Gestion avancée des erreurs et modules personnalisés

Python B2 J3 - 2025/2026

Formation Python niveau intermédiaire : maîtrise des exceptions, création de modules et tests unitaires pour un code robuste et maintenable.

Colin Tenaguillo
formateur_colin.tenaguilo@supdevinci-edu.fr

La gestion complète des exceptions

Python offre un système complet pour gérer les erreurs avec **try/except/else/finally**. Chaque bloc a un rôle spécifique dans la gestion des erreurs.

1. **try**

   Code susceptible de lever une exception

2. **except**

   Gestion spécifique des erreurs

3. **else**

   Exécuté si aucune exception

4. **finally**

   Toujours exécuté (nettoyage)

Exemple pratique : lecture de fichier

```
try:
    fichier = open("data.txt", "r")
    contenu = fichier.read()
except FileNotFoundError:
    print("Fichier non trouvé !")
except PermissionError:
    print("Permission refusée !")
else:
    print("Lecture réussie !")  # Si pas d'exception
finally:
    fichier.close()  # Toujours exécuté
```

Cette structure garantit une gestion propre des ressources même en cas d'erreur.

Lever des exceptions personnalisées

Utilisez `raise` pour déclencher vos propres exceptions quand les conditions ne sont pas respectées.

```
def diviser(a: float, b: float) -> float:
    if b == 0:
        raise ValueError("Division par zéro impossible")
    return a / b

try:
    resultat = diviser(10, 0)
except ValueError as e:
    print(f"Erreur: {e}")
```

Le type d'exception choisi doit être cohérent avec la nature de l'erreur (ValueError, TypeError, etc.).

Créer des exceptions personnalisées

Définition

```
class ErreurBibliotheque(Exception):
    """Exception personnalisée"""
    def __init__(self, message: str,
                 code_erreur: int = 0):
        super().__init__(message)
        self.code_erreur = code_erreur
```

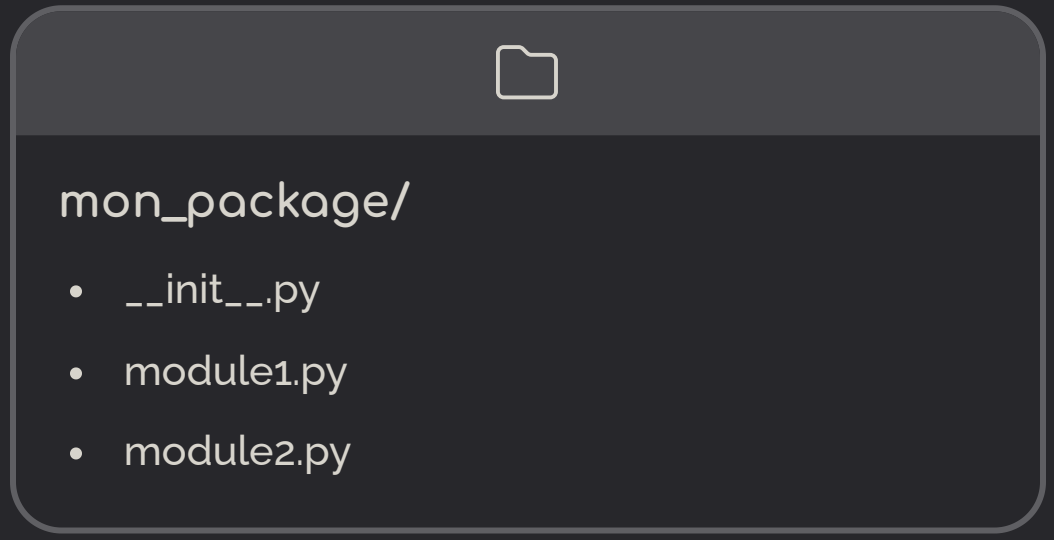
Utilisation

```
raise ErreurBibliotheque(
    "ISBN déjà existant",
    code_erreur=1001
)
```

Les exceptions personnalisées permettent une gestion d'erreur plus précise et métier.

