NSI - première

Python 8 - Exceptions

qkzk

2021/04/27

Une *exception* est une erreur provoquée par Python.

C'est un mécanisme universel en programmation et il existe un moyen d'*attraper* ces exceptions afin d'éviter que le programme ne plante.

Principe des exceptions

Lorsque Python rencontre une instruction impossible il provoque une erreur. Plus précisément on dit qu'il lève une exception.

Par exemple lorsqu'on tente d'accéder à un élément dont l'indice n'existe pas :

```
>>> l = ["a", "b"]
>>> l[0]
'a'
>>>
>>> l[3]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Python lève l'exception `IndexError`

Il en existe beaucoup : `ValueError`, `RuntimeError`, `ZeroDivisionError`, `KeyError`, `FileNotFoundError`

Certaines sont plus surprenantes :

```
>>> while True:
...     print(1)
1
1
1
1
^CTraceback (most recent call last):
  File "<stdin>", line 2, in <module>
KeyboardInterrupt
```

J'ai appuyé sur CTRL + C pour arrêter l'exécution et Python lève l'exception `KeyboardInterrupt`

Le programme est-il foutu dès qu'une exception est rencontrée ?

Non... il est toujours possible d'éviter qu'il ne plante en *attrapant* cette exception.

`try... except`

La syntaxe est la suivante :

```python
nombre = 0

try:
    print( 1 / nombre )

except ZeroDivisionError:

    print("division impossible")

print("cette ligne sera toujours atteinte")
```

Ici on tente une opération, elle est impossible... donc on attrape l'exception `ZeroDivisionError` et on affiche le second message.

Voici la sortie obtenue :

```
division impossible
cette ligne sera toujours atteinte
```

Exercice 1

En utilisant une boucle et un bloc `try...except` produire l'affichage suivant :

```
0.1
0.1111111111111111
0.125
0.14285714285714285
0.16666666666666666
0.2
0.25
0.3333333333333333
0.5
1.0
impossible
-1.0
-0.5
-0.3333333333333333
-0.25
-0.2
-0.16666666666666666
-0.14285714285714285
-0.125
-0.1111111111111111
-0.1
```

ces nombres sont les inverses des entiers de 10 à -10

Différentes exceptions

Lorsqu'aucune exception n'est précisée dans le bloc `except`, Python attrape toutes les erreurs. Ça va plus vite à écrire mais c'est une très mauvaise pratique.

On utilise un bloc `try` lorsqu'on souhaite éviter un problème particulier, pas dans tous les cas.

Lorsqu'un bloc `try` provoque une erreur, la partie du bloc `try` précédent l'erreur est toujours exécutée :

```
>>> a = 1
>>> try:
...   a = 0
...   1 / a
... except ZeroDivisionError:
...   print("impossible")
...
impossible
>>> a
0
```

Exercice 2

1. Quelle est l'exception levée par le programme suivant ?

```
d = {"jean": 82, "ginette": 74, "olivier": 45, "manon": 14}
for n in ["paul", "jean", "olivier"]:
  print(d[n])
```

2. Modifier le programme pour que la boucle se termine et qu'on affiche l'age de toutes les personnes enregistrées dans le dictionnaire.

Compléments

Les exceptions ne sont pas au programme mais vous risquez d'en rencontrer... et pourrez être tenté de les utiliser.

Néanmoins, Python propose d'autres éléments de syntaxe :

```
try:
  toujours_tenté
except MonException:
  fait_autre_chose
else:
  si_pas_derreur
finally:
  sera_toujours_execute
```

Cela peut s'avérer pratique pour être certain que certaines instructions sont exécutées même si le programme plante. Par exemple pour se deconnecter proprement ou fermer un fichier etc.

Le mot clé `assert`

Ce mot clé s'utilise de la façon suivante :

```
assert predicat_vrai, "message d'erreur affiché si le prédicat est faux"
```

Par exemple :

```
assert 2 + 2 == 4, "2 + 2 n'est pas égal à 4 !!!"
```

Ce programme n'affiche rien, justement parce que le booléen `2 + 2 == 4` est évalué à `True`.

Modifions légèrement :

```
assert 2 + 2 == 5, "2 + 2 n'est pas égal à 5 !!!"
```

Cette fois, le booléen `2 + 2 == 5` est évalué à `False` et l'exécution provoque une erreur `AssertionError`. et on voit le message d'erreur "2 + 2 n'est pas égal à 5 !!!" apparaître.

Usage pour tester une fonction

Vous rencontrerez régulièrement des exercices où vous devez écrire le code d'une fonction pour lesquels le comportement est attendu.

Des tests sont proposés afin de s'assurer que la fonction fait ce qu'on attend d'elle.

Ce principe de développement (écrire les tests avant le code) est très courant, il s'appelle "Test Driven Developpement", *développement guidé par les tests*.

Considérons un exemple :

```
def carre(x):
  """renvoie le carré de x"""

assert carre(2) == 4
assert carre(0) == 0
assert carre(1) == 1
assert carre(-1) == 1
assert carre(100) == 10000
assert carre(-5) == 25
```

Lorsqu'on exécute ce script, il lève une erreur. Afin d'éviter cette erreur, on propose le code suivant :

```
def carre(x):
  """renvoie le carré de x"""
  return x * x
```

et maintenant les tests passent tous.

Exercice 2

1. Écrire le code d'une fonction `premier_element` qui renvoie :
   - le premier élément d'une liste non vide,
   - `None` si la liste est vide.
2. Écrire 5 tests avec `assert` vous assurant de vérifier les différentes situations.

Usage pour valider des valeurs

Lors du développement d'un programme complexe, on découpe les étapes afin de les rendre les plus simples et lisibles.

Il arrive donc régulièrement qu'une partie dépende des autres.

Afin de s'assurer qu'une erreur ne puisse se propager trop loin et provoque des erreurs incompréhensibles, on peut vérifier avec `assert` que les données reçues sont valides.

Par exemple, imaginons devoir extraire la première lettre d'une chaîne de caractères... C'est sans difficulté, sauf si l'objet reçu n'est pas une chaîne.

Il suffit de faire `ma_chaine[0]` et on obtient la première lettre. Parfait.

Est-on certain d'obtenir un résultat cohérent quelle que soit le type passé en paramètre ?

- Si `ma_chaine` est du type `str`, tout va bien.
- Si `ma_chaine` est du type `int`, on va rencontrer une erreur. Tout va bien.
- Si `ma_chaine` est du type `list`, on aura un résultat surprenant... C'est mauvais.

Comment obtenir une erreur *lisible* dans tous les cas ?

En utilisant `assert`, par exemple.

```python
def extrait_initiale(mot: str) -> str:
    """Renvoie la première lettre de `mot`"""
    assert isinstance(mot, str), "mot n'est pas du bon type"
    return mot[0]
```

Ainsi, on aura toujours une erreur lisible si les données reçues par `extrait_initiale` ne sont pas du bon type.

SyntaxError et IndentationError

Ces erreurs sont provoquées lorsque Python ne parvient pas à lire la ligne considérées... Il n'est pas possible de les capturer.

La seule solution est de les rectifier.