REPRÉSENTATION DES NOMBRES EN MACHINE

PROBLÉMATIQUE

Dans la mémoire d'un ordinateur, les données sont représentées sous forme de séquences de 0 et de 1.

Par conséquent, toute information mémorisée par l'ordinateur *doit* pouvoir être codée en séquences de 0 et de 1, en particulier les nombres que l'on sera amené à manipuler.

Cependant, un ordinateur ne possède qu'un nombre limité de circuits, donc d'informations. En particulier, la mémoire d'un ordinateur ne peut contenir qu'une quantité finie de nombres. Or, l'ensemble $\mathbb{N}$ des nombres entiers naturels, pour ne citer que cet exemple, est infini : il existe donc des entiers qui ne peuvent pas être codés en machine.

Ainsi, au moment de manipuler des données numériques avec un ordinateur, deux aspects doivent être pris en compte :

1. la représentation des nombres manipulés sous forme de séquences de 0 et de 1 ;
2. les limites de la représentation, par une quantité finie d'informations, d'une quantité de données susceptible d'être infinie.

1 Principes et limites de la représentation binaire des entiers

1.1 Décomposition en base 2 d'un entier naturel

Proposition

Soit $b \in \mathbb{N} \setminus \{0,1\}$.
Pour tout $N \in \mathbb{N}$, il existe $n_0 \in \mathbb{N}$ et des entiers x_0, x_1, ... , x_k, ... , x_{n_0}, tous compris entre 0 et $b-1$, tels que :

$$N = x_{n_0} b^{n_0} + \cdots + x_k b^k + \cdots + x_1 b + x_0.$$

Remarques :

1. Si on impose, dans l'énoncé précédent, que le coefficient x_{n_0} est non nul, alors l'entier n_0 et les coefficients x_0, ... , x_{n_0} sont uniques.
2. L'écriture précédente s'appelle écriture de N en base b.
3. Lorsque $b = 10$, l'écriture précédente s'obtient très facilement.
   Par exemple, l'écriture en base 10 du nombre 271 est : $271 = 2 \times 10^2 + 7 \times 10^1 + 1 \times 10^0$.
4. Le cas particulier de l'écriture en base 2 nous intéresse tout naturellement : dans ce cas, les coefficients x_0, ... , x_{n_0} évoqués dans la proposition précédente appartiennent tous à $\{0,1\}$.
   Ainsi, l'écriture en base 2 permet la représentation de tout entier naturel sous forme de séquence de *bits*.

REPRÉSENTATION BINAIRE D'UN NOMBRE ENTIER

On appelle *représentation binaire d'un nombre entier naturel* N son écriture en base 2, c'est-à-dire la donnée d'un entier naturel n_0 et d'une séquence de bits $(x_0, x_1, \cdots, x_k, \cdots, x_{n_0})$ tels que :

$$N = x_{n_0} \times 2^{n_0} + \cdots + x_k \times 2^k + \cdots + x_1 \times 2 + x_0.$$

NOTATION

L'égalité précédente s'écrit symboliquement sous la forme : $N = (x_{n_0} \cdots x_1 x_0)_2$, ou $N = \underline{x_{n_0} \cdots x_1 x_0}$, voire encore $N = \overline{x_{n_0} \cdots x_1 x_0}$.

Exemples :

1. *Représentation binaire de 271*
   Puisque $271 = 256+8+4+2+1 = 1\times2^8+0\times2^7+0\times2^6+0\times2^5+0\times2^4+1\times2^3+1\times2^2+1\times2^1+1\times2^0$, en reprenant le symbolisme précédent, on peut écrire : $271 = 1\,0000\,1111_2$.
2. *Représentation décimale de 110010_2*
   Le nombre considéré n'est autre que : $0 \times 2^0 + 1 \times 2^1 + 0 \times 2^2 + 0 \times 2^3 + 1 \times 2^4 + 1 \times 2^5 = 50$.

1.2 Addition en binaire

Commençons par quelques considérations simples : $\begin{cases} 0_2 + 0_2 = 0_2 \\ 1_2 + 0_2 = 1_2 \\ 0_2 + 1_2 = 1_2 \\ 1_2 + 1_2 = 10_2 \end{cases}$

On en déduit rapidement que l'addition en binaire s'effectuera sur le même "modèle" que l'addition en écriture décimale, à ceci près que la *retenue* se produit dès qu'on arrive à 2 (au lieu de 10).

