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. Nous connaissons l'interface qui nous a été fournie par l'auteur. 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))
```

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

`3,4,5`

La sortie est :

```
`y = 22.0(x-3) + 44.0`
```

qui est correcte.

`"trois",4,5`

La sortie renvoie une erreur de type `ValueError`, mais c'est celle levée dans le premier `if`, car un élément du tuple n'est pas du bon type.

`3,4,5,6`

La sortie renvoie une erreur de type `ValueError`, mais c'est celle levée dans le premier `if`, car le tuple donné n'est pas de la bonne taille.

`0,3,4`

La sortie renvoie une erreur de type `ValueError`, mais c'est celle levée dans le deuxième `if`, car le tuple donné ne correspond pas à un polynôme de degré 2.

Lever les bonnes erreurs

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

Voici quelques exceptions courantes, et leurs utilisations

Exception	Contexte

Exception	Contexte
`NameError`	accès à une variable inexistante dans l'espace de nom courant
`IndexError`	accès à un indice invalide d'une liste, d'un tuple, d'une chaine de caractères...
`KeyError`	accès à une clé inexistante d'un dictionnaire
`ZeroDivisionError`	division par zéro
`TypeError`	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 renvoyées 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

Exécution de `polynome((2,3))` ou `polynome((4))`

Que se passe-t-il lorsqu'on utilise la fonction `polynome` avec un tuple de 1 ou 2 éléments ?

Dans le cas d'un seul élément, l'erreur ressortie est :

```
a,b,*c = t
TypeError: cannot unpack non-iterable int object
```

Dans le cas de deux éléments, l'erreur levée est :

```
) or not(isinstance(*c,(int, float))) :
TypeError: isinstance expected 2 arguments, got 1
```

Dans les deux cas, on voit apparaître les détails d'implémentation du code de la fonction, qui sont peu clairs en particulier pour le cas n°2.

Exécution de `polynome(3,4,5)`

Une erreur possible, et que vous avez probablement commise, est celle de passer non pas un seul argument sous la forme d'un *tuple* (ou d'une liste) mais de multiples paramètres.

Cette erreur déclenche alors l'exception `TypeError` car les trois arguments passés ne sont pas du bon type.

Une correction possible

Le code suivant est une solution possible (certainement perfectible) à la levée d'erreurs plus explicites :

```
def polynome(*t) :
    try :
        if len(t) == 1 :# si on passe un tuple ou un tableau (1,2,3),
            # *t le convertit en [(1,2,3)]
            t = t[0] # d'où cette ligne
        a, b, *c  = t #Puis on unpack, c étant une liste éventuellement vide
    except TypeError :
        raise TypeError("Must pass three argument or a tuple of 3 element.")
    if len(c) !=1 : # si c est vide ou contient au moins 2 éléments
        raise ValueError("Bad Number of argument.")
    if not(isinstance(a,(int, float))
    ) or not(isinstance(b,(int, float))
    ) or not(isinstance(*c,(int, float))) :
            raise TypeError("argment Error : argument must be a tuple of integers or floats.")
    if a == 0 :
        raise ValueError("First element of tuple must not be 0.")
    return t
```

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'ereurs.

Interception des exceptions

Il arrive souvent en programmation que l'on doive utiliser une instructions oui 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 génèrais 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 pouvaiet 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 excpetions.

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 chaines de caractères suivantes :

- ½
- 3²

▼ Réponse

En fait le problème provient de la *méthode* `isnumeric()`, dont on ne contrôle pas vraiment le fonctionnement. On sait que cette méthode *permet de vérifier si une chaine de caractères est bien constituée uniquement de caractères numériques. Mais sans lire réellement la documentation, qui peut se douter que les fractions définies dans la table utf8, ainsi que les caractères en exposant et en indice**, sont considérés comme des valeurs numériques.

L'intérêt du bloc `try` est qu'il se déclenchera dès qu'il y aura une exception levée. Et cette exception a peu de chance de se produire dans le bloc `if / else`. La fonction devient **dumbproof** (mais si vous réussissez à déclencher une erreur bloquante, signalez-le moi !!).

Enchainer 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
...
```

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...