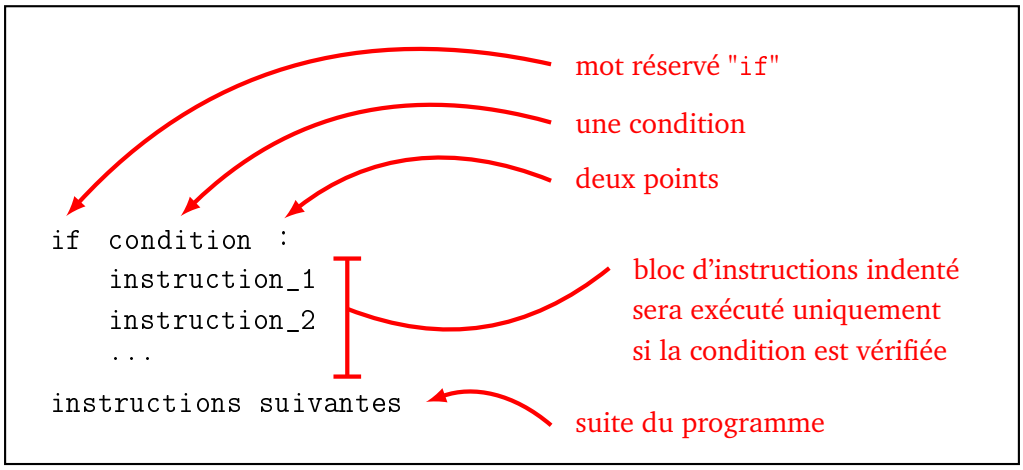
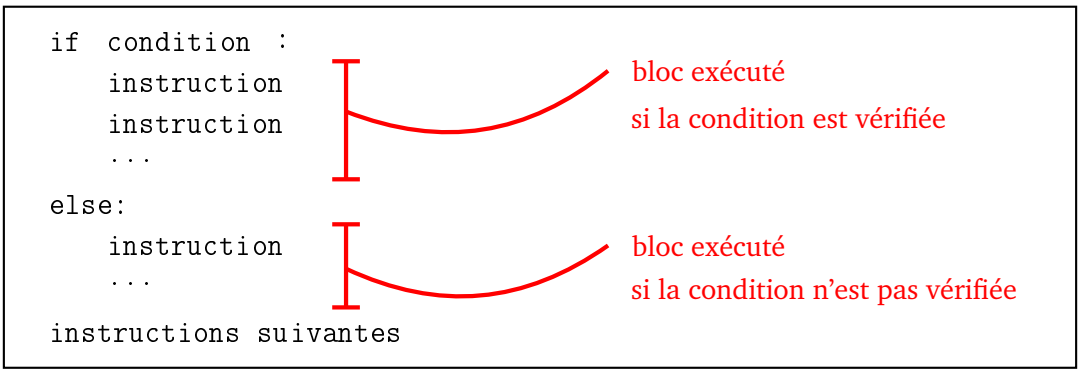
Guide de survie Python

1. Test et boucles

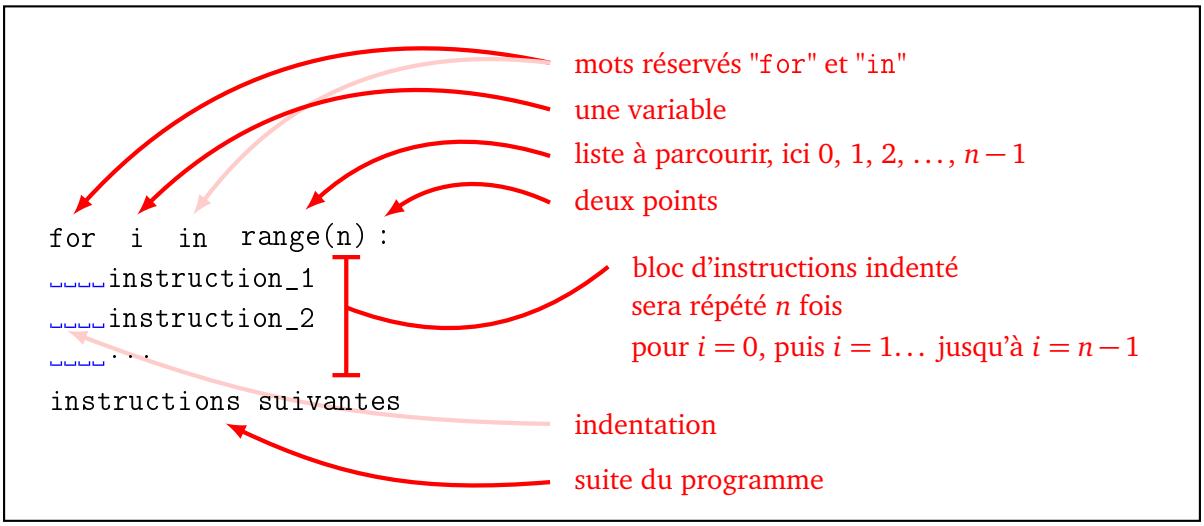
1.1. Si ... alors ...



