Informatique

16

Quelques erreurs

Cours

Quelques erreurs

Ce paragraphe a pour but de vous aider à investiguer lorsque vous rencontrez des erreurs lors de l'exécution de vos codes. Il n'y a pas forcément la solution à votre problème, mais c'est quand même fort probable.

1.I. Variable inexistante

De loin la plus simple, cette erreur est généralement due à une faute d'orthographe ou une minuscule au lieu d'une majuscule…

`Nb_Eleves = 10` `print(Nb_eleves)`	`NameError: name 'Nb_eleves' is not defined` Eleve avec une majuscule

1.II. Curseur

`def f(x):`
`  return 2x+a`
`print(f(2))`
	`return 2x+a`
	`  ^`
	`SyntaxError: invalid syntax`
	Le chapeau indique l'erreur, il fallait écrire 2*x

1.III. Oublie de parenthèses

`Frac = (((10+2/(10+3))`
`print(Frac)`
	`print(Frac)`
	`  ^`
	Il manque une parenthèse fermée à la ligne précédente

Notez que le curseur est placé à la fin du premier mot de la ligne suivant l'erreur.

1.IV. Indentation

`def f(x):`
`  Res = 2*x+2`
`  return Res`
`print(f(2))`
	`return Res`
	`^`
	`IndentationError: unexpected indent`
	Un espace en trop avant return

`def f(x):`
`  # Fonction vide`
`print(f(2))`
	`print(f(2))`
	`  ^`
	`IndentationError: expected an indented block`
	La fonction est laissée vide alors qu'il faut un contenu

1.V. Homogénéité

Quand on fait par exemple une opération entre deux grandeurs non homogènes en type, on est prévenus :

`None + 1`	`TypeError: unsupported operand type(s) for + : 'NoneType' and 'int'`
`1 + '1'`	`TypeError: unsupported operand type(s) for + : 'int' and 'str'`

1.VI. Taille de liste dépassée

`L = [1,2,3]` `L[3]`	`L[3]` `IndexError: list index out of range`

1.VII. « not callable »

Quand on crée une fonction, celle-ci est « appelable », ou « callable ». Pour l'appeler, il faut utiliser des parenthèses. Si vous mettez des parenthèses derrière autre chose qu'une fonction, python vous dira que le type d'objet derrière lequel vous avez mis des parenthèses n'est pas « callable ».

```
a = [1,2,3]
a(1)
```

```
a(1)
TypeError: 'list' object is not callable
```

```
b = None
b(1)
```

```
b(1)
TypeError: 'NoneType' object is not callable
```

1.VIII. « not subscriptable »

Quand on crée une liste (ou un array), celle-ci est « subscriptable », c'est-à-dire qu'on peut aller chercher l'un de ses termes avec des crochets []. Si vous appelez un objet autre qu'une liste avec les crochets, python vous préviendra que cet objet n'est pas « subscriptable ».

```
a = 1
a[0]
```

```
a[0]
TypeError: 'int' object is not subscriptable
```

```
def f(L,x):
    Res = []
    for t in L:
        Res.append(t-x)
L = [1,2,3]
x = 2
LL = f(L,x)
print(LL[0])
```

```
print(LL[0])
TypeError: 'NoneType' object is not subscrip
table
```

Il manque un return Res dans f

Vous rencontrerez cette erreur avec un NoneType quand vous oublier le return ou que vous écrivez return Res.append() dans une fonction qui renvoie une liste, le résultat récupéré étant alors None…

1.IX. « has no attribute »

Si les listes par exemple permettent d'utiliser la « méthode » append en écrivant par exemple L.append(), ce n'est pas le cas de tous les objets :

```
a = 1
a.append(2)
```

```
a.append(2)
AttributeError: 'int' object has no attribut
e 'append'
```

```
def f(L,x):
    Res = []
    for t in L:
        Res.append(t-x)
L = [1,2,3]
x = 2
LL = f(L,x)
LL.append(1)
```

```
LL.append(1)
AttributeError: 'NoneType' object has no att
ribute 'append'
```

Même remarque que la précédente, cela vous arrivera souvent d'avoir l'erreur avec un NoneType du fait de l'oublie d'un return ou return Res.append() dans une fonction renvoyant une liste.

