




Lycée d'enseignement général et technologique international Victor Hugo COLOMIERS		
	La Gestion de Erreurs en Python	
	Devoir Maison	
Nom : _____ Prénom : _____		Date : _____

Table des matières

1	Consignes pour l'AP.....	2
2	Introduction	2
2.1	Qu'est-ce qu'une erreur ?	2
2.2	Les types d'erreur.....	5
3	Gérer les erreurs.....	8
3.1	La documentation	8
3.2	Les assertions	9
3.3	Le mécanisme d'exception.....	12
3.3.1	L'instruction try-except	12
3.3.2	Définir une exception	18



La Gestion de Erreurs en Python

Devoir Maison



1 Consignes de réalisation

Ce devoir est réalisé en pleine autonomie.

Afin de profiter pleinement de cet apprentissage, il vous est conseillé de :

- Lire et les éléments de connaissance présentés
- Ecrire, commenter et exécuter les programmes proposés
- Analyser les résultats obtenus
- Tracer éventuellement (avec des instructions « print ») l'exécution des programmes afin de mieux appréhender la gestion des erreurs en Python
- Exécuter plusieurs fois les programmes en modifiant les jeux de test et/ou certaines instructions afin d'approfondir la mise en œuvre de la gestion d'erreurs

2 Introduction

Lors de l'exécution d'un programme, des erreurs peuvent se produire. Plusieurs causes sont possibles : le code source est mal écrit, ne respectant pas la syntaxe du langage, ou alors des opérations interdites ont été faites, comme une division par zéro. Le pire, ce sont les erreurs de logique, où tout semble bien se passer si ce n'est que le résultat produit finalement n'est pas celui attendu. Ce chapitre présente comment gérer les erreurs dans un programme, notamment avec le mécanisme d'exception utilisé en programmation orientée objet.

2.1 Qu'est-ce qu'une erreur ?

On a déjà rencontré en classe de première de nombreuses situations où l'exécution d'un programme provoque une erreur. Lorsque c'est le cas, l'exécution s'arrête immédiatement et l'interpréteur Python affiche une trace d'erreur.

Cette dernière fournit des informations quant au chemin d'exécution qui a mené jusqu'à l'erreur et sur la cause de cette dernière. Prenons, par exemple, le programme suivant :

```
def pourcentage(note, total):  
    return note / total * 100  
  
print('Alexis a obtenu', pourcentage(18, 20), '%')  
print('Sébastien a obtenu', pourcentage(6, 0), '%')
```

Exécuter le programme, vous obtenez le résultat suivant :

```
Alexis a obtenu 90.0 %  
Traceback (most recent call last):  
  File "program.py", line 5, in <module>  
    print('Sébastien a obtenu', pourcentage(6, 0), '%')  
  File "program.py", line 2, in pourcentage  
    return note / total * 100  
ZeroDivisionError: division by zero
```



La Gestion de Erreurs en Python

Devoir Maison

Victor Hugo

Lycée d'enseignement général et technologique international Victor Hugo COLOMIERS		
	<div>La Gestion de Erreurs en Python</div> <div>Devoir Maison</div>	

On constate que, sur la première ligne, qu'Alexis a obtenu 90%. On ne voit par contre pas la note de Sébastien, mais un message d'erreur à la place. En décortiquant cette trace d'erreur, on peut y identifier plusieurs informations par rapport à l'erreur qui s'est produite :

- la première ligne, qui commence par « Traceback », est le début de la trace d'erreur. Après cette ligne, on retrouve l'ordre des appels qui ont causé l'erreur
- le premier élément de la trace d'erreur indique son origine. Dans ce cas-ci, l'erreur provient de l'exécution de l'instruction de la ligne 5 du fichier program.py :

```
File "program.py", line 5, in <module>
    print('Sébastien a obtenu', pourcentage(6, 0), '%')
```

- comme l'erreur est apparue suite à un appel de fonction, la trace d'erreur fournit plus d'informations quant à l'instruction précise dans la fonction qui a provoqué l'erreur. Dans ce cas-ci, l'erreur provient de l'exécution de l'instruction de la ligne 2, dans la fonction pourcentage :

```
File "program.py", line 2, in pourcentage
    return note / total * 100
```

- enfin, la dernière ligne de la trace d'erreur explique ce qui a provoqué l'erreur. Dans ce cas-ci, il s'agit d'une division par zéro :

```
ZeroDivisionError: division by zero
```

La trace d'erreur permet donc de retracer l'erreur depuis son origine jusqu'à l'instruction précise qui l'a provoquée. Elle essaie également de fournir une explication utile quant à sa cause.

