Nombres réels

Représentation de l'information

Décomposition

En base 10, l'expression 652,375 est une manière abrégée d'écrire :

$$6 \times 10^2 + 5 \times 10^1 + 2 \times 10^0 + 3 \times 10^{-1} + 7 \times 10^{-2} + 5 \times 10^{-3}.$$

Il en va de même pour la base 2. Ainsi, l'expression 110,101 signifie :

$$1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3}.$$

Conversion de binaire en décimal

On peut ainsi facilement convertir un nombre réel de la base 2 vers la base 10. Par exemple :

$$110{,}101_2 = 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} = 4 + 2 + 0{,}5 + 0{,}125 = 6{,}625_{10}.$$

Exercice 1 Transformer les nombres suivants de la base 2 vers la base 10 :

1. $0{,}0101010101_2$
2. $11100{,}10001_2$

Conversion de décimal en binaire

Le passage de base 10 en base 2 est plus subtil. Convertissons, par exemple, 1234,347 en base 2.

— La partie entière se transforme comme vu précédemment : $1234_{10} = 10011010010_2$

— On transforme la partie décimale selon le schéma suivant :

$$\begin{aligned}
0{,}347 \times 2 &= 0{,}694 & 0{,}347_{10} &= 0{,}0_2 \ldots \\
0{,}694 \times 2 &= 1{,}388 & 0{,}347_{10} &= 0{,}01_2 \ldots \\
0{,}388 \times 2 &= 0{,}776 & 0{,}347_{10} &= 0{,}010_2 \ldots \\
0{,}776 \times 2 &= 1{,}552 & 0{,}347_{10} &= 0{,}0101_2 \ldots \\
0{,}552 \times 2 &= 1{,}104 & 0{,}347_{10} &= 0{,}01011_2 \ldots \\
0{,}104 \times 2 &= 0{,}208 & 0{,}347_{10} &= 0{,}010110_2 \ldots \\
0{,}208 \times 2 &= 0{,}416 & 0{,}347_{10} &= 0{,}0101100_2 \ldots \\
0{,}416 \times 2 &= 0{,}832 & 0{,}347_{10} &= 0{,}01011000_2 \ldots \\
0{,}832 \times 2 &= 1{,}664 & 0{,}347_{10} &= 0{,}010110001_2 \ldots \\
0{,}664 \times 2 &= 1{,}328 & 0{,}347_{10} &= 0{,}0101100011_2 \ldots \\
0{,}328 \times 2 &= 0{,}656 & 0{,}347_{10} &= 0{,}01011000110_2 \ldots
\end{aligned}$$

On continue ainsi jusqu'à la précision désirée.

Attention. Un nombre à développement décimal fini en base 10 ne l'est pas forcément en base 2. Cela peut engendrer de mauvaises surprises. Prenons par exemple ce programme Python :

```
i = 0.0
while i < 1000:
    i += 0.01
print(i)
```

Le résultat attendu était 1000.0, mais on obtient 1000.0099999992356. Le problème vient du fait que l'on n'arrive pas à représenter exactement 0,01 en binaire : les erreurs d'arrondi s'accumulent, et après 100 000 additions la variable `i` vaut encore un peu moins que 1000 (999.9999999992356), si bien que la boucle fait un tour de plus.

Anecdote 2. Le 25 février 1991, pendant la Guerre du Golfe, une batterie américaine de missiles Patriot, à Dharan (Arabie Saoudite), a échoué dans l'interception d'un missile Scud irakien. Le Scud a frappé un baraquement de l'armée américaine et a tué 28 soldats. La commission d'enquête a conclu à un calcul incorrect du temps de parcours, dû à un problème d'arrondi. Les nombres étaient représentés en virgule fixe sur 24 bits. Le temps était compté par l'horloge interne du système en dixième de seconde. Malheureusement, 1/10 n'a pas d'écriture finie dans le système binaire : 1/10 = 0,1 (dans le système décimal) = 0,0001100110011001100110011... (dans le système binaire). L'ordinateur de bord arrondissait 1/10 à 24 chiffres, d'où une petite erreur dans le décompte du temps pour chaque 1/10 de seconde. Au moment de l'attaque, la batterie de missile Patriot était allumée depuis environ 100 heures, ce qui a entraîné une accumulation des erreurs d'arrondi de 0,34 s. Pendant ce temps, un missile Scud parcourt environ 500 m, ce qui explique que le Patriot soit passé à côté de sa cible.

