Gestion des Exceptions

Introduction

En Python, la **gestion des exceptions** joue un rôle important dans la fiabilité et la robustesse des programmes. Une exception représente une situation inattendue qui survient pendant l'exécution, comme une division par zéro, l'accès à un fichier inexistant ou l'utilisation d'un type de donnée inapproprié. Lorsqu'une telle situation se produit, Python interrompt temporairement le déroulement normal du programme pour signaler l'erreur.

Et un programme ne doit JAMAIS s'arrêter pour une raison quelconque.

Sans mécanisme de gestion, ces erreurs provoqueraient l'arrêt immédiat du programme. C'est pourquoi Python fournit un système complet de **gestion des exceptions** qui permet non seulement de détecter les erreurs, mais aussi de les traiter de manière contrôlée. Grâce à ce système, un développeur peut anticiper les problèmes potentiels, afficher des messages explicites et assurer la continuité de l'exécution sans interruption brutale.

La gestion des exceptions repose sur l'emploi des mots-clés `try`, `except`, `else` et `finally`. Ensemble, ils permettent d'encadrer les zones de code susceptibles de générer des erreurs, d'intercepter celles qui surviennent, et de définir précisément les actions à entreprendre en cas d'anomalie. Ce mécanisme rend le code plus sûr, plus lisible et plus professionnel, en donnant la possibilité d'anticiper les imprévus et d'y répondre avec clarté.

Exemple

Dans la version suivante de la classe `Fraction`, on introduit une étape : **la validation des données** au moment de l'initialisation.

```
class Fraction:
    def __init__(self, numerateur, denominateur):
        if not isinstance(numerateur, int) or not isinstance(denominateur,
int):
            raise TypeError("Le numerateur et le denominateur doivent etre
des entiers")
        if denominateur == 0:
            raise ValueError("Le denominateur ne peut etre nul")
        self.numerateur = numerateur
        self.denominateur = denominateur

    def quotient(self):
        return self.numerateur / self.denominateur
```

La fonction `isinstance()`

La fonction `isinstance()` est une fonction intégrée de Python. Elle permet de **vérifier le type d'un objet** avant de l'utiliser. Sa syntaxe est la suivante :

```
isinstance(objet, type)
```

Elle renvoie `True` si l'objet correspond bien au type indiqué, et `False` sinon.

Dans notre code :

```
if not isinstance(numerateur, int) or not isinstance(denominateur, int):
    raise TypeError("Le numerateur et le denominateur doivent etre des
entiers")
```

on s'assure que les deux valeurs transmises, `numerateur` et `denominateur`, sont bien des entiers (`int`). En effet, une fraction ne peut contenir que des nombres entiers. C'est la définition même d'une fraction.

Si ce n'est pas le cas, on **interrompt immédiatement la création de l'objet** avec une erreur de type `TypeError`.

Pourquoi protéger l'initialisation des attributs

Lorsqu'on écrit une classe, il est important de **garantir la cohérence des objets** que l'on crée. Si un utilisateur essayait de faire :

```
f = Fraction(1.5, "toto")
```

sans cette vérification, l'objet serait créé avec des données incohérentes (`float` et `str`), ce qui provoquerait plus tard des erreurs difficiles à comprendre. En ajoutant des contrôles dès `__init__`, on s'assure que **chaque instance** de `Fraction` est valide dès le départ.

De la même manière :

```
if denominateur == 0:
    raise ValueError("Le denominateur ne peut etre nul")
```

empêche la création d'une fraction avec un dénominateur nul, ce qui serait **mathématiquement impossible** et causerait une erreur lors du calcul du quotient.

Le mot-clé `raise` et la gestion des erreurs

L'instruction **`raise`** permet de **déclencher volontairement une exception** lorsque quelque chose ne va pas. Ici, on soulève (`raise`) deux types d'exceptions différentes selon la nature du problème :

- `TypeError` → lorsque les types ne sont pas corrects,
- `ValueError` → lorsque la valeur fournie est inacceptable (comme un zéro au dénominateur).

Ces exceptions appartiennent à une **hiérarchie d'exceptions** dans Python. Toutes les erreurs héritent de la classe de base `Exception`, ce qui signifie qu'on peut intercepter une erreur précise ou plus générale selon le contexte.

Le bloc `try/except`

Imaginons le code suivant :

```
print("Début du programme")
tiers = Fraction(1, 0)
print("Fin du programme")
```

Lorsqu'on exécute ce code, Python affiche le message `"Début du programme"`, puis rencontre la ligne qui tente de créer une fraction avec un dénominateur nul. La méthode `__init__` de notre classe `Fraction` détecte cette situation impossible et **lève une exception** de type `ValueError` avec le message `"Le denominateur ne peut etre nul"`.