Comment gérer cette erreur ? Une première manière de faire consiste à rendre la fonction « pourcentage » la plus robuste possible, c'est-à-dire qu'elle ne doit jamais produire d'erreur. Pour cet exemple, on pourrait donc s'assurer qu'elle se comporte bien, peu importe la valeur de total, en renvoyant une valeur spéciale lorsque total vaut 0 :

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

Mais cette solution n'est pas idéale, car il faudrait vérifier que les deux paramètres soient des nombres positifs et que note soit inférieur à total.



La Gestion de Erreurs en Python

Devoir Maison



2.2 Les types d'erreur

Nous pouvons distinguer trois types d'erreurs :

- Une erreur de syntaxe survient lorsque le code source du programme est mal formé. Une telle erreur se produit, par exemple, lorsqu'on oublie la condition d'un if, lorsqu'on a un else sans if associé, lorsqu'on a mal écrit un mot réservé, etc.
- Une erreur d'exécution survient lorsqu'un programme, syntaxiquement correct, effectue une opération interdite. Une telle erreur se produit, par exemple, lorsqu'on tente de faire une division par zéro, lorsqu'on ajoute une liste dans un ensemble, lorsqu'on tente d'accéder à une variable d'instance privée hors de la classe, etc.
- Une erreur logique survient lorsqu'un programme, sans erreur de syntaxe ni d'exécution, ne produit pas le résultat correct attendu. Par exemple, si on a écrit longueur + largeur au lieu de longueur * largeur pour calculer la surface d'un rectangle, le programme se terminera sans erreur, mais pas avec la bonne réponse.



Les erreurs de syntaxe

Lorsqu'un programme Python comporte une erreur de syntaxe, elle ne sera détectée que lors de l'exécution, étant donné qu'il s'agit d'un langage interprété et que l'interpréteur ne passe pas en revue tout le code source avant exécution. Il se peut donc très bien qu'un programme comportant des erreurs de syntaxe se soit toujours exécuté sans erreurs. C'est, par exemple, le cas lorsqu'on a une structure alternative « if-else » dont le bloc « else » n'a jamais été exécuté, alors qu'il comporte une erreur de syntaxe. Examinons le code d'exemple suivant, qui comporte une erreur de syntaxe :

```
note = 12
if note > 10
    print('Vous avez réussi !')
```

Si on tente d'exécuter ce programme, on se retrouve face à l'erreur suivante :

```
File "program.py", line 2
    if note > 10
        ^
SyntaxError: invalid syntax
```

On voit qu'il s'agit d'une erreur de syntaxe grâce à la dernière ligne qui commence par `SyntaxError`. L'interpréteur tente de décrire et situer l'erreur le plus précisément possible. Il indique qu'il y a un souci à la ligne 2 et plus précisément tout à la fin de l'instruction comme pointé par le caractère `^` de la troisième ligne. On se rend en fait compte qu'il manque le `:` après la condition du « if ».

Lycée d'enseignement général et technologique international Victor Hugo COLOMIERS		
	<div>La Gestion de Erreurs en Python</div> <div>Devoir Maison</div>	

Il faut aussi savoir qu'il y a deux cas particuliers d'erreurs de syntaxe pour lesquels le type d'erreur sera différent :

- l'indentation du code n'est pas correcte (IndentationError) ;
- il y a une inconsistance entre l'utilisation d'espaces et de tabulations pour l'indentation d'un même bloc (TabError).

Les erreurs d'exécution

Une erreur d'exécution se produit lorsqu'une opération interdite a été effectuée. Voici plusieurs situations qui peuvent se produire, étant donné ce qu'on a déjà vu :

- 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 pu être importé (ImportError).
- Une variable ou une fonction avec le nom précisé n'a pas pu être trouvée (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).

L'exemple suivant provoque une erreur d'exécution, car on dépasse les bornes d'une liste dont on souhaite afficher les éléments :

```
data = [1, 2, 3]

i = 0
while i <= len(data):
    print(data[i])
    i += 1
```

On commence bien avec 0 comme premier indice, mais on va trop loin pendant la boucle puisqu'on s'arrête à la taille de la liste, alors que le plus grand indice vaut un de moins que cette dernière. Les trois valeurs de la liste sont donc bien affichées, mais s'ensuit une erreur d'exécution :

```
1
2
3
Traceback (most recent call last):
  File "program.py", line 5, in <module>
    print(data[i])
IndexError: list index out of range
```



La Gestion de Erreurs en Python

Devoir Maison



L'erreur logique

Ce type d'erreur est le plus difficile à déceler. Lorsqu'un programme comporte une erreur logique, il s'exécute en effet sans erreur, si ce n'est que le résultat produit n'est pas celui attendu. Supposons, par exemple, que l'on écrive une fonction dont le but est de calculer le périmètre d'un rectangle :

```
def perimetre(longueur, largeur):  
    return longueur + largeur * 2  
  
print(perimetre(2, 1))
```

On n'a malheureusement pas été attentif à la priorité des opérateurs et on a bêtement écrit la formule « longueur plus largeur fois deux » comme appris en primaire. Le problème est que le calcul fait est longueur+(largeur×2); l'exécution du programme affiche donc 4 au lieu de 6. Pour s'en rendre compte, il faudrait pouvoir dire à Python le résultat attendu. Mais d'un autre côté, si on écrit un programme c'est aussi pour que Python calcule ce résultat à notre place !

