Chap 8

Les exceptions

Enseignante: Mme Lamia MANSOURI

1. Présentation des exceptions

- Comme d'autres langages, Python fournit également la gestion d'exceptions à l'aide de **try-except**.
- Pour les erreurs détectées lors de l'exécution, 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.

Il y a deux sortes d'erreurs:

- **Erreur de syntaxe** : Également appelée erreur d’analyse, la plus élémentaire. Apparaît lorsque l'analyseur Python est incapable de comprendre une ligne de code.
- **Exception**: erreurs détectées lors de l'exécution. par exemple: **ZeroDivisionError, TypeError**.

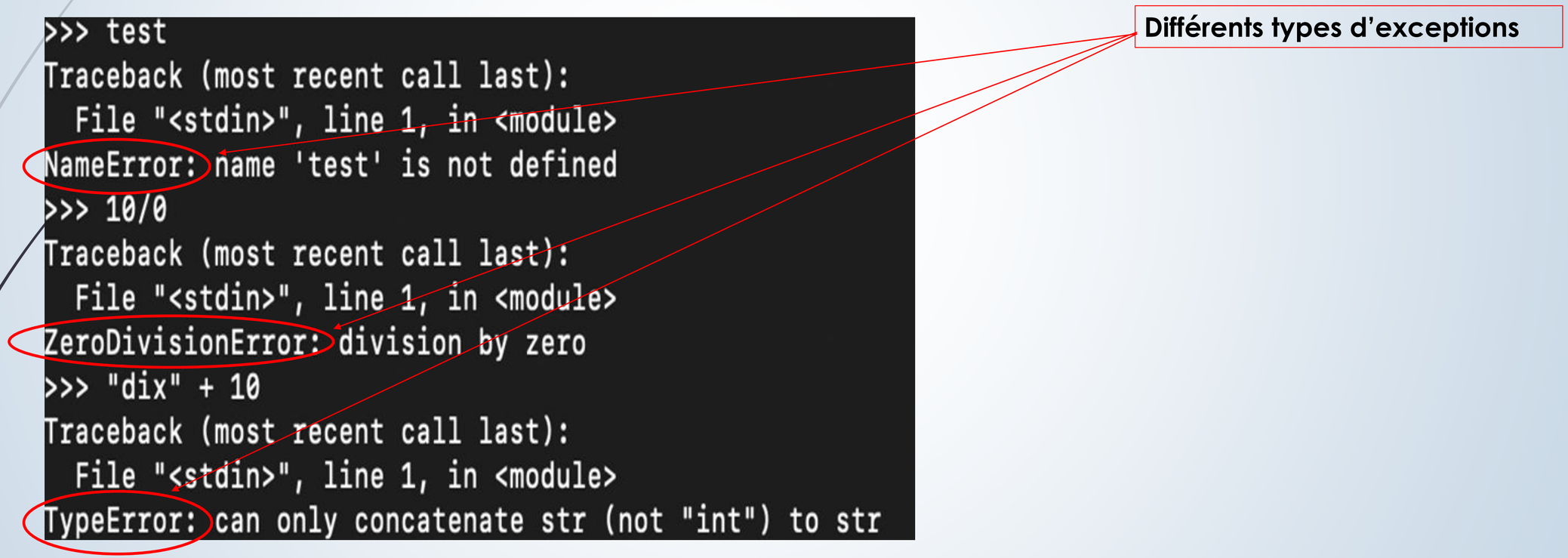
- Une exception est dite **levée** lorsqu'une erreur apparait.
- Une exception est dite **capturée** lorsqu'elle est gérée et traitée.

Le mécanisme des exceptions permet donc au programme de "rattraper" les erreurs, de détecter qu'une erreur s'est produite et d'agir en conséquence afin que le programme ne s'arrête pas.

Dans Python , il existe deux types d'exception :

- celles présentent dans la table des exceptions de Python
- celles créées par l'utilisateur

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



2. Mécanisme des exceptions

Le mécanisme des exceptions permet au programme de "rattraper" les erreurs, de détecter qu'une erreur s'est produite et d'agir en conséquence afin que le programme ne s'arrête pas.

