Exercices_nombres_flottants

June 14, 2019

Exercice nř1 : Python et le plus petit flottant normalisé

L'ensemble des flottants normalisés représentables avec Python sont définis par l'ensemble F(2, 53, -1022, 1023) tenant sur 8 octets. Déterminer les valeurs du plus petit et du plus grand flottant positif non nul. Le plus grand peut-être trouvé à l'aide d'une fonction python.

```
In [ ]: import sys
        sys.float_info
```

1. Donner la valeur du plus grand et plus petit flottant
2. Pour le calcul du plus petit flottant positif non nul représentatble avec Python, un élève propose l'algorithme ci_dessous, rédiger le code python correspondant à l'algorithme 1 et relever la valeur affichée par le programme.

```
// Algorithme 1
réel x <-- 1
tant que (x > 0) faire
    precedent = x
    x         = x / 2
fin tant que
affiche precedent
```

3. Une des spécification de la norme des flottants est de ne pas renvoyer de message d'erreur dans le cas d'un overflow, les calculs peuvent se poursuivre. La norme prévoit des nombres spéciaux (+inf, inf, NaN), en guise d'avertisseur dans le cas d'un dépassement, conservant le type float. Il est alors possible d'écrire `infini = float(inf)`. En vous inspirant de l'algorithme 1 écrire un algorithme 2 permettant d'avoir une valeur approchée du plus grand flottant normalisé positif.

Correction

1. Le plus grand : $max = 1.7976931348623157e + 308$, le plus petit : $min = 2.2250738585072014e - 308$
2. Code Python

```
In [ ]: # Plus petit flottant
        x = 1
        while x > 0:
            precedent = x
            x = x / 2
        print(precedent)
```

```
In [ ]: # Valeur approchée du plus grand flottant
        x = 1
        while x < float('inf'):
            precedent = x
            x = x * 2.0
        print(precedent)
```

Exercice nř2 : Virgule fixe

De même que le système de numération décimal, le système de numération binaire est un système pondéré. Comme nous l'avons vu pour la représentation des nombres entiers, chaque bit d'un nombre binaire fractionnaire a un poids en fonction de son rang (numération de position), la virgule servant de séparateur entre la partie entière et la partie fractionnaire du nombre binaire. La position de la virgule est donc figée :

$$1010,1010_{(2)} = 2^3 + 2 + 2^{-1} + 2^{-3} = 8 + 2 + 0,5 + 0,125 = 10,625_{(10)}$$

Dans le sens inverse donnons un exemple avec 0,82 codé sur 6-bits après la virgule:

$$\begin{array}{ll} 0,82 \times 2 = 1,64 & \text{je pose } \mathbf{1} \\ 0,64 \times 2 = 1,28 & \text{je pose } \mathbf{1} \\ 0,28 \times 2 = 0,56 & \text{je pose } \mathbf{0} \\ 0,56 \times 2 = 1,12 & \text{je pose } \mathbf{1} \\ 0,12 \times 2 = 0,24 & \text{je pose } \mathbf{0} \\ 0,24 \times 2 = 0,48 & \text{je pose } \mathbf{0} \end{array}$$

Au final $0,82_{(10)}$ s'écrit $0,110100_{(2)}$

À l'aide des exemples précédents convertir en

- notation décimale
  - $111,001101$
  - $0,000111011$
- binaire virgule fixe avec arrondi **par défaut** et **au plus près**:
  - 0,54 avec la partie fractionnaire exprimée sur 4-bits fractionnaires
  - 0,9 avec la partie fractionnaire exprimée sur 3-bits fractionnaires
  - 0,24 avec la partie fractionnaire exprimée sur 4-bits fractionnaires

Correction (partielle) :

$111,001101 = 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-3} + 1 \times 2^{-4} + 1 \times 2^{-6} = 7.203125$

Pour la représentation binaire en virgule fixe nous pouvons utiliser le code donné en ANNEXE de l'activité :

```
In [1]: def dec2bin(nombre, precision = 23):
            assert nombre < 1
            binaire = "0."
            for i in range(precision):
                nombre  = nombre * 2
                bit     = int(nombre)
```

```
            nombre  = nombre - bit
            binaire = binaire + str(bit)
        return binaire
```

La conversion se fait avec quelques bits supplémentaires pour pouvoir arrondir suiivant la convention vu en cours

```
In [2]: print(dec2bin(0.54, 7))
        print(dec2bin(0.9, 6))
        print(dec2bin(0.24, 7))
```

```
0.1000101
0.111001
0.0011110
```

Arrondi au plus près :

- 0.1000101 donne sur 4-bits 0.1001
- 0.111001 donne sur 3-bits 0.111
- 0.0011110 donne sur 4-bits 0.0100

Exercice 3 : Influence de l'ordre des opérations sur le résultats

Écrire une suite d'instructions pour faire la somme des 500000 premiers inverses d'entier dans l'ordre croissant puis décroissant. Que remarquez-vous ?

$$\text{ordre croissant : } \frac{1}{1} + \frac{1}{2} + \frac{1}{...} + \frac{1}{500000}$$

Correction

```
In [ ]: somme = 0
        for i in range(1, 500001):
            somme += 1 / i
        print(somme)
```

```
In [ ]: somme = 0
        for i in range(500000, 0, -1):
            somme += 1 / i
        print(somme)
```