

## Séance 5

# Gestion d'erreurs et mécanisme d'exceptions



Ce(tte) œuvre est mise à disposition selon les termes de la Licence Creative Commons Attribution – Pas d'Utilisation Commerciale – Pas de Modification 4.0 International.

# Rappels

- Création d'**interface graphique**
  - Création de la fenêtre principale et de widgets
  - Gestionnaire de mise en page et placement des widgets
  - Séparation de la présentation dans un fichier .kv
- Programmation **évenementielle**
  - Application graphique
  - Gestionnaire d'évènements et binding de fonctions
  - La propriété canvas et dessin de formes

# Objectifs

- Gestion d'**erreurs**
  - Programmation défensive
  - Spécification
  - Instruction assert
- Mécanisme d'**exception**
  - Instruction try-except-finally
  - instruction raise
  - Définition d'une nouvelle exception

# Gestion d'erreurs

always  
make new  
mistakes

(esther dyson)



# Trace d'erreur (1)

- Une erreur d'exécution imprime une **trace d'erreur**

*Chemin d'exécution complet qui a provoqué l'erreur*

```
1 def percentage(score, total):  
2     return score / total * 100  
3  
4 print('Alexis a obtenu', percentage(18, 20), '%')  
5 print('Sébastien a obtenu', percentage(6, 0), '%')
```

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

# Trace d'erreur (2)

- L'erreur a comme **origine** l'exécution de l'instruction en ligne 5

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

- L'erreur provient d'un **appel de fonction**

```
File "program.py", line 2, in percentage
    return score / total * 100
```

- L'erreur est de type **division par zéro**

```
ZeroDivisionError: division by zero
```

# Gestion d'erreurs

- Prendre en compte **tous les cas** possibles d'exécution

*Prévoir une valeur de retour spéciale en cas d'erreur*

```
1 def percentage(score, total):  
2     if total != 0:  
3         return score / total * 100  
4     return None
```

```
Alexis a obtenu 90.0 %  
Sébastien a obtenu None %
```

# Types d'erreur

- On peut considérer **trois types d'erreur** possibles

- **Erreur de syntaxe**

*Code source mal formé*

- **Erreur d'exécution**

*Exécution d'une opération interdite*

- **Erreur logique**

*Programme ne calcule pas ce qu'il faut*

- Le troisième type est le plus **difficile à déceler**

*Il faut pouvoir vérifier que le programme fait ce qu'il faut*

# Erreur de syntaxe

- Erreur détectée **lors de l'exécution** de l'instruction

*Python est en effet un langage interprété*

- Code source du programme contient des **fautes de syntaxe**

*Un peu comme l'orthographe en français*

```
1 score = 12
2 if score > 10
3     print('Vous avez réussi !')
```

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

# Erreur d'exécution

- Erreur produite **durant l'exécution** d'une opération interdite

*Division par zéro, indice en dehors d'une liste...*

```
1 data = [1, 2, 3]
2
3 i = 0
4 while i <= len(data):
5     print(data[i])
6     i += 1
```

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

# Erreur logique

- Le programme ne **calcule pas ce qu'il faut**

*Aucune erreur de syntaxe ou d'exécution ne se produit*

```
1 def perimeter(length, width):  
2     return length + width * 2  
3  
4 print(perimeter(2, 1))
```

4

# Documentation informelle

- La **documentation** d'une fonction décrit le résultat produit

*Permet à un utilisateur d'interpréter le résultat de l'appel*

- Description de **conditions** sur les paramètres

*Et de la valeur de retour si elles ne sont pas satisfaites*

```
1 # Renvoie le pourcentage d'une note étant donné :
2 # - "score" contient la note obtenue (flottant)
3 # - "total" est la note maximale atteignable (flottant)
4 #
5 # Si total <= 0, score < 0 ou score > total, alors renvoie None
6 def percentage(score, total):
7     if total > 0 and (0 <= score <= total):
8         return score / total * 100
9     return None
```

# Spécification (1)

- Documentation formelle de fonctions avec deux éléments
  - Préconditions sur les paramètres
  - Postconditions sur la valeur de retour
- Les préconditions sont à satisfaire avant l'appel

*Conditions sur les paramètres ou l'état global du programme*
- Les postconditions seront garanties après l'appel