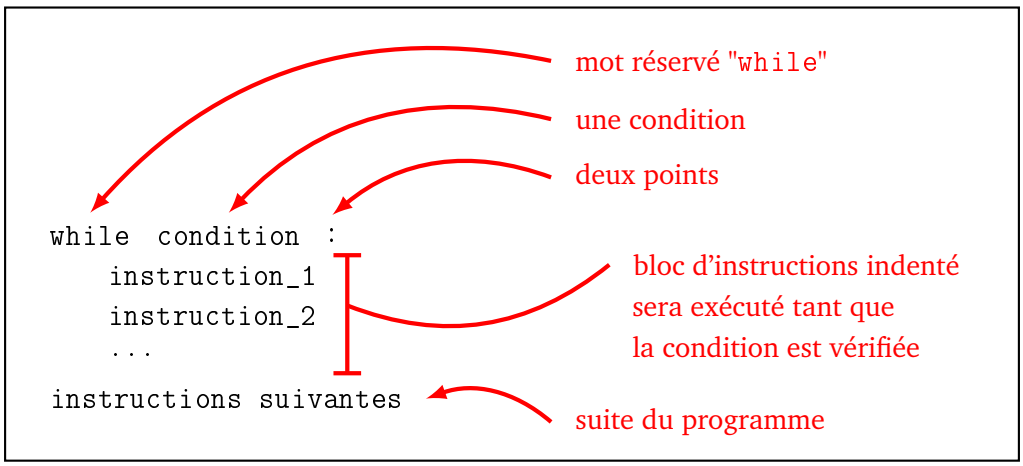
1.2. Si ... alors ... sinon ...



1.3. Boucle pour



1.4. Boucle tant que



1.5. Quitter une boucle

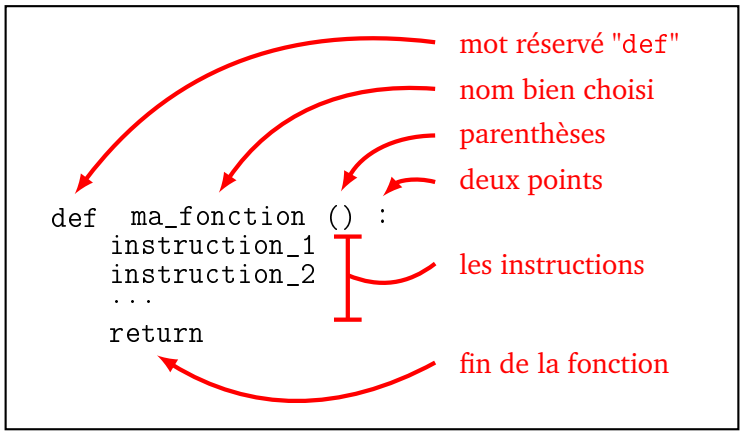
La commande `Python` pour quitter immédiatement une boucle « tant que » ou une boucle « pour » est l'instruction `break`.

2. Type de données

- `int` Entier. Exemples : 123 ou -15.
- `float` Nombre flottant (ou à virgule). Exemples : `4.56`, `-0.001`, `6.022e23` (pour 6.022×10^{23}), `4e-3` (pour $0.004 = 4 \times 10^{-3}$).
- `str` Caractère ou chaînes de caractères. Exemples : `'Y'`, `"k"`, `'Hello'`, `"World !"`.
- `bool` Booléen. `True` ou `False`.
- `list` Liste. Exemple : `[1,2,3,4]`.

