Chapitre 16 - Gestion des erreurs

La phase de débogage d'un programme est très gourmande en temps. Ceci est particulièrement vrai avec python qui est un langage dit non typé.

1- ERREURS DE SYNTAXE :

En écrivant un code, il se glisse souvent des erreurs de syntaxe. Elles empêchent l'interpréteur de comprendre le code et provoquent une erreur avant même son exécution. Ces erreurs sont faciles à trouver :

- Parenthèses, crochets, ou guillemets mal fermés (SyntaxError)
- Mauvaise indentation (IdentationError)
- Oubli des : après un if (Invalid Syntax)

2- ERREURS A L'EXECUTION :

Lorsqu'une erreur apparait à l'exécution, l'interpréteur Python **LEVE** ce que l'on appelle une **EXCEPTION**. Si cette exception n'est pas **CAPTUREE**, le code s'arrête et un message d'erreur apparait dans ce que l'on appelle le **TRACEBACK**.

Les exceptions levées à l'exécution sont plus difficiles à trouver, car elles nécessitent de comprendre l'exécution du code. Elles sont aussi plus variées :

- NameError : un nom de variable a-t-il été mal orthographié ?
- IndexError : l'indice utilisé est-il en dehors de la liste ?
- TypeError : a-t-on essayé d'ajouter un nombre à une chaine de caractère ?

Outre le type d'exception et la ligne l'ayant levée, le traceback contient un historique des appels de fonctions (la pile d'appel) permettant de connaître le contexte d'exécution. La recherche d'une erreur s'apparente alors à une enquête : depuis l'endroit où l'erreur s'est déclarée, on remonte le fil d'exécution pour en déterminer la cause. Le traceback se lit de bas en haut.

coursErreurs.py

```
1  def inverse(x) :
2      return 1/x
3
4  def somme(n):
5      s = 0
6      for i in range(n) :
7          s = s + inverse(i)
8      return s
9  # Main
10 somme(4)
```

```
Traceback (most recent call last):
File "C:\Users\coursErreurs.py", line 10, in <module>
somme(4)
File "C:\Users\\coursErreurs.py", line 7, in somme
s = s + inverse(i)
File "C:\Users\ coursErreurs.py", line 2, in inverse
return 1/x
ZeroDivisionError: division by zero
```

Dans l'exécution ci-dessus, le programme a échoué à cause d'une exception de type ***ZeroDivisionError*** et nous avons la pile d'erreur qui nous indique que le programme s'est interrompu à la ligne 2 suite à l'instruction 1 / x dans la fonction inverse() qui a été appelée par la fonction somme() à la ligne 7, elle-même appelée par le programme à la ligne 10.

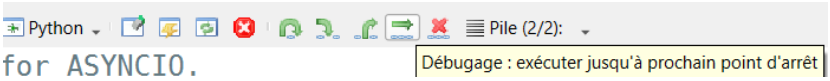
3- DEBOGUEUR:

Un débogueur permet de dérouler un programme pas à pas et de vérifier l'état des variables.

> Un cas de bug devenu célèbre en 1947 : « *Il s'agissait d'un insecte (bug en anglais) ayant provoqué des erreurs de calcul dans un ordinateur Mark II* ».

coursErreurs.py

```
1  def inverse(x) :
2      return 1/x
3
4  def somme(n):
5      s = 0
6      for i in range(n) :
7          s = s + inverse(i)
8      return s
9  # Main
10 somme(4)
```



Environnement

Nom	Type	Repr
i	int	0
inverse	function	<function inverse at 0x00000258593C59D0>
n	int	4
s	int	0
somme	function	<function somme at 0x00000258593C5C10>

Site intéressant : ***https://pythontutor.com/***

4- GERER LES EXCEPTIONS AVEC LA STRUCTURE TRY .. EXCEPT .. ELSE .. FINALLY :

La structure « *try .. except .. else .. finally* » permet de gérer le déroulement d'une exécution lorsqu'une exception est levée.