*Pour autant que les préconditions étaient satisfaites avant l'appel*

# Spécification (2)

- Plus besoin de vérifier la valeur des paramètres

*On suppose que les préconditions sont satisfaites*

```
1 # Calcule le pourcentage correspondant à une note.  
2 #  
3 # Pre: 0 <= score <= total, la note obtenue  
4 #       total > 0, la note maximale atteignable  
5 # Post: La valeur renvoyée contient le pourcentage  
6 #       correspondant à la note obtenue.  
7 def percentage(score, total):  
8     return score / total * 100
```

# Docstring

- Insertion d'un **docstring** pour documenter la fonction

*Chaine de caractères sur plusieurs lignes reconnue par Python*

- Doit être placé **en premier** dans le corps de la fonction

*Automatiquement intégré dans la documentation générée*

```
1 def percentage(score, total):
2     """# Calcule le pourcentage correspondant à une note.
3     #
4     # Pre: 0 <= score <= total, la note obtenue
5     #       total > 0, la note maximale atteignable
6     # Post: La valeur renvoyée contient le pourcentage
7     #       correspondant à la note obtenue.
8     """
9     return score / total * 100
```

# Outil pydoc

```
x combefis@combefis-elementary: ~
+ x ...ombefis-elementary: ~

Help on module program:

NAME
    program

FUNCTIONS
    percentage(score, total)
        # Calcule le pourcentage correspondant à une note.
        #
        # Pre:  0 <= score <= total, la note obtenue
        #       total > 0, la note maximale atteignable
        # Post: La valeur renvoyée contient le pourcentage
        #       correspondant à la note obtenue.

FILE
    /home/combefis/program.py

(END)
```

# Instruction assert (1)

- Vérification de **conditions sensées être vraies** avec assert  
*On vérifie notamment les préconditions avec cette instruction*
- Un programme **doit fonctionner** si on supprime les assertions  
*Elles ne doivent pas faire partie du code fonctionnel*

```
1 def percentage(score, total):  
2     assert total > 0, 'total doit être strictement positif'  
3     assert 0 <= score, 'score doit être positif'  
4     assert score <= total, 'score doit être inférieur à total'  
5     return score / total * 100
```

# Instruction assert (2)

- Arrêt du programme en cas d'erreur d'assertion

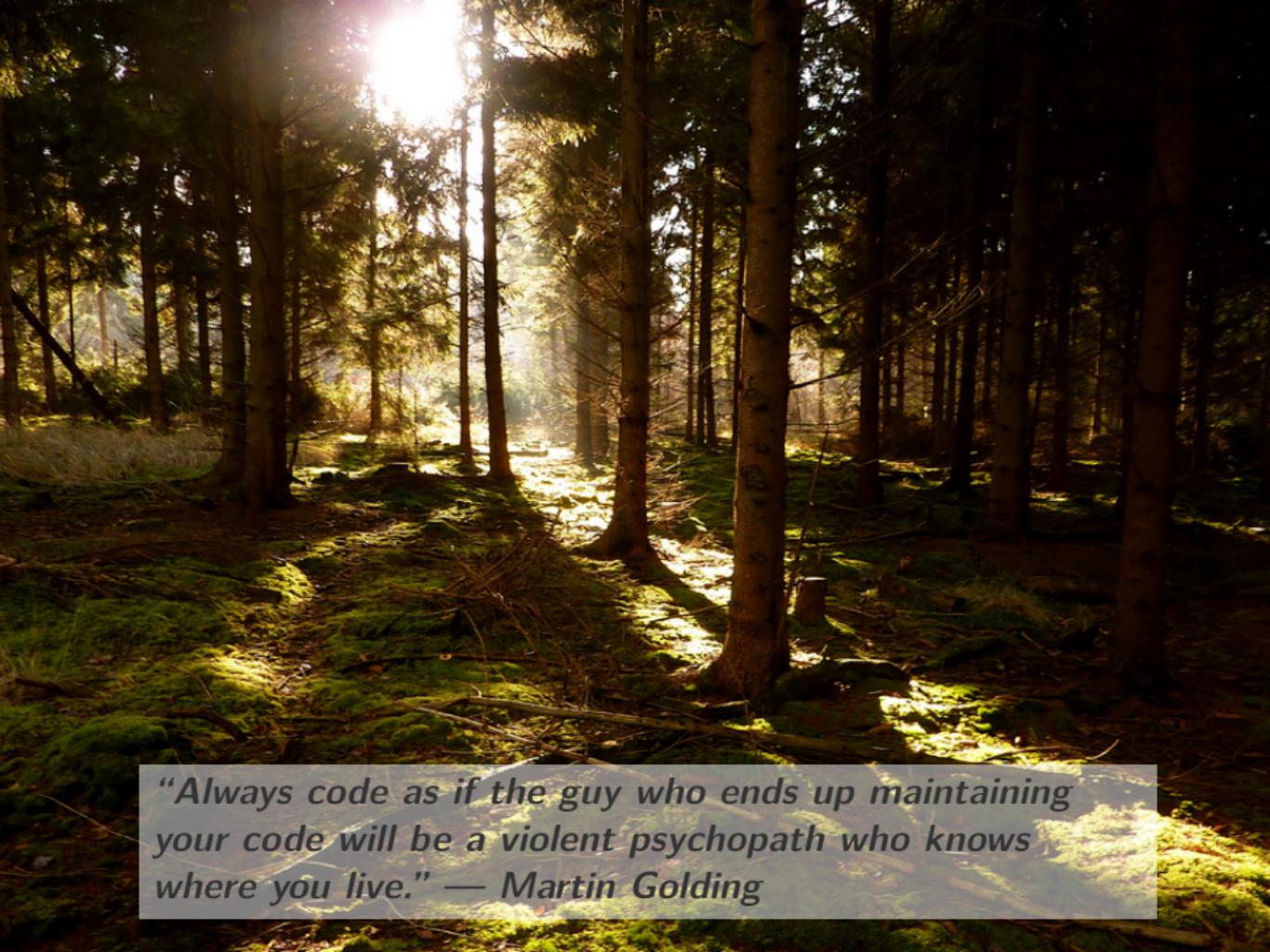
*Avec affichage d'un message d'information*