Exercice 2 Transformer les nombres suivants de la base 10 vers la base 2 :

1. $0{,}5625_{10}$
2. $0{,}15_{10}$
3. $12{,}9_{10}$

Problèmes d'arrondi en Python

Malgré la norme IEEE 754, les arrondis créent des surprises en Python. Le problème fondamental est qu'un nombre décimal fini en base 10 peut avoir un développement infini en base 2.

Exemple 1 :

```
>>> 0.1 + 0.2
0.30000000000000004
>>> 0.1 + 0.2 == 0.3
False
```

Pourquoi ? Car 0,1 n'a pas de représentation finie en binaire. Il s'agit de l'approximation la plus proche, qui n'est pas exactement 0,1.

Solutions pratiques :

Avec `round()` :

```
>>> round(0.1 + 0.2, 10)
0.3
>>> round(0.1 + 0.2, 10) == 0.3
True
```

Avec `math.isclose()` :

```
import math
>>> math.isclose(0.1 + 0.2, 0.3)
True
```

Cette fonction vérifie l'égalité avec une tolérance relative et absolue.

Avec le module `decimal` :

```
from decimal import Decimal
>>> Decimal('0.1') + Decimal('0.2') == Decimal('0.3')
True
```

Valeurs spéciales

La norme IEEE 754 réserve des configurations particulières pour certaines valeurs :

L'infini :

```
>>> float('inf')
inf
>>> float('-inf')
-inf
>>> float('inf') + 1
inf
```

NaN (Not a Number) :

```
>>> float('nan')
nan
>>> float('nan') == float('nan')
False
>>> import math
>>> math.isnan(float('nan'))
True
```

Zéro signé :

```
>>> 0.0
0.0
>>> -0.0
-0.0
>>> 0.0 == -0.0
True
```

Simple et double précision

La norme IEEE 754 définit plusieurs formats :

Simple précision (32 bits) :

— 1 bit de signe

— 8 bits d'exposant

— 23 bits de mantisse

— Environ 7 chiffres décimaux de précision

— Représente des nombres entre $\pm 1{,}4 \times 10^{-45}$ et $\pm 3{,}4 \times 10^{38}$

Double précision (64 bits) :

— 1 bit de signe

— 11 bits d'exposant

— 52 bits de mantisse

— Environ 15 chiffres décimaux de précision

— Représente des nombres entre $\pm 5 \times 10^{-324}$ et $\pm 1{,}8 \times 10^{308}$

En Python, le type `float` utilise toujours la double précision (64 bits). Pour les autres langages, vérifiez la documentation.

La norme IEEE 754

La norme IEEE 754 définit la façon de coder un nombre réel. Cette norme se propose de coder, par exemple, le nombre sur 32 bits et définit trois composantes :

— le signe est représenté par un seul bit, le bit de poids fort

— l'exposant est codé sur les 8 bits consécutifs au signe

— la mantisse (les bits situés après la virgule) sur les 23 bits restants

Étapes de conversion

Pour convertir un nombre décimal en IEEE 754 (32 bits), voici la méthode pas à pas :

1. **Déterminer le signe :** bit 0 si positif, bit 1 si négatif.
2. **Convertir la valeur absolue en binaire :** exprimer le nombre sous forme $1,xxxxxx_2 \times 2^e$ (notation normalisée).
3. **Calculer l'exposant biaisé :** ajouter 127 à l'exposant effectif e (cela donne E).
4. **Extraire la mantisse :** prendre la partie fractionnaire après le 1, et la compléter sur 23 bits.
5. **Assembler les bits :** signe (1 bit) + exposant biaisé (8 bits) + mantisse (23 bits).

