CHAPITRE 4

Administrer les exceptions

1. **Types d’erreur et expérimentation**
2. Types des exceptions
3. Gestion des exceptions

Erreurs de syntaxe

- Les erreurs de syntaxe sont des erreurs d'analyse du code

Exemple :

```
>>> while True print('Hello world')
  File "<stdin>", line 1
    while True print('Hello world')
               ^
SyntaxError: invalid syntax
```

- L'analyseur indique la ligne incriminée et affiche une petite « flèche » pointant vers le premier endroit de la ligne où l'erreur a été détectée.
- L'erreur est causée par le symbole placé avant la flèche.
- Dans cet exemple, la flèche est sur la fonction print() car il manque deux points (':') juste avant. Le nom du fichier et le numéro de ligne sont affichés pour vous permettre de localiser facilement l'erreur lorsque le code provient d'un script.

Exceptions

- Même si une instruction ou une expression est syntaxiquement correcte, elle peut générer une erreur lors de son exécution.
- Les erreurs détectées durant l'exécution sont appelées des exceptions et ne sont pas toujours fatales
- La plupart des **exceptions** toutefois ne sont pas prises en charge par les programmes, ce qui génère des messages d'erreurs comme celui-ci :

```
>>> 10 * (1/0)
Traceback (most recent call last):
ZeroDivisionError: division by zero
```

- La dernière ligne du message d'erreur indique ce qui s'est passé. Les exceptions peuvent être de différents types et ce type est indiqué dans le message : le types indiqué dans l'exemple est ZeroDivisionError