Exemple :

Le calcul de $274 + 53$ s'écrit en binaire :

274	$\longrightarrow$	1	1	1	1	0	1	1	1
+ 53	$\longrightarrow$	0	0	1	1	0	1	0	1

1.3 Autour de la taille d'un nombre écrit en binaire

- Taille d'un nombre codé en binaire

  On remarque rapidement, à l'issue de calculs sur quelques exemples, que la "taille" (nombre de chiffres utilisés) de la représentation binaire d'un entier naturel est supérieure à celle de son écriture décimale.
  - Précisément, étant donné un entier naturel n non nul, il existe un (unique) entier p tel que $2^p \leq n < 2^{p+1}$: l'écriture binaire de n nécessite alors $p+1$ bits.

    On peut exprimer p en fonction de n : par la stricte croissance de la fonction ln, on peut déduire de l'encadrement précédent que $p\ln(2) \leq \ln(n) < (p+1)\ln(2)$, donc, comme $\ln(2) > 0$, $p \leq \dfrac{\ln(n)}{\ln(2)} < p+1$: p est donc le plus grand entier inférieur ou égal à $\dfrac{\ln(n)}{\ln(2)}$.
  - De la même manière, on peut montrer que le nombre de chiffres sollicités dans l'écriture décimale de n est égal au plus grand entier plus petit que $\dfrac{\ln(n)}{\ln(10)}$ augmenté de 1.

  Par exemple, pour coder le nombre 1573 en binaire, on a besoin de 12 bits (alors que l'écriture décimale nécessite 4 chiffres) : le codage binaire ne nécessite que deux symboles, mais le nombre de chiffres binaires utilisés pour coder un entier donné est plus important que le nombre de chiffres que contient ce nombre écrit en numération décimale.

  Les calculs précédents montrent que, pour une valeur de n significativement grande, la représentation binaire d'un entier nécessite environ $\dfrac{\ln(10)}{\ln(2)}$ (c'est-à-dire environ $3,3$) fois plus de chiffres que son écriture décimale.

- Dépassement de capacité

  Rappelons que, si les données sont stockées dans la mémoire d'un ordinateur sous forme de séquences de bits, c'est parce que ces informations sont traitées dans des circuits ne pouvant avoir que deux états (*« le courant passe »* et *« le courant ne passe pas »*).

  Dans la mémoire des ordinateurs, ces circuits sont souvent groupés par huit : les octets. On utilise souvent des nombres exprimés en notation binaire sur un, deux, quatre ou huit octets, soit 8, 16, 32 ou 64 bits, ce qui permet de représenter les nombres :
  - de 0 à $1111\,1111_2 = 255$ sur un octet ;
  - de 0 à $1111\,1111\,1111\,1111_2 = 65\,535$ sur deux octets ;
  - de 0 à $1111\,1111\,1111\,1111\,1111\,1111\,1111\,1111_2 = 4\,294\,967\,295$ sur quatre octets ;
  - de 0 à 18 446 744 073 709 551 615 sur huit octets.

  Ainsi, on ne code pas une infinité d'entiers... alors que l'ensemble $\mathbb{N}$ des entiers naturels est infini !

  Evidemment, cela ne va pas sans poser de problèmes. Par exemple, si on travaille avec une machine manipulant des mots codés sur un octet, le calcul de la somme des nombres 247 et 53 donne 300... qui ne peut pas être codé sur un octet ! Comme la représentation de 300 en binaire est $1\,0010\,1100$, la machine ne retient que l'octet $0010\,1100$, si bien qu'elle affichera : $247 + 53 = 44$.

  On parle alors de *dépassement de capacité* (*overflow* en anglais). Sur certains ordinateurs, les calculs continuent. Sur d'autres, une erreur est signalée, d'une façon différente d'un constructeur à l'autre.

Remarque

N'oublions pas, en outre, que les nombres entiers que l'on manipule peuvent être négatifs (ce qui peut être utile si l'on veut effectuer des soustractions) : il s'agit, à présent, de "compléter" la technique de codage en binaire pour tenir compte du signe des nombres manipulés.