Réserves sur les exposants

Certaines conditions sont à respecter pour les exposants :

— L'exposant 00000000 est réservé : il sert à coder le zéro (+0 et −0) et des nombres très proches de zéro, dits dénormalisés.

— L'exposant 11111111 est réservé : avec une mantisse nulle, il code l'infini ($+\infty$ ou $-\infty$) ; avec une mantisse non nulle, il code NaN, ce qui signifie « Not a number », pour signaler un résultat indéfini.

— Il faut rajouter 127 (01111111) à l'exposant pour une conversion de décimal vers un nombre réel binaire. Notre première intuition serait de dire que l'exposant correspond simplement au « e » de $1, \ldots \times 2^e$. Mais si tel était le cas, comment ferions-nous pour représenter les exposants négatifs ? Aucun bit pour le signe de l'exposant n'a été prévu dans la norme IEEE 754 ; une autre solution a été choisie : pour le format simple précision (codage sur 32 bits), 8 bits sont consacrés à l'exposant, il est donc possible de représenter 256 valeurs, nous allons pouvoir représenter des exposants compris entre -126_{10} et $+127_{10}$ (les valeurs −127 et 128 sont réservées comme indiqué ci-dessus). Pour avoir des valeurs uniquement positives, on procède alors à un décalage en ajoutant systématiquement 127 à la valeur de l'exposant effectif.

La formule d'expression des nombres réels est ainsi la suivante :

$$(-1)^S \times (1 + F) \times 2^{E-127}$$

— S est le bit de signe et l'on comprend alors pourquoi 0 correspond à un nombre positif : $(-1)^0 = 1$,

— E est l'exposant codé : l'exposant effectif vaut $e = E - 127$ (on a ajouté 127 à e pour obtenir E),

— F est la partie fractionnaire.

Exemple détaillé

Traduisons en binaire, en utilisant la norme IEEE 754, le nombre −6,625.

1. **Signe :** négatif, donc bit 1.
2. **Valeur absolue en binaire :** $6{,}625_{10} = 110{,}101_2$.
3. **Notation normalisée :** $110{,}101_2 = 1{,}10101_2 \times 2^2$ (on décale la virgule de 2 rangs).
4. **Exposant biaisé :** $E = 2 + 127 = 129_{10} = 10000001_2$.
5. **Mantisse (23 bits) :** On prend la partie fractionnaire 10101 et on complète avec des zéros : 10101000000000000000000
6. **Résultat final :** 1 10000001 10101000000000000000000.

Exercice 3 Donner la représentation flottante, en simple précision, des nombres suivants :

1. 128
2. −32,75
3. 18,125
4. 0,0625

Exercice 4 Pour chaque situation, expliquez quel est le problème de précision et proposez une solution :

1. Un programme de gestion bancaire doit calculer le total de 100 transactions de 0,01 euro. Pourquoi obtient-on 1,0000000000000007 au lieu de 1,00 ?
2. Vous comparez deux nombres flottants issus de calculs différents (ex : 0.1 + 0.2 vs 0.3). La comparaison échoue. Comment faire ?
3. Un programme scientifique accumule de petites erreurs d'arrondi. Devrait-on utiliser la simple ou la double précision ? Pourquoi ?

Exercice 5 Convertissez en IEEE 754 (32 bits) le nombre $21{,}375_{10}$. Détaillez chaque étape :

1. Conversion en binaire.
2. Notation normalisée.
3. Calcul de l'exposant biaisé.
4. Extraction de la mantisse.
5. Assemblage final en binaire.

Exercice 6 Lancez dans Python les commandes suivantes et expliquez les résultats :

```python
# Expérience 1
x = 0.1 + 0.2 + 0.3
y = 0.6
print(x == y)
print(f"{x} {y}")

# Expérience 2
import sys
print(sys.float_info)

# Expérience 3
a = 10**20
b = a + 1.0
c = b - a
print(c)  # Pourquoi n'obtient-on pas 1.0?

# Expérience 4
from decimal import Decimal
d1 = Decimal('0.1') + Decimal('0.2')
d2 = Decimal('0.3')
print(d1 == d2)
```