Chapitre 3

Gestion des erreurs

Lorsqu'on conçoit un programme où que l'on écrit une fonction, il n'est pas rare de commettre des **erreurs**.

Ces erreurs sont souvent de natures différentes : erreur de syntaxe (oubli du `:` après la déclaration d'une fonction ou après un `while`), erreur de nommage (on utilise une variable qui n'existe pas, car on a ajouté ou oublié une majscule ou une lettre dans le nom de cette variable), erreur d'indice (on essaie d'accéder à un élément d'une liste qui n'existe pas), etc ...

La liste de ces erreurs est longue, et le but de ce cours n'est pas d'en faire une énumération exhaustive, mais plutôt d'apprendre à diagnostiquer et à corriger les plus courantes.

3.1 Diagnostiquer et corriger une erreur

3.1.1 Erreurs de syntaxe

Le premier type d'erreur que l'on rencontre, et celui que l'on commet le plus souvent, surtout au début, est l'erreur de syntaxe. Une erreur de syntaxe est une erreur d'analyse du code, comme par exemple :

```
1 if x = 1:
2     print("La valeur de x est 1")
3 else
4     print("Erreur")
```

Le programme ci-dessus comporte deux erreurs de syntaxe :

- un test d'égalité se fait avec le symbole `==`, et non `=` qui est le symbole d'affectation
- il manque `:` après le `else`

▷ ***Il y a une autre erreur : la variable `x` n'est pas définie !***

Effectivement, c'est une autre erreur, mais ce n'est pas une erreur de syntaxe. Les erreurs de syntaxe sont en quelque sorte les fautes d'orthographe du programme, et ne nécessitent pas de comprendre le programme pour pouvoir les déceler.

Lorsque vous exécutez un programme Python, il y a d'abord une vérification de la syntaxe, puis le programme est exécuté ligne par ligne seulement s'il n'y a pas d'erreurs de syntaxe. Par erreur de syntaxe, on entend :

- une mauvaise utilisation du signe `=`
- une erreur de mot-clé (mauvaise utilisation, mauvaise orthographe, oubli)
- un oubli de parenthèses, d'accolades, de guillemets, de crochets
- une erreur d'indentation (le fameux décalage vers la droite)
- une erreur dans la définition ou l'utilisation d'une fonction

Toutes ces erreurs ont pour conséquence l'apparition d'un désagréable message d'erreur, souvent écrit en rouge :

```
>>> print("erreur)
  File "<stdin>", line 1
    print("erreur)
                  ^
SyntaxError: EOL while scanning string literal
```

On voit dans l'exemple ci-dessus l'erreur renvoyée par la commande `print("erreur)` : il s'agit d'une erreur de syntaxe (d'où le `SyntaxError`) consécutive à l'oubli d'un guillemet. Ici, l'erreur est localisée par le chevron `^`, situé juste en dessous de l'endroit où devrait se refermer le guillemet.

Terminons avec l'exemple suivant :

```
1  def f(x):
2      if x >= 0:
3          print("Positif")
4      elif:
5          print("Négatif")
```

L'exécution de ce code définit simplement la fonction `f`, mais rien n'est affiché, puisque la fonction `f` n'est pas appelée. Cependant, l'erreur de syntaxe s'y trouvant produit une erreur (la voyez-vous ?), et le programme n'est pas exécuté !

Question 01 Relever et corriger les **erreurs de syntaxe** dans les programmes suivants :

```
1  # Programme A
2
3  L = [2;7;"a"]
4  s = 0
5
6  for n in L:
7      s = s + n
8
9    print(s)
```

```
1  # Programme B
2
3  L = [3,0,-2]
4  i = 0
5
6  while i < len(L):
7     print(L(i) / i)
8     i = i + 1
```

3.1.2 Autres erreurs

Une fois la barrière de la syntaxe franchie, le programme Python est exécuté, et bien d'autres erreurs peuvent apparaître.

Erreur de nommage `NameError`

Considérons les instructions suivantes :

```
>>> nombreentier = 3
>>> carre = nombrentier ** 2
```

Aucune erreur de syntaxe ici, mais une **erreur de nommage** (`NameError`).

