Python – Représentation des réels

Erreurs de calcul

Python utilise la virgule flottante en double précision pour représenter les nombres réels (64 bits). Il y a donc des erreurs de calculs possibles. Par exemple :

```
>>> 0.1*3 == 0.3
False
```

Ce n'est bien entendu pas la réponse attendue. On peut demander les valeurs des deux expressions :

```
>>> 0.3
0.3
>>> 0.1*3
0.30000000000000004
```

On pourrait se demander pourquoi une des valeurs est correcte et pas l'autre. En fait, aucune des deux valeurs n'est correcte. Pour s'en rendre compte, il faut utiliser les **f-strings** qui permettent, entre autre, d'afficher un nombre donné de chiffres après la virgule. La syntaxe générale est "`f"{expression:format}"`". Pour afficher `n` chiffres après la virgule, il faut utiliser le format "`.n`".

```
>>> f"{0.3:.20}"
' 0.29999999999999999889'
>>> f"{0.1*3:.20}"
' 0.30000000000000004441'
```

Il n'y a pas autant de chiffres affichés pour les deux nombres. C'est parce que ceux qui manquent sont des 0. On remarque néanmoins qu'aucune des deux valeurs n'est exacte, mais que 0.3 est un peu plus proche de la vraie valeur.
Pour une description plus détaillée des f-strings, vous pouvez consulter :
https://he-arc.github.io/livre-python/fstrings/index.html.
On peut déterminer l'écart entre les deux nombres en faisant la différence :

```
>>> 0.1*3 - 0.3
5.551115123125783e-17
```

On obtient un nombre en écriture scientifique. La notation `e-17` correspond à $\times 10^{-17}$.

```
>>> 1.7e2
170.0
>>> 1.7e-8
1.7e-08
```

L'utilisation d'un nombre en écriture scientifique transforme automatiquement le nombre en réel. Mais si le nombre est entier, sa valeur sera exacte.

```
>>> f"{1.7e2:.20}"
' 170.0'
>>> f"{17e-1:.20}"
' 1.6999999999999999556'
```

Il n'est pas conseillé de tester des égalités sur des réels. Il vaut mieux vérifier que leur différence est infinitésimale.

```
>>> abs(0.3 - 0.1*3) < 1e-16
True
>>> abs(0.1*3 - 0.3) < 1e-16
True
```

La fonction `abs(expr)` donne la valeur absolue de `expr`, c'est-à-dire la valeur sans le signe. Il n'est donc pas nécessaire de connaître à l'avance laquelle des deux valeurs est la plus grande.

```
>>> abs(-7)
7
>>> abs(5-7)
2
```

Application

On considère la fonction suivante qui permet de déterminer si les longueurs `a`, `b` et `c`, `c` étant la plus grande, permettent de former un triangle rectangle dont les côtés mesurent `a`, `b` et `c`.

```
def pythagore_base(a, b, c):
    return a**2 + b**2 == c**2
```

On peut remarquer que la fonction donne le résultat attendu avec des valeurs entières, mais que ce n'est pas forcément le cas avec des valeurs réelles.

```
>>> pythagore_base(3, 4, 5)
True
>>> pythagore_base(0.3, 0.4, 0.5)
True
>>> pythagore_base(0.03, 0.04, 0.05)
False
```

Exercice 1 : Corriger la fonction pour qu'elle donne la bonne réponse, en utilisant une marge d'erreur de 10^{-16}.

```
>>> pythagore_base(0.3, 0.4, 0.5)
True
>>> pythagore_base(0.3, 0.4, 0.500000000001)
False
```

Exercice 2 : Écrire une fonction `pythagore(a, b, c)` qui détermine si on peut faire un triangle rectangle avec les longueurs `a`, `b` et `c`, peu importe l'ordre. Vous pouvez appeler la fonction `pythagore_base` dans cette fonction.

```
>>> pythagore(0.3, 0.5, 0.4)
True
>>> pythagore(0.5, 0.4, 0.3)
True
>>> pythagore(0.4, 0.5, 0.3000000000000001)
False
```

