

# 14. Arithmétique en nombres à virgule flottante : problèmes et limites

Les nombres à virgule flottante sont représentés, au niveau matériel, en fractions de nombres binaires (base 2). Par exemple, la fraction décimale :

---

0.125

---

a la valeur  $1/10 + 2/100 + 5/1000$  et, de la même manière, la fraction binaire :

---

0.001

---

a la valeur  $0/2 + 0/4 + 1/8$ . Ces deux fractions ont une valeur identique, la seule différence est que la première est une fraction décimale, la seconde est une fraction binaire.

Malheureusement, la plupart des fractions décimales ne peuvent pas avoir de représentation exacte en fractions binaires. Par conséquent, en général, les nombres à virgule flottante que vous donnez sont seulement approximatés en fractions binaires pour être stockés dans la machine.

Le problème est plus simple à aborder en base 10. Prenons par exemple, la fraction  $1/3$ . Vous pouvez l'approximer en une fraction décimale :

---

0.3

---

ou, mieux,

---

0.33

---

ou, mieux,

---

0.333

---

etc. Peu importe le nombre de décimales que vous écrivez, le résultat ne vaut jamais exactement  $1/3$ , mais c'est une estimation s'en approchant toujours mieux.

De la même manière, peu importe combien de décimales en base 2 vous utilisez, la valeur décimale 0.1 ne peut pas être représentée exactement en fraction binaire. En base 2,  $1/10$  est le nombre périodique suivant :

---

0.00011001100110011001100110011001100110011001100110011...

---

Arrêtez à n'importe quelle quantité finie de bits, et vous obtiendrez une approximation.

Pour Python, sur une machine typique, 53 bits sont utilisés pour la précision d'un flottant, donc la valeur stockée lorsque vous entrez le nombre décimal 0.1 est la fraction binaire

---

0.00011001100110011001100110011001100110011001100110011010

---

qui est proche, mais pas exactement égale, à  $1/10$ .

Il est facile d'oublier que la valeur stockée est une approximation de la fraction décimale d'origine, du fait de la manière dont les flottants sont affichés dans l'interpréteur. Python n'affiche qu'une approximation décimale de la valeur stockée en binaire. Si Python devait afficher la vraie valeur décimale de l'approximation binaire stockée pour 0.1, il afficherait :

---

```
>>> 0.1
0.1000000000000000055511151231257827021181583404541015625
```

---

C'est bien plus de décimales que ce qu'attendent la plupart des utilisateurs, donc Python affiche une valeur arrondie afin d'améliorer la lisibilité :

---

```
>>> 0.1
0.1
```

---

Il est important de comprendre qu'en réalité, c'est une illusion : la valeur stockée n'est pas exactement 1/10, c'est simplement à *l'affichage* que la valeur stockée est arrondie. Ceci devient évident dès que vous effectuez des opérations arithmétiques avec ces valeurs :

---

```
>>> 0.1 + 0.2
0.30000000000000004
```

---

Ce comportement est inhérent à la nature même de la représentation des nombres à virgule flottante dans la machine : ce n'est pas un bogue dans Python et ce n'est pas non plus un bogue dans votre code. Vous pouvez observer le même type de comportement dans tous les autres langages utilisant le support matériel pour le calcul des nombres à virgule flottante (bien que certains langages ne rendent pas visible la différence par défaut, ou pas dans tous les modes d'affichage).

Une autre surprise est inhérente à celle-ci. Par exemple, si vous tentez d'arrondir la valeur 2.675 à deux décimales, vous obtiendrez

---

```
>>> round(2.675, 2)
2.67
```

---

La documentation de la primitive `round()` indique qu'elle arrondit à la valeur la plus proche en s'éloignant de zéro. Puisque la fraction décimale est exactement à mi-chemin entre 2.67 et 2.68, vous devriez vous attendre à obtenir (une approximation binaire de) 2.68. Ce n'est pourtant pas le cas, car lorsque la fraction décimale 2.675 est convertie en flottant, elle est stockée par une approximation dont la valeur exacte est

---

```
2.67499999999999982236431605997495353221893310546875
```

---

Puisque l'approximation est légèrement plus proche de 2.67 que 2.68, l'arrondi se fait vers le bas.

Si vous êtes dans une situation où les arrondis de nombres décimaux à mi-chemin ont de l'importance, vous devriez utiliser le module `decimal`. Soit dit en passant, le module `decimal` propose aussi un moyen pratique de « voir » la valeur exacte stockée pour n'importe quel flottant.

---

```
>>> from decimal import Decimal
>>> Decimal(2.675)
Decimal('2.67499999999999982236431605997495353221893310546875')
```

---

Une autre conséquence du fait que 0.1 n'est pas exactement stocké 1/10 est que la somme de dix valeurs de 0.1 ne donne pas 1.0 non plus :

---

```
>>> sum = 0.0
>>> for i in range(10):
...     sum += 0.1
...
>>> sum
0.9999999999999999
```

---

L'arithmétique des nombres binaires à virgule flottante réserve beaucoup de surprises de ce genre. Le problème avec « 0.1 » est expliqué en détails ci-dessous, dans la section « Erreurs de représentation ». Voir [The Perils of Floating Point](#) pour une liste plus complète de ce genre de surprises.

Il est vrai qu'il n'existe pas de réponse simple, cependant ne vous méfiez pas trop des nombres à virgule flottante ! Les erreurs, en Python, dans les opérations de nombres à virgule flottante sont dues au matériel sous-jacent, et sur la plupart des machines ne sont pas plus importantes que 1 sur  $2^{53}$  par opération. C'est plus que nécessaire pour la plupart des tâches, mais vous devez garder à l'esprit que ce ne sont pas des opérations décimales, et que chaque opération sur des nombres à virgule flottante peut souffrir d'une nouvelle erreur.

Bien que des cas pathologiques existent, pour la plupart des cas d'utilisations courants vous obtiendrez le résultat attendu à la fin en arrondissant simplement au nombre de décimales désirées à l'affichage. Pour un contrôle fin sur la manière dont les flottants sont affichés, consultez dans [Syntaxe de formatage de chaîne](#) les spécifications de formatage de la méthode `str.format()`

## 14.1. Erreurs de représentation

---

Cette section explique en détail l'exemple du « 0.1 » et montre comment vous pouvez effectuer une analyse exacte de ce type de cas par vous-même. Nous supposons que la représentation binaire des nombres flottants vous est familière.

Le terme *Representation error* signifie que la plupart des fractions décimales ne peuvent être représentées exactement en binaire. C'est la principale raison pour laquelle Python (ou Perl, C, C++, Java, Fortran, et beaucoup d'autres) n'affiche habituellement pas le résultat exact en décimal:

---

```
>>> 0.1 + 0.2
0.30000000000000004
```

---

Pourquoi ? 1/10 et 2/10 ne sont pas représentable de manière exacte en fractions binaires. Cependant, toutes les machines d'aujourd'hui (Juillet 2010) suivent la norme IEEE-754 en ce qui concerne l'arithmétique des nombres à virgule flottante. et la plupart des plateformes utilisent un « IEEE-754 double precision » pour représenter les floats de Python. Les « IEEE-754 double precision » utilisent 53 bits de précision, donc à la lecture l'ordinateur essaie de convertir 0.1 dans la fraction la plus proche possible de la forme  $J/2^{**N}$  avec  $J$  un nombre entier d'exactly 53 bits. Réécrire :

---

$$1 / 10 \approx J / (2^{**N})$$

---

en :