1.X. « concatenate »

On peut concaténer deux listes, c'est-à-dire les mettre bout à bout :

```
>>> [1,2,3]+[5,5]
[1, 2, 3, 5, 5]
```

Vous obtiendrez l'erreur « can only concatenate list (not "NoneType") to list » si vous tentez le + entre deux listes alors que l'une d'elles était un renvoie erroné « None » d'une fonction appelée avant…

1.XI. Gestion des « tuples »

Un tuple est un ensemble d'objets entre parenthèses. A la différence des listes, les tuples ne sont pas modifiables.

```
>>> A = (1,2)

>>> A[0]
1

>>> A[0]=1
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError: 'tuple' object does not support i
tem assignment

>>> A = ([1,2],[3,4])

>>> A[0] = 1
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError: 'tuple' object does not support i
tem assignment

>>> A[0][0] = 5

>>> A
([5, 2], [3, 4])
```

On peut toutefois modifier les éléments dans le tuple.

Vous rencontrez par inadvertance des tuples quand vous renvoyez plusieurs objets dans des fonctions sans les mettre dans une liste ET quand vous les récupérez dans un seul objet :

```
def f(x):
    a = x**2
    b = 2*x
    return [a,b]
Res = f(1)
print(Res)
print(type(Res))
Res[0] = 0
print(Res)
```

```
[1, 2]
<class 'list'>
[0, 2]
```

```
def f(x):
    a = x**2
    b = 2*x
    return a,b
Res = f(1)
print(Res)
print(type(Res))
Res[0] = 0
print(Res)
```

```
(1, 2)
<class 'tuple'>
Res[0] = 0
TypeError: 'tuple' object does not support i
tem assignment
```

Si vous voyez apparaître le mot « tuple », c'est généralement que vous avez fait un return de 2 (ou plusieurs) éléments sans les mettre entre crochets, que vous les récupérez dans un seul objet, et que vous cherchez à modifier l'un d'eux.

Soit la fonction suivante et les 3 suites proposées ensuite :

```
def f(L):
    LL = L.copy()
    LL[0] = 0
    return L,LL
```

`L,LL = f([1,2,3])`
`print(L,LL)`
`L[0] = 1`
`print(L)`
	On récupère les 2 listes, pas de tuple
	`>>> (executing file "<tmp 1>")`
	`[1, 2, 3] [0, 2, 3]`
	`[1, 2, 3]`

`Res = f([1,2,3])`
`print(Res)`
`Res[0][0] = 5`
`print(Res)`
	On récupère un tuple dans Res
	`([1, 2, 3], [0, 2, 3])`
	`([5, 2, 3], [0, 2, 3])`
	On peut modifier les listes dans le tuple

`Res = f([1,2,3])`
`print(Res)`
`Res[0] = 1`
`print(L)`
	`Res[0] = 1`
	`TypeError: 'tuple' object does not support i`
	`tem assignment`
	Mais on ne peut pas modifier le tuple

L'erreur de tuple vient le plus souvent d'une fonction qui renvoie plusieurs choses au lieu d'en renvoyer une seule, donc vous récupérez qu'un seul tuple en ayant oublié qu'il y a plusieurs choses à récupérer.

1.XII. Les erreurs imbriquées

Il reste à aborder les erreurs qui viennent d'une fonction utilisée dans d'autres fonctions.

```
def f(x):
    return 1/x

def g(x):
    return f(x)

def h(x):
    return g(x)

h([1])
```

```
Traceback (most recent call last):
  File "<tmp 1>", line 10, in <module>
    h([1])
  File "<tmp 1>", line 8, in h
    return g(x)
  File "<tmp 1>", line 5, in g
    return f(x)
  File "<tmp 1>", line 2, in f
    return 1/x
TypeError: unsupported operand type(s) for /
: 'int' and 'list'
```

En appelant h pour une liste, l'erreur sera rencontrée dans f au moment d'évaluer 1/x, x ne pouvant être une liste.

Python nous montre d'où vient l'erreur :

- Ligne 10, quand on appelle h([1]), il y a un retour d'erreur, car
- Ligne 8, à l'appel de g(x) il y a un retour d'erreur, car
- Ligne 5, à l'appel fe f(x), il y a un retour d'erreur, car
- Ligne 2, le calcul de 1/x renvoie l'erreur, impossible de diviser un entier par une liste

On comprend donc que dès le départ, x était une liste, et n'aurait pas dû l'être !

1.XIII. Le mode « débogue » de Pyzo

Sans rentrer dans les détails, il est possible d'exécuter pas à pas un code, en utilisant le mode de débogage de Pyzo. Pour cela, il suffit de placer un ou plusieurs points rouges dans la marge puis d'exécuter le code :

```
1 def f(x):
2     a = x**2
3     b = x
4     y = a + b
5     return y
6
7 f(2)
```

Alors apparaissent des outils qui permettent d'avancer ou de mettre fin au mode « débogue » :

On voit par ailleurs dans la console ceci :

```
(f)>>>
```

Nous sommes donc « dans f » et on a accès en cours d'exécution aux variables locales, exemple :

```
(f)>>> a
4

(f)>>> b
2
```