Gestion des erreurs en Python

BO NSI

Mise au point des programmes. Gestion des bugs.	Dans la pratique de la programmation, savoir répondre aux causes typiques de bugs : problèmes liés au typage, effets de bord non désirés, débordements dans les tableaux, instruction conditionnelle non exhaustive, choix des inégalités, comparaisons et calculs entre flottants, mauvais nommage des variables, etc.	On prolonge le travail entrepris en classe de première sur **l'utilisation de la spécification, des assertions, de la documentation des programmes et de la construction de jeux de tests.** Les élèves apprennent progressivement à anticiper leurs erreurs.

Table des matières

1. Introduction

Lors de l'exécution d'un programme, différentes erreurs peuvent produire des erreurs :

- le code source est mal écrit, ne respectant pas la syntaxe du Python,
- des opérations interdites sont employées, comme une division par zéro
- une logique déficiente qui produits des résultats erronés ou inattendus

Ce chapitre présente comment *gérer les erreurs* dans un programme, notamment avec le mécanisme d'*exception* utilisé en programmation orientée objet.

2. Gestion des erreurs

2.1. Trois types d'erreurs principaux :

- **Les erreurs de syntaxe** : code du programme mal formé ex absence des « : »
- **Les erreurs d'exécution** : programme syntaxiquement correct mais effectuant une opération interdite : division par zéro, concaténation d'une chaine de caractères avec un autre type …
- **Les erreurs de logique** : le programme s'exécute sans erreur mais ne renvoie pas la bonne réponse : perimetre = largeur + 2* longueur

Pour les erreurs de syntaxe en général les indications de l'interpréteur Python permettent de remonter à la source de l'erreur.

Pour les erreurs d'exécution on trouve les plus souvent les situations suivantes :

- Une opération arithmétique ne peut pas être effectuée : division par zéro (`ZeroDivisionError`), racine carrée d'un nombre négatif (`ValueError`).
- Un opérateur ou une fonction est utilisé avec une donnée ou variable du mauvais type (`TypeError`).
- Un package n'a pas su être importé (`ImportError`).
- Une variable ou une fonction avec le nom précisé n'a pas su être trouvé (`NameError`).
- Un accès à un élément d'une séquence ne peut pas être effectué : mauvais indice dans une liste (`IndexError`), clé inexistante dans un dictionnaire (`KeyError`).
- Le nombre maximal d'appels récursifs a été atteint (`RecursionError`).

Pour les erreurs de logique il faut mettre en œuvre des tests unitaires. Ce qui consiste à tester chaque brique de code séparément (fonction, classe, méthode) pour vérifier que cellui -ci prend bien en compte les bons arguments et renvoie un résultat correct et fonctionnel. Python est fourni avec un Framework qui permet d'automatiser ces tests : **unittest. En ce qui concerne le programme de NSI on se contente d'utiliser à la main un système d'assertion et de levée d'exceptions.**

Une assertion est une **expression qui doit être évaluée comme vraie**. Si cette évaluation échoue elle peut mettre fin à l'exécution du programme, ou bien lancer une exception.

Une exception est un évènement qui apparaît pendant le déroulement d'un programme et qui empêche la poursuite normale de son exécution.

Enfin une bonne méthode de gestion des erreurs est de fournir une documentation dans l'exemple qui suit on se contentera d'une fonction particulière.

2.2. Exemple de gestion d'erreurs : calcul du taux de réussite d'un test :

On désire calculer le taux de réussite d'un test. Dans un premier temps on écrit un code basique sans réfléchir aux spécifications (conditions de départ et des conditions de retour).

Ex

```
def pourcentage(score, total):
    return score / total * 100

print(f"Alexis a obtenu {pourcentage(18, 20) }")
print(f"Louis a obtenu {pourcentage(18, 0) }")
```

```
Alexis a obtenu 90.0
Traceback (most recent call last):
File " \gestion_erreurs_ex1.py", line 5, in <module>
print(f"Louis a obtenu {pourcentage(18, 0) }")
File " \gestion_erreurs_ex1.py", line 2, in pourcentage
return score / total * 100
ZeroDivisionError: division by zero
```

Le message affiché contient le `traceback`, c'est à dire la pile d'appel — le chemin parcouru par l'interpréteur pour atteindre l'erreur (soit la liste des fonctions traversées pour atteindre l'erreur). Ce message comporte le **type d'exception** levée (`ZeroDivisionError`, `IOError`, `NameError`, `SyntaxError`, etc.) et un **message** qui décrit le problème rencontré.

