	**Langages et programmation**	**NSI T^ale^**
	Gestion des bugs	Cours

*

1. Pour bien commencer

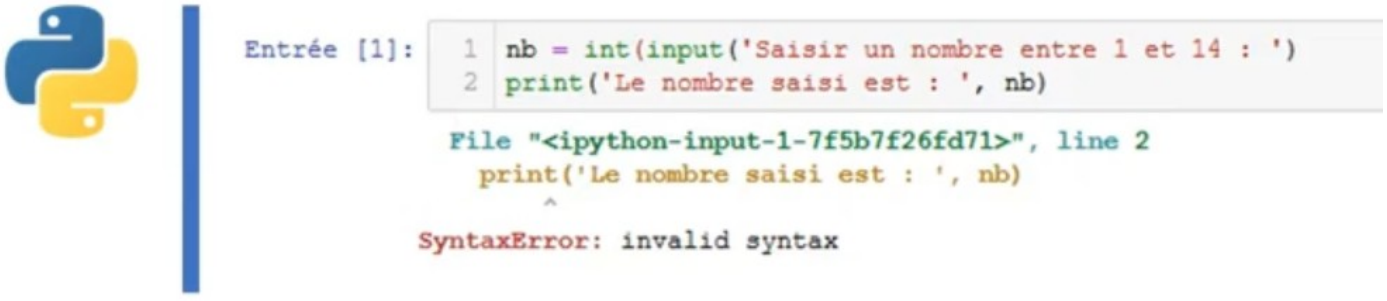
Nous allons étudier les bons ingrédients d'un script Python et plus généralement d'un bon script informatique. Dans la pratique de la programmation, savoir répondre aux causes typiques de bugs, savoir tester de manière fiable ses scripts, est capitale. Nous allons donc explorer quelques pistes pour y parvenir.

2. Les grands principes

Dans les ingrédients d'un bon programme, il y a, entre autres choses, les problèmes liés au typage, les effets de bords non désirés, les débordements dans les tableaux, les instructions conditionnelles non exhaustives, le choix des inégalités, les comparaisons et calculs entre flottants, les mauvais nommages de variables.
Nous allons donc prolonger le travail entrepris en classe de 1ère sur l'utilisation de la spécification, des assertions et de la documentation des programmes dans la construction des jeux de test.
Un bon programme est un programme qui sait anticiper les erreurs.
Voici les grands principes permettant d'éviter au mieux les principales sources de bugs.

2.1. Corriger les bugs évidents

Ce sont ceux qui conduiront immanquablement à des erreurs de syntaxe invalide (« `invalid syntax` »)



Dans n'importe quelle interface Python et en particulier votre IDE préféré, vous pouvez trouver des outils vous permettant d'éviter ces bugs.

Aucune « `invalid syntax` » ne doit être présente dans un code un minimum testé ! Utilisez votre IDE préféré pour vous guider.

2.2. Prévoir toutes les saisies possibles des utilisateurs (imaginez le pire)

Vous demandez la saisie d'un nombre : vérifiez que l'utilisateur n'a pas saisi une chaîne de caractères.

```
Entrée [1]:
1 nb = int(input('Saisir un nombre entre 1 et 14 : '))
2 print('Le nombre saisi est : ', nb)

Saisir un nombre entre 1 et 14 : quatre

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-1-8f7404d41e96> in <module>
----> 1 nb = int(input('Saisir un nombre entre 1 et 14 : '))
      2 print('Le nombre saisi est : ', nb)

ValueError: invalid literal for int() with base 10: 'quatre'
```

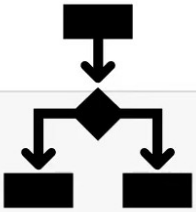
Sinon avec python et le typage des variables, vous risquez d'avoir quelques erreurs. Tentez donc de poser un maximum de garde-fous.

2.3. Effectuer des séries de tests

Vous devez réellement tester toutes les situations possibles correspondant à l'arborescence de votre algorithme :

- Changer les entrées, examiner les sorties
- Chaque cas distinct doit-être examiné au moins une fois : pensez à toutes les structures conditionnelles
- Les cas limites doivent être essayés :
  - le nombre `0`
  - la chaîne vide
  - un seul caractère
  - différences majuscules/minuscules

Par exemple si vous avez une structure condition dans votre script, il va falloir tester tous les cas possibles et tous les passages dans les structures conditionnelles :