Virgule fixe

Nous allons faire une fonction qui transforme un nombre décimal en base 10, en un nombre décimal en binaire. Nous n'allons pas encore utiliser de virgule flottante et juste mettre une partie entière et une partie décimale en indiquant éventuellement la partie qui se répète. Pour cela on considère les deux fonctions suivante.

```
def fois_2(val):
    retenue = 0           # On commence sans retenue
    i = len(val) - 1      # On commence par le chiffre à droite
    val2 = ""             # Le résultat est vide
    while i >= 0 :        # On parcourt de droite à gauche
        c = int(val[i])   # On convertit le symbole en nombre
        double = 2*c + retenue   # On fait le calcul
        retenue = double // 10   # On garde le chiffre des dizaines
        c2 = double % 10         # On garde le chiffre des unités
        if val2 != "" or c2 != 0:   # On regarde si on rajoute le chiffre
            val2 = str(c2) + val2
        i -= 1                   # On se décale à gauche
    if val2 == "":               # Si on n'a rien rajouté
        val2 = "0"               # C'est que le résultat est 0
    return retenue, val2   # On renvoie la retenue et le résultat


def entier_vers_bin(partie_entiere):
    rep = ""
    v = int(partie_entiere)
    while v > 1:
        rep = str(v%2) + rep
        v = v//2
    rep = str(v) + rep
    return rep
```

Les deux fonctions prennent un entier donné sous forme de texte. La fonction `fois_2(val)` calcule le double du nombre et renvoie l'éventuelle retenue et le reste du double, en ayant enlevé les zéros à droite, s'il y en a. La fonction `entier_vers_bin(val)` renvoie la version binaire de l'entier `val`.

```
>>> fois_2("75")       # 2 * 0.75 = 1.5
(1, '5')
>>> fois_2("1954")     # 2 * 0.1954 = 0.3908
(0, '3908')
>>> fois_2("0")
(0, '0')
>>> fois_2("7000")     # 0.7000 = 0.7
(1, '4')
>>> entier_vers_bin("5")
'101'
>>> entier_vers_bin("63")
'111111'
```

Exercice 3 (sur papier) : Faire un tableau pour représenter les différentes valeurs prises par `v` et `rep` au cours de l'exécution de `entier_vers_bin(19)`.

Exercice 4 (sur papier) : Faire un tableau représentant les différentes valeurs de `val2`, `c`, `double` et `retenue` au cours des appels de `fois_2(val)` avec :
1) `val = "1954"` 2) `val = "75"` 3) `val = "7000"`

La fonction `fois_2` renvoyant un couple de valeurs, pour séparer ces valeurs, il faut le faire de la manière suivante :

```
>>> r, v = fois_2("512") # 0.512 * 2 = 1.024
>>> r
1
>>> v
'024'
```

Exercice 5 : Compléter la fonction ci-dessous qui détermine la partie décimale en binaire correspondant à `partie_decimale`. Par exemple `decimal_vers_bin("12")` correspond à la partie décimale de 0,12. Vous pouvez utiliser la fonction `fois_2`.

```
def decimal_vers_bin(partie_decimale):
    vus = [] # liste des restes déjà vus
    rep = "" # la conversion en binaire
    v = partie_decimale
    while v not in vus: # On attend de voir une répétition
        vus.append(v)    # On met v dans ceux qu'on a vu
        ...
        ...
    pos = vus.index(v)   # On regarde à quel moment on a déjà vu la dernière valeur
    if v != "0": # on a un cycle infini
        return rep[:pos] + "[" + rep[pos:] + "]"
    else: # on a une valeur exacte
        return rep[:-1]    # On renvoie tout sauf le dernier chiffre qui est 0
```

```
>>> decimal_vers_bin("1")
'0[0011]'
>>> decimal_vers_bin("2")
'[0011]'
>>> decimal_vers_bin("3")
'0[1001]'
>>> decimal_vers_bin("5")
'1'
>>> decimal_vers_bin("01")
'00[0000101000111101011 1]'
```

Exercice 6 (sur papier) : Faire un tableau pour représenter les différentes valeurs prises par `v` et `vus` lors de l'exécution de `decimal_vers_bin("1")`.

Exercice 7 : Tester les résultats de `decimal_vers_bin` avec des nombres avec plusieurs décimales afin de voir à partir de combien de chiffres après la virgule les calculs deviennent très longs.