3. Définir des fonctions

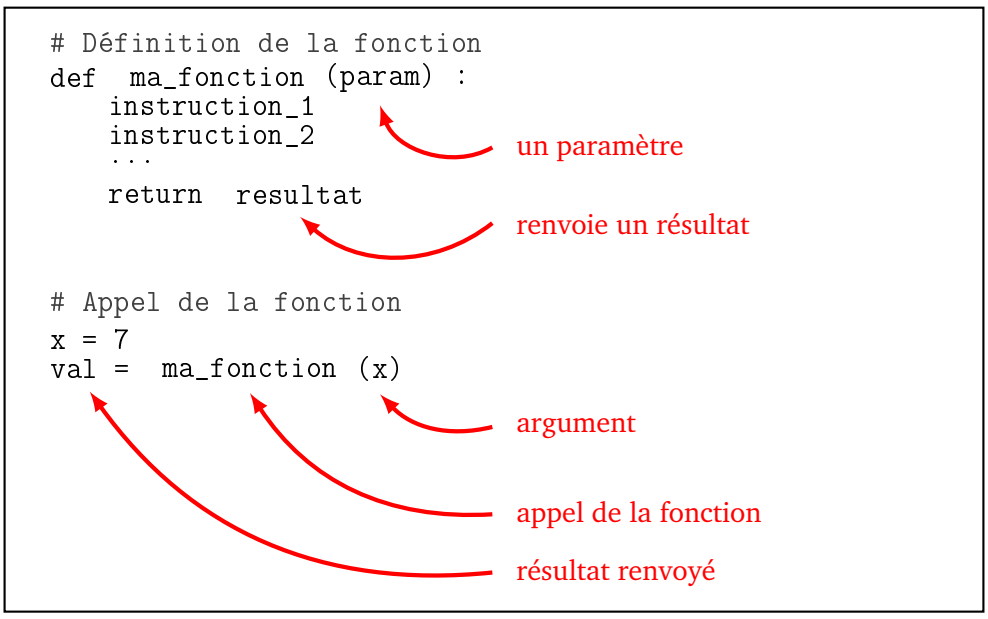
3.1. Définition d'une fonction



3.2. Fonction avec paramètre

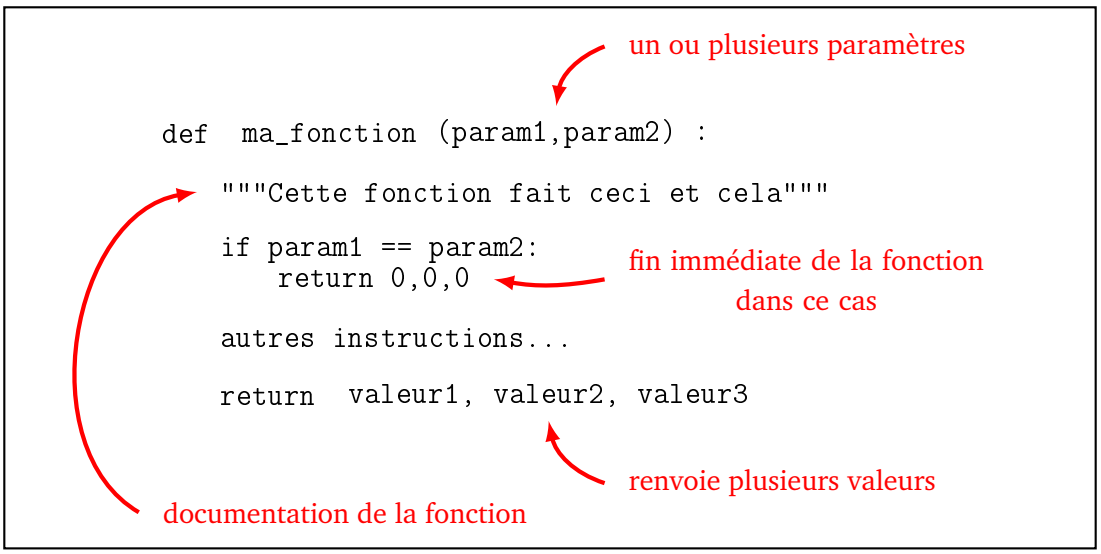
Les fonctions informatiques acquièrent tout leur potentiel avec :

- une ***entrée***, qui regroupe des variables qui servent de ***paramètres***,
- une ***sortie***, qui est un résultat renvoyé par la fonction (et qui souvent dépendra des paramètres d'entrée).



3.3. Fonction avec plusieurs paramètres

Il peut y avoir plusieurs paramètres en entrée, il peut y avoir plusieurs résultats en sortie.