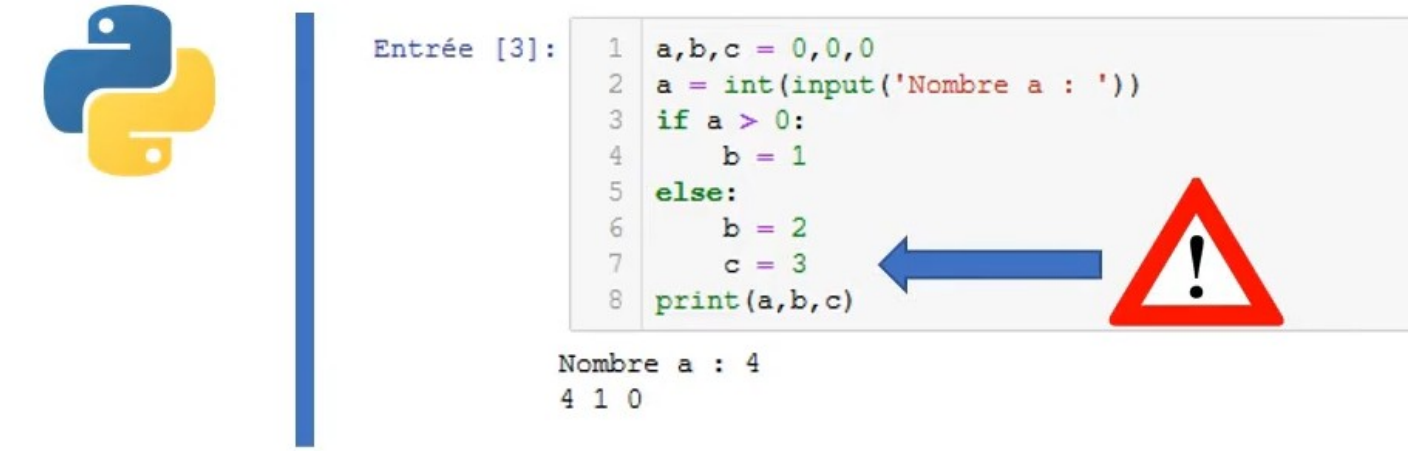


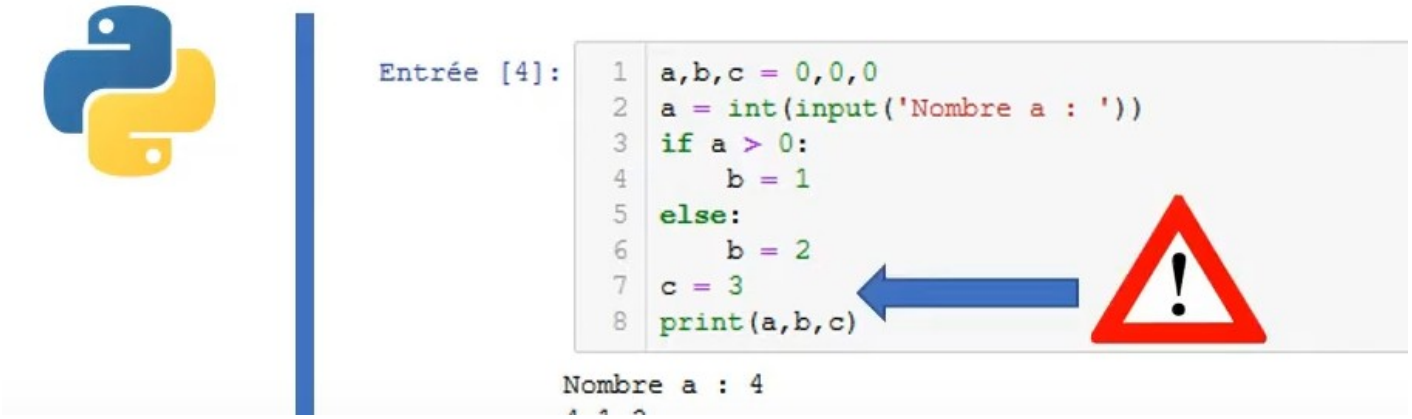
```
Entrée [1]:
1  from math import sqrt
2  a = float(input('Coefficient a : '))
3  b = float(input('Coefficient b : '))
4  c = float(input('Coefficient c : '))
5  delta = b*b-4*a*c
6  if delta < 0:
7      print('Pas de solution réelle')
8  elif delta == 0:
9      print('Une solution : ', (-b/(2*a)))
10 else :
11     print('Deux solutions : {} et {}'.format((-b-sqrt(delta))/(2*a),(-b+sqrt(delta))/(2*a)))

Coefficient a : 1
Coefficient b : 2
Coefficient c : 3
Pas de solution réelle
```

Ici par exemple vous avez un `if` avec un `delta` à tester négatif : il va falloir tester le cas où le `delta` est réellement négatif. Vous avez un `elif` avec un `delta` égal à `0` : il va falloir tester pour rentrer dans cette partie du `elif`. Et enfin vérifier que le `else` fonctionne parfaitement. Pour une sortie conditionnelle avec 3 sorties possibles, vous devez tester les trois !

2.4. Faire attention à l'indentation

Cet élément est spécifique à Python qui tient compte en partie de l'indentation. D'autres langages qui utiliserons en fait des sous-structures, entre accolades par exemple, n'auront pas cette indentation.





