IFT-1004 - Introduction à la programmation

Module 9 : Gestion des exceptions

Honoré Hounwanou, ing.

Département d'informatique et de génie logiciel
Université Laval

Table des matières

Gestion des exceptions

Gestion des exceptions

Objectif

Comprendre l'utilité de la gestion des exceptions, apprendre la syntaxe du Python permettant une gestion de ces exceptions.

À quoi ça sert ?

Les exceptions sont les opérations qu'effectue un interpréteur ou un compilateur lorsqu'une erreur est détectée au cours de l'exécution d'un programme. On dit que le programme **lève une exception**.

Lorsqu'une exception n'est pas traitée par votre code, elle le sera par l'interpréteur qui affichera un message d'erreur.

Dans ce dernier cas cependant, l'exécution de votre programme se terminera abruptement.

Exemple d'exception non traitée

```
>>> valeur = 1/0

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

`ZeroDivisionError` : le type de l'exception,

`division by zero` : le message qu'envoie Python pour vous indiquer plus clairement l'erreur qui vient de se produire.

À quoi ça sert ?

Doit-on traiter les exceptions ?

- ça dépend uniquement de vous et de ce que vous cherchez à accomplir…
- est-ce possible de traiter l'erreur ?
- si oui, où peut-on la traiter ?
- sinon, celle-ci est fatale et il vaut mieux terminer l'exécution du programme avec un message pertinent

À quoi ça sert ?

Comment faire ?

- Python permet de tester un bout de code. S'il ne renvoie aucune erreur, l'exécution continue. Sinon, on peut lui demander d'exécuter une autre action.
- En général, un mécanisme de ce type s'appelle un *mécanisme de traitement des exceptions*. Celui de Python utilise l'ensemble d'instructions `try-except`, qui permet d'intercepter une erreur et d'exécuter un bloc d'instructions spécifique à cette erreur.

Forme minimale du bloc `try`

```
try:
    # bloc à essayer

except:
    # bloc qui sera exécuté en cas d'erreur
```

Les principales classes d'exception

- `NameError` : variable non définie (elle n'existe pas)
- `TypeError` : opérateur incompatible avec les types
- `ZeroDivisionError` : division par 0
- `ValueError` : erreur avec une valeur
- `IndexError` : lorsqu'un index est invalide (par exemple dans une liste)
- `KeyError` : lorsqu'une clé est invalide (par exemple dans un dictionnaire)
- `IOError` : échec d'ouverture ou d'écriture dans un fichier

Exécuter le bloc `except` pour un type d'exception précis

Il est possible (et recommandé) de préciser le type d'exception que nous souhaitons traiter dans un bloc `except` :

```
try:
    resultat = numerateur / denominateur

except NameError:
    print("La variable numerateur ou denominateur n'a pas été définie.")
```

Exécuter le bloc `except` pour un type d'exception précis

On peut intercepter les autres types d'exceptions en faisant d'autres blocs `except` à la suite :

```
try:
    resultat = numerateur / denominateur

except NameError:
    print("La variable numerateur ou denominateur n'a pas été définie.")

except TypeError:
    print("Variables incompatibles avec la division.")

except ZeroDivisionError:
    print("La variable denominateur est égale à 0.")
```

Exemple : ouverture de fichier

Si nous estimons que ce genre de test est susceptible de rendre service à plusieurs endroits dans un programme, nous pouvons aussi l'inclure dans une fonction :

```
def existe(nom_fichier):
    try:
        f = open(nom_fichier, 'r')
        f.close()
        return True
    except IOError:
        return False

nom = input("Veuillez entrer un nom de fichier : ")
if existe(nom):
    print("Ce fichier existe bel et bien.")
else:
    print("Le fichier {} est introuvable.".format(nom))
```

Le mot clé `else`

Le bloc `else` permet d'exécuter une action si aucune erreur n'est trouvée dans le bloc `try`. Peu utilisé en pratique, mais permet de savoir si une exception a été levée ou non.