Voici un exemple d'une fonction avec deux paramètres et deux sorties.

```
def somme_produit(x,y):
    """ Calcule la somme et le produit de deux nombres. """
    S = x + y     # Somme
    P = x*y       # Produit
    return S, P   # Renvoie les résultats

# Appel de la fonction
som, prod = somme_produit(3,7)  # Résultats
print("Somme :",som)            # Affichage
print("Produit :",prod)         # Affichage
```

- Très important ! Il ne faut pas confondre afficher et renvoyer une valeur. L'affichage (par la commande `print()`) affiche juste quelque chose à l'écran. La plupart des fonctions n'affichent rien, mais renvoient une valeur (ou plusieurs). C'est beaucoup plus utile car cette valeur peut être utilisée ailleurs dans le programme.
- Dès que le programme rencontre l'instruction `return`, la fonction s'arrête et renvoie le résultat. Il peut y avoir plusieurs fois l'instruction `return` dans une fonction mais une seule sera exécutée. On peut aussi ne pas mettre d'instruction `return` si la fonction ne renvoie rien.
- Dans les instructions d'une fonction, on peut bien sûr faire appel à d'autres fonctions !

3.4. Commentaires et docstring

- **Commentaire.** Tout ce qui suit le signe dièse `#` est un commentaire et est ignoré par `Python`. Par exemple :

```
# Boucle principale
while r != 0:   # Tant que le reste n'est pas nul
    r = r - 1   # Diminuer le reste
```

- **Docstring.** Tu peux décrire ce que fait une fonction en commençant par un *docstring*, c'est-à-dire une description en français, entourée par trois guillemets. Par exemple :

```
def produit(x,y):
    """ Calcule le produit de deux nombres
    Entrée : deux nombres x et y
    Sortie : le produit de x par y """
```

```
    p = x * y
    return p
```

3.5. Variable locale

Voici une fonction toute simple qui prend en entrée un nombre et renvoie le nombre augmenté de un.

```
def ma_fonction(x):
    x = x + 1
    return x
```

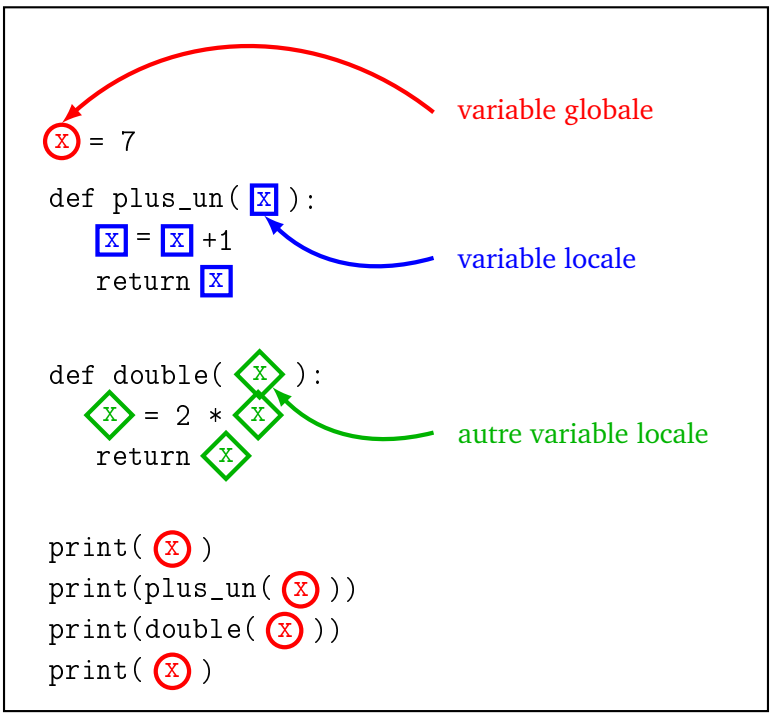
- Bien évidemment `ma_fonction(3)` renvoie `4`.
- Si la valeur de `y` est `5`, alors `ma_fonction(y)` renvoie `6`. Mais attention, la valeur de `y` n'a pas changé, elle vaut toujours `5`.
- Voici la situation problématique qu'il faut bien comprendre :

  ```
  x = 7
  print(ma_fonction(x))
  print(x)
  ```

  — La variable `x` est initialisée à `7`.
  — L'appel de la fonction `ma_fonction(x)` est donc la même chose que `ma_fonction(7)` et renvoie logiquement `8`.
  — Que vaut la variable `x` à la fin ? La variable `x` est inchangée et vaut toujours `7` ! Même s'il y a eu entre temps une instruction `x = x + 1`. Cette instruction a changé le `x` à l'intérieur de la fonction, mais pas le `x` en dehors de la fonction.