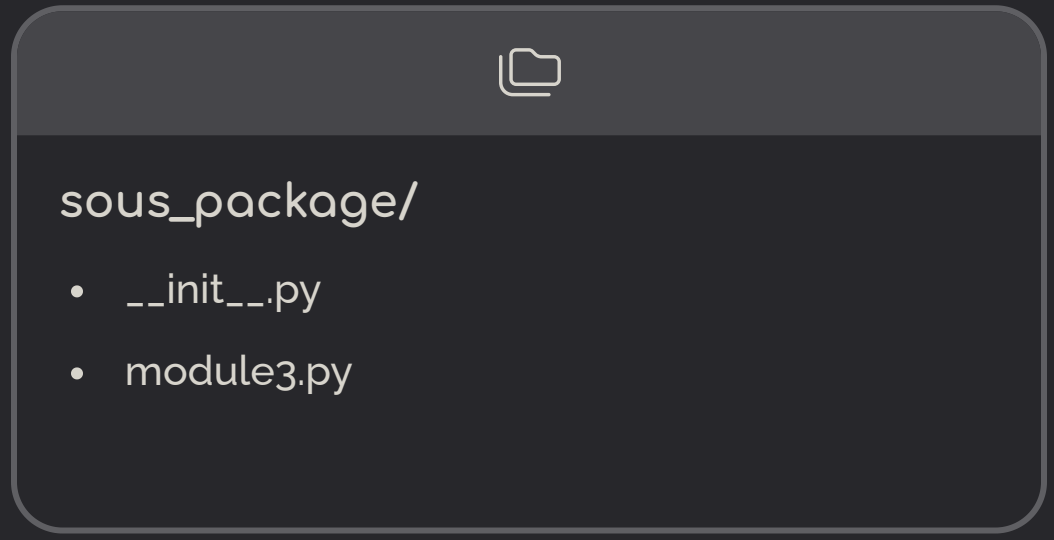
Structure des modules et packages

Un package Python est un dossier contenant un fichier `__init__.py` et des modules.



mon_package/

- __init__.py
- module1.py
- module2.py



sous_package/

- __init__.py
- module3.py

Cette organisation facilite la réutilisabilité et la maintenance du code.

Imports relatifs et absolus

Imports relatifs

```
# Dans module2.py
from .module1 import ma_fonction
from .sous_package.module3 import
autre_fonction
```

Le point (.) indique le package courant

Imports absolus

```
# Dans le programme principal
from mon_package.module1 import
ma_fonction
from mon_package import module2
```

Chemin complet depuis la racine

Préférez les imports absolus pour plus de clarté, sauf dans les packages complexes.

TP Pratique

- class ErreurBibliotheque
- Mettre en place la gestion d'erreur sur la partie gestion de fichier, ou plus
- Split vos fichiers
  - main.py
  - models.py (Class Livre. LivreNumerique, Bibliotheque)
  - file_manager.py (Class BibliothequeAvecFichier(Bibliotheque))
  - exceptions.py (fichiers d'exceptions)

Tests unitaires

Avec pytest - L'après-midi

Les tests unitaires garantissent la fiabilité de votre code et facilitent sa maintenance à long terme.

Pourquoi écrire des tests ?

Vérification

Confirmer le bon fonctionnement de chaque composant

Détection

Identifier les régressions lors des modifications

Maintenance

Faciliter les évolutions et corrections

Documentation

Illustrer l'usage attendu du code

Pourquoi pytest plutôt qu'unittest ?

pytest offre une expérience de test moderne et flexible, avec plusieurs avantages clés par rapport au module `unittest` intégré de Python.

Syntaxe simplifiée

Utilise des assertions Python natives (`assert`), rendant les tests plus lisibles et naturels à écrire.

Fixtures puissantes

Permet une configuration complexe et réutilisable des prérequis de test, injectée directement dans les fonctions de test.

Paramétrisation native

Exécute facilement le même test avec différents jeux de données, réduisant la duplication de code et augmentant la couverture.

Écosystème riche

Vaste collection de plugins pour des fonctionnalités étendues comme des rapports avancés, des couvertures de code, et l'intégration continue.

Messages d'erreur clairs

Fournit des informations détaillées et concises sur les échecs de test, facilitant grandement le processus de débogage.

Standard industriel

Largement adopté et préféré dans l'industrie Python, ce qui facilite la collaboration et l'intégration dans des projets existants.

Installation et configuration de pytest

Pour démarrer avec pytest, il faut d'abord l'installer, puis configurer votre projet pour qu'il détecte correctement vos fichiers de test.

Installation

```
# Installation
pip install pytest pytest-cov
```

Installez les packages `pytest` et `pytest-cov` (pour la couverture de code) en utilisant `pip`.