À ce moment précis, l'exécution du programme s'interrompt immédiatement : Python arrête tout et n'exécute **jamais** la ligne suivante (`print("Fin du programme")`). Autrement dit, le programme **plante** dès qu'une erreur non gérée survient.

Pour éviter ce comportement brutal, Python propose un mécanisme appelé **gestion des exceptions**, qui repose sur le bloc `try/except`. Il permet de **surveiller** une portion de code susceptible de générer une erreur et de **réagir** proprement si une exception survient, sans que le programme entier s'arrête.

Voici la version corrigée du code :

```
print("Début du programme")

try:
    tiers = Fraction(1, 0)
except ValueError as msg:
    print("Erreur :", msg)

print("Fin du programme")
```

Dans ce cas, Python commence l'exécution du bloc `try`. Il essaie de créer l'objet `tiers`, mais l'instruction `raise ValueError(...)` dans `__init__` déclenche une erreur. Au lieu de stopper le programme, Python **intercepte cette erreur** et passe directement dans le bloc `except`, où la variable `msg` contient le message de l'exception.

L'affichage sera alors :

```
Début du programme
Erreur : Le denominateur ne peut etre nul
Fin du programme
```

Le programme continue donc à s'exécuter normalement après avoir signalé l'erreur.

On peut aussi **intercepter plusieurs types d'exceptions** si l'on sait que plusieurs erreurs différentes peuvent se produire :

```
try:
    f = Fraction("a", 3)
except TypeError as msg:
    print("Erreur de type :", msg)
except ValueError as e:
    print("Erreur de valeur :", msg)
```

Ici, Python teste d'abord le bloc `try`. Si le premier argument n'est pas un entier (`"a"` dans cet exemple), une `TypeError` est levée, et c'est le premier `except` qui s'exécute. Si, au contraire, le dénominateur vaut zéro, c'est le second `except` qui prend le relais.

Ce système permet donc de **contrôler les erreurs**, de **personnaliser les messages** affichés à l'utilisateur et surtout de **garantir la stabilité du programme**. Même en cas d'anomalie, l'exécution se poursuit proprement sans interruption brutale.

Le bloc `else`

Le bloc `try/except` peut être enrichi avec deux autres parties importantes : **`else`** et **`finally`**, qui rendent la gestion des erreurs encore plus précise et plus lisible.

Le mot-clé `else` s'utilise pour exécuter du code **uniquement si aucune exception ne s'est produite** dans le bloc `try`. Autrement dit, le code contenu dans `else` ne sera exécuté que si tout s'est bien passé.

Exemple :

```
try:
    f = Fraction(1, 3)
except (TypeError, ValueError) as msg:
    print("Erreur :", msg)
else:
    print("La fraction a été créée avec succès :", f.quotient())
```

Voici ce qu'il se passe pas à pas :

1. Python exécute le code du bloc `try`.
2. Si une erreur est détectée, le bloc `except` s'exécute, et Python ignore complètement le `else`.
3. Si aucune erreur n'est levée, le bloc `else` s'exécute normalement.

L'intérêt du `else` est d'éviter de mélanger le code "normal" (ce qui se passe quand tout va bien) avec le code de gestion d'erreurs. Cela rend la structure du programme plus claire : le `try` pour ce qui peut échouer, le `except` pour les erreurs, et le `else` pour ce qui doit se passer en cas de succès.

Le bloc `finally`

Le mot-clé `finally` permet, lui, de définir une portion de code qui sera **toujours exécutée, qu'il y ait eu une erreur ou non.** Ce bloc est utile pour effectuer des actions de nettoyage ou de fermeture de ressources (par exemple, fermer un fichier ou une connexion réseau) même si une exception s'est produite.

Exemple :

```
try:
    f = Fraction(1, 0)
except ValueError as msg:
    print("Erreur :", msg)
else:
    print("La fraction a été créée avec succès :", f.quotient())
finally:
    print("Fin du traitement.")
```

Voici le déroulement :

- Python essaie d'exécuter le bloc `try`.
- Une erreur est levée (ici `ValueError`) → le bloc `except` s'exécute.
- Le bloc `else` est ignoré, car une exception s'est produite.
- **Le bloc `finally` s'exécute dans tous les cas**, même s'il y a eu une erreur.

L'affichage sera :

```
Erreur : Le denominateur ne peut etre nul
Fin du traitement.
```

Et si le code du `try` ne provoque aucune erreur :

```
La fraction a été créée avec succès : 0.3333
Fin du traitement.
```

Le bloc `finally` s'exécute **quoi qu'il arrive**, même si le programme rencontre une exception non gérée ou quitte le `try` avec un `return`.

