Gestion des erreurs

Introduction à la gestion des erreurs en Python

La gestion des erreurs, ou la gestion des exceptions, est un aspect important de la programmation. Les erreurs peuvent survenir lors de l'exécution d'un programme et peuvent être causées par des erreurs de saisie de l'utilisateur, des problèmes de connectivité réseau, des erreurs de calcul, etc. La gestion des erreurs permet de détecter et de gérer ces situations d'erreur de manière appropriée, ce qui améliore la fiabilité et la robustesse du code.

En Python, les erreurs sont gérées à l'aide du mécanisme des exceptions. Les exceptions sont des événements qui se produisent pendant l'exécution d'un programme et qui perturbent le flux normal d'exécution. Lorsqu'une exception se produit, le programme peut la capturer et la gérer de manière appropriée, au lieu de simplement planter ou générer un comportement inattendu.

Utilisation des blocs try-except

La structure de base pour la gestion des erreurs en Python utilise les blocs try et except. Le code susceptible de provoquer une exception est placé dans un bloc try, et les actions à effectuer en cas d'exception sont spécifiées dans un ou plusieurs blocs except. Voici la syntaxe générale :

```
try:
    # Code susceptible de provoquer une exception
except ExceptionType:
    # Action à effectuer en cas d'exception de type ExceptionType
```

Utilisation des blocs try-except

La structure de base pour la gestion des erreurs en Python utilise les blocs try et except. Le code susceptible de provoquer une exception est placé dans un bloc try, et les actions à effectuer en cas d'exception sont spécifiées dans un ou plusieurs blocs except. Voici la syntaxe générale :

```
try:
    # Code susceptible de provoquer une exception
except ExceptionType:
    # Action à effectuer en cas d'exception de type ExceptionType
```

Lorsque le code dans le bloc try est exécuté, si une exception de type ExceptionType se produit, le code dans le bloc except correspondant est exécuté.

Gestion des exceptions spécifiques

Voici quelques exemples d'exceptions courantes :

ZeroDivisionError : Cette exception est levée lorsque vous essayez de diviser par zéro.
ValueError : Cette exception est levée lorsque vous fournissez une valeur incorrecte à une fonction ou à une opération.
TypeError : Cette exception est levée lorsque vous effectuez une opération sur des types de données incompatibles.
FileNotFoundError : Cette exception est levée lorsque vous essayez d'accéder à un fichier qui n'existe pas.

```
try:
    num = int(input("Entrez un nombre : "))
    result = 100 / num
    print("Le résultat est :", result)
except ZeroDivisionError:
    print("Erreur : Division par zéro")
```

Gestion des exceptions multiples

Il est possible de gérer plusieurs types d'exceptions dans un bloc try-except. Vous pouvez spécifier plusieurs blocs except pour capturer et traiter différentes exceptions. Voici la syntaxe générale :

```
try:
    # Code susceptible de provoquer une exception
except ExceptionType1:
    # Action à effectuer en cas d'exception de type ExceptionType1
except ExceptionType2:
    # Action à effectuer en cas d'exception de type ExceptionType2
...
except ExceptionTypeN:
    # Action à effectuer en cas d'exception de type ExceptionTypeN
```

Utilisation du bloc else

En plus des blocs try et except, vous pouvez également utiliser un bloc else facultatif après les blocs except. Le code dans le bloc else est exécuté uniquement si aucune exception ne se produit dans le bloc try. Voici un exemple :

```
try:
    num = int(input("Entrez un nombre positif : "))
    if num <= 0:
        raise ValueError("Le nombre doit être positif")
except ValueError as err:
    print("Erreur :", str(err))
else:
    print("Le nombre est valide.")
```

Utilisation du bloc finally

En plus des blocs try, except et else, vous pouvez également utiliser un bloc finally facultatif. Le code dans le bloc finally est toujours exécuté, que des exceptions se produisent ou non dans le bloc try. Le bloc finally est souvent utilisé pour effectuer des opérations de nettoyage ou de fermeture, telles que la fermeture d'un fichier ou la libération de ressources.

Voici un exemple qui illustre l'utilisation du bloc finally

```
file = None
try:
    file = open("data.txt", "r")
    # Effectuer des opérations sur le fi
finally:
    if file:
        file.close()
```

Dans cet exemple, le fichier "data.txt" est ouvert dans le bloc try pour effectuer des opérations de lecture. Le bloc finally garantit que le fichier est fermé, quelle que soit l'issue du bloc try. Si le fichier a été ouvert avec succès, il est fermé à l'aide de la méthode close().

Gestion des exceptions personnalisées

En plus des exceptions prédéfinies, vous pouvez également créer vos propres exceptions personnalisées en définissant des classes dérivées de la classe de base Exception. Cela vous permet de gérer des situations d'erreur spécifiques à votre application.

```python
class CustomError(Exception):
    pass

try:
    age = int(input("Entrez votre âge : "))
    if age < 0:
        raise CustomError("L'âge ne peut pas être négatif")
except CustomError as err:
    print("Erreur personnalisée :", str(err))
```

Gestion des exceptions personnalisées

```python
class CustomError(Exception):
    def __init__(self, message):
        self.message = message

    def __str__(self):
        return f"Erreur personnalisée : {self.message}"

    def __repr__(self):
        return f"CustomError('{self.message}')"

def division_secure(a, b):
    try:
        if b == 0:
            raise CustomError("Division par zéro est interdite")
        result = a / b
        return result
    except CustomError as e:
        print(e)  # Affiche le message personnalisé de l'exception
        return None
    except Exception as e:
        print("Erreur :", str(e))
        return None

# Tests
print(division_secure(10, 0))  # Output: Erreur personnalisée : Division par zéro est interdite
print(repr(CustomError("Erreur personnalisée")))  # Output: CustomError('Erreur personnalisée')
```