Heureusement, il existe des techniques, dont une basée sur des tests unitaires, qui permettent de traquer ce type d'erreur. Ces dernières ne seront pas abordées ici.



La Gestion de Erreurs en Python

Devoir Maison



3 Gérer les erreurs

3.1 La documentation

Une première façon de gérer les n'aborderons que la documentation

Le but de la documentation est de savoir comment l'appeler valeur de retour. Elle donne des paramètres et explique quelles seront Voici ce que ça pourrait donner pour la



erreurs passe par la documentation. Ici, nous des fonctions.

permettre à l'utilisateur d'une fonction de correctement et comment interpréter sa conditions à respecter sur les valeurs des les différentes valeurs de retour possibles. fonction « pourcentage »:

```
# Renvoie le pourcentage d'une note étant donné :
# - "note" contient la note obtenue (flottant)
# - "total" est la note maximale atteignable (flottant)
#
# Si total <= 0, note < 0 ou note > total, alors renvoie None
def pourcentage(note, total):
    if total > 0 and (0 <= note <= total):
        return note / total * 100
    return None
```

Ce qu'on vient d'écrire est une documentation informelle. Il n'y a pas vraiment de règles à suivre et on peut se contenter d'un texte en langue naturelle. L'important est de n'oublier aucune information, afin que l'on puisse appeler et utiliser la fonction, et interpréter sa valeur de retour, sans ambiguïté.

L'autre façon de décrire proprement une fonction consiste à établir ses spécifications comme présenté au chapitre 4. Pour rappel, les deux éléments suivants sont à définir :

- les préconditions sont les conditions qui doivent être satisfaites sur les paramètres et l'état global du programme, avant l'appel de la fonction ;
- les postconditions sont 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.

Une spécification est donc un contrat entre celui qui implémente une fonction et celui qui l'utilise. L'utilisateur s'engage à respecter les préconditions avant d'appeler la fonction et le programmeur lui garantit que les postconditions seront satisfaites en retour. Voici une nouvelle version de la fonction pourcentage, avec sa spécification :

```
# Calcule le pourcentage correspondant à une note.
#
# Pre: 0 <= note <= total, la note obtenue
#      total > 0, la note maximale atteignable
# Post: La valeur renvoyée contient le pourcentage
#       correspondant à la note obtenue.
def pourcentage(note, total):
    return note / total * 100
```




La Gestion de Erreurs en Python

Devoir Maison



Étant donné qu'il y a un contrat, le programmeur ne doit plus se soucier du cas où les paramètres sont négatifs, ou de celui où total est nul. Il ne doit garantir le bon comportement de la fonction que pour les cas où les préconditions sont respectées, ce qui simplifie la fonction.

Génération de la documentation

Il est de bon usage, en Python, d'insérer la documentation d'une fonction sous forme d'un « Docstring ». Il s'agit simplement de placer une chaîne de caractères, souvent délimitée par des guillemets triples pour pouvoir l'écrire sur plusieurs lignes, comme première instruction du corps de la fonction. L'exemple précédent se réécrit donc comme suit :

```
def pourcentage(note, total):
    """# Calcule le pourcentage correspondant à une note.
    #
    # Pre:  0 <= note <= total, la note obtenue
    #       total > 0, la note maximale atteignable
    # Post: La valeur renvoyée contient le pourcentage
    #       correspondant à la note obtenue.
    """
    return note / total * 100
```

3.2 Les assertions

Lorsqu'on définit une nouvelle fonction, et qu'on la spécifie, il faut minimiser le nombre de préconditions si on veut la rendre robuste, c'est-à-dire résistante à des mauvaises utilisations. Par contre, si on écrit une fonction à usage interne, c'est moins critique, surtout si le nombre de personnes qui vont l'utiliser n'est pas trop élevé et qu'elles sont de confiance. Dans ce dernier cas, le code de la fonction sera plus simple.

On peut vouloir vérifier que des conditions qui sont censées être satisfaites le sont effectivement, à l'aide du mécanisme d'assertion proposé par Python. Voyons comment l'utiliser pour vérifier les préconditions de la fonction pourcentage :

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

Trois instructions « assert » ont été utilisées pour vérifier les préconditions. Une telle instruction se compose d'une condition (une expression booléenne) éventuellement suivie d'une virgule et d'une phrase en langue naturelle, sous forme d'une chaîne de caractères. L'instruction « assert » teste si sa condition est satisfaite. Si c'est le cas, elle ne fait rien et sinon elle arrête immédiatement l'exécution du programme en affichant éventuellement la phrase qui lui est associée.