- Les variables définies à l'intérieur d'une fonction sont appelées ***variables locales***. Elles n'existent pas en dehors de la fonction.
- Si une variable dans une fonction porte le même nom qu'une variable dans le programme (comme le `x` dans l'exemple ci-dessus), il y a deux variables distinctes ; la variable locale n'existant que dans la fonction.

Pour bien comprendre la portée des variables, tu peux colorier les variables globales d'une fonction en rouge, et les variables locales avec une couleur par fonction. Le petit programme suivant définit une fonction qui ajoute un, et une autre qui calcule le double.



Le programme affiche d'abord la valeur de `x`, donc 7, puis il ajoute un à 7, il affiche donc 8, puis il affiche le double de `x`, donc 14. La variable globale `x` n'a jamais changé, le dernier affichage de `x` est donc encore 7.

3.6. Variable globale

Une ***variable globale*** est une variable qui est définie pour l'ensemble du programme. Il n'est généralement pas recommandé d'utiliser de telles variables, mais cela peut être utile dans certain cas. Voyons un exemple. On déclare la variable globale, ici la constante de gravitation, en début de programme comme une variable classique :

```
gravitation = 9.81
```

La contenu de la variable `gravitation` est maintenant accessible partout. Par contre, si on souhaite changer la valeur de cette variable dans une fonction, il faut bien préciser à `Python` que l'on est conscient de modifier une variable globale !
Par exemple pour des calculs sur la Lune, il faut changer la constante de gravitation qui y est beaucoup plus faible.

```
def sur_la_lune():
    global gravitation   # Oui, je veux modifier cette variable globale !
    gravitation = 1.625  # Nouvelle valeur pour tout le programme
    ...
```

3.7. Arguments optionnels

Il est possible de donner des arguments optionnels. Voici comment définir une fonction (ici qui dessinerait un trait) en donnant des valeurs par défaut :

```
def tracer(longueur, epaisseur=5, couleur="blue"):
```

- La commande `tracer(100)` trace mon trait, et comme je n'ai précisé que la longueur, les arguments `epaisseur` et `couleur` prennent les valeurs par défaut (5 et bleu).
- La commande `tracer(100, epaisseur=10)` trace mon trait avec une nouvelle épaisseur (la couleur est celle par défaut).

- La commande `tracer(100, couleur="red")` trace mon trait avec une nouvelle couleur (l'épaisseur est celle par défaut).
- La commande `tracer(100, epaisseur=10, couleur="red")` trace mon trait avec une nouvelle épaisseur et une nouvelle couleur.
- Voici aussi ce que tu peux utiliser :
  — `tracer(100, 10, "red")` : ne pas préciser les noms des options si on fait attention à l'ordre.
  — `tracer(couleur="red", epaisseur=10, longueur=100)` : on peut nommer n'importe quelle variable ; les variables nommées peuvent être passées en paramètre dans n'importe quel ordre !

4. Modules

4.1. Utiliser un module

- `from math import *` Importe toutes les fonctions du module `math`. Pour pouvoir utiliser par exemple la fonction sinus par `sin(0)`. C'est la méthode la plus simple et c'est celle que nous utilisons dans ce livre.
- `import math` Permet d'utiliser les fonctions du module `math`. On a alors accès à la fonction sinus par `math.sin(0)`. C'est la méthode recommandée officiellement afin d'éviter les conflits entre les modules.

4.2. Principaux modules

- `math` contient les principales fonctions mathématiques.
- `random` simule le tirage au hasard.
- `turtle` la tortue `Python`, l'équivalent de *Scratch*.
- `matplotlib` permet de tracer des graphiques et visualiser des données.
- `tkinter` permet d'afficher des fenêtres graphiques.
- `time` pour l'heure, la date et chronométrer.
- `timeit` pour mesurer le temps d'exécution d'une fonction.

Il existe beaucoup d'autres modules !

5. Erreurs

5.1. Erreurs d'indentation

```
a = 3
 b = 2
```

`Python` renvoie le message d'erreur *IndentationError : unexpected indent*. Il indique le numéro de ligne où se situe l'erreur d'indentation, il pointe même à l'aide du symbole « ^ » l'endroit exact de l'erreur.

