Traquer les erreurs

Laurent Signac – CC-BY-SA – 01-04-22 2141 ec4f3ecc23b9b6e8af1f

Profitez de vos erreurs. Si vous en faites une, le plus important n'est pas de la corriger, mais de savoir pourquoi la faute dans le code a provoqué ce que vous avez constaté. C'est généralement plus difficile que simplement corriger, mais on apprend plus de choses ainsi.

⚠ Ne laissez jamais une erreur inexpliquée, même si vous avez réussi à la corriger.

La lecture de ce document n'a pas pas d'intérêt si le contenu du document *Analyse du traceback et erreurs courantes* n'est pas acquis. L'accent est ici mis sur les erreurs non triviales, c'est à dire dont la ligne et la cause ne sont pas forcément signalées dans le traceback (si on a la chance d'en avoir un…).

1 Erreurs courante

1.1 `if…if…else` n'est pas une instruction

Deux `if` qui se suivent sont deux instructions séparées (alors qu'un `elif` qui suit un `if` fait partie de la même instruction).

Si on veut mettre l'appréciation `A` pour une note supérieure ou égale à 15, `C` pour une note strictement inférieurs à 10 et `B` sinon, il ne faut pas écrire :

```
if note >= 15:
    appreciation = "A"
if note < 10:
    appreciation = "C"
else:
    appreciation = "B"
```

En écrivant ça, personne ne pourra obtenir l'appréciation `A`, même en ayant 20.

Le code correct est :

```
if note >= 15:
    appreciation = "A"
elif note < 10:
    appreciation = "C"
else:
    appreciation = "B"
```

1.2 `sort` et `sorted`

Certaines fonctionnalités sont disponibles par le biais de méthodes et de fonctions. C'est le cas pour le tri d'une liste.

On peut écrire `lst.sort()` qui **fait muter** la liste `lst` qui se retrouve alors triée (ici, `.sort()` est une méthode appliquée sur un objet de type `list`). L'appel à `lst.sort()` **ne renvoie rien**.

Mais on peut aussi écrire `sorted(lst)`, qui **ne fait pas muter** `lst`, mais **renvoie une copie triée** (ici `sorted` est une fonction qui prend en paramètre un objet de type `list`).

Exemple :

```
# Utilisation de sort
lst = [1, 5, 4, 2, 7]
lst.sort()
print(lst) # Maintenant lst est triée
# Utilisation de sorted
lst = [1, 5, 4, 2, 7]
lst2 = sorted(lst)
print(lst2) # lst2 est triée, mais lst ne l'est pas
# Utilisation de sorted
lst = [1, 5, 4, 2, 7]
lst = sorted(lst)
print(lst) # lst est triée (mais ce n'est pas le même lst qu'avant, vérifiez l'id)
```

Une erreur très courante est d'écrire :

```
lst = lst.sort() # Maintenant dans lst, il y a None
```

ou encore :

```
return lst.sort() # Renvoie None à tous les coups
```

On a la même chose (pas tout à fait, mais presque) avec `lst.reverse()` et `reversed(lst)`.

1.3 Raccourcis mal employés

Le raccourci `+=` est connu… mais parfois mal utilisé. Le code suivant **ne calcule pas** la somme des entiers de 1 à 10 (et aucune erreur de syntaxe n'est signalée, puisqu'il n'y a pas d'erreur de syntaxe)

```
s = 0
for i in range(1, 11):
    s =+ i
print(s) # affiche 10 au lieu de 55
```

N'utilisez pas les raccourcis, sauf si vous êtes sûrs de ne *jamais* vous tromper, sinon, écrivez sans raccourci :

```
s = 0
for i in range(1, 11):
    s = s + i
print(s) # affiche 55
```

Autre exemple, on vérifie que la fonction `random` produit effectivement 50 % de nombres entre 0 et 0.5 et 50 % entre 0.5 et 1 :

```
# verification moisie
import random
p, g = 0, 0
for k in range(1000):
    if random.random() < 0.5:
        p += 1
    else:
        g =+ 1
print("Pourcentage < 0.5 : ", round(p / (p + g) * 100, 2))
print("Pourcentage > 0.5 : ", round(g / (p + g) * 100, 2))
```

On obtient le magnifique résultat scientifique (rassurez-vous la génération de nombres pseudo-aléatoire fonctionne mieux que ça…) :

```
Pourcentage < 0.5 :  99.81
Pourcentage > 0.5 :  0.19
```

⚠ N'utilisez **jamais** ces raccourcis sauf si vous êtes certains de ne **jamais** vous tromper.