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'nombrentier' is not defined
```

En effet, il y a une erreur d'orthographe dans le nom de la variable `nombrentier` (il manque un e). Cela re-démontre l'intérêt de bien nommer ses variables, et en particulier de respecter certaines règles de nommage décrites dans la PEP8 [1].

C'est encore cette erreur qui survient lorsqu'on oublie de déclarer une variable avant de l'utiliser, comme un compteur ou une variable de sommation.

1. https://www.python.org/dev/peps/pep-0008/#naming-conventions

Erreur de types `TypeError`

Un exemple d'instruction faisant intervenir une erreur de type est l'addition de deux valeurs a priori non sommables, comme un entier et un caractère :

```
>>> 3 + "a"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Python explique ici clairement qu'il ne sait pas utiliser l'opération `+` avec une variable de type `int` (entier) et une variable de type `str` (chaîne de caractères), cette opération n'ayant a priori pas de sens.

Cette même erreur survient lorsque l'on tente d'accéder à l'élément d'une liste dont l'indice n'est pas un entier :

```
>>> L = [1,2,3]
>>> L[1.5]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: list indices must be integers or slices, not float
```

Enfin, on peut voir apparaître un `TypeError` lorsqu'on appelle une fonction avec un nombre incorrect de paramètres.

```
>>> from math import sqrt
>>> sqrt(3, 4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: sqrt() takes exactly one argument (2 given)
```

La virgule signifie que l'on appelle la fonction `sqrt` avec deux paramètres, 3 et 4, ce que Python refuse en nous signalant que la fonction `sqrt` prend un seul argument, alors que nous lui en avons donné deux.

Question 02 Le programme A de la question 1 fait apparaître une erreur de type. Pourquoi ?

La division par zéro `ZeroDivisionError`

La hantise du mathématicien ! Python interdit la division par zéro et affiche une erreur de type `ZeroDivisionError` lorsque cela se produit.

```
>>> 1 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

Bien sûr, on n'écrit jamais volontairement une division par 0, mais c'est une situation sur laquelle on peut tomber au cours de l'exécution d'un programme.
Prenez par exemple le programme suivant :

```
u = -2

for i in range(10):
    u = (u + 5) / (u + 1)

print(u)
```

Question 03 Vérifier qu'une erreur du type `ZeroDivisionError` apparaît dans ce programme.

Les erreurs d'indices `IndexError` et de clés `KeyError`

Ce genre d'erreur survient lorsque l'on essaie d'accéder à l'élément d'une liste dont l'indice n'existe pas, ou à l'élément d'un dictionnaire dont la clé n'est pas définie.

```
>>> L = [1,2,3]
>>> L[3]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Indice trop grand : `IndexError`

```
>>> panier = {"pommes": 2, "poires": 3}
>>> panier['fraises']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'fraises'
```

Clé non définie : `KeyError`

L' `IndexError` est une erreur très commune, notamment lorsqu'on manipule des indices de listes dans des boucles, comme dans l'exemple suivant :

```
L1 = [1,2,3]
L2 = [4,5,6,7]

for i in range(L2):
    for j in range(L1):
        print(L1[i] + L2[j])
```

Question 04 Identifier l'erreur de type `IndexError` dans le programme ci-dessus.

3.1.3 Identifier et localiser une erreur

En tant que programmeur, il est indispensable que vous sachiez localiser les erreurs que vous commettrez dans vos programmes. Car oui, vous continuerez à commettre des erreurs, parfois de syntaxe (on oublie une virgule, une parenthèse...), mais souvent cette erreur sera imprévue.

Python ne vous laisse pas seul(e), bien au contraire : il vous aide à identifier le type et la source de vos erreurs. Imaginez par exemple qu'un programme s'exécute quand soudaint le message suivant apparaît :

```
Traceback (most recent call last):
  File "module_perso.py", line 8, in <module>
    print(somme(L))
  File "module_perso.py", line 3, in somme
    s = s + L[i]
UnboundLocalError: local variable 's' referenced before assignment
```

Il faut lire ces informations **de bas en haut** :

- `UnboundLocalError` : c'est le type d'erreur rencontrée. Ici, il s'agit d'une variable locale (utilisée à l'intérieur d'une fonction) non définie ou mal définie
- `local variable 's' referenced before assignment` : la variable incriminée est la variable `s`, variable dont on cherche à connaître la valeur alors qu'elle n'a pas été préalablement définie
- `s = s + L[i]` : l'instruction ayant provoqué l'erreur
- `File "module_perso.py", line 3, in somme` : la localisation de l'instruction précédente :
  - fichier *module_perso.py*
  - ligne 3
  - dans une fonction nommée `somme`
- `print(somme(L))` : l'instruction appelant la fonction `somme` dans laquelle s'est produite l'erreur
- `File "module_perso.py", line 8, in <module>` : la localisation de cette instruction
  - fichier *module_perso.py*
  - ligne 8
  - dans le programme principal (noté `<module>`

Grâce à la **pile d'appel**, Python connaît le déroulé exact de la situation menant à l'erreur, et nous l'affiche.

