JOUR 1_3

Gestion avancée des exceptions et des erreurs

Dans tous les programmes, il est possible qu'une erreur se produise. La gestion des erreurs consiste à prendre des mesures pour traiter les erreurs qui se produisent dans votre code.

Types d'erreurs

Il existe deux types principaux d'erreurs :

- **Les erreurs syntaxiques** se produisent lorsque votre code ne respecte pas la syntaxe du langage Python. Ces erreurs sont détectées par le compilateur ou l'interpréteur Python et se traduisent par un message d'erreur.
- **Les erreurs sémantiques** se produisent lorsque votre code est syntaxiquement correct, mais qu'il ne fait pas ce que vous attendez. Ces erreurs peuvent être difficiles à détecter et peuvent entraîner des comportements inattendus de votre programme.

Gestion des erreurs syntaxiques

La gestion des erreurs syntaxiques est automatique. Le compilateur ou l'interpréteur Python détectera les erreurs syntaxiques et affichera un message d'erreur.

Gestion des erreurs sémantiques

La gestion des erreurs sémantiques est plus complexe. Il existe plusieurs techniques que vous pouvez utiliser pour gérer les erreurs sémantiques :

- **Les assertions** permettent de vérifier des conditions dans votre code. Si une condition est fausse, une assertion lève une erreur.
- **Les exceptions** sont des objets qui représentent une erreur. Vous pouvez utiliser des exceptions pour gérer les erreurs dans votre code.
- **Les tests unitaires** sont des tests automatisés qui permettent de tester votre code et de détecter les erreurs.

Assertions

Les assertions permettent de vérifier des conditions dans votre code. Si une condition est fausse, une assertion lève une erreur.

Par exemple, la fonction suivante utilise une assertion pour vérifier que la valeur d'un paramètre est supérieure à 0 :

```
def division(nombre1, nombre2):
    assert nombre2 > 0, "Le second nombre doit être supérieur
à 0"
    return nombre1 / nombre2
```

Si la valeur du paramètre `nombre2` est égale ou inférieure à 0, l'assertion lèvera l'erreur suivante :

```
AssertionError: Le second nombre doit être supérieur à 0
```

Exceptions

Les exceptions sont des objets qui représentent une erreur. Vous pouvez utiliser des exceptions pour gérer les erreurs dans votre code.

Pour lever une exception, vous utilisez le mot-clé `raise`. Par exemple, la fonction suivante lève une exception si la valeur d'un paramètre est égale à 0 :

```
def division(nombre1, nombre2):
    if nombre2 == 0:
        raise ValueError("Le second nombre ne peut pas être
égal à 0")
    return nombre1 / nombre2
```

Si la valeur du paramètre `nombre2` est égale à 0, l'exception `ValueError` sera levée.

Pour gérer les exceptions, vous pouvez utiliser des blocs `try` / `except`. Un bloc `try` / `except` permet de tester si une exception est levée. Si une exception est levée, le bloc `except` est exécuté.

Par exemple, le code suivant utilise un bloc `try` / `except` pour gérer les exceptions levées par la fonction `division()` :

```
try:
    resultat = division(10, 0)
except ValueError:
    print("Le second nombre ne peut pas être égal à 0")
else:
    print("Le résultat est", resultat)
```

Si la valeur du paramètre `nombre2` est égale à 0, l'exception `ValueError` sera levée et le bloc `except` sera exécuté. Sinon, le bloc `else` sera exécuté.

Tests unitaires

Les tests unitaires sont des tests automatisés qui permettent de tester votre code et de détecter les erreurs.

Les tests unitaires sont un excellent moyen de gérer les erreurs sémantiques. Ils vous permettent de tester votre code de manière

exhaustive et de détecter les erreurs avant qu'elles ne se produisent dans un environnement de production.

Pour écrire des tests unitaires, vous pouvez utiliser une bibliothèque de tests unitaires, telle que `unittest`.

Hierarchie des exceptions et gestion de contexte

En Python, les exceptions sont des objets qui représentent une erreur. Elles sont utilisées pour gérer les erreurs dans votre code.

Les exceptions sont hiérarchisées. Cela signifie que certaines exceptions sont des sous-classes d'autres exceptions. Par exemple, l'exception `ValueError` est une sous-classe de l'exception `Exception`.

Hiérarchie des exceptions

La hiérarchie des exceptions en Python est la suivante :

```
Exception
    |- BaseException
        |- SystemExit
        |- KeyboardInterrupt
        |- GeneratorExit
        |- Exception
            |- StopIteration
            |- OSError
                |- IOError
                |- OSError
                |- ...
            |- ArithmeticError
                |- ZeroDivisionError
                |- OverflowError
                |- ...
```

```
            |- ValueError
                |- UnicodeDecodeError
                |- UnicodeEncodeError
                |- ...
            |- KeyError
            |- IndexError
            |- ...
            |- AttributeError
            |- NameError
            |- ...
```

La classe `Exception` est la classe mère de toutes les exceptions. Les exceptions `BaseException`, `SystemExit`, `KeyboardInterrupt`, et `GeneratorExit` sont des exceptions spéciales qui ne sont pas levées par le code utilisateur.

Les exceptions `StopIteration`, `OSError`, `ArithmeticError`, `ValueError`, `KeyError`, `IndexError`, `AttributeError`, et `NameError` sont des exceptions courantes qui sont levées par le code utilisateur.

Gestion de contexte

La gestion de contexte est une technique qui permet de gérer les ressources de manière sûre et propre.

La gestion de contexte utilise les blocs `with` / `as`. Un bloc `with` / `as` permet de créer un contexte et de le gérer de manière sûre et propre.