1.4 Étourderie

L'oubli du mot `range` dans une boucle *répéter pour* donne (parfois) une instruction syntaxiquement correcte, et l'erreur est donc pénible à détecter :

```
for i in (2, 10, 2):
    print(i)
```

au lieu de :

```
for i in range(2, 10, 2):
    print(i)
```

1.5 Une variable ne contient pas une formule...

Le programme suivant est une tentative pour lister les multiples de 3 dont le carré est inférieur à 10000 (mais il ne fonctionne pas) :

```
n = 0
carre = n ** 2
while  carre < 1e4:
    n = n + 3
```

L'erreur se solde par une boucle infinie, ce qui ne passe pas inaperçu. Un raisonnement simple sur la boucle permet de trouver l'erreur.

Bien entendu, on a simplement oublié de mettre à jour la valeur de la variable `carre`, ce qu'il est nécessaire de faire à chaque fois que la valeur de `n` change.

```
n = 0
carre = n ** 2
while  carre < 1e4:
    n = n + 3
    carre = n ** 2
```

1.6 Copies superficielles ou profondes

La copie des types simples ne pose pas de problème particulier. Il faut en revanche être prudent avec les collections. On peut copier une collection ainsi :

```
a = [1,2,[5,6,7]]
b = list(a)
```

Mais c'est une copie superficielle (*shallow copy*). Si un des éléments est modifiable (si un des éléments de la liste est une liste par exemple), ça peut être un problème. Le test suivant permet de comprendre ce qui se passe :

```
>>> a = [1, [1, 2]]
>>> id(a), id(a[0]), id(a[1])
---> (140409461203840, 140409515960624, 140409461264192)
>>> b = list(a)
>>> id(b), id(b[0]), id(b[1])
---> (140409460884416, 140409515960624, 140409461264192) # b[0] et b[1] partagés
>>> a, b
---> ([1, [1, 2]], [1, [1, 2]])
>>> b[0] = 42
>>> a, b
---> ([1, [1, 2]], [42, [1, 2]])   # contenu de a non modifié
>>> b[1][0] = 54
>>> a, b
---> ([1, [54, 2]], [42, [54, 2]]) # contenu (indirect) de a modifié
```

Pour obtenir un comportement différent, on dispose d'un module copy qui offre une copie en profondeur (*deep copy*) :

```
>>> import copy
>>> a = [1, [1, 2]]
>>> b = copy.deepcopy(a)
>>> b[1][0] = 54
>>> a, b
---> ([1, [1, 2]], [1, [54, 2]])
```

1.7 `tuple` non modifiable mais...

Un tuple n'est pas modifiable. Mais il peut contenir une liste qui l'est (ce qui n'est pas forcément une bonne idée...) :

```
>>> l = (3, 4, [1, 2, 3])
>>> l[2].append(4)
---> (3, 4, [1, 2, 3, 4])
```

2 Erreurs plus complexes

2.1 Évaluation paresseuse

Certaines fonctions renvoient des *itérateurs*. C'est le cas de `reversed`, qui ne renvoie pas directement une séquence *à l'envers* mais un itérateur qui peut parcourir les éléments en sens inverse. Mais attention aux pièges :

```
>>> t = [1, 2, 3]
>>> u = reversed(t) # le contenu n'est pas encore parcouru
>>> t[1] = 10
>>> v = list(u)     # il est parcouru maintenant
>>> v
---> [3, 10, 1]
>>> t = [1, 2, 3]
>>> it = reversed(t)
>>> u = list(it)    # le contenu est parcouru, it est épuisé
>>> v = list(it)    # plus rien dans it...
>>> u
---> [3, 2, 1]
>>> v
---> []
```

2.2 L'erreur n'est pas là

Retrouvez le code ici : https://gitlab.com/laurent.signac/teaching_snippets/-/tree/master/erreur_pas_la

Dans un programme Python, la ligne où une exception est levée, et donc une erreur révélée, n'est pas forcément la ligne qui contient la faute réelle.

