Chapitre 8

programmation

Initiation à la programmation défensive

« Attrapez-les toutes ! »

À retenir

La *programmation défensive* est un ensemble de règles pour concevoir des programmes assurés de fonctionner dans le pire des cas.
Les principales sont

- ne pas faire confiance aux données entrées par les utilisateurs ;
- gérer les exceptions avec `try` / `except` ;
- écrire des tests avec `assert` ;
- ne pas chercher à réinventer la roue : utiliser des modules qui ont fait leurs preuves.

I Gestion des exceptions

1 L'utilisateur et ses facéties

On a malheureusement déjà vu ceci :

Code Python

```
def inverse(x : float) -> float :
^^Ireturn 1 / x
```

L'utilisateur évalue `inverse(0)` et obtient :

```
Traceback (most recent call last):
File "fonction1.py", line 4, in <module>
inverse(0)
```

```
File "fonction1.py", line 2, in inverse
return 1 / x
ZeroDivisionError: division by zero
```

Le message d'erreur peut se lire ainsi : Il y a une erreur quand on appelle `inverse(0)` car lorsqu'on doit renvoyer `1/x` cela provoque une erreur de type `ZeroDivisionError`.

De la même manière lorsque l'utilisateur évalue `inverse('chaussette')` il obtient :

```
Traceback (most recent call last):
File "fonction1.py", line 4, in <module>
inverse(0)
File "fonction1.py", line 2, in inverse
return 1 / x
TypeError: unsupported operand type(s) for /: 'int' and 'str'
```

Effectivement l'opérateur `/` ne peut pas diviser un `int` par un `str`.

`ZeroDivisionError` et `TypeError` sont deux représentants de ce qu'on appelle des *exceptions*. On dit qu'*une exception est levée* (*an exception is raised* en Anglais) lorsque l'interpréteur PYTHON rencontre un problème qu'il ne peut résoudre ou bien que le programme lui même indique que ce doit être le cas. Les exceptions les plus courantes sont ces deux premières ainsi que

- `NameError` pour une variable non définie ;
- `IndexError` pour un indice de liste trop grand ;
- `KeyError` pour une clé de dictionnaire inexistante ;

On pourra trouver la liste de toutes les exceptions ici.

Définition : exception

Une exception est un objet de PYTHON servant à représenter une erreur.

2 Syntaxe de la gestion des exceptions

Ne pas faire confiance à l'utilisateur c'est prévoir ces exceptions. La syntaxe PYTHON est simple :

- la partie du code susceptible de lever une exception est mise dans un bloc `try` ;
- si une exception est levée on la gère avec un bloc `except`.

Code Python

```
def inverse(x: float) -> float:
    try:
        return 1 / x
    except ZeroDivisionError:
        print('Erreur - 1 /', x, ': division par zéro.')
    except TypeError:
        print('Erreur - 1 /', x, ': type incorrect.')
    return 0
```

Dans l'exemple ci-dessus les erreurs de type et de division par zéro sont gérées par un message d'affichage et le renvoi de la valeur zéro (zéro n'est l'inverse d'aucun nombre).

Remarque

La gestion des erreurs permet d'éviter que le programme s'arrête brusquement mais elle n'élimine pas les erreurs, elle fait en sorte de les signaler à l'utilisateur du programme (ou du module). C'est donc à lui de rectifier ce qui produit l'erreur.

Conseil

Quand on gère les exceptions, on doit toujours le faire en précisant le type de l'exception qui a été levée. Le programme suivant fonctionne mais n'est pas recommandé car lorsqu'une exception a lieu, on ne sait pas laquelle.

```
def inverse(x: float) -> float:
    try:
        return 1 / x
    except:
        print('Erreur avec 1 /', x)
        return 0
```

II Tests unitaires

Définition : test unitaire

Un (ou des) test(s) unitaire(s) sert à vérifier qu'une partie d'un programme (une *unité*) fonctionne comme on l'a prévu.

Conseil

Lorsqu'on écrit une partie d'un programme destinée à effectuer une tâche particulière, il est préférable d'écrire quelques tests *avant même d'écrire le programme*.
Les tests doivent être pensés pour aborder le cas général ainsi que les cas particuliers.
Cette démarche aide à clarifier les idées pour programmer correctement.