Dans le programme d'exemple suivant, le premier appel s'exécutera sans erreur tandis que le second provoquera une erreur d'exécution due à une assertion non satisfaite :

```
print(pourcentage(15, 20), '%')
print(pourcentage(22, 20), '%')
```



La Gestion de Erreurs en Python

Devoir Maison



```
75.0 %
Traceback (most recent call last):
  File "program.py", line 8, in <module>
    print(pourcentage(22, 20), '%')
  File "program.py", line 4, in pourcentage
    assert note <= total, 'note doit être inférieur à total'
AssertionError: note doit être inférieur à total
```

Le mécanisme d'assertion est là pour empêcher des erreurs qui ne devraient pas se produire, en arrêtant prématurément le programme, avant d'exécuter le code qui aurait produit une erreur. Si une telle erreur survient, c'est que le programme doit être modifié pour qu'elle n'arrive plus. Dans cet exemple, on se rend immédiatement compte qu'un appel ne respectant pas les préconditions a été fait, et qu'il faut donc le changer. C'était peut-être 12/20 au lieu de 22/20...

Enfin, il faut savoir que le mécanisme d'assertion est une aide au développeur, et ne doit en aucun cas faire partie du code fonctionnel d'un programme. En supprimant toutes les instructions `assert`, le programme doit continuer à fonctionner normalement.

La programmation défensive

Ce mode de programmation, qui exploite les assertions pour vérifier les préconditions, est appelé programmation défensive. Dans ce type de programmation, on suppose que les fonctions sont appelées comme il faut, dans le respect de leurs préconditions. On prévoit néanmoins un garde-fou avec des instructions « `assert` » pour vérifier que les préconditions sont effectivement bien satisfaites.

En pratique, on va utiliser ce type de programmation pour du code sur lequel on a le contrôle. Par exemple, lorsqu'on écrit un module, on peut programmer défensivement pour les fonctions qui ne sont destinées qu'à être appelées au sein de ce dernier. Lorsqu'on écrit des fonctions destinées à être utilisées par d'autres personnes, on va plutôt faire une gestion active des erreurs, notamment avec l'instruction `if-else` et en prévoyant des valeurs de retour spéciales.

Dans les deux cas, on doit spécifier les fonctions que l'on définit. En programmation défensive, on peut se permettre d'avoir des préconditions et faire l'hypothèse qu'elles seront toujours respectées. Par contre, lorsque les fonctions sont destinées à être utilisées par d'autres, on va limiter au maximum le nombre de préconditions.



La Gestion de Erreurs en Python

Devoir Maison



Prenons l'exemple du test d'une sous-chaine. Voici le code de la fonction « `issubsequence` », avec sa documentation :

```
def issubsequence(subseq, seq):
    """Teste si une séquence est une sous-séquence d'une autre.

    Pre : -
    Post: La valeur renvoyée contient True si 'subseq' est une
           sous-séquence de 'seq' et False sinon.
    """
    # Vérification des paramètres
    if type(subseq) != str or type(seq) != str:
        return False
    if len(subseq) > len(seq):
        return False
    # Teste la sous-séquence à toutes les positions possibles
    for i in range(0, len(seq) - len(subseq) + 1):
        if _issubsequenceat(subseq, seq, i):
            return True
    return False
```

On peut faire plusieurs observations sur la définition de cette fonction qui est, pour rappel, destinée à être appelée par d'autres programmeurs :

- on a limité drastiquement le nombre de préconditions, les ramenant même au nombre de zéro ;
- on a pratiqué la gestion active des erreurs, en vérifiant que les paramètres reçus par la fonction sont corrects, et en renvoyant False le cas échéant ;
- on a fait appel à une fonction « `_issubsequenceat` » au sein de la boucle, qui permet de tester si `subseq` est une sous-chaine de `seq` en commençant à son indice `i`.

Voyons maintenant le code de cette dernière fonction appelée par « `issubsequence` ». Comme elle n'est pas destinée à être utilisée en dehors du module, mais uniquement par les autres fonctions du même module, on va pouvoir pratiquer la programmation défensive :

```
def _issubsequenceat(subseq, seq, pos):
    """Teste si une séquence est une sous-séquence d'une autre
    en commençant à un indice donné.

    Pre : len(subseq) <= len(seq)
          0 <= pos <= len(seq) - len(subseq)
    Post: La valeur renvoyée contient True si 'subseq' est une
           sous-séquence de 'seq' commençant à l'indice 'pos'
           et False sinon.
    """
    # Vérification des préconditions
    assert type(subseq) == str and type(seq) == str
    assert type(pos) == int
    assert len(subseq) <= len(seq)
    assert 0 <= pos <= len(seq) - len(subseq)
    # Teste la sous-séquence à la position 'pos'
    for i in range(len(subseq)):
        if seq[pos + i] != subseq[i]:
            return False
    return True
```