1.4 Codage en machine d'un entier relatif

- Une première idée de codage des entiers relatifs
  Le codage d'un entier relatif en machine doit être un prolongement du codage précédent.
  Une idée simple peut consister à "réserver" un bit pour coder le signe de l'entier considéré (0 pour +, 1 pour −, par exemple).
  Par exemple, dans le cas d'une machine manipulant des mots de 8 bits, en utilisant 1 bit pour le signe, et les 7 restants pour coder la valeur absolue de l'entier considéré, on peut représenter tous les entiers relatifs compris entre $-111\,1111_2 = -127$ (qui serait codé sous la forme 1111 1111) et $+111\,1111_2 = 127$ (qui, lui, serait représenté par 0111 1111).
  Cela dit, cette façon de coder les entiers relatifs présente plusieurs inconvénients, parmi lesquels :
  - l'existence de deux zéros : dans l'exemple précédent, 1000 0000 et 0000 0000 désignent le même nombre ;
  - les régles d'addition définies pour les entiers naturels ne se prolongent pas naturellement aux entiers relatifs : on peut essayer, par exemple, le calcul de $81 + (-81)$ avec des mots d'un octet...

  Pour éviter ces problèmes, on utilise une autre convention pour coder les entiers relatifs : la *représentation en complément à 2*.

- Méthode du complément à 2
  Restons sur l'exemple de mots de 8 bits : dans ce cas, on peut coder $2^8 = 256$ nombres : si l'on ne codait que des entiers naturels, on pourrait représenter tous les entiers compris entre 0 et $2^8 - 1$.
  On va s'attacher à proposer une méthode permettant de coder tous les entiers compris entre -2^7 et $2^7 - 1$ (il y en a bien 256).
  Pour cela, on convient que :
  - si un entier n est compris entre 0 et $2^7 - 1$, il est codé par sa représentation binaire "classique", décrite précédemment ;
  - si un entier n est compris entre -2^7 et -1, il est représenté par le codage en binaire de l'entier *positif* $2^8 + n$, qui est compris entre $2^8 - 2^7 = 2^7$ et $2^8 - 1$.

  Par cette méthode, l'entier -1 est représenté par le codage de 255 en binaire, ce qui donne $1111\,1111_2$, -2 est représenté par le codage de 254 en binaire , et ainsi de suite, jusqu'à -128, qui est représenté par le codage de 128 en binaire.
  On réalise alors que, en suivant cette convention :
  - le premier bit qui apparaît dans le codage d'un entier relatif vaut 0 si ce nombre est positif, 1 si ce nombre est négatif : voilà une façon simple de détecter rapidement le signe d'un entier ;
  - les règles d'addition des entiers relatifs prolongent celles des entiers naturels.

- Dépassement de capacité
  De nouveau, le codage des entiers ne peut être réalisé que dans les limites définies par la taille des mots manipulés par la machine considérée : les erreurs signalées avec les entiers naturels peuvent se reproduire.
  Une addition de deux nombres positifs ou négatifs peut entraîner un dépassement de capacité... mais celui-ci peut être détecté en regardant le signe du résultat par rapport au signe des deux opérandes. En effet :
  - si la somme de deux nombres positifs dépasse la capacité de codage, le résultat sera un nombre négatif ;
  - si la somme de deux nombres négatifs dépasse la capacité de codage, le résultat sera un nombre positif.

1.5 Avec Python...

En Python les nombres entiers (relatifs) sont représentés par des objets qui appartiennent au type `int` (pour *integer*) :

```
>>> type (5)
<class 'int'>
```

Les valeurs possibles pour un objet de ce type ne sont limitées que par les capacités de l'ordinateur. Les calculs sont réalisés de manière exacte ou n'aboutissent pas. Dans les précédentes versions de Python [1] (c'est-à-dire Python 2.x) ce type correspond à la notion de « long integer ». Les entiers de type `int` étaient alors codés sur un nombre déterminé de bits : lorsque la capacité des entiers machine (32 ou 64 bits) était dépassée, les nombres étaient suivis du marqueur L, qui explicitait qu'on passait dans le type appelé `long`.

En Python 3.x, les types `int` et `long` sont fusionnés, et les entiers sont toujours de taille illimitée [2]. Le marqueur L n'est plus utilisé... mais certains calculs peuvent ne pas aboutir...

1. ou dans d'autres langages
2. Un codage un peu sophistiqué des nombres permet de "dépasser" les problèmes de dépassements de capacité "classiques"

2 Codage en machine d'un nombre réel

PRINCIPE DU CODAGE D'UN RÉEL EN MACHINE

La manipulation de nombres réels avec un nombre fini de chiffres n'a rien de nouveau pour nous.