1 Utilisation de `assert`

Le mot-clé **`assert`** sert à vérifier qu'une *assertion* (un test de condition) est vraie. Si c'est le cas le programme continue, sinon une exception du type **`AssertionError`** est levée. Ainsi on peut la gérer avec une structure **`try`** / **`except`** comme vu précédemment.

Code Python

```
def inverse(x: float) -> float:
    try:
        assert x != 0
    except AssertionError:
        print('x ne doit pas être nul !')
        return 0
    return 1 / x
```

2 Un exemple

On veut coder une fonction **`sort`** (qui signifie *trier* en Anglais) qui

- en entrée prend une liste d'`int` ;
- en sortie renvoie une liste qui contient les mêmes valeurs que la liste d'entrée, mais triées dans l'ordre croissant.

La stratégie conseillée est d'écrire les tests d'abord :

Code Python

```
def sort(l: list) -> list:
    return []


assert sort([]) == []
assert sort([0, 2, 3, 1]) == [0, 1, 2, 3]
assert sort([1, 0, 1, 2]) == [0, 1, 1, 2]
```

Le premier test vérifie qu'il n'y a pas d'erreur avec une liste vide (cas particulier qui peut arriver). Le deuxième vérifie qu'il y a effectivement tri, le troisième vérifie que `sort` fonctionne sur une liste avec doublons.

Attention

En mathématiques, on sait que « quelques exemples ne peuvent servir à prouver une généralité ». De la même manière quelques tests ne constituent pas une *preuve absolue* qu'un programme fonctionne correctement.

III Utiliser des modules déjà écrits

C'est ce que l'on veut dire par « ne pas chercher à réinventer la roue » : quand on doit écrire un programme, il est plus que probable que les fonctionnalités qu'on veut implémenter l'aient déjà été par quelqu'un d'autre, que la communauté l'utilise et en est satisfaite. Si c'est le cas, autant l'utiliser, à moins que

- vous-même ne soyez pas satisfaits des performances du module (trop lent, trop gourmand en mémoire) ;
- une situation technique particulière fait que vous ne pouvez pas l'utiliser dans votre projet ;
- vous êtes élève/étudiant et votre professeur·e vous a donné cet exercice/projet pour vous former.

Exercices

Exercice 1

On veut coder une fonction `age` qui demande à l'utilisateur majeur d'entrer son âge, on attend donc qu'il entre un entier entre 18 et 120.

- la fonction ne prend rien en entrée ;
- elle renvoie un `int` compris entre 18 et 120, et tant que l'utilisateur n'entre pas un âge valide elle lui demande d'entrer son âge.

Programmer cette fonction de manière défensive (gérer les exceptions).

Exercice 2

On veut coder une fonction `is_sorted` qui

- en entrée prend une liste d'`int` éventuellement vide ;
- renvoie `True` si elle est triée par ordre croissant (ou vide ou bien avec un seul élément) et `False` dans le cas contraire.

Écrire un ensemble de tests pertinents pour cette fonction.

Exercice 3

Reprendre le module `fractions_custom` du chapitre précédent et gérer toutes les exceptions :

- l'utilisateur crée n'importe quoi ;
- l'utilisateur divise par zéro.

On pourra utiliser la fonction `isinstance()` qui

- prend en entrée une variable et un type de variable ;
- renvoie `True` si la variable est bien ce ce type et `False` sinon.
  Par exemple `isinstance(2,str)` renvoie `False`.

Exercice 4

On veut créer une fonction `merge` (qui veut dire *fusionner* en Anglais) qui

- prend en entrée 2 listes d'entiers triées par ordre croissant éventuellement vides ;
- renvoie une liste composée des valeurs des 2 listes, triées par ordre croissant. Par exemple `merge([1, 3, 8], [4, 5, 10])` doit renvoyer `[1, 3, 4, 5, 8, 10]`.

1. Écrire des tests pertinents pour cette fonction.
2. **Compliqué :** programmer cette fonction. Le plus simple est de la programmer de manière récursive. On peut la programmer de manière impérative mais c'est plus long et sans doute plus compliqué.

Projet optionnel : module `kb_mouse`

Reprendre le module `kb_mouse` du chapitre précédent et gérer les exceptions.