Messages d'erreur

I. Introduction

Programmer, c'est résoudre des problèmes... mais aussi en rencontrer ! Il est tout à fait normal de faire des erreurs, que l'on débute ou que l'on soit expérimenté. L'essentiel, ce n'est pas de ne jamais se tromper, mais d'apprendre à identifier les erreurs, comprendre leur origine et savoir les corriger. C'est une compétence fondamentale en informatique, qui fait partie du travail du programmeur.

Les messages d'erreur générés par Python sont là pour nous aider dans cette démarche. Plutôt que de paniquer à leur apparition, il faut apprendre à les lire et les interpréter : ils donnent des indications précieuses sur ce qui s'est mal passé et à quel endroit. Avec un peu d'entrainement, nous apprendrons à les comprendre et les corriger rapidement, ce qui nous rendra plus autonome.

Un message d'erreur Python se compose de deux parties :

- ⋆ **Traceback** : où et comment l'erreur s'est produite.
- ⋆ **Type d'erreur** (dernière ligne du message) : nom de l'erreur et message explicatif.

Bonnes pratiques pour comprendre une erreur

Le meilleur conseil qu'on puisse donner pour comprendre un message d'erreur et le corriger rapidement est de lire **en premier** (et souvent uniquement) la dernière ligne qui concerne le type de l'erreur et donc d'**ignorer** la Traceback et aussi **les numéros des lignes concernées** !
En effet :

- ⋆ il est indispensable de savoir ce qu'on cherche avant d'essayer de le trouver !
- ⋆ les lignes données dans le message sont souvent trompeuses : ce sont les lignes où se produisent l'erreur mais ce ne sont pas toujours les lignes qui contiennent l'erreur (c'est-à-dire les lignes où il faut effectuer une correction).

II. Analyse détaillée d'exemples simples

1) Exemple 1

Considérons ce programme :

```
1 def diviser(a, b):
2     return a / b
3
4 x = diviser(10, 0)
```

Après exécution, ce code produit l'erreur suivante :

```
Traceback (most recent call last):
  File "exemple.py", line 4, in <module>
    x = diviser(10, 0)
  File "exemple.py", line 2, in diviser
    return a / b
ZeroDivisionError: division by zero
```

Les 5 premières lignes du message d'erreur concernent la localisation de l'erreur et la dernière ligne seulement concerne le type de l'erreur.

On lit donc d'abord (et souvent uniquement) la dernière ligne qui nous donne :

- ⋆ le type de l'erreur : `ZeroDivisionError`
- ⋆ un message explicatif (en anglais bien sûr) : `division by zero`

Ces informations sont très souvent suffisantes pour retrouver l'erreur et la corriger.

Si nécessaire, on peut également étudier la Traceback qui nous donne la liste des fichiers et fonctions concernées par l'erreur produite (du plus récent au plus ancien). Dans notre exemple, on a deux parties :

⋆ `File "exemple.py", line 4, in <module>` :

- Fichier `"exemple.py"`
- Ligne `4`
- Fonction `<module>` : ce qui signifie en dehors de toute fonction

La ligne `x = diviser(10, 0)` a produit une erreur. Comme cette ligne fait appel à une fonction, on remonte le fil d'exécution pour localiser l'erreur dans la fonction ce qui est fait dans la deuxième partie de la Traceback.

⋆ `File "exemple.py", line 2, in diviser` :

- Fichier `"exemple.py"`
- Ligne `2`
- Fonction `diviser`

La ligne `return a / b` a produit une erreur.

2) Exemple 2

Considérons ce programme :

```
1  from math import sqrt
2
3  def discriminant(a, b, c):
4      delta = b**2 - 4 * a * c
5      return delta
6
7  def solutions(a, b, c):
8      delta = discriminant(a, b, c)
9
10     if delta > 0:
11         x1 = (-b - sqrt(delta))/(2 * a)
12         x2 = (-b + sqrt(delta))/(2 * a)
13         return [x1, x2]
14     elif delta == 0:
15         x0 = -b / (2 * a)
16         return [x0]
17     else:
18         return []
19
```

```
20 a = input("a = ")
21 b = input("b = ")
22 c = input("c = ")
23
24 S = solutions(a, b, c)
25 print(f"Les solutions de l'équation sont S = {S}")
```

Après exécution et réponses aux `input`, ce code produit l'erreur suivante :

```
a = 1
b = 3
c = 2
Traceback (most recent call last):
  File "exemple.py", line 24, in <module>
    S = solutions(a, b, c)
  File "exemple.py", line 8, in solutions
    delta = discriminant(a, b, c)
  File "exemple.py", line 4, in discriminant
    delta = b**2 - 4 * a * c
            ~^^~
TypeError: unsupported operand type(s) for ** or pow(): 'str' and 'int'
```

Ici, la Traceback nous donne 3 numéros de lignes : 4, 8 et 24. Si vous êtes assez attentifs vous comprendrez qu'**aucune de ces 3 lignes ne comporte d'erreur !**

En lisant la dernière ligne du message, on comprend qu'on a affaire à une `TypeError` (erreur de type) et qu'on a essayé de calculer une puissance entre un objet de type `str` et un objet de type `int`.