5.2. Erreurs de syntaxe

- ```
  while x >= 0
      x  = x - 1
  ```

  `Python` renvoie le message d'erreur *SyntaxError : invalid syntax* car il manque les deux points après la condition `while x >= 0 :`
- `chaine = Coucou le monde` renvoie une erreur car il manque les guillemets pour définir la chaîne de caractères.
  ```

- `print("Coucou"` Python renvoie le message d'erreur *SyntaxError : unexpected EOF while parsing* car l'expression est mal parenthésée.
- `if val = 1:` Encore une erreur de syntaxe, car il faudrait écrire `if val == 1:`.

5.3. Erreurs de type

- **Entier**

```
n = 7.0
for i in range(n):
    print(i)
```

Python renvoie le message d'erreur *TypeError : 'float' object cannot be interpreted as an integer*. En effet `7.0` n'est pas un entier, mais un nombre flottant.

- **Nombre flottant**

```
x = "9"
sqrt(x)
```

Python renvoie le message d'erreur *TypeError : a float is required*, car `"9"` est une chaîne de caractères et pas un nombre.

- **Mauvais nombre d'arguments**
`gcd(12)` Python renvoie le message d'erreur *TypeError : gcd() takes exactly 2 arguments (1 given)* car la fonction `gcd()` du module `math` a besoin des deux arguments, comme par exemple `gcd(12,18)`.

5.4. Erreurs de nom

- `if y != 0: y = y - 1` Python renvoie le message *NameError : name 'y' is not defined* si la variable `y` n'a pas encore été définie.
- Cette erreur peut aussi se produire si les minuscules/majuscules ne pas scrupuleusement respectées. `variable`, `Variable` et `VARIABLE` sont trois noms de variables différents.
- `x = sqrt(2)` Python renvoie le message *NameError : name 'sqrt' is not defined*, il faut importer le module `math` pour pouvoir utiliser la fonction `sqrt()`.
- **Fonction non encore définie**

```
produit(6,7)

def produit(a,b):
    return a*b
```

Renvoie une erreur *NameError : name 'produit' is not defined* car une fonction doit être définie avant d'être utilisée.

5.5. Exercice

Corrige le code ! Python doit afficher `7 5 9`.

```
a == 7
if (a = 2) or (a >= 5)
    b = a - 2
     c = a + 2
else
b = a // 2
c = 2 * a
```

```
print(a b c)
```

5.6. Autres problèmes

Le programme se lance mais s'interrompt en cours de route ou bien ne fait pas ce que tu veux ? C'est là que les ennuis commencent, il faut décafardiser le code ! Il n'y a pas de solutions générales mais seulement quelques conseils :

- Un code propre, bien structuré, bien commenté, avec des noms de variables et de fonctions bien choisis est plus facile à relire.
- Teste ton algorithme à la main avec papier/crayon pour les cas faciles.
- N'hésite pas à afficher les valeurs des variables, pour voir leur évolution au cours du temps. Par exemple `print(i,liste[i])` dans une boucle.
- Une meilleure méthode pour inspecter le code est de visualiser les valeurs associées aux variables à l'aide des fonctionnalités *debug* de ton éditeur `Python` favori. Il est aussi possible de faire une exécution pas à pas.
- Est-ce que le programme fonctionne avec certaines valeurs et pas d'autres ? As-tu pensé aux cas extrêmes ? Est-ce que n est nul alors que ce n'est pas autorisé ? Est-ce que la liste est vide, alors que le programme ne gère pas ce cas ? etc.

Voici quelques exemples.

- Je veux afficher les carrés des entiers de 1 à 10. Le programme suivant ne renvoie pas d'erreur mais ne fait pas ce que je veux.

  ```
  for i in range(10):
      print(i ** 2)
  ```

  La boucle itère sur les entiers de 0 à 9. Il faut écrire `range(1,11)`.
- Je veux afficher le dernier élément de ma liste.

  ```
  liste = [1,2,3,4]
  print(liste[4])
  ```

  `Python` renvoie le message d'erreur *IndexError : list index out of range* car le dernier élément est celui de rang 3.
- Je veux faire un compte à rebours. Le programme suivant ne s'arrête jamais.

  ```
  n = 10
  while n != "0":
      n = n - 1
      print(n)
  ```

  Avec une boucle « tant que » il faut prendre grand soin de bien écrire la condition et de vérifier qu'elle finit pas être fausse. Ici, elle est mal formulée, cela devrait être `while n!= 0:`.