2.1 Instruction try-except

Syntaxe:

```
try: # On test un bout de code
    # Code à tester
except:
    # Code éxecuté en cas d'erreur
```

Comment fonctionne try()?

- La première clause **try** est exécutée, c'est-à-dire le code entre **try** et **except** clause.
- S'il n'y a pas d'exception, alors seule la clause **try** sera exécutée.
- Si une exception survient, la clause **try** sera ignorée et la clause **except** sera exécutée.
- Si une exception se produit, mais que la clause **except** ne la gère pas, elle est transmise aux instructions **try** externes.
- Si l'exception n'est pas gérée, l'exécution s'arrête.
- Une instruction **try** peut avoir plus d'une clause except

Exemples :

```
try:
    resultat = numerateur / denominateur
except NameError:
    print( "La variable numérateur ou dénominateur n'a pas été définie.  ")
except TypeError:
    print("Le numérateur ou dénominateur est d'un type incompatible avec une division. ")
except ZeroDivisionError:
    print("La variable dénominateur est égale a 0. "  )
```

```
valid = False
while not valid:
    an = input(" Année de naissance ? ")
    try:
        an=int(an)
        if 0 <= an <= 2020:
            valid = True
        else:
            print("L'année doit être comprise entre 0 et 2020.")
    except :
        print(" Veuillez entrer un nombre naturel. ")

print(" Tu as ", 2020 - an, " ans. ")
```

Le type Exception

Dans un programme , 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 :

- 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
```

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 1er 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.

Exemple d'exécution du programme :

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

2.2 La clause else

Syntaxe:

```
try: # On test un bout de code
    # Code à tester
except:
    # Code exécuté en cas d'erreur
else:
    #Code else
```

Le code entre dans le bloc **else** uniquement si la clause **try** ne déclenche pas d'exception.

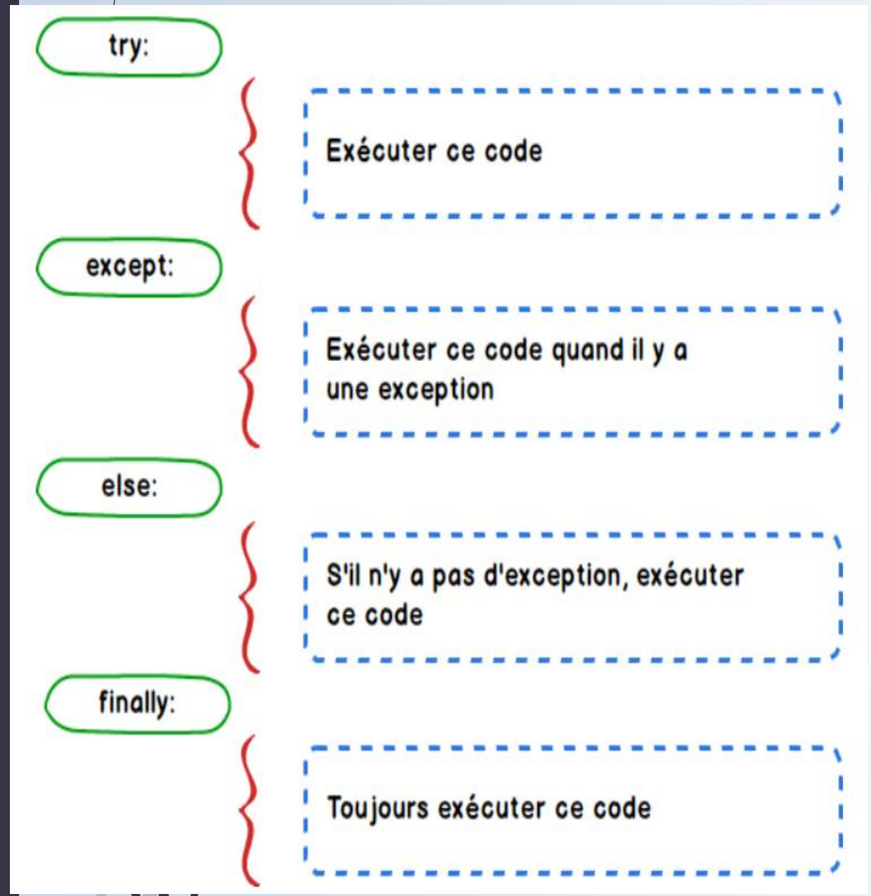
2.3 la clause Finally

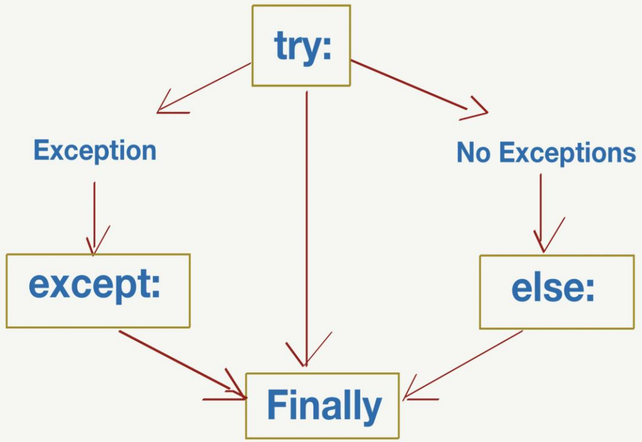
Syntaxe:

