

Programmation défensive

Chapitre 11

NSI2

8 novembre 2023

Ce n'est pas ça...



Gestion des exceptions

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

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

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
```

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

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'
```


`ZeroDivisionError` et `TypeError` sont deux représentants de ce qu'on appelle des *exceptions*.

`ZeroDivisionError` et `TypeError` sont deux représentants de ce qu'on appelle des *exceptions*.

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 :

<https://docs.python.org/fr/3.8/library/exceptions.html>.

Syntaxe de la gestion des exceptions

- la partie du code susceptible de lever une exception est mise dans un bloc `try`;

- 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**.

Exemple

```
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
```

Les **try** .. **except** n'éliminent pas les erreurs mais permettent de les « intercepter ».

Toujours préciser le type de l'exception qui a été levée sinon ce n'est pas clair.

Toujours préciser le type de l'exception qui a été levée sinon ce n'est pas clair.

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

Toujours préciser le type de l'exception qui a été levée sinon ce n'est pas clair.

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

Dans le bloc **except** on ne sait pas quelle erreur est survenue.

Tests unitaires

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

Écrire quelques tests *avant même d'écrire le programme*.

Les tests doivent être pensés pour aborder le cas général ainsi que les cas particuliers.

`assert` sert à vérifier qu'une *assertion* (un test de condition) est vraie.

`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.

Exemple

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.

On écrit d'abord les tests

On écrit d'abord les tests

```
def sort(l: list) -> list:  
    ...
```

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

Une batterie de tests ne constitue pas une preuve.

Différence `try ... except`
et
`assert`.

`try ... except` pour détecter et contrôler les erreurs.

`try ... except` pour détecter et contrôler les erreurs.

`assert` pour vérifier conditions, résultats.

`try ... except` pour détecter et contrôler les erreurs.

`assert` pour vérifier conditions, résultats.

Les premières ont vocation à rester dans le programme, les secondes non.

Utiliser des modules déjà écrits

Parce que c'est crétin
de chercher à
réinventer la roue.

sauf...

- pas satisfait des performances du module (trop lent, trop gourmand en mémoire);

- pas satisfait 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;

- pas satisfait 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;
- le prof le demande, pour se former!