Voici un exemple. On dispose d'une liste, contenant des mots et leur classement (peu importe ce qu'est ce classement). Par exemple :

```
data =  [ (1, "Minerva"), (2, "Albus"), (3, "Severus"), (4, "Sibylle"), (5, "Gilderoy") ]
```

Puis, on dispose d'une fonction, qui prend en paramètre le nom d'une personne, et renvoie le nombre de voyelles que contient ce mot :

```
def nb_voyelles(nom: str) -> int:
    nom_m = nom.lower()
    for voyelle in "aeiou":
        c = c + nom_m.count(voyelle)
    return c
```

À présent, pour chaque nom de la liste `data`, on souhaite afficher le nombre de voyelles qu'il contient :

```
for nom in data:
    print(nb_voyelles(nom))
```

Si on l'exécute, on obtient le traceback suivant qui indique une erreur sur la ligne `nom_m = nom.lower()` :

```
Traceback (most recent call last):
  File "<tmp 1>", line 11, in <module>
    print(nb_voyelles(nom))
  File "<tmp 1>", line 4, in nb_voyelles
    nom_m = nom.lower()
AttributeError: 'tuple' object has no attribute 'lower'
```

Pourtant l'erreur est ailleurs. Saurez-vous expliquer ce qui se passe ?

2.3 Méli mélo de noms de variables

https://gitlab.com/laurent.signac/teaching_snippets/-/tree/master/meli_melo_noms_variables

Une erreur assez courante, et parfois difficile à détecter, est la réutilisation d'un même nom de variable pour deux choses différentes. Par exemple, lorsqu'on traite les composantes rouge vertes et bleues d'une image,

et qu'il faut parcourir toute l'image (balayage selon les abscisses et le ordonnées), on voit parfois :

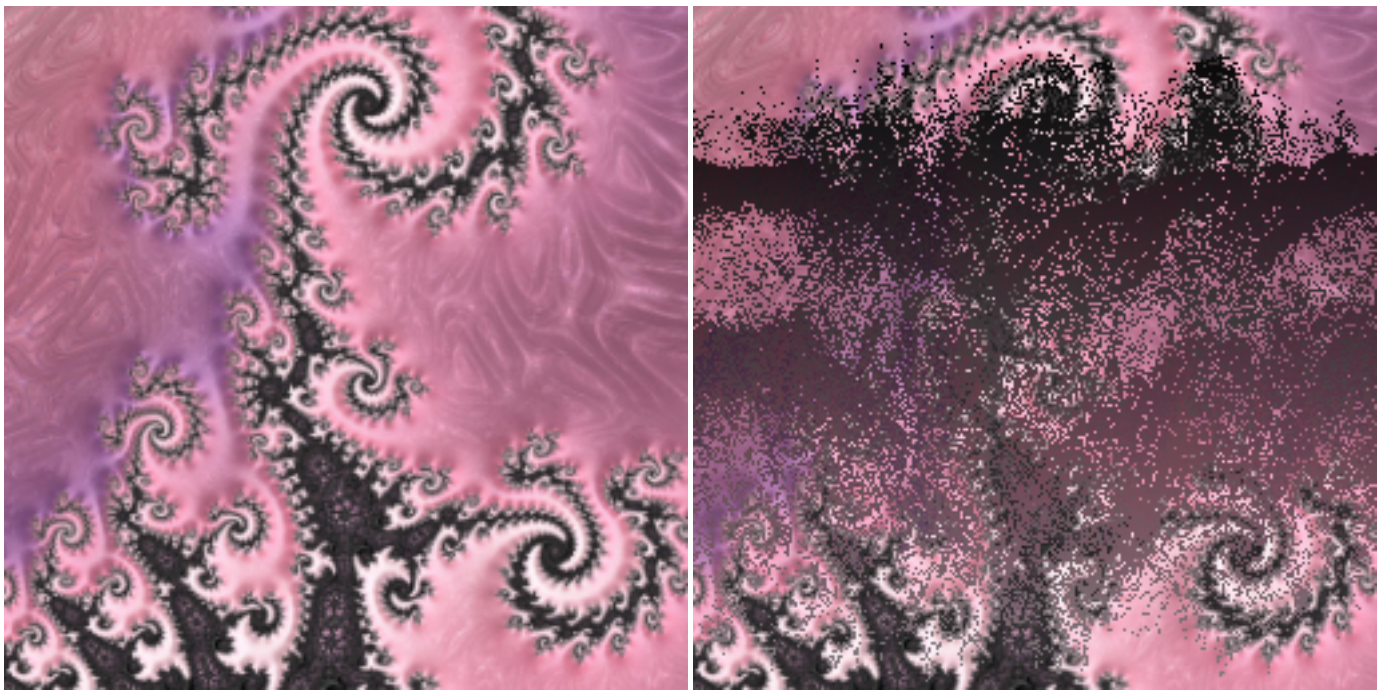
```
from PIL import Image

image = Image.open("sample_image.png")

for a in range(image.size[0]):
    for b in range(image.size[1]):
        r, v, b = image.getpixel((a, b))
        # Traitement pour assombrir par exemple :
        r1, v1, b1 = r//2, v//2, b//2
        image.putpixel((a, b), (r1, v1, b1))

image.show()
```

