

Gestion des exceptions

1. Un exemple d'erreur à ne pas commettre

Utilisation malheureuse d'un module

Reprenons le module `secondDegre.py`, mais en temps qu'utilisateur·trice. Nous connaissons l'interface qui nous a été fournie par l'auteur·trice. Pour tester le module nous lançons le script suivant, nommé `testModule.py`, et situé dans le même dossier que `secondDegre.py` :

```
import secondDegre as sD

p = input("Donnez les coefficients du polynome séparés par des virgules :")
p = tuple(map(float, p.split(",")))

p = sD.polynome(p)
print(sD.tangente(p,3))
```

```
1 import secondDegre v1 as sD
2
3 p = input("Donnez les coefficients du polynome sé
4
5 p = tuple(map(float, p.split(",")))
6
7 p = sD.polynome(p)
8
9 print(sD.tangente(p,3))
10
11
```



Un problème ?

Copiez-collez le code ci-dessus dans un fichier `testModule.py`, puis exécutez-le en saisissant :

1. 3,4,5 ;
2. "trois",4,5
3. 3,4,5,6
4. 0,3,4

Dans chacun des cas, qu'obtient-on en sortie ? Pourquoi ?

Solution



Lever les bonnes erreurs

Dans l'exemple précédent, les trois erreurs, pourtant très différentes, **sont signalées par le même message**. L'utilisateur·trice, **qui lui ne connaît pas l'implémentation**, ne peut donc pas savoir d'où provient son erreur** (ce qui peut donner des séances de débogage particulièrement frustrantes). Il est donc nécessaire de préciser mieux les erreurs commises par l'utilisateur·trice, pour qu'il ou elle n'ait pas à se préoccuper des détails d'implémentation.

Il est par exemple possible de **rajouter un message** lorsque l'erreur est levée, en la passant en paramètre directement dans l'instruction `ValueError()` :

```
def polynome(t) :  
    a,b,*c = t  
    if len(c) >1 :  
        raise ValueError("length of tuple argument greater than 3")  
    if not(isinstance(a,(int, float))  
    ) or not(isinstance(b,(int, float))  
    ) or not(isinstance(*c,(int, float))) :  
        raise ValueError("argument Error : argument must be a tuple integers or float")  
    if a == 0 :  
        raise ValueError("First element of tuple must not be 0")  
    return t
```

2. Tyres d'exceptions

Voici quelques exceptions courantes, et leurs utilisations

Exception	Contexte
<code>NameError</code>	accès à une variable inexistante dans l'espace de nom courant
<code>IndexError</code>	accès à un indice invalide d'une liste, d'un tuple, d'une chaîne de caractères...
<code>KeyError</code>	accès à une clé inexistante d'un dictionnaire
<code>ZeroDivisionError</code>	division par zéro
<code>TypeError</code>	opération appliquées à un ou des objets incompatibles

Lever des exceptions

Une exception peut être **levée** (c'est-à-dire volontairement déclenchée) par l'intermédiaire de l'instruction `raise`.

Dans ce cas **le programme est interrompu**, et *la pile d'erreurs* est renvoyée dans le terminal à l'utilisateur.

Corriger le code""

Malgré nos corrections, il reste plusieurs possibilités d'erreurs dans l'utilisation de la fonction `polynome(t)`.

Quelles sont-elles et comment les corriger pour lever une exception explicite ?

Solution



3. Intercepter des exceptions

Vous avez constaté dans la solution précédente un bloc que nous n'avons encore jamais utilisé :

```
try :
    if len(t) == 1 :
        t = t[0]
    a, b, *c = t
except TypeError :
    raise TypeError("Must pass three argument or a tuple of 3 element.")
```

On a ici l'utilisation d'une structure spéciale : l'interception d'erreurs.



Interception des exceptions

Il arrive souvent en programmation que l'on doive utiliser une instructions ou une série d'instruction dont on sait à l'avance qu'elle peuvent générer des erreurs. La structure suivante est là pour ça :

```
try :
    # Bloc try
except error :
    # Bloc except
```

Le code du bloc `try` va être exécuté, et si une erreur du type fournie en argument de l'instruction `except` est levée, alors le code du bloc `except` est exécuté.



Exemple :

En première nous avons vu l'importance de rendre parfois un code **dumbproof**, et que cela généraient parfois de nombreuses difficultés. Le simple fait de coder une fonction demandant à un utilisateur de saisir un nombre entier entre 1 et 10 inclus pouvait rapidement pénible à écrire. Les deux onglets ci-dessous donnent deux versions d'une fonction permettant de réaliser cette fonction, la version utilisée en première, et celle levant des exceptions.

Version avec des structures conditionnelles

```
def askIntFrom1To10() :
    while True :
        nb = input("Entrez un entier entre 1 et 10 :)")
        if nb.isnumeric() and "." not in nb :
            nb = int(nb)
            if 1 <= nb and nb <= 10 :
                return nb
            else :
                print("L'entier saisi n'est pas entre 1 et 10. Veuillez recommencer")
        else :
            print("Ce n'est pas un entier, veuillez recommencer !")
```

Version avec interception d'erreurs

```
def askIntFrom1To10() :
    while True :
        try :
            nb = int(input("Entrez un entier entre 1 et 10 :"))
            if 1 <= nb and nb <= 10 :
                return nb
            else :
                print("L'entier saisi n'est pas entre 1 et 10. Veuillez recommencer")
        except ValueError :
            print("Ce n'est pas un entier, veuillez recommencer !")
```

? Exercice

Evidemment, la différence ne saute pas vraiment aux yeux... Pourquoi faire tout un plat d'une seule ligne gagnée ?

Essayez donc, pour chacune des 2 fonctions précédentes, avec les chaînes de caractères suivantes (à copier-coller) :

- `½`
- `32`

🔗 Réponse



i Enchaîner les interceptions

Il est aussi possible d'avoir plusieurs blocs `except` successifs, en utilisant :

```
try :  
    # Bloc try  
except error1 :  
    # Bloc except1  
except error2 :  
    # Bloc except2  
...
```

i Etendre la gestion des exceptions

Il existe de nombreuses autres possibilités utilisant la levée d'exceptions, mais elles dépassent largement le programme de Terminale.

Les plus curieux parmi vous pourront toujours aller lire la [doc Python](#)", qui reste la référence absolue...