OFFICE DE LA FORMATION PROFFESSIONELLEET
DE LA PROMOTION DU TRAVAIL

RAPPORT DES EXCEPTIONS PYTHON

Le 06/08/2024

Option : DEVELOPPEMENT DIGITAL

ETABLIS PAR : Yasser KHIAT, Saad AIT ALLOUCH,
Souhaib EDDAHMANI, Mohammed Reda MENYANI

En informatique, lors de l'exécution d'un programme, il y a toujours un risque d'erreur. C'est la raison pour laquelle tout bon langage possède une gestion de base des erreurs qui correspond concrètement en l'affichage d'un message d'erreur nous informant sur le type d'erreur détectée et (souvent) en l'arrêt de l'exécution du programme après que l'erreur ait été détectée.

Dans cette partie, nous allons apprendre à intercepter les erreurs renvoyées par Python et à les gérer.

Les types d'erreurs et pourquoi gérer les erreurs

On programmation, on peut rencontrer principalement deux types d'erreurs :

- Les erreurs de syntaxe dans le code faites par le développeur ;
- Les erreurs d'environnement qui ne sont pas du fait du développeur.

Par exemple, se servir d'une variable non déclarée dans un script, utiliser des opérateurs avec des types de données qui ne les supportent pas ou oublier un guillemet ou un deux points sont des erreurs de syntaxe faites par le développeur.

En revanche, tenter d'importer un module qui ne serait pas disponible à un temps t, ou demander à un utilisateur d'envoyer un chiffre et recevoir une chaine par exemple va produire une erreur qui n'est pas sûr au développeur.

Comme les erreurs de syntaxe sont des erreurs que nous avons faites, nous pouvons les corriger directement et c'est ce qu'on s'efforcera à faire et modifiant nos scripts. Pour les autres erreurs, en revanche, il va falloir mettre en place un système de gestion d'erreurs qui indiquera au Python quoi faire si telle ou telle erreur est rencontrée.

Il est essentiel de fournir une gestion des erreurs d'environnement afin de garantir que le script ne plante pas dans certaines situations et pour garantir l'intégrité et la sécurité des données ainsi qu'une bonne expérience pour l'utilisateur qui n'a pas envie de voir des messages d'erreur Python.

En Python, nous allons pouvoir intercepter certaines erreurs pour les prendre en charge nous mêmes et pour décider si le script doit continuer de s'exécuter ou pas.

Les classes exception

En Python, les erreurs détectées durant l'exécution d'un script sont appelées des exceptions car elles correspondent à un état "exceptionnel" du script.

Si vous essayez de déclencher des erreurs manuellement, vous pouvez constater que Python analyse le type d'erreur et renvoie un message différent selon l'erreur détectée :

```
>>> test
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'test' is not defined
>>> 10/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> "dix" + 10
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

Ici, nous avons trois types d'exceptions différentes : une exception `NameError`, une exception `ZeroDivisionError` et une exception `TypeError`. Comment fait Python pour analyser les types d'erreurs et renvoyer des messages différents en fonction ?

En fait, vous devez savoir que Python possède de nombreuses classes d'exceptions natives et que toute exception est une instance (un objet) créé à partir d'une classe exception.

Afin de bien comprendre la hiérarchie des classes d'exceptions, vous pouvez retenir que la classe d'exception de base pour les exceptions natives est `BaseException`. Toutes les autres classes d'exception vont dériver de cette classe. Ensuite, nous avons également quatre autres classes d'exception de base (qui dérivent de `BaseException`) :

- La classe `Exception` est la classe de base pour toutes les exceptions natives qui n'entraînent pas une sortie du système et pour toutes les exceptions définies par l'utilisateur (nous sommes l'utilisateur dans ce cas) ;
- La classe `ArithmeticError` est la classe de base pour les exceptions natives qui sont levées pour diverses erreurs arithmétiques et notamment pour les classes `OverflowError`, `ZeroDivisionError` et `FloatingPointError` ;
- La classe `BufferError` est la classe de base pour les exceptions levées lorsqu'une opération liée à un tampon ("buffer") ne peut pas être exécutée ;
- La classe `LookupError` est la classe de base pour les exceptions qui sont levées lorsqu'une clé ou un index utilisé sur un tableau de correspondances ou une séquence est invalide.

De nombreuses classes dérivent ensuite de ces classes de base. En fonction de l'erreur rencontrée par l'analyseur Python, un objet exception appartenant à telle ou telle classe

exception va être crée et renvoyé. C'est cet objet là que nous allons pouvoir intercepter et manipuler.

Python nous fournit des structures permettant de gérer manuellement certaines exceptions. Nous allons voir comment mettre en place ces structures dans cette leçon.

L'instruction try… except

Les clauses `try` et `except` fonctionnent ensemble. Elles permettent de tester (try) un code qui peut potentiellement poser problème et de définir les actions à prendre si une exception est effectivement rencontrée (except).

Imaginons par exemple un script qui demande à un utilisateur d'envoyer deux nombres grâce à la fonction `input()` qui permet de recevoir des données d'utilisateurs externes.

Le but de notre script va être de calculer le quotient de ces deux nombres entiers. Ici, on peut déjà anticiper des cas d'erreurs qui peuvent se présenter : l'utilisateur peut nous envoyer autre chose qu'un nombre entier ou peut nous envoyer un dénominateur égal à 0, ce qui est interdit en mathématique.

On va vouloir gérer ces deux cas exceptionnels. Pour cela, on va pouvoir utiliser deux blocs `try` et `except` comme cela :