En résumé

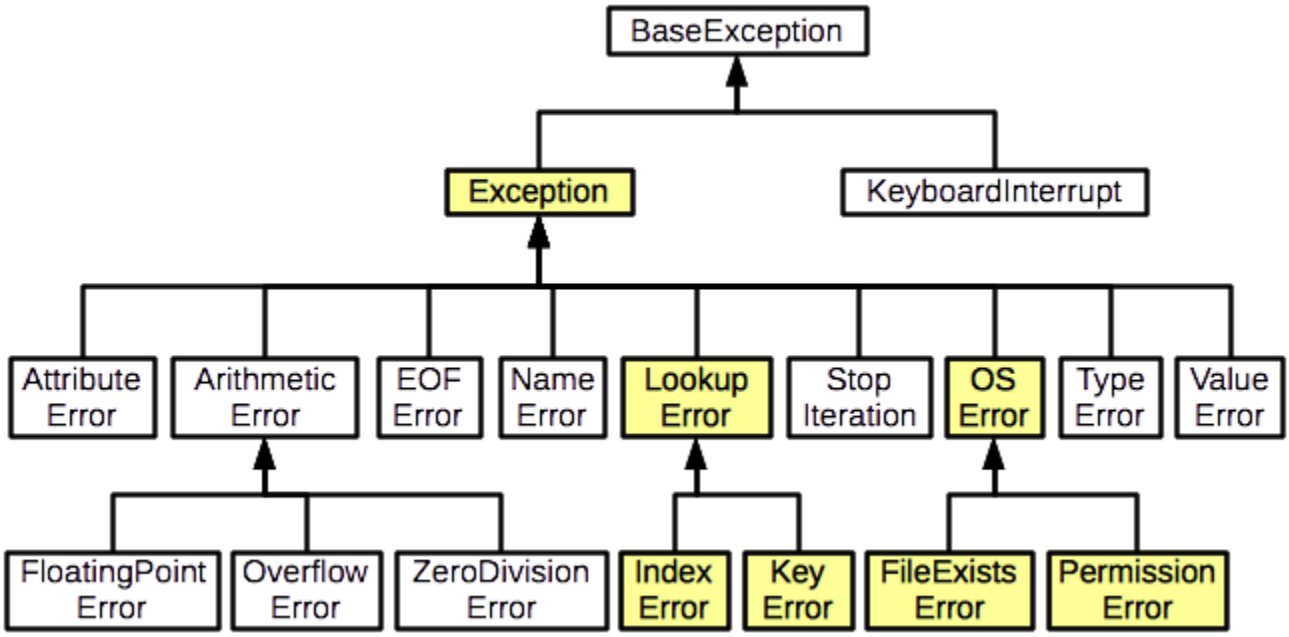
- **`try`** → contient le code à surveiller.
- **`except`** → intercepte et traite les erreurs.
- **`else`** → s'exécute seulement si tout s'est bien passé.
- **`finally`** → s'exécute toujours, qu'il y ait une erreur ou non.

Cette structure complète rend le code plus robuste et plus prévisible, tout en clarifiant le comportement du programme dans chaque situation possible.

Hierarchie des exceptions

En Python, toutes les erreurs sont organisées sous forme d'une **hiérarchie de classes**, un peu comme une arborescence où chaque type d'erreur hérite d'un type plus général.

Cette hiérarchie permet de **capturer plusieurs erreurs différentes** en n'utilisant qu'une seule catégorie commune.



Prenons cet exemple :

```
liste = [0, 1, 2, 3]
dico = {'nom': 'Doe'}

try:
    print(dico['prenom'])
    print(liste[7])
except (KeyError, IndexError) as msg:
    print(msg)
```

Voici ce qui se passe :

- La première ligne du bloc `try` provoque une **KeyError**, car la clé `'prenom'` n'existe pas dans le dictionnaire `dico`.
- Le programme s'arrête donc immédiatement à cette ligne et ne lit pas la suivante (`liste[7]`), puisque l'exception a déjà été levée.
- L'exception est interceptée par le bloc `except`, qui capture à la fois `KeyError` et `IndexError`.
- Le message d'erreur s'affiche alors sans que le programme plante.

Le bloc `except` fonctionne ici parce que les deux types d'erreurs possibles (`KeyError` pour un dictionnaire et `IndexError` pour une liste) ont été regroupés dans un même tuple :

```
except (KeyError, IndexError) as msg:
```

Mais il existe une approche encore plus élégante.

En effet, `KeyError` et `IndexError` sont deux sous-classes d'une même classe parente : `LookupError`. Cette classe regroupe toutes les erreurs liées à une **recherche invalide** dans une structure de données indexée ou associée, comme une liste ou un dictionnaire.