Par exemple, nous n'explicitons jamais le *réel* π avec toutes ses décimales (et pour cause : il y en a une infinité !), mais nous nous contentons de l'approcher par le nombre *décimal* $3,14$, ou $3,15$, ou $3,1416$, ou autre, selon la précision souhaitée.

Cette façon d'approcher un nombre réel à l'aide d'un nombre « à virgule », ayant un nombre fini de chiffres après la virgule, de façon arbitrairement précise, s'appelle *approximation décimale d'un nombre réel*.

Plus précisément :

- $3,14$ est l'approximation décimale de π par défaut au centième près ;
- $3,15$ est l'approximation décimale de π par excès au centième près ;
- $3,1416$ est l'approximation décimale de π par excès au dix millième près.

Pour le codage d'un nombre réel dans un ordinateur, on reprend l'idée d'approximation d'un nombre réel par un autre nombre... mais en utilisant la base 2, et en essayant de contourner certains défauts de cette représentation.

2.1 Nombres à virgule, virgule flottante

- NOMBRES DÉCIMAUX, NOMBRE À VIRGULES EN BINAIRE

  On appelle *nombre décimal* tout réel qui peut s'écrire sous la forme $\dfrac{n}{10^m}$, où n est un entier relatif, et m est un entier naturel..

  Par exemple, $x = \dfrac{719}{100}$ est un nombre décimal, que l'on écrit $7,19$, cette dernière écriture portant le nom de *notation décimale* de x.

  En notation décimale, les chiffres situés à gauche de la virgule représentent des entiers, des dizaines, des centaines, etc. et ceux placés à droite de la virgule des dizièmes, des centièmes, des millièmes, etc.

  On peut, par analogie, écrire un nombre à virgule en notation binaire en utilisant les puissances négatives de 2. Par exemple :

$$\begin{aligned} 11,0101_2 &= 1\times 2^1 + 1\times 2^0 + 0\times 2^{-1} + 1\times 2^{-2} + 0\times 2^{-3} + 1\times 2^{-4} \\ &= 2 + 1 + 0 + 0,25 + 0 + 0,0625 \\ &= 3,3125 \end{aligned}$$

- PROBLÈME DE TAILLE

  S'il s'agit d'"économiser" le nombre de chiffres à utiliser pour fournir une approximation décimale d'un réel donné (ou, plus simplement, pour écrire un nombre décimal donné), on réalise rapidement que cette écriture n'est pas adaptée à tous les ordres de grandeurs :

  ↪ un « très grand » nombre, tel que le nombre d'Avogadro N_A, (environ $6,022\times 10^{23}$) va mobiliser une grande quantité de données pour être écrit : l'écriture décimale d'une approximation de N_A à l'entier près nécessite au moins 24 chiffres ;

  ↪ le problème se pose dans les mêmes proportions pour la représentation des quantités « très petites » : par exemple, la charge élémentaire d'un électron, qui vaut, en Coulombs, environ $-1,602\times 10^{-19}$, se représente, sous forme décimale, à l'aide d'une approximation nécessitant au moins 20 chiffres.

  La "simple" écriture d'un nombre à virgule en notation binaire présente les mêmes inconvénients pour la représentation des « très grands » et des « très petits » nombres.

  Pour pallier ce défaut de l'écriture décimale, on est amené à utiliser l'*écriture scientifique d'un nombre décimal*, également appelée *écriture à virgule flottante*, que l'on va adapter aux nombres écrits en binaire.

- ÉCRITURE À VIRGULE FLOTTANTE

  ↪ *Notation scientifique d'un nombre décimal*

  La notation scientifique d'un nombre décimal x consiste à préciser son signe s ($+$ ou $-$), et à déterminer un nombre décimal m (appelé *mantisse de* x) situé dans l'intervalle $[1,10[$ et un entier relatif n (appelé *exposant de* x) permettant d'écrire :

$$x = s\,m\,10^n.$$

  Par exemple, l'écriture scientifique du nombre décimal $173,95$ est $+1,7395\times 10^2$.

  La notation scientifique des nombres décimaux est la représentation généralement utilisée pour l'affichage des valeurs avec une calculatrice : elle permet d'économiser le nombre de chiffres à utiliser pour représenter un nombre décimal, ou une approximation décimale d'un nombre réel.

  Les nombres représentés sous cette forme sont appelés *nombres à virgule flottante* : la virgule de la mantisse peut être « déplacée » par une modification de l'exposant.