Exemple

L'exemple suivant utilise la gestion de contexte pour ouvrir et fermer un fichier :

```
with open("fichier.txt", "r") as f:
    contenu = f.read()
```

Ce code ouvrira le fichier `fichier.txt` en mode lecture et assignera le contenu du fichier à la variable `contenu`. Lorsque le bloc `with` sera terminé, le fichier sera fermé automatiquement.

Conclusion

La gestion des exceptions et la gestion de contexte sont deux techniques importantes de la programmation Python. Elles vous permettent de garantir que votre code fonctionne correctement, même en cas d'erreur, et de gérer les ressources de manière sûre et propre.

Recommandations

Voici quelques recommandations pour la gestion des exceptions et la gestion de contexte :

- **Gérez toutes les exceptions.** Ne laissez pas les exceptions se propager sans être gérées.
- **Utilisez la gestion de contexte pour gérer les ressources.** Cela vous aidera à éviter les fuites de mémoire et d'autres problèmes.
- **Rendez votre code plus lisible en utilisant des blocs `try` / `except` et `with` / `as` explicites.**

Création de vos propres exceptions

Vous pouvez créer vos propres exceptions pour représenter des erreurs spécifiques à votre code. Cela vous permet de mieux contrôler la façon dont les erreurs sont gérées.

Création d'une exception

Pour créer une exception, vous devez créer une classe qui hérite de la classe `Exception`.

Voici un exemple de création d'une exception :

```
class MaPropreException(Exception):
    pass
```

Cette classe crée une exception appelée `MaPropreException`. Cette exception est une sous-classe de l'exception `Exception`.

Attributs d'une exception

Les exceptions peuvent avoir des attributs. Ces attributs peuvent être utilisés pour fournir des informations sur l'erreur.

Voici un exemple d'utilisation d'attributs dans une exception :

```
class MaPropreException(Exception):
    def __init__(self, message, code):
        super().__init__(message)
        self.code = code
```

Cette classe définit deux attributs :

- `message` : le message de l'erreur
- `code` : un code d'erreur

Lever une exception

Pour lever une exception, vous utilisez le mot-clé `raise`.

Voici un exemple de levée d'une exception :

```
try:
    raise MaPropreException("L'erreur s'est produite", 123)
except MaPropreException as e:
    print(e.message, e.code)
```

Ce code lève une exception `MaPropreException` avec le message "L'erreur s'est produite" et le code d'erreur 123.

Gestion d'une exception

Pour gérer une exception, vous pouvez utiliser des blocs `try` / `except`.

Voici un exemple de gestion d'une exception :

```
try:
    raise MaPropreException("L'erreur s'est produite", 123)
except MaPropreException as e:
    print(e.message, e.code)
```

Ce code lève une exception `MaPropreException`. Le bloc `except` est exécuté et imprime le message et le code d'erreur de l'exception.

Conclusion

La création de vos propres exceptions vous permet de mieux contrôler la façon dont les erreurs sont gérées. Cela peut être utile pour représenter des erreurs spécifiques à votre code.

Utilisation des gestionnaires de contexte (with statement)

Les gestionnaires de contexte (with statement) sont une fonctionnalité de Python qui permet de gérer les ressources de manière sûre et propre.

Les ressources peuvent être des objets tels que des fichiers, des bases de données, des sockets, ou d'autres ressources qui doivent être fermées ou libérées après utilisation.

Syntaxe

La syntaxe des gestionnaires de contexte est la suivante :

```
with <objet> as <variable>:
    # Code à exécuter dans le contexte
```

L'objet `<objet>` est la ressource que vous souhaitez gérer. La variable `<variable>` est une variable qui contient la ressource pendant l'exécution du code dans le contexte.

Exemple

L'exemple suivant utilise un gestionnaire de contexte pour ouvrir et fermer un fichier :

```
with open("fichier.txt", "r") as f:
    contenu = f.read()
```

Ce code ouvrira le fichier `fichier.txt` en mode lecture et assignera le contenu du fichier à la variable `contenu`. Lorsque le bloc `with` sera terminé, le fichier sera fermé automatiquement.

Fonctionnement

Lorsque vous exécutez un bloc `with`, Python appelle la méthode `__enter__()` de l'objet `<objet>`. La méthode `__enter__()` est responsable de l'initialisation de la ressource.

Si la méthode `__enter__()` ne lève aucune exception, Python exécute le code dans le bloc `with`.

Lorsque le bloc `with` est terminé, Python appelle la méthode `__exit__()` de l'objet `<objet>`. La méthode `__exit__()` est responsable de la fermeture ou de la libération de la ressource.

Avantages

Les gestionnaires de contexte présentent plusieurs avantages :

- **Sécurité :** Les gestionnaires de contexte garantissent que les ressources seront fermées ou libérées, même en cas d'erreur.
- **Lissibilié :** Les gestionnaires de contexte rendent le code plus lisible et plus facile à maintenir.
- **Efficacité :** Les gestionnaires de contexte peuvent améliorer l'efficacité de votre code en évitant les fuites de mémoire et autres problèmes.

Recommandations

Voici quelques recommandations pour l'utilisation des gestionnaires de contexte :

- **Utilisez des gestionnaires de contexte pour toutes les ressources que vous utilisez.**
- **Gérez les exceptions dans le bloc `except` du gestionnaire de contexte.**
- **Utilisez la méthode `contextlib.closing()` pour créer facilement des gestionnaires de contexte pour des objets qui n'en ont pas.**

Conclusion

Les gestionnaires de contexte sont une fonctionnalité puissante qui peut vous aider à écrire du code Python plus sûr, plus lisible et plus efficace.