Après lecture de ces informations, on peut conclure que l'erreur est provoquée par l'utilisation d'une variable `s` non déclarée dans la fonction `somme` située ligne 3, appelée à la ligne 8 par l'instruction `print(somme(L))` dans le fichier `module_perso.py`.

☞ *C'est cette interprétation qu'il faut savoir faire lorsqu'une erreur survient dans un programme!*

Voici le programme qui a amené cette erreur :

```
1  def somme(L):
2      for i in L:
3          s = s + L[i]
4
5      return s
6
7  L = [1,6,2]
8  print(somme(L))
```

Question 05 Corriger le programme en conséquence.

Question 06 Exécuter alors le programme. Expliquer l'erreur qui se produit, l'expliquer et la corriger.

3.1.4 Terminaison et Correction d'un programme

Il y a certaines erreurs que Python ne voit pas! Considérons la fonction `produit` suivante, dont le but est de calculer le produit de deux nombres entiers a et b :

```
1  def produit(a,b):
2      i = 0
3      p = 1
4
5      while i <= b:
6          p += a
7
8      return p
9
10 print(produit(5,3))
```

Question 07 Justifier que ce programme ne se termine jamais.

Ce programme est juste au niveau de la syntaxe et de toutes les opérations qui s'y déroulent. Il n'y a donc pas de raison de produire une erreur. Python est en effet incapable de distinguer une boucle infinie d'un programme un peu trop long.

☞ *Vérifier que le programme se termine, c'est vérifier la* ***terminaison*** *de ce programme*

Question 08 Une fois l'erreur précédente corrigée, le programme affiche-t-il le résultat attendu? Justifier.

Il arrive qu'un programme ne donne pas toujours le résultat attendu. C'est une erreur, mais Python ne peut pas le savoir! C'est à vous de vérifier que l'algorithme que vous programmez affiche le résultat que vous souhaitez obtenir, et ce quel que soit le cas de figure envisagé. On dit alors que l'algorithme (ou le programme) est correct.

☞ *Vérifier que le programme affiche toujours la valeur attendue, c'est vérifier la* ***correction*** *de ce programme*

Question 09 Corriger le programme précédent afin qu'il soit correct.

3.2 Erreurs et Exceptions

Une **exception** est un mécanisme d'interruption du programme utilisé pour signaler que quelque chose d'anormal est en train de se produire. On les rencontre dans de nombreux cas, mais souvent, c'est dans le cadre d'erreurs.

À chaque fois que Python affiche une erreur, il se produit une exception (on dit qu'une exception est **levée**).

Ainsi, les `IndexError` et autres `ZeroDivisionError` sont des exceptions.

Vous trouverez la hiérarchie des exceptions (et donc des erreurs) Python sur la page suivante. Il n'est bien entendu pas utile de toutes les connaître!

`https://docs.python.org/3/library/exceptions.html#exception-hierarchy`

☞ *On voit que une exception levée pour mauvaise indentation (* `IndentationError` *) est une exception qui « découle » d'une exception de type* `SyntaxError` *. Cette notion de « découler » s'appelle l'**héritage** en programmation orientée objet*

3.2.1 Lever une exception

Le mot-clé `raise`

Normalement, c'est Python qui s'occupe de lever des exceptions lorsqu'une survient une erreur. Mais on peut également lever une exception soi-même avec le mot clé `raise` :

```
raise NomDeLException(message)
```

Imaginons la fonction suivante :

```
1  def donner_code(code):
2      if code == 1234:
3          print("Vous pouvez entrer")
4      else:
5          raise ValueError("Mauvais mot de passe")
```

```
>>> donner_code(1234)
Vous pouvez entrer
```

```
>>> donner_code(4321)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "fichier.py", line 5, in donner_code
    raise ValueError("Mauvais mot de passe")
ValueError: Mauvais mot de passe
```

On décide de lever une exception de type `ValueError` lorsque le mot de passe renseigné n'est pas le bon. En général, on ne lève pas une exception pour si peu. Le principal inconvénient est que le programme s'interrompt lors de la levée d'une exception.