<div>Lycée d'enseignement général et technologique international Victor Hugo COLOMIERS</div> <div></div>		<div>NSI</div> <div>NUMÉRIQUE ET SCIENCES INFORMATIQUES</div>
<div></div>	<div>La Gestion de Erreurs en Python</div>	<div>Victor Hugo</div>
	<div>Devoir Maison</div>	

On peut faire plusieurs observations sur la définition de cette fonction qui est, pour rappel, destinée à n'être appelée que depuis les autres fonctions du même module, en l'occurrence par « issubsequence » :

- on a ajouté `_` au début du nom de la fonction, suivant la convention Python, signalant ainsi qu'elle est privée au module ;
- on n'a pas hésité à définir plusieurs préconditions qui décrivent les paramètres valides ;
- on a pratiqué la programmation défensive, en vérifiant les préconditions à l'aide d'assertions.

3.3 Le mécanisme d'exception

On a vu que l'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 orientés objet, qui permet de gérer des exécutions exceptionnelles qui ne se produisent qu'en cas d'erreur.

3.3.1 L'instruction try-except

La construction de base à utiliser est l'instruction try-except qui se compose de deux blocs de code. On place le code « risqué » dans le bloc try et le code à exécuter en cas d'erreur dans le bloc except. Partons d'un exemple pour comprendre son utilisation :

```
anneeNaissance = input('Année de naissance ? ')

try:
    print('Tu as', 2021 - int(anneeNaissance), 'ans.')
except:
    print('Erreur, veuillez entrer un nombre.')

print('Fin du programme.')
```

On demande donc à l'utilisateur son année de naissance, grâce à la fonction prédéfinie input qui renvoie, pour rappel, une chaîne de caractères. On souhaite ensuite calculer son âge en soustrayant son année de naissance à 2021. Pour cela, il faut convertir la valeur de la variable anneeNaissance en un int. Cette conversion peut échouer si la chaîne de caractères entrée par l'utilisateur n'est pas un nombre. Voyons deux scénarios possibles d'exécution.

- Si l'utilisateur entre un nombre entier, l'exécution se passe sans erreur et son âge est calculé et affiché :

```
Année de naissance ? 1994
Tu as 27 ans.
Fin du programme.
```

- Si l'utilisateur entre une chaîne de caractères quelconque, qui ne représente pas un nombre entier, un message d'erreur est affiché :

```
Année de naissance ? deux
Erreur, veuillez entrer un nombre.
Fin du programme.
```

<div>Lycée d'enseignement général et technologique international Victor Hugo COLOMIERS</div> <div></div>		<div></div>
<div></div>	<div>La Gestion de Erreurs en Python</div>	<div></div>
	<div>Devoir Maison</div>	

Dans le premier cas, la conversion s'est passée sans souci, et le bloc try a donc pu s'exécuter intégralement sans erreur. L'exécution du programme se poursuit donc après l'instruction « try-except ». Dans le second cas, une erreur se produit dans le bloc try, lors de la conversion. L'exécution de ce bloc s'arrête donc immédiatement et passe au bloc except, avant de continuer également après l'instruction try-except.

On place donc uniquement le code qui est susceptible de générer une erreur dans le bloc try, avec le code qui en dépend. Il est important de ne pas placer trop de code dans ce dernier. Améliorons l'exemple précédent pour demander à l'utilisateur son année de naissance, en boucle jusqu'à ce qu'il fournisse une valeur correcte :

```
valid = False
while not valid:
    anneeNaissance = input('Année de naissance ? ')
    try:
        anneeNaissance = int(anneeNaissance)
        if 0 <= anneeNaissance <= 2021:
            valid = True
        else:
            print("L'année doit être comprise entre 0 et 2021.")
    except:
        print('Veuillez entrer un nombre naturel.')
print('Tu as', 2021 - anneeNaissance, 'ans.')
```

On initialise une variable valid à la valeur False. On rentre ensuite dans une boucle « while » qui va se répéter tant que la variable valid ne passe pas à True. Le corps de la boucle commence par demander à l'utilisateur son année de naissance, à l'aide de la fonction prédéfinie « input ». On entre ensuite dans une zone critique, placée donc dans un bloc try. On commence par tenter de convertir la variable anneeNaissance en un entier de type int. Si la conversion réussit, on s'assure que la valeur est bien comprise entre 0 et 2021 et, dans ce cas, on passe la valeur de valid à True, sinon on affiche un message d'erreur. Dans le bloc except, on affiche également un message d'erreur. On finit par calculer l'âge et l'afficher, en dehors de la boucle while.