- Empêche des erreurs qui ne devraient pas se produire

*Le programme peut être modifié pour les éviter*

```
1 print(percentage(15, 20), '%')
2 print(percentage(22, 20), '%')
```

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

A photograph of a forest floor covered in green moss and fallen branches. Sunlight filters through the tall, thin trunks of coniferous trees, creating bright highlights and deep shadows. The overall atmosphere is mysterious and natural.

*“Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live.” — Martin Golding*

# Programmation défensive (1)

## ■ Programmation défensive

- Utilisation de l'instruction assert
- On suppose les préconditions remplies
- Peut être pratiquée

## ■ Gestion d'erreur

- Utilisation de l'instruction if-else
- On vérifie les conditions nécessaires sur les données
- Doit être pratiquée

# Programmation défensive (2)

- **Programmation défensive** au sein d'un module

*Se pratique sur du code dont vous avez le contrôle*

- **Gestion d'erreur** pour interface avec l'extérieur

*Vérification de toutes données hors contrôle*

- **Spécifications** dans les deux cas

*Le moins de préconditions possible vers l'extérieur*

# Sous-chaine (1)

- Tester si `s` est une **sous-chaine** de `string`

*À une position donnée `pos` dans la chaîne `string`*

- Fonction auxiliaire en programmation défensive

*Vérification des préconditions avec l'instruction `assert`*

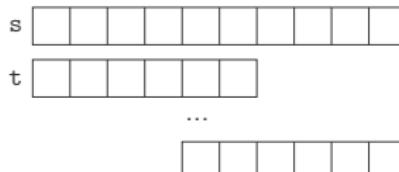
```
1 def _issubsequenceat(subseq, seq, pos):
2     # Vérification des préconditions
3     assert type(subseq) == str and type(seq) == str
4     assert type(pos) == int
5     assert len(subseq) <= len(seq)
6     assert 0 <= pos <= len(seq) - len(subseq)
7     # Teste la sous-séquence à la position 'pos'
8     for i in range(len(subseq)):
9         if seq[pos + i] != subseq[i]:
10             return False
11     return True
```

# Sous-chaine (2)

- La fonction principale vérifie les paramètres

*Renvoi de False en cas de souci*

```
1 def issubsequence(subseq, seq):
2     # Vérification des paramètres
3     if type(subseq) != str or type(seq) != str:
4         return False
5     if len(subseq) > len(seq):
6         return False
7     # Teste la sous-séquence à toutes les positions possibles
8     for i in range(0, len(seq) - len(subseq) + 1):
9         if _issubsequenceat(subseq, seq, i):
10             return True
11     return False
```



A photograph of a forest scene. In the foreground, a large, multi-trunked tree with twisted branches and green leaves stands prominently. Behind it, a dense stand of tall, thin trees with light-colored, textured bark reaches towards a bright sky. The ground is covered in green grass.