Si on exécute le code fourni, avec l'image dans le répertoire courant, aucune exception n'est levée. Et l'image résultat s'affiche. Mais elle ne ressemble pas à ce qu'on devrait obtenir (original à gauche, image transformée à droite) :



À noter que le code échoue sur une image plus petite (par exemple sur une image 128x128). L'exception levée signale alors :

```
Traceback (most recent call last):
  File "<tmp 2>", line 10, in <module>
    image.putpixel((a, b), (r1, v1, b1))
  File "/usr/lib/python3.9/site-packages/PIL/Image.py", line 1758, in putpixel
    return self.im.putpixel(xy, value)
IndexError: image index out of range
```

Pourtant même sur une petite image, le code peut ne pas lever d'exception… à condition que l'image soit suffisamment sombre…

Voyez-vous où se situe le problème ?

2.4 Paramètres par défaut mutables

Utiliser un paramètre par défaut mutable est syntaxiquement correct, mais c'est une erreur de programmation dans la plupart des cas.

Voici l'exemple d'une fonction (jouet) qui prend en paramètre une chaîne. Elle compte les voyelles et les consonnes, puis les ajoute à un éventuel décompte préalable passé en paramètre (une liste de deux entiers) :

```
>>> compte_lettres("Wingardium")
--> [4, 5] # 4 consonnes et 5 voyelles

>>> compte_lettres("Wingardium", [6, 10]) # Ajout au décompte précédent
--> [10, 15]
```

Pour faire ceci, on utilise un paramètre dont la valeur par défaut est `[0, 0]` (qui est une liste, et donc qui est mutable…)

```
import string
# Attention, ce code est mauvais...
def compte_lettres(chaine, compte=[0, 0]):
    for c in chaine.lower():
        if c in "aeiou":
            compte[0] += 1
        elif c in string.ascii_lowercase:
            compte[1] += 1
    return compte
```

Mais tout ne se passe pas comme prévu :

```
>>> compte_lettres("Leviosa", [1, 1])
--> [5, 4]

>>> compte_lettres("Leviosa")
--> [4, 3]

>>> compte_lettres("Leviosa")
--> [8, 6]       <=========== Pas normal.... :(
```

Vous êtes invités à réfléchir aux causes du problème. En attendant, la bonne pratique est d'utiliser uniquement des paramètres par défaut non mutables :

```
import string
def compte_lettres(chaine, compte=None):
    if compte is None:
        compte = [0, 0]
    for c in chaine.lower():
        if c in "aeiou":
            compte[0] += 1
        elif c in string.ascii_lowercase:
            compte[1] += 1
    return compte
```

2.5 `random` et fonction impure

Une fonction *pure* est une fonction sans état, qui renvoie le même résultat si on lui donne les mêmes paramètres, et qui n'a pas d'effet de bord.

La fonction `random.random()` renvoie un nombre aléatoire entre 0 et 1. Bien sûr c'est une fonction impure, avec un état (sinon, on aurait toujours le même nombre aléatoire, ce qui n'est pas très utile).

Le code suivante fait des tirages aléatoires, avec `random` et comptabilise le nombre de tirages dans les intervalles $[0, \frac{1}{3}[$, $[\frac{1}{3}, \frac{2}{3}[$, $[\frac{2}{3}, 1[$.

Le tirage est censé être uniforme et pourtant… :

```
import random

N = 100000
tiers_1, tiers_2, tiers_3 = 0, 0, 0
for k in range(N):
    if random.random() < 1/3:
        tiers_1 += 1
    elif 1/3 <= random.random() < 2/3:
        tiers_2 += 1
    else:
        tiers_3 += 1

print(tiers_1, tiers_2, tiers_3)
```

affiche :

```
33200 22552 44248
```

Le nombre de valeurs dans le premier tiers semble correct, mais pas dans les autres.

Une idée du problème ?