```
>>> try:
...     x = int(input("Entrez un numérateur : "))
...     y = int(input("Entrez un dénominateur : "))
...     print("Résultat de la division :", x / y)
... except:
...     print("Erreur : impossible d'effectuer la division")
...
Entrez un numérateur : 10
Entrez un dénominateur : 2
Résultat de la division : 5.0
>>>
>>> try:
...     x = int(input("Entrez un numérateur : "))
...     y = int(input("Entrez un dénominateur : "))
...     print("Résultat de la division :", x / y)
... except:
...     print("Erreur : impossible d'effectuer la division")
...
Entrez un numérateur : "dix"
Erreur : impossible d'effectuer la division
>>>
>>> try:
...     x = int(input("Entrez un numérateur : "))
...     y = int(input("Entrez un dénominateur : "))
...     print("Résultat de la division :", x / y)
... except:
...     print("Erreur : impossible d'effectuer la division")
...
Entrez un numérateur : 10
Entrez un dénominateur : 0
Erreur : impossible d'effectuer la division
```

Ici, on place le code à tester à l'intérieur du bloc `try`. Si aucune erreur n'est détectée par Python lors de l'exécution du code, c'est-à-dire si aucun objet exception n'est créé, ce qui se situe dans la clause `except` va être ignoré.

En revanche, si une exception intervient pendant l'exécution du code dans `try`, le reste du code dans cette clause est ignoré et on rentre dans l'instruction `except`. Gérer les exceptions comme cela permet notamment d'éviter l'arrêt de l'exécution du script dans des cas où cela ne nous semble pas nécessaire.

Notre code ci-dessus est cependant loin d'être optimise car notre clause `except` est bien trop large. Lorsqu'on gère les exceptions manuellement, on voudra toujours apporter la gestion la plus fine possible pour éviter de capturer toutes les erreurs n'importe comment.

Pour cela, nous allons préciser le type d'erreur qu'une instruction `except` doit intercepter. Si on souhaite gérer plusieurs types d'exceptions, on pourra préciser autant d'instructions `except` qu'on le souhaite à la suite d'un `try`.

Regardez plutôt le code ci-dessous :

```
>>> try:
...     x = int(input("Entrez un numérateur : "))
...     y = int(input("Entrez un dénominateur : "))
...     print("Résultat de la division :", x / y)
... except ValueError:
...     print("Erreur : la valeur entrée n'est pas un entier valide")
... except ZeroDivisionError:
...     print("Erreur : impossible de diviser par zéro")
...
Entrez un numérateur : 10
Entrez un dénominateur : 5
Résultat de la division : 2.0
>>>
>>> try:
...     x = int(input("Entrez un numérateur : "))
...     y = int(input("Entrez un dénominateur : "))
...     print("Résultat de la division :", x / y)
... except ValueError:
...     print("Erreur : la valeur entrée n'est pas un entier valide")
... except ZeroDivisionError:
...     print("Erreur : impossible de diviser par zéro")
...
Entrez un numérateur : 10
Entrez un dénominateur : "maison"
Erreur : la valeur entrée n'est pas un entier valide
>>>
>>> try:
...     x = int(input("Entrez un numérateur : "))
...     y = int(input("Entrez un dénominateur : "))
...     print("Résultat de la division :", x / y)
... except ValueError:
...     print("Erreur : la valeur entrée n'est pas un entier valide")
... except ZeroDivisionError:
...     print("Erreur : impossible de diviser par zéro")
...
Entrez un numérateur : 10.0
Erreur : la valeur entrée n'est pas un entier valide
>>>
>>> try:
...     x = int(input("Entrez un numérateur : "))
...     y = int(input("Entrez un dénominateur : "))
...     print("Résultat de la division :", x / y)
... except ValueError:
...     print("Erreur : la valeur entrée n'est pas un entier valide")
... except ZeroDivisionError:
...     print("Erreur : impossible de diviser par zéro")
...
Entrez un numérateur : 10
Entrez un dénominateur : 0
Erreur : impossible de diviser par zéro
```

Ce code est déjà beaucoup plus optimisé puisqu'il nous permet une gestion plus fine des erreurs et ne va en l'occurence capturer que deux types d'exceptions : le cas où les données entrées ne sont pas des entiers et le cas où le dénominateur est égal à 0.

Ce code demande cependant bien évidemment de connaitre les classes d'exception Python, mais il suffit de consulter la documentation en cas de doute.

Notez que rien ne nous empêche de préciser un `except` sans classe à la fin si on souhaite absolument donner des ordres dans le cas où Python capturerait tout autre type d'exceptions que ceux correspondant aux clauses `except` précédentes.

La clause else

Nous allons également pouvoir ajouter une clause `else` en fin d'instruction `try… except`. Le code contenu dans cette clause sera exécuté dans le cas où aucune exception n'a été levée par la clause `try`.

Il est considéré comme une bonne pratique de placer le code "non problématique" dans la clause `else` plutôt que dans la clause `try`.

On pourrait ainsi réécrire notre exemple précédent de la façon suivante :