```
try :
    # Code à tester
except Exception as e :
    # Code exécuté en cas d'erreur
finally :
    # Le bloc finally est toujours exécuté.
```

La gestion des exception complète est **try-except-finally.** Le bloc **finally** est toujours exécuté, que l'exception se produise ou non dans le bloc **try**.

Syntaxe complète :





Exemple :

```
try:
    x=int(input("saisir un numérateur:"))
    y=int(input("saisir un dénominateur:"))
    z=x/y
except ValueError:
    print("Error : valeur invalide")
except ZeroDivisionError:
    print("Error : division par zéro!!")
else :
    print("résultat de la division ", z)
finally:
    print("fin du programme ! ")
```

```
saisir un numérateur:1
saisir un dénominateur:0
Error : division par zéro!!
fin du programme !
```

2.4 Lever une exception

- L'interpréteur est responsable de la levée de toutes les exceptions évoquées jusqu'à présent. Ces exceptions résultent de la rencontre d'erreurs lors de l'exécution.

- Un programmeur peut lui aussi souhaiter lancer une exception face à une entrée erronée par exemple grâce à l'instruction **raise.**
- Plusieurs méthodes pour l'utilisation de **raise :**

Méthode 1: raise exception_type(message)

```
def inverse(x):
    if x == 0:
        raise ValueError("valeur nulle interdite!!")
    y = 1.0 / x
    return y

try:
    print(inverse(0))  # erreur
except ValueError as exc:
    print("erreur, message :", exc)
```

Méthode 2

```
def inverse(x):
if x == 0:
        raise ValueError
    y = 1.0 / x
    return y

try:
    print(inverse(0))  # erreur
except ValueError:
    print("erreur de type ValueError")
```

Méthode 3

```
def inverse(x):
    if x == 0:
        raise
    y = 1.0 / x
    return y

try:
    print(inverse(0))  # erreur
except :
    print("erreur, message :")
```

2.5 Propagation des erreurs

12

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 chaine :

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

def compute():
    fun()

compute()
```

```
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
```

Le programme appelle la fonction **compute** qui, elle-même, appelle la fonction **fun**.
Cette dernière produisant une erreur à cause de la division par zéro.