CHAPITRE 4

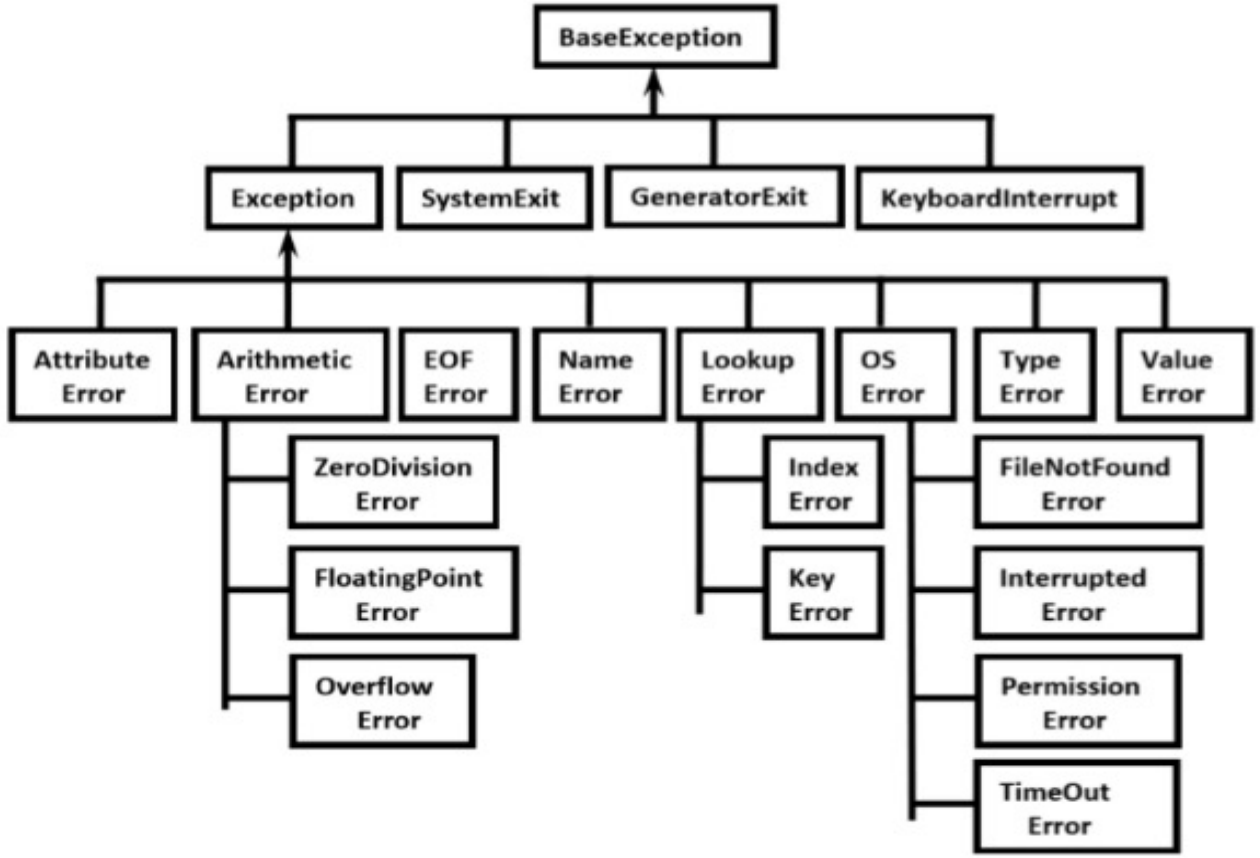
Administrer les exceptions

1. Types d’erreur et expérimentation
2. **Types des exceptions**
3. Gestion des exceptions

- En Python, les erreurs détectées durant l'exécution d'un script sont appelées des exceptions car elles correspondent à un état "exceptionnel" du script
- Python analyse le type d'erreur déclenché
- Python possède de nombreuses classes d'exceptions natives et toute exception est une instance (un objet) créée à partir d'une classe exception
- La classe d'exception de base pour les exceptions natives est **BaseException**
- Quatre classes d'exception dérivent de la classe **BaseException** à savoir :

Exception	• Toutes les exceptions intégrées non-exit du système sont dérivées de cette classe • Toutes les exceptions définies par l'utilisateur doivent également être dérivées de cette classe
SystemExit	• Déclenchée par la fonction sys.exit() si la valeur associée est un entier simple, elle spécifie l'état de sortie du système (passé à la fonction exit() de C)
GeneratorExit	• Levée lorsque la méthode close() d'un générateur est appelée
KeyBoradInterrupt	• Levée lorsque l'utilisateur appuie sur la touche d'interruption (normalement Control-C ou Delete)

- Il y a également d'autres classes **d'exception** qui dérivent de Exception telles que:
  - 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 **LookupError** est la classe de base pour les exceptions qui sont levées lorsqu'une clé ou un index utilisé sur un tableau de correspondances où une séquence est invalide.
- De nombreuses classes dérivent ensuite de ces classes.
- En fonction de l'erreur rencontrée par l'analyseur Python, un objet exception appartenant à telle ou telle classe exception va être créé et renvoyé. Cet objet est intercepté et manipulé.



CHAPITRE 4

Administrer les exceptions

1. Types d’erreur et expérimentation
2. Types des exceptions
3. **Gestion des exceptions**

Détection et traitement des exceptions en Python

- On peut détecter les exceptions en plaçant les instructions qui peuvent générer des exceptions dans un bloc **try**.
- Il existe 2 formes d'expressions try mutuellement exclusives (on ne peut en employer qu'une à la fois) : **try-except et try-finally.**
- Une instruction **try peut être accompagnée d'une ou plusieurs clauses except**, d'une **seule clause finally** ou d'une combinaison **try-except-finally**.

Exemple :

- On souhaite calculer l'âge saisi par l'utilisateur en soustrayant son année de naissance à 2016. Pour cela, il faut convertir la valeur de la variable birthyear en un int.
- Cette conversion peut échouer si la chaine de caractères entrée par l'utilisateur n'est pas un nombre.

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


try:
    print('Tu as', 2016 - int(birthyear), 'ans.')  #code risqué
except:
    print('Erreur, veuillez entrer un nombre.') #code à exécuter en cas d'erreur


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

Instruction try-except

```
birthyear = input('Année de naissance ? ')
try:
    print('Tu as', 2016 - int(birthyear), 'ans.')  #code risqué
except:
    print('Erreur, veuillez entrer un nombre.') #code à exécuter en cas d'erreur

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

- Dans le premier cas, la conversion s'est passée normalement, et le bloc try a donc pu s'exécuter intégralement sans erreur.
- 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.

1er cas

- 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 22 ans.
Fin du programme.
```

2ème cas

- Si l'utilisateur entre une chaine 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.
```

Type Exception

- 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**.

```
try:
    i = int(input('i ? '))      #code à risque
    j = int(input('j ? '))      #code à risque
    print(i, '/', j, '=', i / j)  #code à risque
except Exception as e:    # variable e de type Exception
    print(type(e))            #afficher le type de l'exception
    print(e)                  #afficher l'exception
```

- On récupère donc l'objet de type Exception dans la variable e.
- Dans le bloc except, on affiche son type et sa valeur.

Type Exception

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 :

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

- Si on fournit une valeur de 00 pour b, on aura une division par zéro qui produit une erreur de type ZeroDivisionError :

