Analyse du traceback et erreurs courantes

Laurent Signac – CC-BY-SA – 01-07-22 0938 7c53cb3991c2ae30c171

There are two ways to write error-free programs; only the third one works. – Alan Perlis

1 Analyse du traceback

Le traceback est l'ensemble des lignes affichées par l'interpréteur Python lorsqu'une exception est levée. Comprendre le traceback est **très utile** pour trouver rapidement une faute et l'expliquer.

Voici un exemple de traceback affiché lors de l'exécution d'un code un peu obscur :

```
1  def truc(c):
2      a = [1, 2, 3]
3      a[c] = (1 / c)
4
5  def machin(a, b):
6      d = min(a, b) * 2
7      truc(d)
8
9  for i in range(0, 5):
10     machin(5 - i, i)
```

Le traceback est :

```
Traceback (most recent call last):
  File "/home/signac/obscur.py", line 10, in <module>
    machin(5 - i, i)
  File "/home/signac/obscur.py", line 7, in machin
    truc(d)
  File "/home/signac/obscur.py", line 3, in truc
    a[c] = (1 / c)
ZeroDivisionError: division by zero
```

Ce traceback est constitué d'une liste de lignes en défaut (il y en a 3 ici), puis d'un message qui précise le type d'exception (c'est la dernière ligne).

En lisant le traceback de haut en bas, on avance dans le temps. Ce qui s'est produit en dernier est donc **à la fin** du traceback.

En le lisant de haut en bas, on note (notez la correspondance entre ce qui est indiqué ci-dessous et le contenu du traceback) :

- exécution de la ligne 10 «`machin(5 - i, i)`», donc on entre dans la fonction `machin` qui a provoqué
- l'exécution de la ligne 7 «`truc(d)`», donc on entre dans la fonction `truc` qui a provoqué
- l'exécution de la ligne 3 «`a[c] = (1 / c)`» qui a provoqué
- la levée d'exception `ZeroDivisionError: division by zero`

La plupart du temps, la lecture des **trois dernières lignes** du traceback permet de cerner le problème :

```
File "/home/signac/obscur.py", line 3, in truc
    a[c] = (1 / c)
ZeroDivisionError: division by zero
```

ce qui signifie : Exécution de la ligne 3 «`a[c] = (1 / c)`» qui a provoqué l'exception `ZeroDivisionError`.

La toute dernière ligne du traceback est le message associé à l'exception qui a été levée. Ce message indique quel genre d'erreur s'est produite. Est-ce que c'est un problème d'indentation, de noms de variable non défini ou autre ? Ici, il s'agit d'une division par 0.

Les deux lignes précédentes indiquent à quel moment de l'exécution cette exception a été levée. Ici, en calculant «`a[c] = (1 / c)`».

Il est donc très facile de deviner que le problème est dû au calcul de `1 / c` (c'est la seule division de la ligne), alors que `c` est nul.

Cette analyse (les trois dernières lignes du traceback) peut être suffisante pour corriger l'erreur ou non.

Si on a besoin de plus de contexte, on remonte dans le traceback, pour savoir **pourquoi** on est arrivé jusqu'à la ligne «`a[c] = (1 / c)`» avec une valeur de `c` manifestement nulle.

À partir de maintenant, lorsque vous obtenez une erreur lors de l'exécution de code Python, vous devez (dans cet ordre) :

1. regarder la dernière ligne du traceback pour connaître la nature de l'exception
2. regarder les deux lignes qui précèdent la dernière pour avoir le numéro de ligne et son contenu
3. et ensuite uniquement, regarder votre code, à l'endroit indiqué, pour comprendre la faute

Pour corriger la faute, se contenter de ces trois points peut suffire, ou il peut être nécessaire de remonter dans le traceback. Si on veut par exemple, empêcher que `c` soit nul, c'est peut-être la ligne 9 qu'il faut corriger :

```
1  def truc(c):
2      a = [1, 2, 3]
3      a[c] = (1 / c)
4
5  def machin(a, b):
6      d = min(a, b) * 2
7      truc(d)
8
9  for i in range(1, 5):
10     machin(5 - i, i)
```

Cette fois-ci, l'exécution donne :

```
Traceback (most recent call last):
  File "/home/signac/obscur.py", line 10, in <module>
    machin(5 - i, i)
  File "/home/signac/obscur.py", line 7, in machin
    truc(d)
  File "/home/signac/obscur.py", line 3, in truc
    a[c] = (1 / c)
IndexError: list assignment index out of range
```

Le message d'exception n'est plus le même. Il y a une autre faute. Peut-être que cette faute a été ajoutée par notre modification, ou peut-être que cette faute était déjà présente, mais qu'auparavant, nous ne l'avions pas atteinte, le programme ayant stoppé son exécution avant : la correction effectuée a peut-être permis d'aller plus loin dans l'exécution.

