Pense bête 4

Variables locales, variables globales

Lorsqu'une fonction est appelée, Python réserve pour elle (dans la mémoire de l'ordinateur) **un espace de noms**.

Cet espace de noms **local** à la fonction est à distinguer de l'espace de noms **global** où se trouvait les variables du programme principal.

A chaque fois que nous définissons des variables à l'intérieur du corps d'une fonction, ces variables ne sont accessibles qu'à la fonction elle-même.

On dit que ces variables sont des **variables locales** à la fonction.

Une variable locale peut avoir le même nom qu'une variable de l'espace de noms global mais elle reste néanmoins indépendante. Dans ce cas, pour la fonction, la variable globale est masquée par la variable locale de même nom.

Les contenus des variables locales sont stockés dans **l'espace de noms local qui est inaccessible depuis l'extérieur de la fonction**.

Les variables définies à l'extérieur d'une fonction sont **des variables globales**. **Leur contenu est « visible » de l'intérieur d'une fonction, mais la fonction ne peut pas le modifier.**

Exemple :

```
#Définition de la fonction
def test():
    b = 5
    print ("Dans la fonction : a = ", a, "b = ",b)
#Programme principal
a = 2
b = 7
test()
print ("Dans le programme principal: a = ", a, "b = ",b)
```

Résultat :

Dans la fonction : a = 2 b = 5

Dans le programme principal : a = 2 b = 7

Nous voyons que la variable a, du programme principal est visible dans la fonction puisque celle-ci en affiche la valeur.

La fonction ayant une variable locale de même nom b, qu'une variable globale du programme principal, ne voit plus cette variable globale.

Elle ne peut donc pas modifier cette variable globale.

Utilisation d'une variable globale modifiable par une fonction

Il peut se faire que l'on ait besoin de définir une fonction qui soit capable de modifier une variable globale. Pour atteindre ce résultat, il suffit d'utiliser l'instruction **global**. Cette instruction permet d'indiquer - **à l'intérieur de la définition d'une fonction** - quelles sont les variables à traiter globalement.

```
#Définition de la fonction
def test():
    global b
    b = 5
    print ("Dans la fonction : a = ", a, "b = ",b)
#Programme principal
a = 2
b = 7
print ("Dans le programme principal, avant appel à la fonction: a = ", a, "b = ",b)
test()
print ("Dans le programme principal, après appel à la fonction: a = ", a, "b = ",b)
```

Résultat

Dans le programme principal, avant appel à la fonction : a = 2 b = 7

Dans la fonction : a = 2 b = 5

Dans le programme principal, après appel à la fonction : a = 2 b = 5

L'instruction **global b** dans la fonction, indique à python qu'il ne doit pas créer une nouvelle variable b, locale à la fonction.

La fonction travaille alors directement sur la variable globale du programme principal.

Les exceptions

Un système de gestion des exceptions permet de gérer les conditions exceptionnelles pendant l'exécution du programme.

Lorsqu'une exception se produit, l'exécution normale du programme est intérrompue et l'exception est traitée.

On rencontre des exceptions dans de nombreux cas, mais souvent, c'est dans le cadre d'erreurs : division par zéro, accès à une zone interdite de la mémoire…

Non seulement une exception interrompt le programme, mais elle collecte des informations sur la source de l'erreur afin qu'au moment où ça crash, le développeur ait de quoi déboguer.

Une exception va donc normalement déclencher un affichage à la fin avec toutes ces infos, et sur la dernière ligne, le type de l'exception (ValueError, IndexError, etc) ainsi qu'une description de ce qui a causé l'erreur.
Néanmoins, le plus intéressant est ce qu'il y a au dessus : le gros pâté de texte. C'est ce qu'on appelle une stack trace, et ça représente la pile d'appels qui ont amené à cette erreur. Chaque ligne de la stack trace va vous donner le chemin d'un fichier de code, et une ligne.
Une erreur se lit donc à l'envers, de bas en haut.
Vous lisez d'abord le nom de l'erreur et sa cause, puis, vous remontez la stack trace ligne à ligne, pour essayer de trouver quelle ligne de quel fichier de code vous devez déboguer.