```
i ? 5
j ? 0
#affiche:
#<class 'ZeroDivisionError'>
#division by zero
```

Capture d'erreur spécifique

- Chaque type d'erreur est donc défini par une classe spécifique.
- Il est possible d'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.

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

Exemple :

- Lorsqu'une erreur se produit dans le bloc try 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.**

Gestionnaire d'erreur partagé

- Il est possible d'exécuter le même code pour différents types d'erreur, 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:
    i= int(input('i ? '))
    j = int(input('j ? '))
    print(i, '/', j, '=', i / j)
except (ValueError, ZeroDivisionError) as e:
    print('Erreur de calcul :', e)  # exécuter le même code pour différents types d'exceptions
except:
    print('Autre erreur.')
```

Bloc finally

- Le mot réservé **finally** permet d'introduire un bloc qui sera exécuté soit après que le bloc try se soit exécuté complètement sans erreur, 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**

```
print('Début du calcul.')
try:
    i = int(input('i ? '))
    j = int(input('j ? '))
    print('Résultat :', i / j)
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'affichage est le suivant :

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

- Si une erreur se produit l'affichage est le suivant :

```
Début du calcul.
i ? 2
j ? 0
Erreur.
Nettoyage de la mémoire.
Fin du calcul.
```

Dans les 2 cas le bloch finally a été exécuté

PARTIE 3

Génération d'erreur

- Il est possible de générer une erreur dans un programme grâce à l'instruction **raise**.
- Il suffit en fait simplement d'utiliser le mot réservé **raise** suivi d'une référence vers un objet représentant une exception.

Exemple :

```
def factoriel(a):
    if a < 0:
        raise ArithmeticError()  #signaler une erreur de type ArithmeticError si n<0
    if a == 0:
        return 1
    return n * factoriel(a - 1)
```

- Si n est strictement négatif, une exception de type **ArithmeticError** est générée.

- Le programme suivant permet de capturer spécifiquement l'exception de type **ArithmeticError** lors de l'appel de la fonction **fact**.

```
try:
    a = int(input('Entrez un nombre : '))    #code à risque
    print(factoriel(a))  #code à risque
except ArithmeticError:  #capturer  de l'exception ArithmeticError
        print('Veuillez entrer un nombre positif.')  #afficher le message si l'exception
                                                     #ArithmeticError est capturée
except:   #capturer les autres types d'exception
  print('Veuillez entrer un nombre.')  #afficher le message si d'autres types
#d'exceptions sont capturées
```

PARTIE 3

Créer un type d'exception

- Il est parfois plus pratique et plus lisible de définir nos propres types d'exceptions
- Pour cela, il suffit de définir une nouvelle classe qui hérite de la classe Exception

Exemple :

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

- Cette classe est tout simplement vide puisque son corps n'est constitué que de l'instruction pass

Exemple :

- Définissons une fonction **trinomial** qui calcule et renvoie les racines d'un trinôme du second degré de la forme ax2+bx+c et qui génère une erreur lorsqu'il n'y a pas de racine réelle :

```
from math import sqrt
def trinomial(a, b, c):
    delta = b ** 2 - 4 * a * c
    # Aucune racine réelle
    if delta < 0:
        raise NoRootException() #lever une exception de type 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)
```

Exception paramétrée

- Lorsqu'on appelle la fonction **trinomial**, 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. Pour cela, on appelle donc la fonction trinomial en lui passant en paramètres 1, 0 et 2 puisque x+2 correspond à a=1, b=0 et c=2

```
try:  #Attraper une exception avec le bloc try-except
    print(trinomial(1, 0, 2))
except NoRootException:
    print('Pas de racine réelle.')
```

- Pour l'exemple précédent, il pourrait être utile de connaitre la valeur du discriminant
- Lorsqu'aucune racine réelle n'existe. Pour cela il faut ajouter une variable d'instance et
- Un accesseur à la classe **NoRootException**

```
class NoRootException(Exception):
    def __init__(self, delta): #définition d'un constructeur avec paramètre
        self.__delta = delta
    @property
    def delta(self):
        return self.__delta
```

Exception paramétrée

- Il est possible de récupérer la valeur du discriminant dans le bloc **except**, à partir de l'objet représentant l'exception qui s'est produite

```
from math import sqrt
def trinomial(a, b, c):
    delta = b ** 2 - 4 * a * c
    # Aucune racine réelle
    if delta < 0:
        raise NoRootException(delta) #lever une exception avec paramètre
    # 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)
```

```
try:
    print(trinomial(1, 0, 2))
except NoRootException as e:
    print('Pas de racine réelle.')
    print('Delta =', e.delta) #récupérer la valeur du paramètre
```