Mais ici sur les 2 codes présents, vous voyez bien la différence à la ligne 7 où le `c` est indenté dans le premier cas et pas dans le second. Ce script ne produit aucune erreur. En revanche il ne donne pas le même résultat. Si vous rentrez dans la boucle `else` dans le premier cas, le `c` est exécuté, mais si vous n'y rentrez pas le `c` ne change pas de valeur. Alors que dans le second cas, quelque soit le résultat de la structure conditionnelle, le `c` changera de valeur.
Donc, soyez très prudents, et en particulier dans la détection de bugs, tenez compte des indentations.

2.5. Éviter les effets de bords

On parle d'effet de bords quand une fonction modifie la valeur d'une variable dite globale.
Le paradigme de programmation qui se propose d'éviter au maximum les effets de bords est la programmation fonctionnelle (cf : paradigmes de programmation).

Entrée [5]:

```
1  def fct():
2      global i
3      i = i + 1
4
5  i = 3
6  fct()
7  print (i)
```

Chaque fonction possède ses variables et ces variables ne sont pas visibles du programme principal. Or dans Python il est possible de contourner cet effet grâce au mot clé **`global`**.
Les effets de bords sont responsables de nombreuses erreurs et rendent aussi difficile la lecture des programmes.
Évitez autant que possible d'utiliser des variables globales !

3. Gérer les erreurs de saisie

3.1. Définir une exception

Python permet, comme de nombreux langages, de définir ce que l'on appelle des exceptions.
Gérer une exception permet d'intercepter une erreur afin d'éviter un arrêt du programme. C'est un élément capital de la gestion des erreurs.

Une exception sépare :

- d'un côté la séquence d'instruction à exécuter lorsque tout se passe bien
- d'un autre côté une ou plusieurs séquences d'instruction à exécuter en cas d'erreur.

Le mécanisme s'effectue en 2 phases :

- la levée d'exception
- le traitement approprié de celle-ci

Quand Python rencontre une erreur dans votre code, il lève une exception… et vous avez déjà vu des exceptions levées par Python :

Entrée [1]:

```
1  nb = int(input('Saisir un nombre entre 1 et 14 : '))
2  print('Le nombre saisi est : ', nb)
```

```
Saisir un nombre entre 1 et 14 : quatre

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-1-8f7404d41e96> in <module>
----> 1 nb = int(input('Saisir un nombre entre 1 et 14 : '))
      2 print('Le nombre saisi est : ', nb)

ValueError: invalid literal for int() with base 10: 'quatre'
```

Sur cet exemple où l'on demande de saisir un nombre, on rentre une chaîne de caractères et la conversion en entier ne peut pas s'effectuer. Vous avez donc une « **`ValueError`** » et vous avez donc là l'observation d'une exception levée par Python.

Python lève des exceptions quand il trouve une erreur, soit dans le code (erreur de syntaxe), soit dans l'opération qu'il effectue… mais les indications données sont plus ou moins limpides.

3.2. Syntaxe d'une exception

```
try:
    # Bloc à exécuter (bloc principal)
except:
    # Bloc qui sera exécuté en cas d'erreur dans le bloc principal
```

Il y a deux étapes :

- Le bloc `try` : le bloc à exécuter, le bloc principal du programme qui doit s'exécuter normalement.
- Le bloc `except` : si une exception est levée, Python renvoie automatiquement à ce bloc. Ce bloc ne sera exécuté qu'en cas d'erreur dans le bloc principal.

Entrée [11]:

```
1 try:
2     nb = int(input('Saisir un nombre entre 1 et 14 : '))
3     print('Le nombre saisi est : ', nb)
4 except:
5     print('Saisie invalide')
```

```
Saisir un nombre entre 1 et 14 : quatre
Saisie invalide
```

Sur l'exemple, si la saisie et la conversion se passent bien à la ligne 2, le print s'exécute. Sinon, on passera dans `l'except` avec un message d'erreur « `Saisie invalide` ».

Comparaison avec ou sans `try`/`except` pour une division par zéro :
Dans le second cas on affiche un message d'erreur mais le code peut continuer.

Sans try/except :

Entrée [8]:

```
1 a = 1
2 b = 0
3 print('a/b = ', a/b)
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-8-7ba251f6cd28> in <module>
      1 a = 1
      2 b = 0
----> 3 print('a/b = ', a/b)

ZeroDivisionError: division by zero
```

Avec try/except

Entrée [9]:

```
1 a = 1
2 b = 0
3 try:
4     # Effectuer un calcul
5     print('a/b = ', a/b)
6 except:
7     print('Erreur lors du calcul')
```