On peut donc simplifier le code en écrivant :

```
liste = [0, 1, 2, 3]
dico = {'nom': 'Doe'}

try:
    print(dico['prenom'])
    print(liste[7])
except LookupError as msg:
    print("Erreur de recherche :", msg)
```

Dans ce cas, `LookupError` capture aussi bien :

- les erreurs de type `KeyError`, lorsqu'on cherche une clé inexistante dans un dictionnaire,
- que les erreurs de type `IndexError`, lorsqu'on tente d'accéder à un indice inexistant dans une liste.

Ce mécanisme illustre parfaitement la **hiérarchie des exceptions** en Python : on peut intercepter soit une erreur très précise, soit un groupe d'erreurs plus large selon le niveau de généralité souhaité.

Ainsi :

- utiliser `KeyError` ou `IndexError` permet de traiter des cas spécifiques,
- utiliser `LookupError` permet de gérer d'un coup toutes les erreurs liées à la recherche d'un élément inexistant dans une collection.

Ce système hiérarchique rend la gestion des erreurs **plus souple et plus expressive**, en laissant au développeur le choix du niveau de précision adapté à chaque situation.

Pourquoi une hiérarchie d'exceptions ?

Une question très pertinente, et beaucoup d'apprenants se la posent au début : **pourquoi existe-t-il une hiérarchie d'exceptions**, et pourquoi ne pas simplement tout intercepter avec une seule commande comme :

```
except BaseException:
    print("Une erreur est survenue")
```

Pour comprendre l'intérêt de cette hiérarchie, il faut d'abord savoir que **toutes les erreurs en Python sont des objets**, et que ces objets sont organisés en classes, avec des relations de parenté.

Tout en haut de la hiérarchie, on trouve la classe `BaseException`, dont **toutes les autres exceptions héritent**. Elle est la racine de tout le système d'erreurs du langage.

Parmi ses sous-classes, on trouve par exemple :

- `Exception` (la plupart des erreurs courantes en dépendent),
- `SystemExit` (utilisée quand le programme se ferme proprement),
- `KeyboardInterrupt` (déclenchée quand on interrompt un programme avec `Ctrl + C`).

Sous `Exception`, on trouve des catégories plus précises, comme :

- `ArithmeticError` pour les erreurs mathématiques (`ZeroDivisionError` par exemple),
- `LookupError` pour les recherches invalides (`KeyError`, `IndexError`),
- `ValueError`, `TypeError`, etc.

Cette structure permet d'être **plus précis** dans le traitement des erreurs. On peut ainsi choisir de n'intercepter que les erreurs qui concernent le contexte dans lequel on se trouve.

Prenons un exemple :

```
try:
    resultat = 10 / 0
except ZeroDivisionError:
    print("Division par zéro interdite")
```

Ici, on ne capture que l'erreur liée à une division par zéro. Si une autre erreur survient (par exemple une erreur de type ou une erreur de fichier), elle remontera naturellement, car elle n'a rien à voir avec cette partie du programme.

Cela rend le code **plus sûr** : on ne masque pas des erreurs inattendues ou graves qui méritent d'être corrigées, au lieu d'être simplement ignorées.

Pourquoi il ne faut pas intercepter `BaseException` (ou même `Exception`) partout

Si vous attrapez toutes les exceptions avec :

```
try:
    # du code
except BaseException as e:
    print("Erreur :", e)
```

vous capturez **absolument tout**, y compris des erreurs que Python utilise pour fonctionner normalement. Par exemple :

- `KeyboardInterrupt` empêche l'utilisateur d'arrêter le programme avec `Ctrl + C`,

- `SystemExit` empêche le programme de se fermer proprement,
- ou encore d'autres erreurs système qui devraient interrompre l'exécution.

En interceptant tout sans distinction, on risque de **cacher des erreurs graves** ou de **bloquer des comportements normaux** du langage. Le programme semble "tenir bon", mais il devient imprévisible, voire impossible à déboguer.

Pour conclure :

- La **hiérarchie des exceptions** permet de choisir le bon niveau de précision : on peut gérer une erreur spécifique ou une famille entière d'erreurs.
- Elle rend le code plus **lisible, maîtrisé et sûr**, en évitant de masquer des problèmes inattendus.
- Utiliser `BaseException` ou même `Exception` de façon trop générale, c'est un peu comme dire "quelle que soit l'erreur, fais semblant que tout va bien", ce qui est dangereux, car on perd la trace de ce qui s'est réellement passé.

En d'autres termes : il vaut mieux **attraper uniquement ce que l'on sait gérer**, et laisser Python nous avertir pour tout le reste.