Saurez-vous répondre à ces questions :

- est-ce que la nouvelle faute a été ajoutée ou bien était-elle déjà présente ?
- quelle ligne de code a provoqué la nouvelle exception ?
- quelle est la raison immédiate de la levée d'exception ?
- quelle était la valeur de `i` lorsque l'exception a été levée ?

- il est peu probable que la ligne qui a provoqué la levée de l'exception soit celle à corriger. Vous pouvez proposer des corrections minimales (tout effacer, par exemple, est une correction qui enlève généralement les erreurs… mais ne présente pas un grand intérêt par ailleurs) qui permettent de ne plus avoir ce message (même si ici, ça n'a pas trop de sens puisqu'on ne sait pas à quoi peut servir cette portion de code)

Depuis Python 3.11, le positionnement de l'exception est mieux décrit dans le Traceback. Certaines améliorations sont déjà présentes dans la version 3.10 (messages plus explicites par exemple).
Pour une erreur de ce type :

```
>>> x = {'a': 1, 'b': {'c': None}}
>>> v = x['b']['c'][4]
TypeError: 'NoneType' object is not subscriptable
```

Le traceback avant Python 3.11 ne permettait pas de savoir si la valeur à `None` était `x`, `x['b']` ou `x['b']['c']`.
Depuis Python 3.11, la portion de la ligne en défaut est indiquée par des `^` :

```
v = x['b']['c'][4]
    ~~~~~~^^^^^
TypeError: 'NoneType' object is not subscriptable
```

2 Erreurs courantes

Certaines erreurs sont détectées à la lecture du code, et d'autres à l'exécution. Les erreurs de syntaxe et d'indentation sont détectées à la lecture du code. Il n'y a donc pas de traceback. Les erreurs de nom ou de type (et les autres) sont détectées à l'exécution. Il y a un traceback. Ça signifie aussi que ces erreurs peuvent passer inaperçues lors d'une exécution (si le code concerné n'est pas exécuté ou que les valeurs en entrée ne conduisent pas à l'erreur).

2.1 `SyntaxError`

Les causes sont multiples :

- Oubli ou mauvais placement d'un délimiteur : parenthèse, crochet, guillemet

- Oubli des `:` pour annoncer un bloc. Certaines instructions (`if`, `elif`, `else`, `for`, `def`, `while` pour les plus courantes) sont suivies d'un bloc de code. Elles doivent être suivies de `:` puis de lignes indentées délimitant le bloc.
- Caractère manquant ou qui ne peut pas se trouver là (un chiffre en début de nom de variable, un caractère spécial comme `!` à un endroit impossible…)

Certaines instructions peuvent courir sur plusieurs lignes. Il n'est donc **pas rare qu'une erreur de syntaxe soit détectée et donc signalée sur la ligne qui suit l'endroit où elle se trouve réellement**.

Exemples :

```
1 lst = [2, 3, 4]
2 a = lst[0] * (6 + 2
3 b = 5
```

```
  File "<tmp 1>", line 3
    b = 5
    ^
SyntaxError: invalid syntax
```

```
a = 5
if a == 4
    print("a vaut 4")
```

```
  File "<tmp 1>", line 2
    if a == 4
             ^
SyntaxError: invalid syntax
```

```
a = 3
b = 3a
```

```
  File "<tmp 1>", line 2
    b = 3a
         ^
SyntaxError: invalid syntax
```

```
a = 5!3
```

```
  File "<tmp 1>", line 1
    a = 5!3
         ^
SyntaxError: invalid syntax
```

2.2 IndentationError

Comme indiqué précédemment, certaines instructions sont suivies de `:`, puis d'un bloc indenté. La valeur préconisée pour l'indentation est de 4 espaces, mais elle n'est pas imposée (c'est une mauvaise idée d'indenter avec autre chose que 4 espaces, mais ce n'est pas une faute).

Les erreurs d'indentation peuvent être de 3 types :

- le bloc est indenté sans raison : `unexpected indent`
- pas de bloc indenté alors qu'il devrait y en avoir un : `expected an indented block`
- après un bloc indenté, le niveau d'indentation trouvé ne correspond à aucun bloc englobant : `unindent does not match any outer indentation level`

Une erreur assez courante et pénible à détecter (qui arrive de moins en moins souvent car les éditeur de code la gèrent bien) est de mélanger les indentations avec 4 espaces et avec une tabulation. Ce sont deux niveaux d'indentation différents, alors que visuellement, ils sont similaires sur l'écran.