A présent, ne perdons pas de vue que l'on travaille avec des ordinateurs, qui codent toutes les informations en binaire : cette écriture doit donc être adaptée en base 2.

↪ *Nombre binaire à virgule flottante*

L'adaptation de la notation scientifique en base 2 est définie dans la norme *IEEE 754* (*Institute of Electrical and Electronics Engineers*).

Dans cette démarche, un nombre réel x est représenté sous la forme $x = sm2^n$, où s est le *signe* de x, m est un réel de $[1, 2[$ (encore appelé *mantisse*), et n est un entier relatif (encore appelé *exposant*).

Par exemple, le nombre $x = 3,3125 = 11,0101_2$ s'écrit aussi $x = +1,65625 \times 2^1$, ou encore $x = +1,10101_2 \times 2^1$: il a pour mantisse $1,65625$ et pour exposant 1.

2.2 Virgule flottante en machine

Par exemple, quand on utilise 64 bits pour représenter un nombre à virgule, la norme IEEE 754 convient qu'on utilise 1 bit pour le signe, 11 bits pour l'exposant et 52 bits pour la mantisse :

- le signe + est représenté par 0 et le signe − par 1 ;
- l'exposant n est un entier relatif compris entre -1022 et 1023, représenté par l'entier naturel $n + 1023$, qui est compris entre 1 et 2046 ;
- la mantisse m est un nombre binaire à virgule dans l'intervalle $[1, 2[$, comprenant 52 chiffres après la virgule.

Remarques :

1. L'exposant étant codé à l'aide de 11 bits, il devrait pouvoir prendre $2^{11} = 2\,048$ valeurs, alors que la description précédente n'en propose que 2 046 : les exposants codés par 0 et 2 047 sont en fait réservés pour des situations exceptionnelles :
   - un nombre ayant un exposant codé à 2 047 est assimilé à $+\infty$ ou $-\infty$;
   - un nombre ayant un exposant codé par 0 est dit *dénormalisés* : il s'agit d'un nombre proche de 0 dont la mantisse n'obéit pas à la règle implicite décrite ci-dessous...
2. Comme la mantisse appartient à $[1, 2[$, elle a toujours un seul chiffre avant la virgule et ce chiffre est toujours un 1 : il est donc inutile de le représenter. De cette façon, on utilise les 52 bits pour représenter les 52 chiffres après la virgule.

   Ainsi, la précision réelle de la mantisse est de 53 bits.
3. Un premier problème se pose : on remarque rapidement qu'il n'est pas possible de représenter 0 sous la forme $s\,m\,2^n$, où s est un signe, $m \in [1, 2[$ et $n \in \mathbb{Z}$. Il est pourtant bien commode de pouvoir le coder en machine, car on est amené à l'utiliser fréquemment.

   Par convention, on décide qu'un nombre vaut zéro si et seulement si tous les bits de son exposant et de sa mantisse valent 0. Il reste un choix pour le bit de signe : il y a donc un zéro positif et un zéro négatif dans l'ensemble des nombres à virgule flottante.

Exemple :

Déterminer le codage du nombre décimal $-243,25$ dans la norme IEEE 754.

2.3 Limites de la représentation des réels en machine

Une infinité de nombre à représenter avec une quantité finie de données

On peut montrer que tout nombre réel non nul x peut s'écrire sous la forme $x = s\,m\,2^n$, où s décrit le signe de x, n est un entier relatif, et m est un *réel* appartenant à $[1, 2[$.

Cependant, les quantités n et m, quand elles sont codées dans un ordinateur, ne peuvent prendre qu'un nombre fini de valeurs : tout réel ne peut donc pas être codé en machine.

Ce problème peut apparaître y compris si l'on décide de ne manipuler que des nombres déjà codés en machine : le résultat d'un calcul faisant intervenir deux de ces nombres peut "sortir" des capacités de représentation. Il y a essentiellement deux types de problèmes que cette limitation peut engendrer...

- Dépassement de capacité

  Les nombres à virgule flottante étant représentés sur un nombre donné de bits, il existe forcément un nombre maximal représentable dans ce format.

  Plus précisément, un nombre à virgule flottante en base 2 n'est plus représentable sur 64 bits si sa représentation demande :

  → un exposant supérieur à 1 023 ;

  → ou un exposant égal à 1 023 et une mantisse supérieure à la plus grande mantisse représentable sur 52 bits, c'est-à-dire 1 (implicite) suivi de 52 chiffres binaires tous égaux à 1 après la virgule.