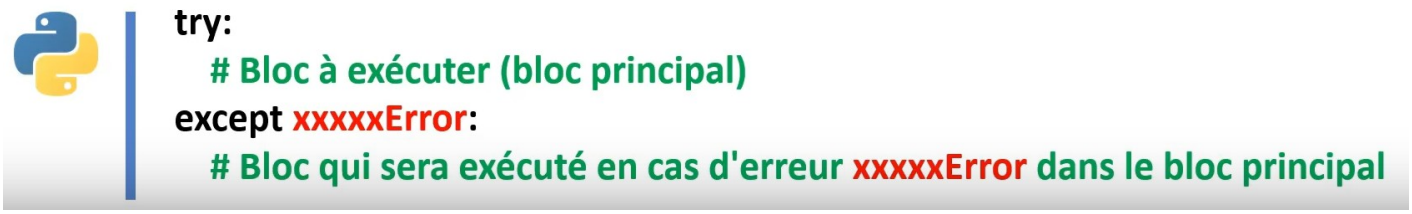
```
Erreur lors du calcul
```

3.3. Repérer les différentes erreurs possibles

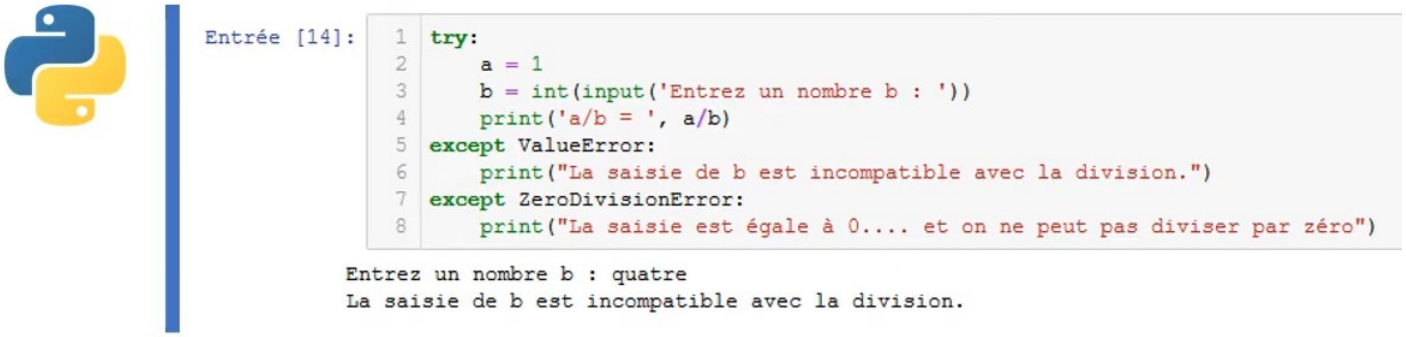
Suivant le type d'erreur, l'exception est différente :

- `NameError` : l'une des variables n'existe pas
- `TypeError` : l'une des variables ne peut diviser ou être divisée
- `ZeroDivisionError` : très classique… mais problématique !

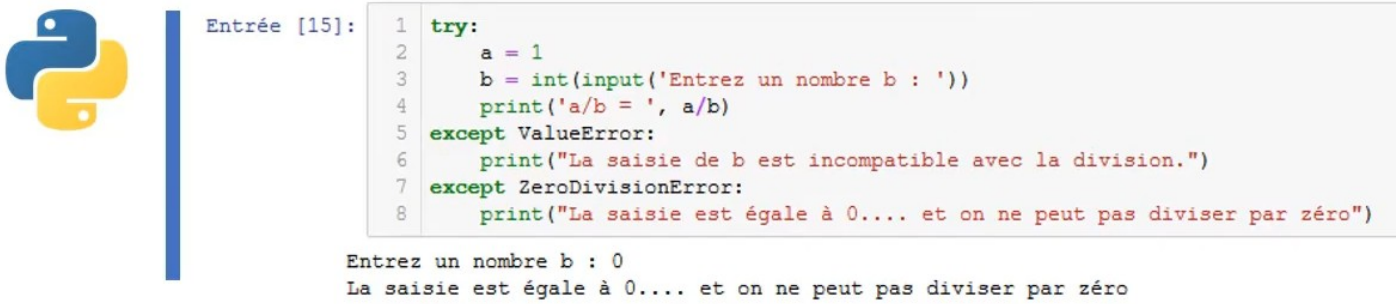
Vous pouvez traiter différemment le type d'erreur en modifiant après le mot `except` le nom du type d'erreur à examiner.



Vous pouvez donc lister tous les types d'erreur les uns derrière les autres avec plusieurs `except`.
Exemple pour la saisie d'un caractère à la place d'un entier :



On a donc une `ValueError` et on rentre dans le premier `except`.
Le même code en saisissant `0` :



Cette fois-ci on a une `ZeroDivisionError` et on rentre dans le second `except`.

On peut ainsi adapter le message d'erreur à la saisie de l'utilisateur.

3.4. Pour être encore plus précis

Nommer l'erreur :

Vous pouvez récupérer l'erreur en lui donnant un nom avec `as` puis préciser le nom de l'exception.