Mécanisme d'exception

# Instruction try-except (1)

- **Code risqué** placé dans un bloc try

*N'y placer que le code risqué et tout code qui en dépend*

- **Erreurs capturées** dans le bloc except

*Y placer le code à exécuter en cas de capture d'une erreur*

```
1 from datetime import *
2
3 birthyear = input('Année de naissance ? ')
4
5 try:
6     now = datetime.now()
7     age = now.year - int(birthyear)
8     print('Tu as', age, 'ans')
9 except:
10     print('Erreur')
```

# Instruction try-except (2)

- Si l'utilisateur entre un nombre entier, **pas d'erreurs**

```
Année de naissance ? 1984  
Tu as 31 ans
```

- Si l'utilisateur n'entre pas un nombre entier, **erreur capturée**

```
Année de naissance ? BLA  
Erreur
```

# Validité d'une donnée

- Demande d'une valeur à l'utilisateur **en boucle**

*Tant que la valeur demandée n'est pas du bon type*

```
1 from datetime import *
2
3 valid = False
4 while not valid:
5     birthyear = input('Année de naissance ? ')
6     try:
7         birthyear = int(birthyear)
8         valid = True
9     except:
10        print('Veuillez entrer un nombre entier')
11
12 now = datetime.now()
13 age = now.year - birthyear
14 print('Tu as', age, 'ans')
```

# Vérifier le type d'erreur (1)

- Plusieurs **types d'erreur** sont possibles

*Division par zéro, erreur de conversion...*

- Toutes les erreurs sont capturées par l'**instruction except**

*Possibilité de capturer les erreurs de manière spécifique*

```
1 try:  
2     a = int(input('a ? '))  
3     b = int(input('b ? '))  
4     print(a, '/', b, '=', a / b)  
5 except:  
6     print('Erreur')
```

# Vérifier le type d'erreur (2)

- Une **exception** est un objet qui représente une erreur

*L'objet est généralement de type Exception*

- Types spécifiques pour différencier les **types d'erreurs**

*ZeroDivisionError, ValueError...*

```
1 import sys
2
3 try:
4     a = int(input('a ? '))
5     b = int(input('b ? '))
6     print(a, '/', b, '=', a / b)
7 except Exception as e:
8     print(type(e))
9     print(e)
```

# Vérifier le type d'erreur (3)

- Division par zéro

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

- Erreur de conversion

```
a ? 2
b ? 0
<class 'ZeroDivisionError'>
division by zero
```

# Capturer une erreur spécifique (1)

- Gestionnaire d'erreurs différent pour chaque type d'erreur

*Il suffit de déclarer un bloc `except` par erreur à capturer*

- Attention à l'**ordre de capture** (de haut en bas)

*Il faut classer les erreurs de la plus à la moins spécificité*

```
1 import sys
2
3 try:
4     a = int(input('a ? '))
5     b = int(input('b ? '))
6     print(a, '/', b, '=', a / b)
7 except ValueError:
8     print('Erreur de conversion')
9 except ZeroDivisionError:
10    print('Division par zéro')
11 except:
12    print('Autre erreur')
```

## Capturer une erreur spécifique (2)

```
1 import sys
2
3 try:
4     a = int(input('a ? '))
5     b = int(input('b ? '))
6     print(a, '/', b, '=', a / b)
7 except Exception:
8     print('Erreur')
9 except ValueError:
10    print('Erreur de conversion')
11 except ZeroDivisionError:
12    print('Division par zéro')
```

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

# Information sur une erreur

- L'objet de l'exception peut contenir de l'information

*On peut accéder à des propriétés ou à des méthodes*

```
1 try:  
2     import mymod  
3 except SyntaxError as e:  
4     print(e)  
5     print('File:', e.filename)  
6     print('Line:', e.lineno)  
7     print('Text:', e.text)
```

```
can't assign to literal (mymod.py, line 1)  
File: /Users/combefis/Desktop/mymod.py  
Line: 1  
Text: 2 = x
```