  Un calcul dépassant cette limite donne lieu à un dépassement de capacité. En Python, ce dépassement de capacité peut retourner à l'utilisateur la valeur `inf` (pour ∞) ou simplement un message d'erreur comportant la mention `Overflow`.

Une situation similaire – qui ne se présentait cependant pas pour les entiers – se produit lorsque l'on veut représenter un nombre à virgule flottante trop proche de 0, c'est à dire un nombre dont l'exposant est inférieur à $-1\,022$. Si le résultat d'un calcul amène un tel nombre, alors il peut être arrondi à 0, ou produire une erreur. On parle alors de *dépassement de capacités par valeurs inférieures* (*underflow*).

Exemple :
Observer les réactions de Python lorsqu'on lui demande de retourner le flottant 2^{1024}, le flottant $2^{1023} \times 2$ ou 2^{-1075}.

- Le résultat d'un calcul faisant intervenir deux nombres à virgule flottante peut donner résultat non représentable dans la norme précédente.
  Même sans effectuer de calcul, la plupart des nombres décimaux ne sont pas codables dans ce format avec 64 bits.
  Ainsi, par exemple, le nombre $0,4$ admet $0,011001100110011\cdots_2$ pour développement en base 2 : il s'agit d'un développement infoni périodique.
  La représentation en virgule flottante sera donc forcément une valeur approchée de ce nombre. Par défaut, la norme IEEE 754 impose que les nombres à virgule soient arrondis à la valeur représentable la plus proche. Dans le cas de $0,4$, la valeur approchée choisie est :

$$0,0110011001100110011001100110011001100110011001100110011010_2 \approx 0,4000000000000000022204460492503.$$

Exemple :
Quand on demande à Pyhton d'"imprimer" $1.94 - 1$ ou $1.94 - 0.1$, il se passe des choses étranges...
Moralité
Les nombres binaires en virgule flottante ne permettent pas de représenter exactement la plupart des nombres décimaux, plus précisément tous ceux qui ne s'écrivent pas sous la forme $\frac{k}{2^n}$. Ainsi, des nombres qui, "habituellement", ne posent pas de problème dans les calculs en mathématiques deviennent ainsi une source d'erreurs multiples.

3 Type `int` et type `float`

On a vu que les nombres entiers, dans Python, appartiennent au type `int`.

En Python (et dans beaucoup d'autres langages) les objets qui représentent les nombres en virgule flottante sont de type `float`.

Un nombre entier peut être vu, par Python, dans ces deux types. Par exemple, 2 est du type `int`, tandis que 2. est du type `float`.

Au-delà des modes de représentations décrits précédemment, ces deux types ont des spécificités dont on doit tenir compte au moment de choisir le point de vue que l'on va adopter sur un nombre :

- certaines opérations sont propres à un type : la division euclidienne, notamment, n'existe que pour les entiers ;
- les résultats des calculs effectués avec le type `int` sont exacts, ce qui n'est pas le cas avec le type `float` ;
- la manipulation de grandeurs physiques est plus pertinente avec le type `float` (qui "tient compte" de la notion d'ordre de grandeur) ;
- si l'on doit effectuer des calculs sur des données dont les ordres de grandeur sont très différents, les flottants ne sont pas appropriés ;
- si l'on doit effectuer des tests du type *« le nombre a est-il égal au nombre b ?*, les flottants ne sont pas appropriés.

Exemples :

1. Comparer le résultat des deux séquences d'instructions suivantes :

```
>>> a=9
>>> a=a*a
>>> a=a*a
```

```
>>> a=9.
>>> a=a*a
>>> a=a*a
```

2. Si on affecte la valeur 10^{-323} à une variable, est-elle, selon Python, nulle ? et si on divise cette même variable par 10 ?
3. Additionner des nombres d'ordres de grandeurs trop différents entraîne une *perte de chiffres significatifs.*
   Par exemple, si on écrit `>>> a=9` puis `>>> b=1e-16`, `>>> b` est-il nul ? `>>> a+b` est-il distinct de `>>> a` ?
4. Dans un registre proche, observer le résultat que Python propose pour $\left(1.23 + 3 \times 10^{-3}\right) + 4 \times 10^{-4}$ et $1.23 + \left(3 \times 10^{-3} + 4 \times 10^{-4}\right)$.