Le mot-clé `assert`

L'utilisation de `assert` permet de lever une exception de type `AssertionError` lorsqu'une certaine condition n'est pas remplie, accompagnée d'un message d'erreur. Par exemple, avant de calculer la racine carrée d'un nombre, on peut s'assurer que ce dernier est positif à l'aide d'une assertion :

```
1  from math import sqrt
2
3  def racineCarree(nombre):
4      assert nombre >= 0, "Le nombre donné doit être positif ou nul"
5      return sqrt(nombre)
```

```
>>> racineCarree(4)
2.0
```

```
>>> racineCarree(-2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "fichier.py", line 4, in racineCarree
    assert nombre >= 0, "Le nombre donné doit
    ↪  être positif ou nul"
AssertionError: Le nombre donné doit être
↪  positif ou nul
```

Les assertions sont là pour empêcher des erreurs qui ne devraient pas se produire. Si une telle erreur survient, c'est que le programme doit être modifié pour qu'elle n'arrive plus.

3.2.2 Attraper une exception

Les exceptions ne sont pas un simple mécanisme de debuggage. Elles servent d'abord et avant tout à gérer les cas exceptionnels, et on peut donc les détecter, et réagir quand elles surviennent, à l'aide de l'instruction `try/except` .

Considérons la fonction suivante :

```
1  def inverse(liste):
2      for x in liste:
3          print(1 / x)
```

Le rôle de cette fonction est d'afficher les inverses des éléments d'une liste. Par exemple :

```
>>> inverse([1,2,3])
1.0
0.5
0.3333333333333333
```

Mais dans certains cas, une exception sera levée :

```
>>> inverse([1,0,"a"])
1.0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in inverse
ZeroDivisionError: division by zero
```

On voit dans ce cas que l'inverse de 1 est calculé, mais le calcul de l'inverse de 0 lève une exception de type `ZeroDivisionError`, et le programme s'interrompt.

On peut modifier le programme précédent afin d'empêcher une exception de type `ZeroDivisionError` d'interrompre notre programme en l'**attrapant** :

```
1 def inverse(liste):
2     for x in liste:
3         try: # On essaie de calculer l'inverse
4             print(1 / x)
5         except ZeroDivisionError: # En cas de division par 0, on le signale
6             print("Division par 0")
```

```
>>> inverse([1,0,"a"])
1.0
Division par 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in <module>
    print(1 / x)
TypeError: unsupported operand type(s) for /: 'int' and 'str'
```

On voit que le calcul de l'exception levée par le calcul de l'inverse de 0 a été capturée, et le programme a poursuivi son exécution, mais le message `Division par 0` a été affiché.
Cependant, une seconde exception `TypeError` a été levée lorsque le programme a cherché à calculer l'inverse de `"a"`. On peut alors la capturer elle aussi et traiter le cas particulier :

```
1 def inverse(liste):
2     for x in liste:
3         try: # On essaie de calculer l'inverse
4             print(1 / x)
5         except ZeroDivisionError: # En cas de division par 0, on le signale
6             print("Division par 0")
7         except TypeError: # En cas de type de donné non numérique, impossible
8             print("Mauvais type de variable utilisé)
```

```
>>> inverse([1,0,"a"])
1.0
Division par 0
Mauvais type de variable utilisé
```

Notre programme se termine sans interruption!

▷ ***Et si un autre type d'exception est levé, le programme va encore être interrompu!***

On peut capturer toutes les exceptions en ne précisant pas un type après le mot-clé `except` :

```
1  def inverse(liste):
2      for x in liste:
3          try: # On essaie de calculer l'inverse
4              print(1 / x)
5          except ZeroDivisionError: # En cas de division par 0, on le signale
6              print("Division par 0")
7          except TypeError: # En cas de type de donné non numérique, impossible
8              print("Mauvais type de variable utilisé)
9          except: # Si une autre exception survient...
10             print("Erreur inconnue. Contactez le programmeur au plus vite !")
```

Ainsi, si une erreur inconnue survient, elle ne fera pas planter notre programme.
Attention cependant : ce n'est pas une bonne habitude à prendre. Il est préférable d'anticiper tous les cas de figure pouvant lever des exceptions.

Question 10 Proposer une fonction `racineCarree(n)` retournant la racine carrée de `n` si `n` est un entier positif, sinon afficher une erreur, en capturant des exceptions.

☞ *On utilisera la fonction `sqrt` du module `math`, et on fera des tests pour voir les différentes exceptions possibles.*

Question 11 Proposer une fonction `valeurDico(dictionnaire, cle)` qui retourne la valeur correspondante à la clé `cle` dans `dictionnaire`. Si la clé n'existe pas, on affichera une erreur, en capturant des exceptions.