# Gestionnaire d'erreurs partagé

- Même gestionnaire d'erreurs pour différents types

*Tuple d'exception fourni à l'instruction except*

```
1 try:  
2     a = int(input('a ? '))  
3     b = int(input('b ? '))  
4     print(a / b)  
5 except (ZeroDivisionError, ValueError):  
6     print('Erreur de calcul')  
7 except:  
8     print('Erreur')
```

```
a ? 1  
b ? 0  
Erreur de calcul
```

# Propagation d'erreur (1)

- Une **erreur non capturée** remonte les appels de fonction

*Jusqu'à être attrapée ou remonté jusqu'au bout*

- La **trace d'erreur** montre le trajet pris par l'exception

*En la lisant à l'envers, on peut suivre la propagation*

# Propagation d'erreur (2)

## ■ Passage de fun à compute au programme principal

```
1 def fun():
2     print(1 / 0)
3
4 def compute():
5     fun()
6
7 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
```

# Propagation d'erreur (3)

- Exception **interceptée** dans la fonction `compute`

```
1 def fun():
2     print(1 / 0)
3
4 def compute():
5     try:
6         fun()
7     except:
8         print('Erreur.')
9
10 compute()
```

```
Erreur.
```

# Bloc finally (1)

- Le bloc finally s'exécute **dans tous les cas**

*Après le bloc try ou l'except en cas d'erreur*

- Notamment utilisé pour faire du **nettoyage**

*Par exemple pour libérer des ressources qui ont été allouées*

# Bloc finally (2)

- Bloc finally exécuté à tous les coups avant la fin du calcul

```
1 print('Début du calcul.')
2 try:
3     a = int(input('a ? '))
4     b = int(input('b ? '))
5     print('Résultat :', a / b)
6 except:
7     print('Erreur.')
8 finally:
9     print('Nettoyage de la mémoire.')
10 print('Fin du calcul.)
```

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

# Générer une erreur

- L'instruction `raise` permet de générer une erreur

*Création d'un objet du type de l'exception*

```
1 def fact(n):
2     if n < 0:
3         raise ArithmeticError()
4     if n == 0:
5         return 1
6     return n * fact(n - 1)
7
8 try:
9     n = int(input('Entrez un nombre : '))
10    print(fact(n))
11 except ArithmeticError:
12     print('Veuillez entrer un nombre positif.')
13 except:
14     print('Veuillez entrer un nombre.')
```

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

# Définir une erreur (1)

- Définition d'une erreur en définissant une nouvelle classe

*La classe est créée à partir de la classe Exception*

- L'instruction `pass` ne fait rien

```
1 from math import sqrt
2
3 class NoRootException(Exception):
4     pass
5
6 def trinomialroots(a, b, c):
7     delta = b ** 2 - 4 * a * c
8     if delta < 0:
9         raise NoRootException()
10    if delta == 0:
11        return -b / (2 * a)
12    x1 = (-b + sqrt(delta)) / (2 * a)
13    x2 = (-b - sqrt(delta)) / (2 * a)
14    return (x1, x2)
```

## Définir une erreur (2)

- Capture de la nouvelle erreur avec l'instruction `except`

*Le nouveau type d'erreur est maintenant connu par Python*

```
1 try:  
2     print(trinomialroots(1, 0, 2))  
3 except NoRootException:  
4     print('Pas de racine réelle.')  
5 except:  
6     print('Erreur')
```

Pas de racine réelle.

# Exception paramétrée

- Stockage d'un **paramètre** dans l'exception

```
1 class NoRootException(Exception):
2     def __init__(self, delta):
3         self.__delta = delta
4
5     @property
6     def delta(self):
7         return self.__delta
8
9     def trinomialroots(a, b, c):
10        # ...
11        if delta < 0:
12            raise NoRootException(delta)
13        # ...
14    except NoRootException as e:
15        print('Pas de racine réelle (delta = ', e.delta, ')', sep='')
```

```
Pas de racine réelle (delta = -8)
```

# Quand utiliser les erreurs ?

- Toujours vérifier les données provenant de l'**extérieur**

*Lecture avec `input`, lecture d'un fichier...*

- Lors d'un **appel à une fonction** d'un module

*Lire la documentation de la fonction, pour les erreurs potentielles*

- Quand on définit une **librairie**

*Pour les fonctions publiques offertes à l'extérieur*

# Crédits

- <https://www.flickr.com/photos/mstibbetts/1401175133>
- <http://www.flickr.com/photos/tetezinharamana/7152072635/>
- <https://www.flickr.com/photos/mbiskoping/510673513>