Entrée [19]:

```
try:
    a = 1
    b = int(input('Entrez un nombre b : '))
    print('a/b = ', a/b)
except ZeroDivisionError as exception_retournee:
    print("L'erreur suivante a été trouvée :", exception_retournee)
```

```
Entrez un nombre b : 0
L'erreur suivante a été trouvée : division by zero
```

Sur cet exemple, la division par zéro est en `as exception_retournee`, vous pouvez donc afficher le message (en anglais…) de l'erreur.

Instruction si tout se passe bien :

Vous pouvez effectuer également une instruction si tout se passe bien avec le mot-clé `else` :

Entrée [21]:

```
try:
    a = 1
    b = int(input('Entrez un nombre b : '))
    q = a/b
except ValueError:
    print("La saisie de b est incompatible avec la division.")
except ZeroDivisionError:
    print("La saisie est égale à 0.... et on ne peut pas diviser par zéro")
else:
    print('a/b = ', q)
```

```
Entrez un nombre b : 2
a/b =  0.5
```

Si vous n'avez pas de levée d'exception alors la dernière ligne est exécutée.

3.5. La méthode complète

On peut exécuter des instructions après le bloc `try` quel que soit le résultat avec `finally` même si Python trouve une instruction `return` dans le bloc `except`.

Entrée [22]:

```
try:
    a = 1
    b = int(input('Entrez un nombre b : '))
    q = a/b
except ValueError:
    print("La saisie de b est incompatible avec la division.")
except ZeroDivisionError:
    print("La saisie est égale à 0.... et on ne peut pas diviser par zéro")
else:
    print('a/b = ', q)
finally :
    print("Cela s'affiche dans tous les cas...")
```

```
Entrez un nombre b : 2
a/b =  0.5
Cela s'affiche dans tous les cas...
```

Si l'on souhaite juste ignorer une erreur, on peut se contenter d'ajouter le mot clé `pass` dans le bloc `except`.

4. Les mécanismes d'assertion

4.1. Le principe

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.

Les assertions sont un moyen simple de s'assurer, avant de continuer, qu'une condition est respectée. Elles seront usuellement insérées dans des fonctions ou des blocs `try`/`except`.

On pourra positionner des assertions pour tester un programme ou rechercher des bugs.
Elle est essentiellement basée sur les structures conditionnelles.

4.2. La syntaxe

assert *test_booleen*, 'texte à afficher si *test_booleen* est faux'

Exemple pour tester une saisie utilisateur (nombre > 4) :

```
Entrée [23]:
1 a = 1
2 b = int(input('Entrez un nombre b positif supérieur à 4 : '))
3 assert b > 0, 'b doit être strictement positif'
4 assert b>=4, 'b doit être supérieur ou égal à 4'
5 print('a/b = ', a/b)

Entrez un nombre b positif supérieur à 4 : 0

---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-23-7b316b4ccaed> in <module>
      1 a = 1
      2 b = int(input('Entrez un nombre b positif supérieur à 4 : '))
----> 3 assert b > 0, 'b doit être strictement positif'
      4 assert b>=4, 'b doit être supérieur ou égal à 4'
      5 print('a/b = ', a/b)

AssertionError: b doit être strictement positif
```

Vous pouvez adapter le message à afficher lors du déclenchement de l'assertion.

Ce dispositif est très utile dans un bloc `try`/`except` pour afficher les raisons de l'arrêt du programme (ou pour la recherche de bugs).

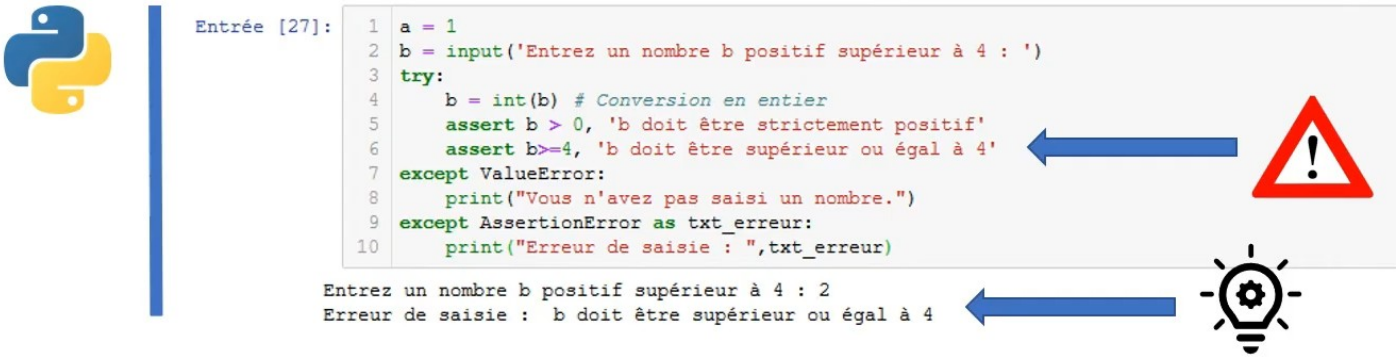
4.3. L'intérêt des assertions ?