Une première manière de rendre le code plus **robuste** consiste à s'assurer que l'erreur ne puisse être commise soit :

```
def pourcentage(score, total):
    if total == 0 :
        return None
    else:
        return score / total * 100
```

Mais il reste d'autres points à vérifier que les nombre soient positifs que le score soit inférieur au total que la fonction prend en compte des entiers ou des réels

Pour une gestion plus poussée des erreurs (de logique ici) on va définir une documentation dans laquelle on va préciser :

- **les préconditions** : les conditions qui doivent être satisfaites sur les paramètres et l'état global du programme, avant l'appel de la fonction.
- **les postconditions :** les conditions qui seront satisfaites sur la valeur de retour et sur l'état global du programme, après l'appel de la fonction, si les préconditions ont été respectées.

```
def pourcentage(score, total):
    """
    # Calcule le pourcentage de réussite  à un test.
    # - "score" contient la note obtenue (flottant)
    # - "total" est la score maximal atteignable (flottant)
    # Pre:  0 <= score <= total, le score obtenu
```

```
    #       total > 0, la score maximal atteignable
    # Post: La valeur renvoyée contient le pourcentage
    #       correspondant au score obtenu.

    """
    return score / total * 100
```

La documentation permet d'établir une sorte de contrat entre le codeur et l'utilisateur de code qui dans le meilleur des mondes ici ne devrait pas mettre total à 0 et utiliser uniquement des entiers positifs.

Elle permet aussi de fournir une documentation sous forme de **Docstring** immédiatement disponible en faisant un help dans le shell.

help(pourcentage**)**
Help on function pourcentage in module __main__:

pourcentage(score, total)
# Calcule le pourcentage de réussite à un test.
# - "score" contient la note obtenue (flottant)
# - "total" est la score maximal atteignable (flottant)
# Pre: 0 <= score <= total, le score obtenu
# total > 0, la score maximal atteignable
# Post: La valeur renvoyée contient le pourcentage
# correspondant au score obtenu.

2.3. Utilisation d'assertions :

Si le code est destiné à des personnes de confiance on peut utiliser des assertions pour limiter le nombre de préconditions.

```
def pourcentage(score, total):
    assert total > 0, 'total doit être strictement positif'
    assert 0 <= score, 'score doit être positif'
    assert score <= total, 'score doit être inférieur à total'
    return score / total * 100

print(f"test 1 réussite  : {pourcentage(18, 20) }")
print(f"test2  réussite :  {pourcentage(-18, 10) }")
```

```
test 1 réussite : 90.0
Traceback (most recent call last):
File " gestion_erreurs_ex1.py", line 28, in <module>
print(f"test2 réussite : {pourcentage(-18, 10) }")
File gestion_erreurs_ex1.py", line 15, in pourcentage
assert 0 <= score, 'score doit être positif'
AssertionError: score doit être positif
```

L'assertion assert 0 <= score, 'score doit être positif' est bien détectée et renseigne sur l'erreur mais elle arrête aussi le programme avant d'exécuter le code qui produit une erreur. Le code en question n'est donc pas fonctionnel et ne doit pas être utilisé en l'état.

On peut aussi utiliser une assertion pour vérifier un résultat

ex `assert pourcentage(100,100) == 100` mais là aussi si le résultat n'est pas conforme le programme est interrompu.

Pour un code fonctionnel il faut lever les exceptions : chapitre suivant ou/et faire de la **programmation défensive** : supprimer les sources d'erreurs en conseillé de blindant son code.

```
def pourcentage(score, total):
    """
    # Calcule le pourcentage de réussite  à un test.
    # - "score" contient la note obtenue (flottant)
    # - "total" est la score maximal atteignable (flottant)
    # Pre:  0 <= score <= total, le score obtenu
    #       total > 0, la score maximal atteignable
    # Post: La valeur renvoyée contient le pourcentage correspondant au score obtenu.
    """
    if total > 0 and (0 <= score <= total):
        return score / total * 100
    else :
        return None
```

3. Mécanisme d'exception

On peut donc gérer les erreurs à l'aide de l'instruction `if-else` et en prévoyant des valeurs de retour spéciales. Cette technique n'est malheureusement pas toujours utilisable, notamment lorsque la fonction définie ne renvoie rien.

Voyons maintenant le *mécanisme d'exception*, présent dans les langages de programmation orienté objet, qui permet de gérer des exécutions exceptionnelles qui ne se produisent qu'en cas d'erreur.