Exemples :

```
for i in range(10):
print(i)
```

```
  File "<tmp 1>", line 2
    print(i)
    ^
IndentationError: expected an indented block
```

```
a = 5
    b = 6
```

```
  File "<tmp 1>", line 2
    b = 6
IndentationError: unexpected indent
```

```
for i in range(10):
    if i % 3 == 0:
        print("Plouf")
   print(i)
```

```
  File "<tmp 1>", line 4
    print(i)
          ^
IndentationError: unindent does not match any outer indentation level
```

2.3 NameError

Le nom de variable (ou de fonction) n'est pas connu de Python. Ce type d'erreur est souvent dû à une faute de frappe dans un nom, ou à une variable non initialisée.

Exemples :

```
lst = [5, 6, 7]
for v in lst:
    s = s + v
print(s)
```

```
Traceback (most recent call last):
  File "<tmp 1>", line 3, in <module>
    s = s + v
NameError: name 's' is not defined
```

```
liste = [4, 5, 6]
for v in l1ste:
    print(v)
```

```
Traceback (most recent call last):
  File "<tmp 1>", line 2, in <module>
    for v in l1ste:
NameError: name 'l1ste' is not defined
```

```
def pdt_des_elts(lst):
    p = 1
    for v in lst:
        p = p * v
    return p

lst = [1, 2, 3, 4, 5]
print(pdt_des_elements(lst))
```

```
Traceback (most recent call last):
  File "<tmp 1>", line 8, in <module>
    print(pdt_des_elements(lst))
NameError: name 'pdt_des_elements' is not defined
```

Dans l'ordre, les causes des erreurs sont :

- oubli d'initialiser `s` avant la boucle
- mauvaise écriture de `liste` (on a écrit `l1ste` avec un `1`)
- mauvaise écriture du nom de fonction (`pdt_des_elements` au lieu de `pdt_des_elts`)

2.4 `TypeError`

Ce type d'erreur est par exemple obtenu si on essaie de faire une opérations sur des objets dont le type ne convient pas :

- additionner une chaîne et un nombre (ou toute autre erreur similaire)
- indicer une séquence avec autre chose qu'une tranche (*slice*) ou un entier
- passer un nombre incorrect de paramètres à une fonction
- utilisation de parenthèses (*callable*) lorsque ça n'a pas de sens
- utilisation de crochets (*subscriptable*) lorsque ça n'a pas de sens

Exemples :

```
1 a = "une fois sur"
2 b = 2
3 print(a + b)
```

```
Traceback (most recent call last):
  File "<tmp 1>", line 3, in <module>
    print(a + b)
TypeError: can only concatenate str (not "int") to str
```

```
1 a = 10
2 lst = [1, 2, 3, 4, 5]
3 print(lst[a / 2])
```

```
Traceback (most recent call last):
  File "<tmp 3>", line 3, in <module>
    print(lst[a / 2])
TypeError: list indices must be integers or slices, not float
```

⚠ Le problème signalé ci-dessus n'est pas un indice hors limite (il y a un indice hors limite, mais l'exception signalée est levée **avant** que le problème de l'indice hors limite ne se produise).

```
1 lst = [5, 6, 7]
2 print(lst(0))
```

```
Traceback (most recent call last):
  File "<tmp 1>", line 3, in <module>
    print(lst(0))
TypeError: 'list' object is not callable
```

L'interpréteur indique que le type `list` (ici `lst`) n'est pas *callable*. Seuls les objets *callable* (par exemple les fonctions, les classes…) peuvent être suivis de parenthèses.

```
1 def mafonction(n):
2     return n ** 2 + 2 * n + 1
3
4 print(mafonction[5])
```

```
Traceback (most recent call last):
  File "<tmp 1>", line 5, in <module>
    print(mafonction[5])
TypeError: 'function' object is not subscriptable
```

L'interpréteur indique que le type `function` (ici `mafonction`) n'est pas *subscriptable*. Seuls les objets *subscriptables* (par exemple les listes, les tuples, les dictionnaires…) peuvent être suivis d'une paire de crochets.

```
1 import math
2 print(math.pow(3, 4, 5))
```

```
Traceback (most recent call last):
  File "<tmp 3>", line 2, in <module>
    print(math.pow(3, 4, 5))
TypeError: pow expected 2 arguments, got 3
```

Dans le cas de `TypeError`, le message d'erreur (par exemple « `pow expected 2 arguments, got 3` » ci-dessus) permet souvent de mieux circonscrire le problème. Il faut donc le lire et le comprendre.

2.5 IndexError

Cette erreur indique qu'un indice (d'une liste par exemple) est hors limite. On rappelle que les indices des séquences (liste, tuple, chaîne) commencent à 0 et se terminent à `n - 1` si `n` est la longueur de la séquence.

```
1 lst = [1, 2, 3]
2 i = 0
3 while i < 4:
4     print(lst[i])
5     i = i + 1
```

```
Traceback (most recent call last):
  File "<tmp 1>", line 4, in <module>
    print(lst[i])
IndexError: list index out of range
```

C'est par ailleurs une mauvaise idée de parcourir une liste avec une boucle `while` plutôt que `for`.