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.

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.

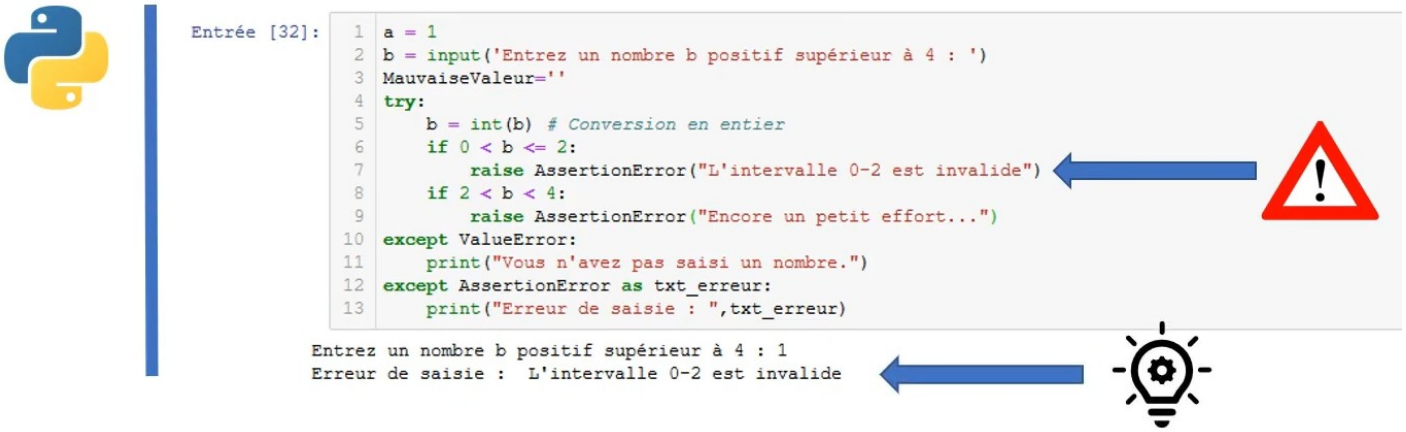
Coupler des assertions à un bloc `try`/`except` comporte de nombreux avantages :



L'exception due aux assertions est récupérée dans `l'except` en la renommant en `txt_erreur`, ce qui permet de récupérer le message d'erreur de l'assertion.

Associée à une boucle `while`, une assertion permet de réaliser un contrôle très fin des saisies des utilisateurs, en rendant plus clairs les messages d'erreurs.

4.4. Peut-on lever volontairement une exception ?



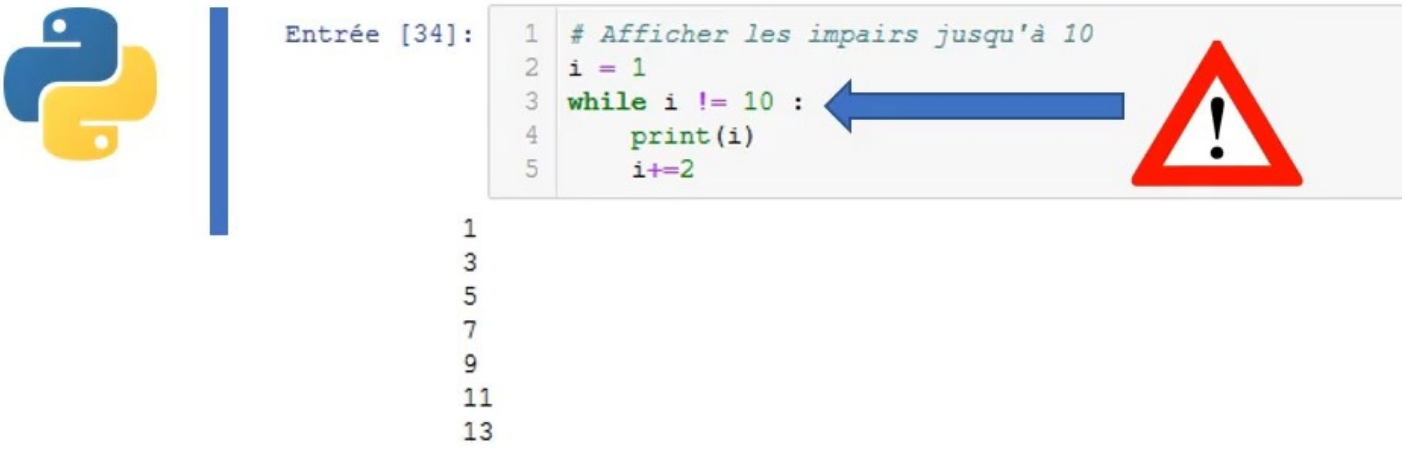
Il existe le mot-clé `raise` qui est utilisable à tout endroit du code et pas seulement dans un bloc `try`. Il peut être sans valeur dans un bloc `except`, il permet ainsi de ne pas « bloquer » une exception et de la propager. On peut donc l'utiliser pour déclencher à nouveau une erreur dans un bloc `except` non typé. Ici on lève une erreur d'assertion qui est récupérée par le bloc `except`.

