Gestion des exceptions

1. Un exemple d'erreur à ne pas commettre

Utilisation malheureuse d'un module

Reprenons le module `secondDegre.py` de la partie précédente, mais cette fois ci 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))
```

Un problème ?

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

1. `3,4,5` ;
2. `3,4,5,6`
3. `0,3,4`

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

Solution

Lever les bonnes erreurs

Dans l'exemple précédent, les deux erreurs, très différentes, **sont signalées par leur type**, accompagné d'un message plus ou moins clair. L'utilisateur·trice, qui ne connait pas l'implémentation, **ne peut parfois 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·trice, pour qu'il ou elle 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()` ou `TypeError()`. On peut en outre améliorer le code en s'assurant que les éléments du tuple `t` sont bien `int` ou `float` :

```
def polynome(t) :
    if len(t)>3 :
        raise ValueError("length of tuple argument greater than 3")
        a, b, c = t
    if not(isinstance(a,(int, float))
    ) or not(isinstance(b,(int, float))
    ) or not(isinstance(c,(int, float))) :
            raise TypeError("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. Tyes d'exceptions

Voici quelques exceptions courantes ainsi que leurs utilisations

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 appliquée 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

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 générais 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 chaines de caractères suivantes (à copier-coller) :

- `½`
- `3²`

Réponse

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...

4. Les assertions

Une autre méthode pour interrompre explicitement un programme est d'utiliser des **assertions** créées avec l'instruction `assert`, sous la forme suivante :

```
assert expression booléenne, chaine de caractère avec message d'erreur
```

Cette instruction lèvera alors une `AssertionError`, avec le message passé entre parenthèse. Mais attention, le programme sera alors interrompu !

Les assertions sont souvent utilisées pour s'assurer que les **préconditions** d'une fonction sont bien remplies, c'est-à-dire que les arguments fournis correspondent bien aux exigences du programme.

Exemple

Dans le cadre d'une application graphique, on veut pouvoir déplacer un point $M(x;y)$ avec un vecteur $\vec{u}(a,b)$ avec x,y,a,b entiers. On veut créer une fonction `translation(p : tuple,v :tuple) -> tuple` qui prend en argument deux tuples de dimension 2, et qui renvoie le tuple obtenu en additionnant les éléments de même indice. Cependant, pour que la fonction soit utilisable, il faut impérativement que l'utilisateur ait bien fourni deux tuples de dimension 2 et d'entiers.

```
def translation(t : tuple , v : tuple) -> tuple :
    assert type(t) == tuple and len(t) == 2, "bad t argument, is not a tuple of length 2"
    assert type(v) == tuple and len(v) == 2, "bad v argument, is not a tuple of length 2"
    assert type(t[0]) == int and type(t[1]) == int, "tuple t must only contains integers"
    assert type(v[0]) == int and type(v[1]) == int, "tuple v must only contains integers"
    return (t[0]+v[0], t[1]+v[1])
```