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

def compute():
    try:
        fun()
    except:
        print('Erreur!!!')

compute()
```

```
Erreur!!!
```

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**.

2.6 Instructions try, except imbriquées

Il est possible d'imbriquer les portions protégées de code les unes dans les autres.

Dans l'exemple qui suit, la première erreur est l'appel à une fonction non définie, ce qui déclenche l'exception NameError.

Exemple 1

```
def inverse(x):
    y = 1.0 / x
    return y

try:
    try:
        print(inverses(0))  # fonc. inexistante --> exception NameError
        print(inverse(0))   # division par zéro --> ZeroDivisionError
    except NameError:
        print("appel à une fonction non définie")
except ZeroDivisionError as exc:
    print("erreur", exc)
```

appel à une fonction non définie

Exemple 2

```
def inverse(x):
    y = 1.0 / x
    return y

try:
    try:
        print(inverse(0))   # division par zéro --> ZeroDivisionError
        print(inverses(0))  # fonc. inexistante --> exception NameError
    except NameError:
        print("appel à une fonction non définie")
except ZeroDivisionError as exc:
    print("erreur", exc)
```

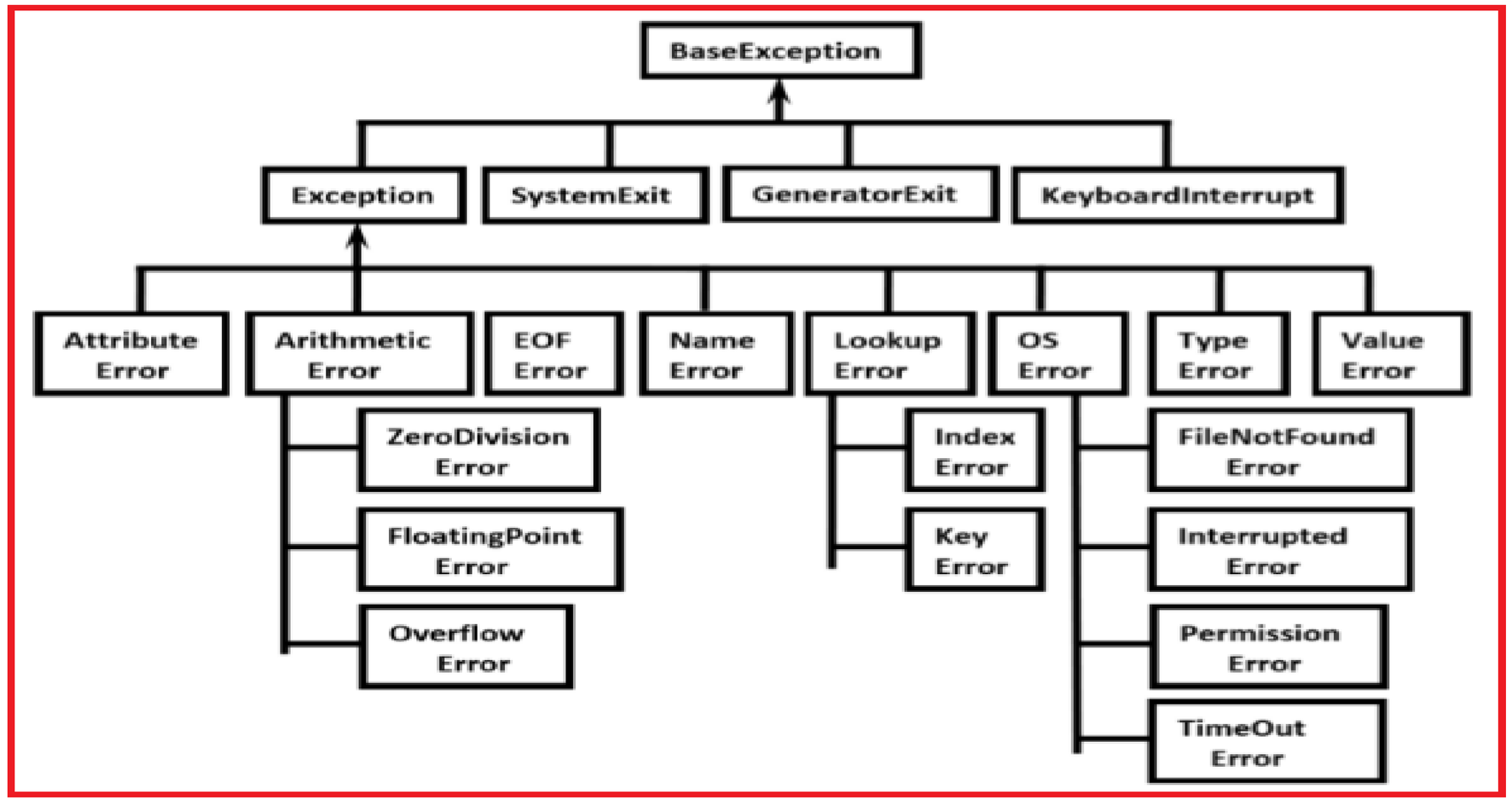
erreur float division by zero

3. Les classes exception

- Python possède de nombreuses classes d'exceptions natives.
- Toute exception est une instance (un objet) créé à partir d'une classe exception.

Voici est le modèle d'hiérarchie des exceptions en Python:

- La classe la plus élevée est **BaseException**
- Les sous-classes directes



Quelques 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

- 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*** ;

Exemple