Pour éviter que le programme ne se termine définitivement ou qu'il se termine sans que la raison n'en soit explicite, il est conseillé de **gérer les exceptions**.

3.1. Instruction try-except-else-finally

La gestion des exceptions repose sur quelques mots clés : `try`, `except`, `else` et `finally` et sur une liste de types d'exceptions dont la liste peut être consultée dans la documentation en ligne de Python :

- `try` : le bloc de code qui suit ce mot clé est exécuté séquentiellement. En cas de problème, l'exécution est interrompue et l'interpréteur passe au bloc d'instructions suivant le mot clé except.
- `except NomErreur` : ce bloc d'instructions est exécuté si une erreur a été détectée dans le bloc try et si son type correspond à NomErreur. Plusieurs clauses except peuvent être utilisées. Il n'est pas nécessaire d'indiquer le type de l'erreur (tous les types sont alors traités de façon identique).

- `else` : cette directive permet d'isoler dans la partie try la ou les instructions qui peuvent poser problème. Toutes les instructions suivantes (qui ne doivent donc être exécutées que si aucun problème intervient) peuvent être placées dans le bloc else.
- `finally` : le bloc qui suit est exécuté dans tous les cas de figure, qu'une exception ait été levée ou pas. C'est donc ici que l'on peut s'assurer qu'un fichier ouvert dans le bloc try est correctement fermé, quoi qu'il arrive.

Exemple reprise de pourcentage

```
def pourcentage(score, total):
    try :
        resultat =  score/total
    except ZeroDivisionError :
        print("Division par Zéro total ne doit pas
être égal à  0")
    else :
        return resultat

pourcentage(20,100)
pourcentage(50,0)
print("Le programme continue !")
```

Sortie :

```
0.2

Division par Zéro total ne doit pas être
égal à 0
None
Le programme continue !
```

On peut aussi affiner les sources d'exceptions en utilisant plusieurs except

```
def pourcentage():
    try :
        score  = int(input("Donner un score "))
        total = int(input("Donner le total des
points"))
        resultat = score/total
    except ZeroDivisionError :
        print("Division par Zéro total ne doit pas
être égal à  0")
    except ValueError :
        print(" score et total sont des entiers")
    else :
        return resultat

print(pourcentage())
print(pourcentage())
print("Le programme continue !")
```

Sortie :

```
Donner un score 1.2
 score et total sont des entiers
None
Donner un score 100
Donner le total des points0
Division par Zéro total ne doit pas
être égal à  0
None

Le programme continue !
```

On peut aussi grouper les origines des exceptions dans un seul except

```
except (ZeroDivisionError,ValueError) :
        print("Oups problème")
```

Sortie :

```
Donner un score 10
Donner le total des points0
Oups problème
None
```

Le finally est toujours executé à la fin

```python
def pourcentage():
    try :
        score  = int(input("Donner un score "))
        total = int(input("Donner le total des points"))
        resultat = score/total
    except (ZeroDivisionError,ValueError) as e:

        print(f"Origine du pb {e}")
    else :
        return resultat
    finally :
        print("le finally est au final toujours eexecuté")


print(pourcentage())
print("Le programme continue !")
```

Sortie :
Donner un score 10
Donner le total des points0
Origine du pb division by zero
le finally est au final toujours executé
None
Le programme continue !

3.2. Lever d'une exception « à la main » - raise

Il est possible de signaler une exception grâce au mot-clé `raise`.

```python
def pourcentage():
    try :
        score  = int(input("Donner un score "))
        total = int(input("Donner le total des points"))
        if score < 0 :
            raise ValueError("score doit être positif")
        resultat = score/total
    except (ZeroDivisionError,ValueError) as e:

        print(f"Origine du pb {e}")

    else :
        return resultat
    finally :
        print("le finally est au final toujours executé")


print(pourcentage())
print("Le programme continue !")
```

Sortie :
Donner un score -10
Donner le total des points300
Origine du pb score doit être positif
le finally est au final toujours executé
None
Le programme continue !

Ici dans le try on met le code litigieux. Dans le cas où l'utilisateur entre autre chose qu'un entier l'interpréteur python détecte une erreur d'exécution ce qui entraîne l'apparition d'une exception (et sans précaution le programme s'arrête) le code dans le except est exécuté mais le programme n'est pas interrompu.

