Exceptions

Gestion des erreurs

Lorsque l'exécution d'un programme ne se déroule pas comme prévu, elle se termine souvent par une erreur. Cela s'appelle une **exception**. Considérons la fonction ci-dessous :

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

Si on essaie de calculer l'inverse de 0, on obtient l'erreur suivante :

```
>>> inverse(0)
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
  File "exceptions.py", line 11, in inverse
    return 1/x
ZeroDivisionError: division by zero
```

Cette erreur est composée de 3 parties :

- La **pile d'appels** qui nous indique, en lisant de bas en haut, que l'erreur est survenue lors de l'exécution de la ligne 11 du fichier `exceptions.py`, dans la fonction `inverse` qui a été appelée dans l'interpréteur (`pyshell`).
- L'exception est `ZeroDivisionError`, ce qui fait clairement allusion à une division par 0.
- Le texte d'explication confirme que c'est bien une division par 0.

Cela permet donc de mieux comprendre ce qui s'est passé et à quel moment. Par contre, l'absence d'exception ne veut pas dire qu'il n'y a pas d'erreur dans le programme. Il peut s'exécuter "normalement", sans pour autant faire ce qui était prévu. Dans ce cas, on a affaire à une erreur de programmation, comme une erreur dans de calcul, qui ne peut pas être détectée par l'interpréteur, à moins d'utiliser des **`assert`**.
Il existe plusieurs types d'exceptions et voici les principales :

Exception	Explication
`NameError`	Variable ou fonction inconnue.
`IndexError`	Indice invalide pour un tableau.
`KeyError`	Clef inconnue pour un dictionnaire.
`ZeroDivisionError`	Division par zéro.
`TypeError`	Opération entre deux valeurs incompatibles.

La commande **`raise`** permet à l'utilisateur de lever des exceptions :

```
>>> raise ValueError("On avait dit positif !")
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
ValueError: On avait dit positif !
```

Lors de la création d'un objet, cela peut permettre de renvoyer une erreur si les paramètres donnés aux constructeurs ne sont pas valides.

Catch me if you can

Une exception n'est pas forcément synonyme de fin d'exécution d'un programme. Si le programme s'arrête, c'est que l'exception n'a pas été "rattrapée" avant de remonter jusqu'à l'interpréteur. Lorsqu'une exception est levée, elle remonte la pile d'appels en stoppant l'exécution des fonctions en cours, jusqu'à ce que l'une d'entre elle l'intercepte.
Pour intercepter une exception, il faut utiliser la structure ci-contre.

```
try:
    bloc qui peut
    provoquer une
    exception
except:
    à faire en cas
    d exception
```

Cela peut permettre de remonter rapidement une grande pile d'appels en ne gérant l'erreur que dans la fonction initiale. Par exemple, dans une interface graphique dans laquelle un module essaie de télécharger des données par Internet. En cas d'erreur, la fonction qui fait le téléchargement peut lever une exception et c'est la fonction principale qui contiendra le `try ... except` pour pouvoir afficher un message indiquant qu'il y a un problème de connexion.

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

def double_inverse(x):
    return 2 * inverse(x)

try:
    double_inverse(0)
except ZeroDivisionError:
    print("On ne divise pas par 0 !")
```

On peut également préciser l'exception que l'on veut rattraper, en laissant passer les autres. Dans cet exemple, l'exception est levée par `inverse` mais n'est pas rattrapée par la fonction `double_inverse`.
Cela permet aussi de réagir de façons différentes à plusieurs exceptions :

```
tab = [5, 1, 6, 0]

try:
    i = int(input("Donner un indice pour le tableau"))
    print(tab[i])
except IndexError:
    print("On est hors du tableau")
except ValueError:
    print("Il faut donner un entier")
```

En conclusion

Les exceptions sont un mécanisme qui permet de simplifier la gestion des erreurs afin de ne pas multiplier les tests après les appels de fonctions et en n'en mettant que lorsque c'est réellement nécessaire. C'est néanmoins une technique avancée et qui nécessite du temps pour être maîtrisée.
Il faut aussi faire attention car la simplicité gagnée sur l'écriture des fonctions rend plus opaque le fonctionnement du programme si la levée de l'exception et son interception sont éloignées l'une de l'autre, par exemple par l'appel imbriqué de plusieurs fonctions.
Pour être plus explicite, il est possible de définir de nouvelles exceptions si nécessaire, mais on sort du programme de NSI.