Voici le résultat d'une exécution du programme :

```
Année de naissance ? BLA
Veuillez entrer un nombre naturel.

Année de naissance ? -12
L'année doit être comprise entre 0 et 2021.

Année de naissance ? 1973
Tu as 48 ans.
```

La boucle s'est donc exécutée trois fois avant que l'utilisateur n'entre une valeur valide et obtienne ainsi son âge.



La Gestion de Erreurs en Python

Devoir Maison



L'Objet Exception

Comme on l'a vu en début de chapitre, différents types d'erreurs peuvent survenir. Lorsqu'on utilise l'instruction try-except, le bloc except capture toutes les erreurs possibles qui peuvent survenir dans le bloc try correspondant. Une exception est en fait représentée par un objet, instance de la classe Exception. On peut récupérer cet objet en précisant un nom de variable après except comme dans cet exemple :

```
try:
    a = int(input('a ? '))
    b = int(input('b ? '))
    print(a, '/', b, '=', a / b)
except Exception as e:
    print(type(e))
    print(e)
```

On récupère donc l'objet de type Exception dans la variable e. Dans le bloc except, on affiche son type et sa valeur. Voici deux exemples d'exécution qui révèlent deux types d'erreurs différents :

- Si on ne fournit pas un nombre entier, il ne pourra être converti en int et une erreur de type ValueError se produit :

```
a ? trois
<class 'ValueError'>
invalid literal for int() with base 10: 'trois'
```

- si on fournit une valeur de 0 pour b, on aura une division par zéro qui produit une erreur de type ZeroDivisionError :

```
a ? 5
b ? 0
<class 'ZeroDivisionError'>
division by zero
```

Capter une erreur spécifique

Chaque type d'erreur est donc défini par une classe spécifique. On va pouvoir associer plusieurs blocs except à un même bloc try, pour exécuter un code différent en fonction de l'erreur capturée. Lorsqu'une erreur se produit, les blocs except sont parcourus l'un après l'autre, du premier au dernier, jusqu'à en trouver un qui corresponde à l'erreur capturée. Réécrivons l'exemple précédent en capturant les exceptions spécifiques pour les deux cas d'erreur qu'on a pu observer (erreur de conversion et division par zéro) :

```
try:
    a = int(input('a ? '))
    b = int(input('b ? '))
    print(a, '/', b, '=', a / b)
except ValueError:
    print('Erreur de conversion.')
except ZeroDivisionError:
    print('Division par zéro.')
except:
    print('Autre erreur.')
```



La Gestion de Erreurs en Python

Devoir Maison



Remarquez tout d'abord qu'on n'est pas obligé de spécifier une variable lorsqu'on capture une exception spécifique. Cette fois-ci, lorsqu'une erreur se produit dans le bloc try, ce sera l'un des blocs except seulement qui sera exécuté, selon le type de l'erreur qui s'est produite. Le dernier bloc except est là pour prendre toutes les autres erreurs.

L'ordre des blocs except est très important et il faut les classer du plus spécifique au plus général, celui par défaut devant venir en dernier. En effet, si on commence par un bloc except pour une exception de type Exception, il sera toujours exécuté et tous les autres qui le suivent ne le seront jamais. Dans l'exemple suivant, ce sera donc toujours « Autre erreur. » qui sera affiché dès qu'une erreur se produit dans le bloc try :

```
try:
    a = int(input('a ? '))
    b = int(input('b ? '))
    print(a, '/', b, '=', a / b)
except Exception:
    print('Autre erreur.')
except ValueError:
    print('Erreur de conversion.')
except ZeroDivisionError:
    print('Division par zéro.')
```

On aura, par exemple, le résultat suivant si on spécifie une valeur nulle pour le dénominateur :

```
a ? 2
b ? 0
Autre erreur.
```

Enfin, notez qu'il est possible d'exécuter le même code pour différents types d'erreurs, en les listant dans un tuple après le mot réservé except. Si on souhaite exécuter le même code pour une erreur de conversion et de division par zéro, il faudrait écrire :

```
try:
    a = int(input('a ? '))
    b = int(input('b ? '))
    print(a, '/', b, '=', a / b)
except (ValueError, ZeroDivisionError) as e:
    print('Erreur de calcul : ', e)
except:
    print('Autre erreur.')
```

Le deuxième bloc except capture donc les erreurs de type ValueError et ZeroDivisionError. L'exception capturée est stockée dans la variable e que l'on affiche pour avoir des informations sur la cause de l'erreur apparue. On pourrait, par exemple, observer le résultat suivant lors d'une exécution du programme :

```
a ? 2
b ? 0
Erreur de calcul : division by zero
```




La Gestion de Erreurs en Python

Devoir Maison



La propagation d'erreurs

Que se passe-t-il lorsqu'on ne capture pas une exception à l'aide d'une instruction try-except ? Cette dernière va en fait remonter la séquence des appels de fonctions. Prenons, par exemple, le programme suivant qui comporte deux fonctions, appelées en chaîne :

```
def fun():  
    print(1 / 0)  
  
def compute():  
    fun()  
  
compute()
```