```
try:
    resultat = numerateur / denominateur

except NameError:
    print("La variable numerateur ou denominateur n'a pas été définie.")

except TypeError:
    print("Variables incompatibles avec la division.")

except ZeroDivisionError:
    print("La variable denominateur est égale à 0.")

else:
    print("Le résultat obtenu est {}".format(resultat))
```

Le mot clé `finally`

Le bloc `finally` permet d'exécuter une action après un bloc `try`, quelle que soit l'issue de l'exécution dudit bloc.

```
try:
    # Instructions

except:
    # Gestion des erreurs

finally:
    # Instructions exécutées qu'il y ait une erreur ou non
```

Le bloc `finally` peut sembler équivalent à mettre le code juste après le bloc `try-except`, mais celui-ci sera exécuté dans tous les cas de figure. Par exemple, si une instruction `return` se trouve dans le bloc `try`, le bloc `finally` sera tout de même exécuté avant le retour.

Syntaxe générale

```
try:
    [...]  # Exécution normale

except Nom_1 [as valeur]:
    [...]  # Éxécuté si une exception Nom_1 est levée

except (Nom_2, Nom_3) [as valeur]:
    [...]  # Éxécuté si une exception Nom_2 ou Nom_3 est levée

else:
    [...]  # Exécuté si aucune exception

finally:
    [...]  # Exécuté à la fin, dans tous les cas
```

Énoncés `raise` et `as`

Syntaxe :

```
raise TypeDeLException("message à afficher")
```

L'instruction `raise` interrompt l'exécution de la fonction en cours. Elle lance *vers le haut* l'exception levée. L'instruction `as` permet d'obtenir une variable locale qui contient l'exception. Exemple :

```
annee = input()

try:
    annee = int(annee)
    if annee <= 0:
        raise ValueError("l'année entrée est <= 0")

except ValueError as e:
    print("Exception: {}", e)
```

Exemple d'utilisation : validation de préconditions

```
def chiffrer(texte, cle):
    """Chiffre le texte avec la clé reçue (chiffrement de César).

    Args:
        cle (int): La clé, entre 0 et 25.

    Returns:
        str: le texte chiffré.

    """
    if not isinstance(cle, int):
        raise TypeError("erreur ...")

    if cle < 0 or cle > 25:
        raise ValueError("erreur ...")

    [...]
```

Exemple d'utilisation : validation dans une interface

Avec une interface graphique, la validation des entrées peut aussi se faire à l'aide des exceptions.

```
self.etiquette_erreur = Label(root, foreground='red')
[...]

try:
    cle = int(self.entree_cle.get())

except:
    self.etiquette_erreur['text'] = "Erreur : ..."
```

Classes d'exceptions

On peut créer ses propres exceptions en dérivant une nouvelle classe de la classe `Exception`.

```
class MonErreur(Exception):
    pass
```

On peut ensuite lancer cette exception comme n'importe quelle autre exception.

```
>>> raise MonErreur("erreur très spéciale")

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
__main__.MonErreur: erreur très spéciale
```

Classes d'exceptions

```
try:
    raise MonErreur("erreur très spéciale")

except MonErreur as e:
    print(e)
```

Les classes d'exception sont des classes comme les autres :

- on peut définir un constructeur (méthode `__init__`)
- on peut définir une méthode `__repr__`, etc.
- par défaut, le constructeur d'une `Exception` accepte un nombre arbitraire d'arguments et les stocke dans l'objet.
- par défaut, l'affichage d'un objet `Exception` produit l'affichage de tous les arguments reçus à la construction.

Erreurs et exceptions : à retenir

Trois questions à se poser lorsqu'on valide des entrées :

- ma variable a-t-elle le bon **type**? (`int`, `float`, `str`, etc.)
- ma variable a-t-elle le bon **format**? (date, téléphone, code postal, etc.)
- ma variable respecte-t-elle les **pré-conditions**? (valeur minimale/maximale, etc.)

Lectures et travaux dirigés

Lectures et travaux dirigés

- Lectures : revoir la fin du chapitre 9 (pages 117 et suivantes)
- Travaux dirigés : Héritage de la classe `Canvas` en un `CanvasEchiquier` + utilisation de la gestion des exceptions

Questions ?