3.3. Utilisation des mécanismes de lever d'exception

Les erreurs présentées dans les parties précédentes peuvent être évitées avec une programmation défensive néanmoins la levée des exceptions permet

- de ne pas interrompre le programme en cas d'exception (erreurs)
- de tester des partie de code dont on ne peut pas affirmer le résultats typiquement quand on utilise des bibliothèques avec des méthodes ou fonctions pouvant entraîner des erreurs pas à priori décelable (la flemme de lire la documentation)

3.3.1. Les fichiers

On distingue couramment deux types de fichier, qui seront manipulés à des niveaux différents :

- Un *fichier texte* est constitué d'une séquence de caractères, permettant de stocker une chaine de caractères sur disque. Un fichier `.py` contenant le code source d'un programme Python est un exemple d'un tel fichier.
- Un *fichier binaire* est constitué d'une séquence de bits, organisés en paquets de huit, appelés *octets*. Un fichier `.png` avec une image est un exemple d'un tel fichier.

La seule raison pour laquelle on différencie ces deux types de fichiers est car Python propose des fonctions spécifiques différentes permettant de facilement les manipuler.

Deux erreurs peuvent survenir lorsqu'on manipule un fichier. Tout d'abord, il se peut que le fichier que l'on tente d'ouvrir n'existe pas, dans lequel cas une erreur de type `FileNotFoundError` est générée. Ensuite, durant la lecture ou l'écriture, différentes situations d'erreur peuvent survenir comme le disque qui devient plein, l'utilisateur qui n'a pas les droits suffisants pour lire/écrire un fichier, etc. Dans toutes ces situations, une erreur de type `IOError` survient, signalant en fait une erreur d'entrée/sortie. Si on veut un programme robuste, il faudra les traiter à l'aide d'un `try-except`.

L'exception `IOError` est en fait une erreur générique d'entrée/sortie et on peut se limiter à gérer cette dernière. Néanmoins, il est parfois utile de gérer ses cas particuliers, parmi lesquels on a :

- `FileNotFoundError` si le fichier n'existe pas ;
- `FileExistsError` si le fichier existe déjà ;
- `PermissionError` si le programme n'a pas les droits d'accès nécessaires sur le fichier ;
- `IsADirectoryError` si le fichier est en fait un dossier.

3.3.1.1. Ouverture/Lecture/ Codage

Ouverture/Lecture	Ecriture

`try:`
`  file = open('data.txt')`
`  print(file)`
`  file.close()`
`except FileNotFoundError:`
`  print('Fichier introuvable.')`
`except IOError:`
	`try:`
	`  file = open('data.txt', 'w')`
	`  file.write('Table de 7 :\n')`
	`  for i in range(10):`
	`  file.write('{} x 7 = {}\n'.format(i, i * 7))`
	`  file.close()`
	`except IOError:`

`    print('Erreur d\'ouverture.')`	`    print('Erreur d\'entrée/sortie.')`
Pas de fichier : Fichier introuvable Fichier existant : <_io.TextIOWrapper name='data.txt' mode='r' encoding='cp1252'>	

3.3.1.2. Problème de codage de caractère

```
try :
    file = open('data.txt', 'w',
encoding='ascii')
    file.write('€')
    file.close()
except IOError as e :
    print(e)
```

file.write('€')
UnicodeEncodeError: 'ascii' codec can't encode character '\u20ac' in position 0: ordinal not in range(128)

"€" est un caractère UTF-8 dont le code est supérieur aux 128 possibilité en ASCII classique

3.3.1.3. Fermeture

Un fichier ouvert doit être fermer on peut ajouter un finally pour en être sûr :

```
try:
    file = open('data.txt')
    print(file.read())
except FileNotFoundError:
    print('Fichier introuvable.')
except IOError:
    print('Erreur d\'entrée/sortie.')
finally:
    print("On ferme le fichier")
    file.close()
```

Sortie :

On ferme le fichier

Instruction `with`

Python propose l'*instruction* `with` pour proprement gérer un contexte et automatiquement libérer les ressources au moment où le contexte est quitté.

```
try:
    with open('data.txt') as file:
        print(file.read())
except FileNotFoundError:
    print('Fichier introuvable.')
except IOError:
        print('Erreur d\'entrée/sortie.')
```

3.3.2. La récursion

```python
def fact(n):
    if n < 0:
        raise ArithmeticError("n doit être positif")
    if  n == 0 :
        return  1
    else :
        return n*fact(n-1)

try:
    print(fact(1000))
except (RecursionError, ArithmeticError) as e:
    print(e)
finally :
    print("Finally")
```

Sortie :
la profondeur de récursivité maximale est dépassée
Finally