Exemple :

```
 8 def uneFonction():
 9     return 1/0
10
11 def uneAutreFonction():
12     uneFonction()
13
14 uneAutreFonction()
15
```

Affichage dans la console :

runfile('D:/Python/scrtach_python/Tutoriels/gestionErreur.py', wdir='D:/Python/scrtach_python/Tutoriels')

Traceback (most recent call last):

File "<ipython-input-1-707d21c8b768>", line 1, in <module>

runfile('D:/Python/scrtach_python/Tutoriels/gestionErreur.py', wdir='D:/Python/scrtach_python/Tutoriels')

File "D:\Anaconda3\lib\site-packages\spyder_kernels\customize\spydercustomize.py", line 827, in runfile

execfile(filename, namespace)

File "D:\Anaconda3\lib\site-packages\spyder_kernels\customize\spydercustomize.py", line 110, in execfile

exec(compile(f.read(), filename, 'exec'), namespace)

File "D:/Python/scrtach_python/Tutoriels/gestionErreur.py", line 14, in <module>

uneAutreFonction()

File "D:/Python/scrtach_python/Tutoriels/gestionErreur.py", line 12, in uneAutreFonction

```
    uneFonction()

  File "D:/Python/scrtach_python/Tutoriels/gestionErreur.py", line 9, in
uneFonction
    return 1/0

ZeroDivisionError: division by zero
```

On lit sur la dernière ligne : ZeroDivisionError: division par zero

Elle a lieu dans mon fichier gestionErreur.py à la ligne 9 dans la fonction uneFonction.

Cette fonction uneFonction est elle-même appelée à la ligne 12 par uneAutreFonction.

Etc.

Ici la correction est simple et il suffit de corriger la ligne 9 de uneFonction.

Mais dans ce programme :

```
 8 def uneFonction(diviseur):
 9     return 1/diviseur
10
11 def uneAutreFonction():
12     uneFonction(0)
13
14 uneAutreFonction()
15
```

runfile('D:/Python/scrtach_python/Tutoriels/gestionErreur.py', wdir='D:/Python/scrtach_python/Tutoriels')

Traceback (most recent call last):

File "<ipython-input-2-707d21c8b768>", line 1, in <module>

```
  runfile('D:/Python/scrtach_python/Tutoriels/gestionErreur.py',
wdir='D:/Python/scrtach_python/Tutoriels')


 File "D:\Anaconda3\lib\site-
packages\spyder_kernels\customize\spydercustomize.py", line 827, in runfile
   execfile(filename, namespace)

 File "D:\Anaconda3\lib\site-
packages\spyder_kernels\customize\spydercustomize.py", line 110, in execfile
   exec(compile(f.read(), filename, 'exec'), namespace)

 File "D:/Python/scrtach_python/Tutoriels/gestionErreur.py", line 14, in
<module>
   uneAutreFonction()

 File "D:/Python/scrtach_python/Tutoriels/gestionErreur.py", line 12, in
uneAutreFonction
   uneFonction(0)

 File "D:/Python/scrtach_python/Tutoriels/gestionErreur.py", line 9, in
uneFonction
   return 1/diviseur

ZeroDivisionError: division by zero
```

Ici ce n'est pas la ligne 9 dans uneFonction qu'il faut corriger, mais la ligne 12 dans uneAutreFonction qui n'appelle pas la fonction uneFonction avec le bon paramètre.

Attraper une exception

Les exceptions ne sont pas un simple mécanisme de debuggage. Elles servent d'abord et avant tout à gérer les cas exceptionnels, et on peut donc les détecter, et réagir quand elles surviennent, à l'aide de l'instruction :

Try/except

```
#fonction
def uneFonction(diviseur):
    try:
        nombre = 1/diviseur
        return nombre
    except ZeroDivisionError:
        print("Le divisuer fourni est nul")
        return  None
#programme principal
nombre = int(input("Entrez le diviseur: "))
nombre2 = uneFonction(nombre)
print("nombre 2 : ",nombre2)
```

Résultat

Entrez le diviseur : 3

nombre 2 : 0.3333333333333333

Un autre essai en entrant la valeur 0

Entrez le diviseur: 0

Le divisuer fourni est nul

nombre 2 : None

La fonction a détecté la division par zéro : le programme ne plante plus, mais avertit l'utilisateur qu'il n'a pas fourni une valeur correcte.

None = rien