5. Les problèmes liés aux boucles

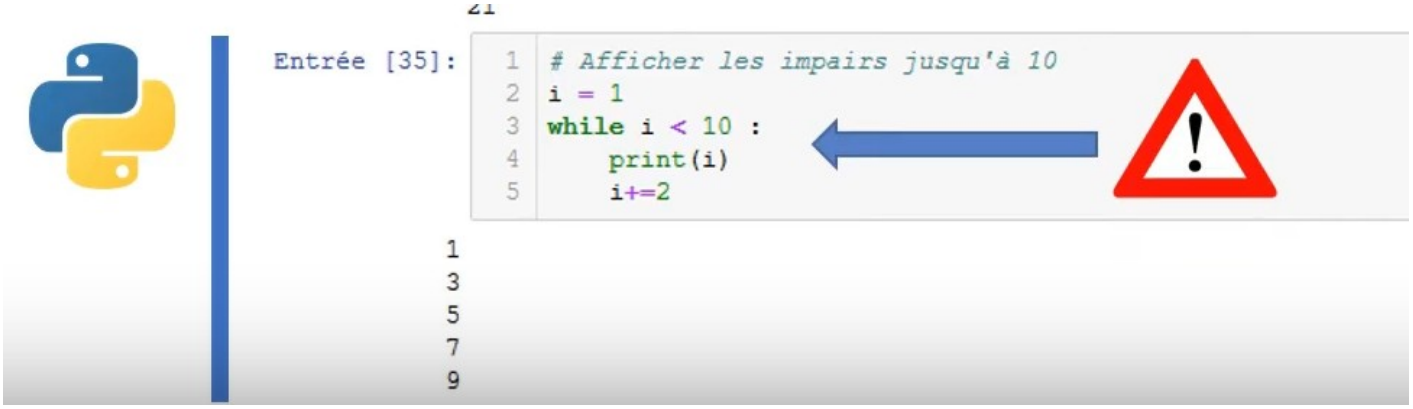
Il s'agit principalement de problèmes liés aux boucles `while`.

5.1. Le principe

Il faut toujours prouver qu'une boucle `while` se termine, c'est à dire que la structure conditionnelle indiquée dans le `while` puisse passer de `True` à `False`.
Ce n'est pas toujours évident lorsqu'il s'agit de compteurs, comme dans l'exemple suivant :



Le compteur correspond à des valeurs impaires et la boucle est infinie. Pour corriger, il faut modifier la structure conditionnelle pour qu'elle soit plus restrictive :



5.2. Fonctionnement du while / condition

Votre condition booléenne doit toujours pouvoir passer à l'état `False` : il faut le vérifier
Si vous utilisez un compteur, n'oubliez pas de l'incrémenter et de vérifier que la condition peut passer à l'état `False` ! Sinon vous créez ce qu'on appelle une boucle infinie, puisque la condition du `while` est toujours vraie…
Si votre code se lance dans une boucle infinie, vous devrez arrêter vous-même le programme, Python ne le fera pas tout seul car, pour lui, il se passe bel et bien quelque chose. De toute façon, il est incapable de différencier une boucle infinie d'une boucle finie.

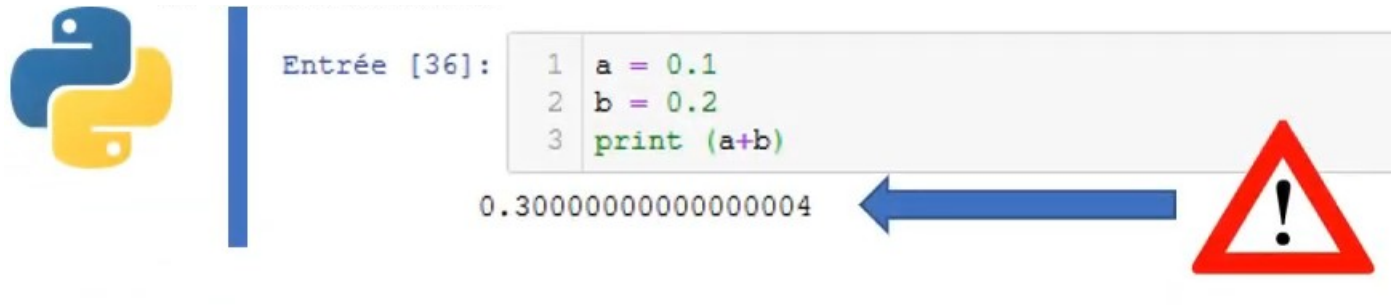
Savoir si un programme se termine n'est pas une chose facile…

Pour étudier un algorithme, le plus efficace est de la faire tourner « à la main » : on exécute l'algorithme en utilisant uniquement une feuille et un crayon. Il convient alors de prouver sa terminaison de manière théorique et mathématique.