Configuration (pyproject.toml)

```
[tool.pytest.ini_options]
testpaths = ["tests"]
python_files = ["test_*.py"]
python_classes = ["Test*"]
python_functions = ["test_*"]
```

Le fichier `pyproject.toml` (ou `pytest.ini`) permet de définir où trouver les tests et quelles conventions de nommage utiliser.

Structure des tests avec pytest

Pour une organisation efficace, les tests pytest sont généralement regroupés dans un répertoire `tests/` à la racine de votre projet. Cette structure facilite leur découverte et leur exécution.

```
bibliotheque/
├── __init__.py
├── models.py
├── file_io.py
└── tests/
    ├── __init__.py
    ├── conftest.py        # Fixtures partagées
    ├── test_models.py     # Tests des modèles
    └── test_file_io.py    # Tests I/O
```

Les fichiers de test doivent commencer par `test_` (ex: `test_models.py`) et `conftest.py` est utilisé pour définir des fixtures partagées, rendant vos tests plus modulaires et réutilisables.

Premier test pytest

```
# tests/test_models.py
from bibliotheque.models import Bibliotheque, Livre

def test_creation_bibliotheque():
    """Test simple avec assertions normales"""
    biblio = Bibliotheque("Test")
    assert biblio.nom == "Test"
    assert len(biblio) == 0

def test_ajouter_livre():
    biblio = Bibliotheque("Test")
    livre = Livre("Python", "A. Dev", "123")

    resultat = biblio.ajouter_livre(livre)

    assert resultat is True
    assert len(biblio) == 1
    assert livre in biblio.livres
```

Voici un premier exemple simple de test pytest pour la gestion d'une bibliothèque. Ce fichier, nommé tests/test_models.py, contient des tests pour la création de la bibliothèque et l'ajout de livres, illustrant la simplicité de la syntaxe pytest.

Les fonctions de test doivent commencer par test_ pour être automatiquement découvertes par pytest. L'utilisation directe de l'instruction assert rend le code de test concis et facile à lire.

Fixtures pytest - Réutilisation du code

Les fixtures pytest sont des fonctions spéciales qui fournissent un contexte réutilisable pour vos tests. Elles sont idéales pour configurer un état initial (setup) et le nettoyer (teardown), garantissant des environnements de test isolés et cohérents.

Définition des Fixtures (conftest.py)

```
# tests/conftest.py
import pytest
from bibliotheque.models import Bibliotheque, Livre

@pytest.fixture
def bibliotheque_vide():
    """Fixture pour une bibliothèque vide."""
    return Bibliotheque("Bibliothèque Test")

@pytest.fixture
def livre_python():
    """Fixture pour un livre 'Python'."""
    return Livre("Python", "A. Developer", "123-456")
```

Utilisation dans les Tests

```
# tests/test_models.py
def test_avec_fixtures(bibliotheque_vide, livre_python):
    """Teste l'ajout d'un livre à une bibliothèque vide."""
    bibliotheque_vide.ajouter_livre(livre_python)

    assert len(bibliotheque_vide) == 1
    assert livre_python in bibliotheque_vide.livres
```

En nommant une fixture comme argument d'une fonction de test, pytest la détecte et l'exécute automatiquement avant le test, simplifiant grandement la gestion des prérequis.

Ce mécanisme centralise la logique de setup, rendant vos tests plus modulaires, plus faciles à lire et à maintenir.

Tester les exceptions avec pytest

`pytest.raises()` est un outil puissant pour tester le comportement de votre code lorsqu'il rencontre des erreurs. Il permet de vérifier qu'une fonction lève l'exception attendue dans des conditions spécifiques, garantissant ainsi la robustesse de votre application.

```
# tests/test_bibliotheque.py
from bibliotheque.models import ErreurBibliotheque, Bibliotheque, Livre
import pytest

def test_exception_livre_doublon(bibliotheque_vide, livre_python):
    bibliotheque_vide.ajouter_livre(livre_python)

    # Créons un livre avec un ISBN déjà existant pour provoquer l'erreur
    livre_doublon = Livre("Autre Titre", "Autre Auteur", "123-456")

    # Vérifier que l'exception ErreurBibliotheque est levée
    with pytest.raises(ErreurBibliotheque) as exc_info:
        bibliotheque_vide.ajouter_livre(livre_doublon)

    # Assurer que le code d'erreur est correct
    assert exc_info.value.code_erreur == 1001
    # Assurer que le message de l'exception contient le texte attendu
    assert "existe déjà" in str(exc_info.value)
```

Le gestionnaire de contexte `with pytest.raises(ErreurBibliotheque) as exc_info:` est la clé ici. Il exécute le code à l'intérieur et vérifie si une instance de `ErreurBibliotheque` est levée.