La seule puissance qu'on a calculée étant à la ligne 4 avec `b**2`, on comprend donc que `b` est une chaine de caractères au lieu d'être un nombre ! Effectivement `b` est défini à la ligne 21 par `b = input("b = ")` et la fonction `input` renvoie toujours une chaine de caractères.

On corrige donc cette ligne par `b = float(input("b = "))` et, par la même occasion, les lignes 20 et 22 qui comportent la même erreur (mais qui ne s'est pas produite) lors de l'exécution.

Dans cet exemple, on a donc corrigé les lignes 20, 21 et 22 alors que les lignes données dans le message d'erreur étaient 4, 8 et 24 !

III. Différents types d'erreurs et leurs origines probables

Étudions maintenant un par un les principaux types d'erreurs que l'on peut rencontrer et leurs origines les plus probables.

1) `SyntaxError`

C'est l'erreur la plus fréquente mais aussi celle qui a le plus d'origines possibles.

Signification : Python ne comprend pas le code car il ne respecte pas la syntaxe du langage.

Causes fréquentes :

- ⋆ Oubli d'une parenthèse fermante (de loin le plus fréquent).
- ⋆ Oubli des `:` à la fin de la première ligne d'une structure de contrôle (`if`, `elif`, `else`, `for`, `while`, `def`).
- ⋆ Syntaxe d'affectation inversée (`valeur = variable` au lieu de `variable = valeur`).
- ⋆ Oubli du symbole `*` dans une multiplication (`2x` au lieu de `2 * x`).

- ⋆ Nom de variable invalide.
- ⋆ Définition d'une chaine de caractères incorrecte (oubli de fermer les délimiteurs, mauvais choix entre `"` et `'`, etc).

2) `NameError`

Signification : Utilisation d'une variable ou d'une fonction qui n'a pas été défini.

Causes fréquentes :

- ⋆ Utilisation d'une variable avant sa définition notamment à cause d'un oubli d'initialiser la variable (`x += 1` sans initialisation).
- ⋆ Erreur de typo dans le nom d'une variable ou d'une fonction qui fait qu'on l'écrit deux fois de deux façons différentes.
- ⋆ Oubli d'exécuter le code de définition d'une fonction avant de l'utiliser ou exécution du code produisant une erreur que l'on ignore : si on exécute une fonction et qu'il y a une erreur, la fonction est inconnue de Python.

3) `TypeError`

Signification : Opération entre deux variables dont les types sont incompatibles ou fonction qui a reçu un paramètre dont le type est invalide.

Causes fréquentes :

- ⋆ Utilisation d'une variable de type `str` comme un nombre notamment à cause d'un oubli de conversion en une variable numérique après un `input`.
- ⋆ Appel d'une fonction sur le mauvais type d'argument.
- ⋆ Variable `a` ou `b` non entières dans la syntaxe `range(a, b)`.
- ⋆ Tentative de modification d'un caractère d'une chaine de caractères (`chaine[i] = "a"`).

4) `ValueError`

Signification : Une fonction native de Python ou issue d'un module importé a reçu un paramètre dont la valeur est invalide.

Causes fréquentes :

- ⋆ Valeur négative dans la fonction `sqrt` du module `math`.
- ⋆ Conversion de type impossible (`int("abc")`).
- ⋆ etc.

5) `IndentationError`

Signification : Le code n'est pas correctement indenté ou Python ne comprend pas la structure de blocs.

Causes unique : Indentation incorrecte avec une ou plusieurs structures de contrôle (`if`, `elif`, `else`, `for`, `while`, `def`). Souvent c'est lié à un oubli initial des `:` que l'on a corrigé ensuite sans corriger l'indentation (lorsqu'on écrit les `:` l'indentation se fait automatiquement).

6) `IndexError`

Signification : Accès à un élément d'une liste avec un indice invalide (trop petit ou trop grand).

Causes fréquentes :

- ⋆ Dans une boucle du type `for i in range(len(liste)):` qui contient un accès du type `liste[i+1]`.

7) `KeyError`

Signification : Accès à un élément d'un dictionnaire avec une clé inexistante.

Causes fréquentes :

- ⋆ Erreur de typo dans l'écriture de la clé.

8) `AttributeError`

Signification : Application d'une méthode à une variable qui n'a pas le bon type.

Causes fréquentes :

- ⋆ Utilisation d'une méthode sur les listes (`append`, `pop`, ...) à une variable de type différent (`int`, `str`, ...).
- ⋆ Erreur de typo dans le nom d'une méthode (`L.apend(3)`).