6. Le problème des flottants

6.1. Le principe

Le stockage des nombres réels (vu en 1ère) pose de nombreux problèmes d'arrondis.
Exemple pour l'arrondi suivant :

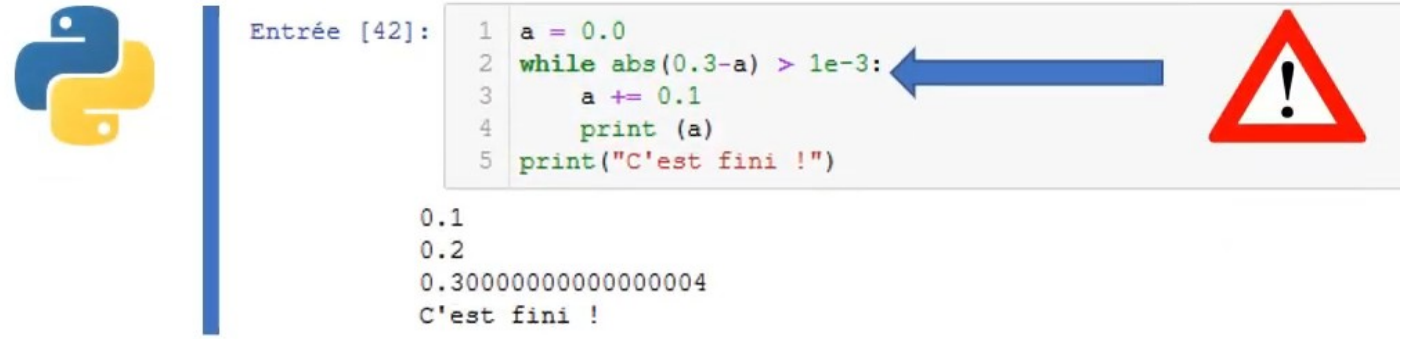


Si on se sert d'une valeur exacte dans une boucle :

On crée une boucle infinie qui ne devrait pas arriver théoriquement. Il faudra donc trouver une parade.

6.2. Les astuces pour contourner le problème

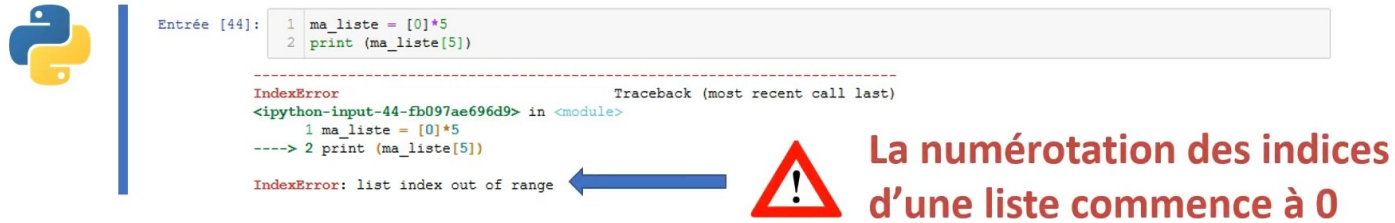
Lors des comparaisons des nombres flottants, il est préférable de raisonner en comparant « le reste d'une soustraction »… en utilisant au passage la valeur absolue pour s'affranchir des problèmes de signes !
Exemple :



7. Problèmes avec les listes (tableaux)

7.1. Parcourir une liste

Parcourir une liste et y récupérer une valeur est une chose commune dans un programme.
Avant de récupérer une valeur, il convient de s'assurer que l'indice existe bien afin d'éviter des erreurs.
Attention : l'indice de la liste commence à `0`



ma_liste →

Indice	0	1	2	3	4
Valeur	0	0	0	0	0

8. Pour aller plus loin

- Adaptez vos programmes pour contrôler toutes les erreurs possibles et éviter les arrêts intempestifs grâce aux `try`/`except`
- Sécurisez les saisies des utilisateurs avec des boucles
- Vérifiez impérativement les conditions d'arrêt des boucles `while`
- Testez vos scripts avec des assertions
- Méfiez-vous des calculs avec les flottants…

Il est toujours préférable d'effectuer une batterie de tests avant de fournir un code à des utilisateurs… et surtout ne pas oublier d'imaginer les pires des saisies possibles.

Ce mode de programmation qui exploite les assertions pour vérifier les préconditions, est appelé **programmation défensive**.

Source : https://www.youtube.com/watch?v=cCFuNC7L85w