Afin de découper le nombre entre la partie entière et la partie décimale, nous allons utiliser la fonction `split` qui s'utilise de la manière suivante :

```
>>> "211,3".split(",")
['211', '3']
>>> "211".split(",")
['211']
>>> "211,3,2,,71".split(",")
['211', '3', '2', '', '71']
```

Cette commande découpe la chaîne associée en une liste. À chaque fois qu'on croise le symbole donné en paramètre, un nouvel élément est ajouté à la liste. On peut remarquer que s'il n'y a pas de virgules, la liste ne contient qu'un seul élément. Il faut donc distinguer deux cas : celui où la liste obtenue n'a qu'un seul élément, c'est donc un nombre entier, ou alors il y a deux éléments : la partie entière et la partie décimale.

Exercice 8 : Écrire une fonction `reel_vers_bin(reel)` qui prend un réel positif donné sous forme de texte et qui renvoie la version binaire, toujours sous forme de texte.

```
>>> reel_vers_bin("9")
'1001'
>>> reel_vers_bin("9,5")
'1001,1'
>>> reel_vers_bin("9,3")
'1001,0[1001]'
```

Exercice 9 : Rajouter la gestion des nombres négatifs dans `reel_vers_bin`. Si vous avez besoin d'utiliser une chaîne de caractères, sans prendre le premier symbole, vous pouvez utiliser `chaine[1:]`. On n'utilisera pas de complément à 2, mais plutôt le symbole "-" devant les nombres négatifs.

```
>>> reel_vers_bin("-9")
'-1001'
>>> reel_vers_bin("-9,3")
'-1001,0[1001]'
```

Virgule flottante

On considère la fonction suivante qui permet de transformer un entier donné en binaire en entier en base 10.

```
def bin_vers_entier(binaire):
    rep = 0
    for b in binaire:
        rep = rep * 2
        if b == "1":
            rep = rep + 1
    return rep
```

Pour convertir le nombre, il faut commencer par calculer la valeur de l'exposant et soustraire 127. On obtient alors le nombre e. Si on note m_1, m_2… les chiffres de la mantisse, on obtient alors la valeur du réel en faisant $2^e + m_1 \times 2^{e-1} + m_2 \times 2^{e-2} + \ldots + m_{23} \times 2^{e-23}$. Pour rappel, enlever 1 à l'exposant revient à diviser par 2 le coefficient précédent. Il ne reste plus qu'à appliquer le signe pour obtenir le résultat.

Exercice 10 : Écrire une fonction `flottant_vers_reel(flottant)` qui prend un triplet `flottant = (signe, exposant, mantisse)` et qui retourne le nombre décimal correspondant en base 10. Les 3 parties du triplet sont des nombres binaires donnés sous forme de texte. Le signe est composé d'un bit, l'exposant de 8 et la mantisse de 23.

```
>>> flottant_vers_reel(('0', '10000110', '10100110100110000000000'))
211.296875
>>> flottant_vers_reel(('1', '01111101', '10000000000000000000000'))
-0.375
```

Vous pouvez tester votre fonction en utilisant ce site :
https://www.h-schmidt.net/FloatConverter/IEEE754.html

Pour aller plus loin

Faire la conversion dans l'autre sens est un peu plus compliqué. L'algorithme global est le suivant :

- Découper le nombre en partie entière et partie décimale.
- Déterminer le signe.
- Si la partie entière n'est pas nulle, la convertir en binaire, en enlevant le 1 de gauche et calculer la puissance de 2 correspondant à ce 1.
  Sinon, commencer à calculer la partie décimale en binaire et chercher le premier 1 et la puissance de 2 correspondant.
- Calculer l'exposant en prenant la puissance de 2 trouvée et en ajoutant 127
- Convertir cet exposant en binaire sur 8 bits.
- Calculer la mantisse jusqu'à arriver à 23 bits.
- Si le premier chiffre de la partie décimale restante est 5 ou plus, ajouter 1 à la mantisse.

