utiliser des tests d'égalité stricte sur les nombres flottants, mais la fonction `isclose(a, b)` de la bibliothèque `math` :

```
from math import isclose
print(0.1 + 0.2 == 0.3, isclose(0.1 + 0.2, 0.3)) # False, True
print(10e20 - 1 < 10e20, isclose(10e20 - 1, 10e20)) # False, True
```

Une autre source de problèmes dus à ces approximations est l'accumulation d'erreurs. Dans l'exemple suivant, on voit qu'il est préférable de faire une multiplication plutôt qu'une suite d'additions :

```
s = 0.0
for i in range(1000):
    s += 0.1
print(s, 0.1 * 1000) # 99.999999999986, 100.0
```

10.3.11 Nommage des variables

La lisibilité d'un programme est grandement facilitée si l'on donne des noms expressifs aux fonctions, variables, classes et méthodes. Des noms bien choisis rendent le code plus facile à comprendre et donc les erreurs plus faciles à détecter. Ils réduisent aussi le risque de masquage de nom qui peut être une erreur difficile à identifier.

En Python, il y a deux situations où un nom peut être masqué par un autre :

- on importe une bibliothèque et on déclare localement une fonction qui a le même nom qu'une fonction importée. Ce risque est particulièrement élevé si on importe toutes les fonctions d'une bibliothèque avec `from m import *` car on ne sait pas quels noms sont importés !
- on définit une fonction avec un paramètre qui a le même nom qu'une variable globale. À l'intérieur de cette fonction, le paramètre masque la variable globale, qui ne peut être utilisée.

Pour éviter de masquer un nom exporté par un module, il est recommandé d'utiliser l'instruction `import m` qui oblige à qualifier tous les noms de `m` avec la notation pointée `m.f`.

À défaut, il est recommandé de nommer les fonctions importées (`from m import f1, f2, ...`) et d'éviter l'importation globale (`from m import *`).

Pour les variables globales, la meilleure approche est de leur donner des noms explicites, par exemple qui commencent par `g_`.

10.3.12 Autres erreurs fréquentes

Chaque langage de programmation a ses propres particularités qui peuvent être source d'erreurs. Pour le langage Python, on peut citer celles-ci :

- erreur dans l'indentation du code ou mélange entre indentation par des espaces ou des caractères de tabulation ;
- oubli de la valeur de retour dans une instruction `return` ou surtout de l'instruction `return` elle-même (sans instruction `return`, la fonction renvoie `None`)
- oubli du paramètre `self` comme premier paramètre dans la définition d'une méthode.
⚠ L'erreur ne se manifestera que lors de l'appel de la méthode !
- oubli de la notation pointée `self.x` pour référencer un attribut `x` de l'objet courant dans une méthode. L'utilisation de `x = 0` ne provoquera pas d'erreur car `x` sera considéré comme variable locale, mais l'effet sera bien sûr différent de `self.x = 0` .

Exercice 10.9

Un test unitaire consiste à vérifier qu'une unité de programme, par exemple une fonction, se comporte comme prévu. On distingue les tests "boîte noire", définis à partir de la seule spécification de l'unité de programme, des tests "boîte blanche", conçus à partir de son code.

On veut ici définir une fonction qui retourne le deuxième plus grand élément d'un tableau de nombres. Par exemple dans le tableau `[4, 2, 7, 5, 9, 1, 5, 3]`, la valeur cherchée est 7. On veut s'assurer que la fonction est codée correctement en créant un jeu de tests.

1. Écrire la spécification de la fonction demandée.
2. Écrire un ensemble de tests "boîte noire" de cette fonction.
3. Écrire le code de la fonction selon l'approche suivante :
 - a) écrire d'abord une fonction qui calcule la valeur maximale d'un tableau

- b) puis une fonction qui calcule la plus grande valeur d'un tableau inférieure à une valeur donnée
 - c) utiliser ces deux fonctions pour écrire une troisième fonction qui calcule la valeur demandée ;
4. Écrire le code de la fonction qui calcule directement la valeur demandée.
 5. Lancer les tests et corriger les erreurs éventuelles. Il pourra être utile de créer des tests "boîte blanche" pour détecter ces erreurs...