```
>>> try:
...     x = int(input("Entrez un numérateur : "))
...     y = int(input("Entrez un dénominateur : "))
...     z = x / y
... except ValueError:
...     print("Erreur : la valeur entrée n'est pas un entier valide")
... except ZeroDivisionError:
...     print("Erreur : impossible de diviser par zéro")
... else:
...     print("Résultat de la division : ",z)
...
Entrez un numérateur : 10
Entrez un dénominateur : 5
Résultat de la division :  2.0
>>>
>>> try:
...     x = int(input("Entrez un numérateur : "))
...     y = int(input("Entrez un dénominateur : "))
...     z = x / y
... except ValueError:
...     print("Erreur : la valeur entrée n'est pas un entier valide")
... except ZeroDivisionError:
...     print("Erreur : impossible de diviser par zéro")
... else:
...     print("Résultat de la division : ",z)
...
Entrez un numérateur : "dix"
Erreur : la valeur entrée n'est pas un entier valide
>>>
>>> try:
...     x = int(input("Entrez un numérateur : "))
...     y = int(input("Entrez un dénominateur : "))
...     z = x / y
... except ValueError:
...     print("Erreur : la valeur entrée n'est pas un entier valide")
... except ZeroDivisionError:
...     print("Erreur : impossible de diviser par zéro")
... else:
...     print("Résultat de la division : ",z)
...
Entrez un numérateur : 10
Entrez un dénominateur : 0
Erreur : impossible de diviser par zéro
```

La clause finally

La dernière clause à connaitre est la clause `fin ally`. Le code qu'elle contient sera exécuté dans tous les cas, qu'une exception ait été levée par la clause `try` ou pas.

Cette clause va s'avérer très utile lorsqu'on voudra terminer certaines opérations (fermer un fichier par exemple) quel que soit l'état du script.

```
>>> try:
...     x = int(input("Entrez un numérateur : "))
...     y = int(input("Entrez un dénominateur : "))
...     z = x / y
... except ValueError:
...     print("Erreur : la valeur entrée n'est pas un entier valide")
... except ZeroDivisionError:
...     print("Erreur : impossible de diviser par zéro")
... else:
...     print("Résultat de la division : ",z)
... finally:
...     print("Ce message est toujours affiché")
...
Entrez un numérateur : 10
Entrez un dénominateur : 2
Résultat de la division :  5.0
Ce message est toujours affiché
>>>
>>> try:
...     x = int(input("Entrez un numérateur : "))
...     y = int(input("Entrez un dénominateur : "))
...     z = x / y
... except ValueError:
...     print("Erreur : la valeur entrée n'est pas un entier valide")
... except ZeroDivisionError:
...     print("Erreur : impossible de diviser par zéro")
... else:
...     print("Résultat de la division : ",z)
... finally:
...     print("Ce message est toujours affiché")
...
Entrez un numérateur : "dix"
Erreur : la valeur entrée n'est pas un entier valide
Ce message est toujours affiché
```

Définir nos propres classes d'exception

Pour finir, vous devez savoir que Python nous laisse la possibilité de créer nos propres classes d'exception.

Pour cela, nous allons toujours créer des classes à partir de la classe de base `Exception`.

```
>>> class CustomError(Exception):
...     def __init__(self, erreur, message):
...             #Code à imaginer !
```

Définir nos propres exceptions va s'avérer être une fonctionnalité très utile notamment dans le cas où on souhaite distribuer un module Python et que certaines de nos fonctions peuvent déclencher des exceptions non prises en charge par les classes d'exception standard Python.

Conclusion :

Python est un langage qui a été créé avec l'objectif d'être facilement compréhensible. Pour cette raison, et pour la structure du langage en général qui pousse les développeurs à coder plus proprement et à adopter de bonne pratiques, il est l'un des langages les plus recommandés aux personnes souhaitant s'initier à la programmation.

Si Python est simple à apprendre, il n'en est pas moins puissant et versatile : ce langage peut être utilisé pour effectuer des tâches complexes et pour mener à bien différents projets très différents, que ce soit la création d'applications Web ou de programmes divers.