Exercice 11 : Écrire une fonction `entier_vers_bin_expo(entier)` qui renvoie l'exposant correspondant à la puissance de 2 associée au 1 le plus à gauche et renvoie le reste du nombre en binaire. On suppose que le nombre donné est sous forme de texte et strictement positif.

```
>>> entier_vers_bin_expo("14") # 14 -> 1110
(3, '110')
>>> entier_vers_bin_expo("5")  # 5 -> 101
(2, '01')
>>> entier_vers_bin_expo("1")  # 1 -> 1
(0, '')
```

Exercice 12 : Écrire une fonction `entier_vers_bin_taille(entier, taille)` qui renvoie l'écriture binaire de l'entier `entier`, avec `taille` bits.

```
>>> entier_vers_bin_taille(52, 8)
'00110100'
>>> entier_vers_bin_taille(135, 8)
'10000111'
>>> entier_vers_bin_taille(0, 8)
'00000000'
```

Exercice 13 : Écrire une fonction `reel_vers_flottant(reel)` qui renvoie le triplet `(signe, exposant, mantisse)` qui correspond au réel donné sous forme de texte.

```
>>> reel_vers_flottant("125")
('0', '10000101', '11110100000000000000000')
>>> reel_vers_flottant("0,3")
('0', '01111101', '00110011001100110011010')
>>> reel_vers_flottant("-51,79")
('1', '10000100', '10011110010100011110110')
```

Vous pouvez utiliser la fonction suivante pour ajouter 1 à la mantisse en cas d'arrondi par excès :

```
def ajouter_1(nb):
    rep = ""
    retenue = 1
    i = len(nb)
    while i>0:
        i -= 1
        if retenue == 1:
            if nb[i] == "1":
                rep = "0" + rep
            else:
                rep = "1" + rep
                retenue = 0
        else:
            rep = nb[i] + rep
    return rep
```

Dans la suite, vous trouverez une version à trous de la fonction `reel_vers_flottant`, puis la version complète.

Version à trous

```
def reel_vers_flottant2(reel):
    decomp = reel.split(",")
    signe = "0"
    partie_entiere = decomp[0]
    # On soccupe du signe -
    if partie_entiere[0] == "-":
        signe = "1"
        partie_entiere = partie_entiere[1:]
    if len(decomp) > 1:
        # Il y a une partie decimale
        partie_decimale = decomp[1]
    else:
        # Il n'y a pas de partie decimale
        partie_decimale = "0"
    if partie_entiere != "0":
        # On convertit la partie entière
        exposant, mantisse = entier_vers_bin_expo(partie_entiere)
    else:
        # On cherche le 1e 1 dans les decimales
        d = 0
        exposant = 0
        while d == 0:
            ...
            ...
        mantisse = ""
    exposant = exposant + 127
    # On convertit l'exposant en binaire sur 8 bits
    exposant = entier_vers_bin_taille(exposant, 8)
    # On complete jusqu'a avoir 23 chiffres
    while len(mantisse) < 23:
        ...
        ...
    # On arrondit si le chiffre suivant est 5 ou plus
    if int(partie_decimale[0]) >= 5:
        ...
    return (signe, exposant, mantisse)
```

Version complète

```
def reel_vers_flottant2(reel):
    decomp = reel.split(",")
    signe = "0"
    partie_entiere = decomp[0]
    # On soccupe du signe -
    if partie_entiere[0] == "-":
        signe = "1"
        partie_entiere = partie_entiere[1:]
    if len(decomp) > 1:
        # Il y a une partie decimale
        partie_decimale = decomp[1]
    else:
        # Il n'y a pas de partie decimale
        partie_decimale = "0"
    if partie_entiere != "0":
        # On convertit la partie entière
        exposant, mantisse = entier_vers_bin_expo(partie_entiere)
    else:
        # On cherche le 1e 1 dans les decimales
        d = 0
        exposant = 0
        while d == 0:
            d, partie_decimale = fois_2(partie_decimale)
            exposant -= 1
        mantisse = ""
    exposant = exposant + 127
    # On convertit l'exposant en binaire sur 8 bits
    exposant = entier_vers_bin_taille(exposant, 8)
    # On complete jusqu'a avoir 23 chiffres
    while len(mantisse) < 23:
        d, partie_decimale = fois_2(partie_decimale)
        mantisse = mantisse + str(d)
    # On arrondit si le chiffre suivant est 5 ou plus
    if int(partie_decimale[0]) >= 5:
        mantisse = ajouter_1(mantisse)
    return (signe, exposant, mantisse)
```