Par exemple, le code suivant entraine rapidement un levée d'exception et un arrêt de l'exécution :

```
def inverser(l,n)  :
    """
        l : liste de valeurs quelconques
        n : longueur de la liste
        Inverse les éléments de la liste en paramètre
    """
    for i in range(n) :
        l[i] = 1 / l[i]

# Main
l = [1,0,3,4,'NSI']
inverser(l,6)
print(l)
```

```
>>> (executing file "coursErreurs.py")
Traceback (most recent call last):
File "C:\Users \coursErreurs.py", line 35, in <module>
inverser(l,6)
File "C:\Users\coursErreurs.py", line 10, in inverser
l[i] = 1 / l[i]
ZeroDivisionError: division by zero
```

```python
def inverser(l,n)  :
    """
        l : liste de valeurs quelconques
        n : longueur de la liste
        Inverse les éléments de la liste en paramètre
    """
    for i in range(n) :
        try :
            l[i] = 1 / l[i]
        except ZeroDivisionError as e :
            print(f"i = {i} , {e}")
            print(type(e),e)
        except TypeError as e :
            print(f"i = {i}  , {e} ")
        except IndexError as e:
            print(f"i = {i} , {e}")
        else :
            l[i] = l[i] + 100
        finally :
            print(f"L'indice {i} à été traité")

# Main
l = [1,0,3,4,'NSI']
inverser(l,6)
print(l)
```

En intégrant à ce code une structure *try .. except*, on peut éviter l'arrêt de l'exécution, en gérant chacune des exceptions individuellement :

```
>>> (executing file "coursErreurs.py")
L'indice 0 à été traité
i = 1 , division by zero
<class 'ZeroDivisionError'> division by zero
L'indice 1 à été traité
L'indice 2 à été traité
L'indice 3 à été traité
i = 4  , unsupported operand type(s) for /: 'int' and 'str'
L'indice 4 à été traité
i = 5 , list index out of range
L'indice 5 à été traité
[101.0, 0, 100.33333333333333, 100.25, 'NSI']
```

5- LEVER UNE EXCEPTION :

Il est possible de créer soi-même une nouvelle exception à l'aide du mot clé raise.

Cas de l'exception crée et non gérée ensuite :

```python
def age(anneeNaissance) :
    if anneeNaissance > 2025 :
        raise ValueError("n doit être inférieur à 2026")
    return 2025 - anneeNaissance

a = age(2050)
print(f"age en 2025 : {a} ans")
```

```
>>> (executing file "coursErreurs.py")
Traceback (most recent call last):
File "C:\Users \coursErreurs.py", line 63, in <module>
a = age(2050)
File "C:\Users \coursErreurs.py", line 60, in age
raise ValueError("n doit être inférieur à 2026")
ValueError: n doit être inférieur à 2026
```

L'exécution de la commande ***raise*** entraine l'arrêt de l'exécution car cette exception n'est pas gérée.

Avec le code ci-dessous, l'exception est gérée et n'entraine pas d'arrêt du code :

```
def age(anneeNaissance) :
    if anneeNaissance > 2025 :
        raise ValueError("n doit être inférieur à 2026")
    return 2025 - anneeNaissance

try :
    a = age(2050)
    print(f"age en 2025 : {a} ans")
except ValueError as e :
    print(e)
```

```
>>> (executing file "coursErreurs.py")
n doit être inférieur à 2026
```

6- INTEGRER DES TESTS DANS SON CODE EN INTEGRANT DES ASSERTIONS :

Une assertion échoue (exception AssertionError) si l'expression booléenne qui suit le mot clé **assert** est fausse. Dans ce cas l'interpréteur Python lève une exception. L'exécution s'arrête alors si l'exception n'est pas gérée. Si l'assertion est vraie, l'exécution continue sans erreur.

On peut insérer des assertions dans une fonction pour tester les valeurs des paramètres (test de *précondition*) ou après exécution d'une fonction pour tester les valeurs retournées (test de *postcondition*).

```
def racineCarree(x):
    assert x >= 0 , "le nombre doit être positif"
    return x/2

assert racineCarree(9) == 3 , "Test racineCarre(9) à échoué"
```

```
>>> (executing file "coursErreurs.py")
Traceback (most recent call last):
File "C:\Users \coursErreurs.py", line 7, in <module>
assert racineCarree(9) == 3 , "Test racineCarre(9) à échoué"
AssertionError: Test racineCarre(9) à échoué
```

7- INTEGRER UNE DOCUMENTATION DANS SON CODE :

Un docstring permet de renseigner l'utilisateur sur son domaine de validité de la fonction :

```
def age(anneeNaissance) :
    """
        anneeNaissance : entier inférieur à 2026
        fonction qui renvoie un entier qui correspond à l'age
    """
    if anneeNaissance > 2025 :
        raise ValueError("n doit être inférieur à 2026")
    return 2025 - anneeNaissance
```

```
>>> help(age)
Help on function age in module __main__:

age(anneeNaissance)
    anneeNaissance : entier inférieur à 2026
    fonction qui renvoie un entier qui correspond à l'age
```