Si l'exception est levée comme prévu, le test continue. Sinon, ou si une exception différente est levée, le test échoue. L'objet exception est capturé dans `exc_info`, ce qui permet d'effectuer des assertions détaillées sur ses attributs, comme son `code_erreur` ou son message, pour une vérification précise.

Tests paramétrés

```
import pytest

@pytest.mark.parametrize("titre,auteur,isbn,attendu", [
    ("Python", "A. Dev", "111", True),
    ("", "A. Dev", "222", False),  # Titre vide
    ("Java", "", "333", False),    # Auteur vide
])
def test_validation_livre(titre, auteur, isbn, attendu):
    """Test avec multiples jeux de données"""
    # Implémentation de la validation
    est_valide = bool(titre and auteur and isbn)
    assert est_valide == attendu
```

Les tests paramétrés avec `pytest.mark.parametrize` permettent d'exécuter une même fonction de test plusieurs fois avec différents jeux de données. Cela réduit considérablement la duplication de code et rend vos tests plus concis et maintenables.

Dans cet exemple, la fonction `test_validation_livre` est exécutée trois fois, testant la logique de validation d'un livre avec des scénarios valides et invalides (titre ou auteur manquant), tout en vérifiant le résultat attendu.

Marqueurs Pytest - Organisation et Exécution Sélective

```
import pytest

@pytest.mark.slow
def test_import_gros_fichier():
    """Test marqué comme lent - peut être exclu lors des exécutions rapides"""
    # Ce test pourrait impliquer le traitement de grandes quantités de données
    # ou des requêtes réseau prolongées.
    pass

@pytest.mark.integration
def test_cycle_complet_utilisateur():
    """Test d'intégration simulant un flux utilisateur complet"""
    # Ce test vérifierait l'interaction entre plusieurs composants du système.
    pass

# Exécution sélective depuis la ligne de commande :
# pytest -m "not slow"    # Exclut tous les tests marqués comme 'slow'
# pytest -m integration   # Exécute uniquement les tests marqués comme 'integration'
# pytest -m "slow or integration" # Exécute les tests lents OU d'intégration
```

TP - Mise en place pytest

Cet atelier pratique vise à vous guider dans la configuration d'un environnement de test robuste en utilisant pytest, en suivant ces étapes clés :

1

Installation de pytest

Installez pytest et ses dépendances via pip. C'est la première étape essentielle pour commencer à écrire des tests.

2

Création de **tests/conftest.py**

Ce fichier centralisera vos fixtures réutilisables, fournissant un contexte isolé et cohérent pour vos tests.

3

Implémentation de **test_models.py**

Rédigez les tests unitaires pour la logique de vos modèles (par exemple, création de bibliothèque, ajout de livres).

4

Implémentation de **test_file_io.py**

Développez des tests spécifiques pour les fonctions gérant les opérations d'entrée/sortie de fichiers (par exemple, lecture, écriture).

5

Configuration de **pyproject.toml**

Optimisez l'exécution de pytest et définissez des options spécifiques à votre projet dans ce fichier de configuration.

Commandes Pytest Essentielles

Pytest fournit une ligne de commande intuitive pour exécuter vos tests, filtrer les exécutions et générer des rapports. Voici quelques-unes des commandes les plus utiles :

```
# Exécute tous les tests découverts
pytest

# Exécute tous les tests avec un niveau de verbosité élevé
pytest -v

# Exécute uniquement les tests contenus dans un fichier spécifique
pytest tests/test_models.py

# Exécute un test spécifique dans un fichier
pytest tests/test_models.py::test_ajouter_livre

# Exécute les tests et génère un rapport de couverture de code
pytest --cov=bibliotheque

# Exécute les tests et génère un rapport de couverture HTML détaillé
pytest --cov=bibliotheque --cov-report=html
```

Ces commandes vous permettent de contrôler précisément l'exécution de vos tests, du simple lancement à l'analyse avancée de la couverture de code, essentielle pour garantir la qualité de votre application.