Le programme appelle donc la fonction compute qui, elle-même, appelle la fonction fun, cette dernière produisant une erreur à cause de la division par zéro. Voici le résultat que l'on obtient en exécutant ce programme :

```
Traceback (most recent call last):  
  File "program.py", line 7, in <module>  
    compute()  
  File "program.py", line 5, in compute  
    fun()  
  File "program.py", line 2, in fun  
    print(1 / 0)  
ZeroDivisionError: division by zero
```

Dans un sens, la trace d'erreur permet de suivre le déroulement de l'exécution, partant du programme principal vers la fonction fun, en passant par la fonction compute. Dans l'autre sens, on peut suivre la propagation de l'erreur depuis la fonction fun vers le programme principal, en passant par la fonction compute.

Une erreur qui n'est pas capturée va donc se propager, en remontant la séquence des appels de fonctions qui ont été faits. On peut modifier le programme en capturant, par exemple, l'erreur dans la fonction compute :

```
def fun():  
    print(1 / 0)  
  
def compute():  
    try:  
        fun()  
    except:  
        print('Erreur.')  
compute()
```

Dans ce cas, l'erreur qui est générée dans la fonction fun va juste se propager dans la fonction compute où elle sera arrêtée par l'instruction try-except. Le résultat de l'exécution est donc :

```
Erreur.
```

Notez qu'il est important de capturer l'erreur à l'endroit où son traitement est le plus approprié.



La Gestion de Erreurs en Python

Devoir Maison



Le bloc « finally »

Parfois, on souhaite exécuter des instructions dans tous les cas, avant que l'exécution ne continue après le bloc try-except. Le problème est que lorsqu'une erreur se produit dans le bloc try, l'exécution de ce dernier est interrompue pour se poursuivre dans le bloc except correspondant.

Pour cela, on peut utiliser le mot réservé finally qui permet d'introduire un bloc qui sera exécuté soit après l'exécution complète sans erreur du bloc try, soit après avoir exécuté le bloc except correspondant à l'erreur qui s'est produite lors de l'exécution du bloc try. On obtient ainsi une instruction try-except-finally dont voici un exemple d'utilisation :

```
print('Début du calcul.')
try:
    a = int(input('a ? '))
    b = int(input('b ? '))
    print('Résultat :', a / b)
except:
    print('Erreur.')
finally:
    print('Nettoyage de la mémoire.')
print('Fin du calcul.')
```

Si l'utilisateur fournit des valeurs correctes pour a et b, l'exécution du bloc try se passera sans erreur, c'est-à-dire que le résultat de la division de a par b sera affiché, puis que la phrase signalant que la mémoire est nettoyée sera affichée :

```
Début du calcul.
a ? 2
b ? 8
Résultat : 0.25
Nettoyage de la mémoire.
Fin du calcul.
```

Le bloc finally a donc bien été exécuté, après l'exécution sans erreur du bloc try et avant que l'exécution du programme ne continue après l'instruction try-except.

Par contre, si une erreur se produit, alors le résultat ne sera pas affiché puisque le bloc try sera arrêté ; on aura simplement l'affichage d'un message d'erreur. Par contre, la phrase signalant que la mémoire est nettoyée sera de nouveau affichée témoignant bien du fait que le bloc finally a été exécuté, cette fois-ci après le bloc except, mais toujours avant la reprise de l'exécution après l'instruction try-except :

```
Début du calcul.
a ? 2
b ? 0
Erreur.
Nettoyage de la mémoire.
Fin du calcul.
```

Enfin, notez que l'on peut se limiter à une instruction try-finally, sans définir un seul bloc except. Dans ce cas, si une erreur se produit dans le bloc try, elle sera propagée après exécution du bloc finally.



La Gestion de Erreurs en Python

Devoir Maison



3.3.2 Définir une exception

Terminons en voyant comment définir ses propres exceptions. Comme on a déjà pu le voir, une exception est un objet et du coup, il faut qu'il existe quelque part une classe pour qu'on puisse en créer des instances. Définir un nouveau type d'exception revient donc à définir une nouvelle classe.

Génération d'erreurs

Avant de définir nos propres exceptions, voyons comment générer une erreur dans un programme grâce à l'instruction « raise ». Il suffit en fait d'utiliser le mot réservé « raise » suivi d'une référence vers un objet représentant une exception. L'exemple suivant reprend la fonction récursive permettant de calculer la factorielle d'un nombre naturel positif :

```
def fact(n):
    if n < 0:
        raise ArithmeticError()
    if n == 0:
        return 1
    return n * fact(n - 1)

print(fact(-12))
```

Dans cet exemple, nous avons ajouté une instruction if testant si n est strictement négatif, cas dans lequel on génère une exception de type ArithmeticError, qui représente une erreur arithmétique.