```
def division(a, b):
    try:
        d = a//b
    except ArithmeticError:
        print("Erreur")
```

- 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.

Exemple

```
T = [1, 2, 3]
try:
    print(T[4])
except LookupError:
    print("Indice est supérieur à la taille du tableau")
```

- La classe **AttributeError** est déclenchée lorsqu'une référence d'attribut ou une attribution échoue, par exemple lorsqu'un attribut non existant est référencé.

```
class Personne:
    def __init__(self, nom):
        self.nom = nom

 try:
    p = Personne(" ahmed")
    print(p.prenom)
except AttributeError:
    print("Erreur d'attribut")
```

4. Définir une exception

Il est possible de définir un nouveau type d'exception pour un type d'erreur spécifique. Pour cela, il faut créer une nouvelle classe dérivant de la classe **BaseException** ou d'une de ses sous-classes (Exception par exemple).

L'exemple suivant crée une exception **AucunChiffre** qui est lancée par la fonction conversion lorsque la chaîne de caractères qu'elle doit convertir ne contient pas que des chiffres.

```
class AucunChiffre(Exception):
    """
    chaîne de caractères contenant aussi autre chose que des chiffres
    """
    pass

def conversion(s):
    if not s.isdigit():
        raise AucunChiffre(s)
    return int(s)

try:
    s = "123a"
    print(s, " = ", conversion(s))
except AucunChiffre as exc:
    # on affiche ici le commentaire associé à la classe d'exception
    # et le message associé
    print(AucunChiffre.__doc__, " : ", exc)
```

chaîne de caractères contenant aussi autre chose que des chiffres
: 123a

5. Les assertions

Les assertions sont un moyen simple de s'assurer, avant de continuer, qu'une condition est respectée.

Ce sont simplement des expressions booléennes qui vérifie si les conditions renvoient true ou non. Si l'expression est vraie, le programme ne fait rien et passe à la ligne de code suivante. Cependant, si elle est fausse, une exception **AssertionError** est levée.

Syntaxe

```
assert <condition>
```

```
assert <condition>,<error message>
```

```
num=int(input('Entrer un nombre: '))
assert num>=0
print('vous avez saisi: ', num)
```

L'instruction d'impression s'affichera si le nombre entré est supérieur ou égal à 0. Les nombres négatifs entraînent l'abandon du programme après l'affichage de l'AssertionError.

```
Entrez un nombre: 100
Vous avez entré 100

Entrez un nombre: -10
Traceback (dernier appel le plus récent):
  Fichier "C: /python36/xyz.py", ligne 2, dans <module>
    assert num> = 0
AssertionError
```

```
num=int(input(' Entrer un nombre : '))
assert num>=0, " Seuls les nombres positifs sont acceptés."
print('vous avez saisi : ', num)
```

```
Entrez un nombre: -10
Traceback (dernier appel le plus récent):
  Fichier "C: /python36/xyz.py", ligne 2, dans <module>
    assert num> = 0, "Seuls les nombres positifs sont acceptés."
    AssertionError: Seuls les nombres positifs sont acceptés.
```