Si on exécute ce programme qui tente de calculer la factorielle de -12, il va s'arrêter brutalement avec l'erreur suivante :

```
Traceback (most recent call last):
  File "program.py", line 8, in <module>
    print(fact(-12))
  File "program.py", line 3, in fact
    raise ArithmeticError()
ArithmeticError
```

L'erreur observée est de type ArithmeticError, et c'est précisément celle qu'on a générée à la deuxième ligne du corps de la fonction fact avec l'instruction raise. On va, évidemment, pouvoir se protéger de cette erreur à l'aide d'un try-except lorsqu'on appelle la fonction. Le programme suivant capture spécifiquement l'exception de type ArithmeticError :

```
try:
    n = int(input('Entrez un nombre : '))
    print(fact(n))
except ArithmeticError:
    print('Veuillez entrer un nombre positif.')
except:
    print('Veuillez entrer un nombre.')
```

Lycée d'enseignement général et technologique international Victor Hugo COLOMIERS		
	<div>La Gestion de Erreurs en Python</div> <div>Devoir Maison</div>	

Cette fois-ci, on n'aura plus de trace d'erreur puisque l'erreur est capturée et gérée. On obtient, par exemple, le résultat d'exécution suivant :

```
Entrez un nombre : -12
Veuillez entrer un nombre positif.
```

Si on avait entré du texte ne correspondant pas à un nombre entier, le gestionnaire d'erreur par défaut (le dernier except) aurait été exécuté.

Enfin, il faut savoir que, dans une fonction, l'instruction raise se comporte comme l'instruction return, à savoir qu'elle a pour conséquence que l'exécution du corps de la fonction est immédiatement quittée. L'erreur se propage ensuite en remontant la séquence des appels de fonctions, jusqu'à être attrapée. Si elle n'est jamais attrapée, le programme se termine en erreur.

Créer un type d'exception

Dans l'exemple précédent, on a généré une exception de type ArithmeticError, dont la classe existe déjà en Python. On pourrait se limiter aux types d'exceptions déjà existants, voire utiliser le type générique Exception, mais il est parfois plus pratique et plus lisible de définir ses propres types. Pour cela, il suffit de définir une nouvelle classe, comme le montre l'exemple suivant :

```
class NoRootException(Exception):
    pass
```

Cette classe ne fait rien puisque son corps n'est constitué que de l'instruction pass. Mais intéressons-nous à sa définition. On remarque une petite différence par rapport à une définition « usuelle » de classe, c'est le (Exception) qui a été ajouté entre parenthèses après le nom de la classe. Sans rentrer dans le détail, cela permet de signifier à Python que cette classe est de « type exception », ce qui permettra de l'utiliser avec try-except et raise.

Une fois cette classe définie, on va pouvoir l'utiliser. Définissons une fonction « trinomialroots » qui calcule et renvoie les racines d'un trinôme du second degré de la forme « ax^2+bx+c », qui génère une erreur lorsqu'il n'y a pas de racine réelle :

```
from math import sqrt

def trinomialroots(a, b, c):
    delta = b ** 2 - 4 * a * c
    # Aucune racine réelle
    if delta < 0:
        raise NoRootException()
    # Une racine réelle double
    if delta == 0:
        return -b / (2 * a)
    # Deux racines réelles simples
    x1 = (-b + sqrt(delta)) / (2 * a)
    x2 = (-b - sqrt(delta)) / (2 * a)
    return (x1, x2)
```



La Gestion de Erreurs en Python

Devoir Maison



La quatrième ligne du corps de la fonction génère une erreur de type `NoRootException` à l'aide de l'instruction `raise`, dans le cas où le trinôme n'admet pas de racine réelle. Pour que ce programme fonctionne, il faut évidemment que la classe `NoRootException` soit définie dans le même fichier que la fonction `trinomialroots`.

Lorsqu'on appelle cette fonction, on va donc pouvoir utiliser l'instruction `try-except` pour attraper cette erreur, lorsqu'elle survient. Essayons, par exemple, de calculer et d'afficher les racines réelles du trinôme x^2+2 . Pour cela, on appelle donc la fonction `trinomialroots` en lui passant en paramètres 1, 0 et 2 puisque x^2+2 correspond à $a=1$, $b=0$ et $c=2$:

```
try:
    print(trinomialroots(1, 0, 2))
except NoRootException:
    print('Pas de racine réelle.')
```

Puisque ce trinôme n'admet pas de racine réelle, la parabole correspondante se trouvant au-dessus de l'axe des x , une erreur sera générée et capturée par le bloc `except`, comme on le constate sur le résultat de l'exécution du programme :

```
Pas de racine réelle.
```

Tout ceci est possible car on a défini une nouvelle classe de « type exception », permettant de générer une erreur de ce type avec `raise` et de l'attraper avec `except`.

**** Fin du document ****