Aix*Marseille université
Initiative d'excellence
Année 2018-19

Diplôme Inter-Universitaire
Enseigner l'informatique au lycée

Semaine 2 : TP 1

Codage des nombres

1 Représentation des entiers

1.1 Conversions de base en Python

1.1.1 Affichage d'un entier dans une base

En Python (comme pour tout les langages de programmation), il est très facile d'obtenir la représentation d'un entier en base 2, 10 ou 16.

Pour cela, on peut utiliser la méthode format sur les chaînes de caractères. Cette méthode permet de transformer une chaîne de caractères en remplaçant chaque champs de remplacement (sous-chaîne entourée d'accolades {}) par un des arguments de la fonction potentiellement transformé pour correspondre à un format donné.

Par exemple, le code suivant :

```
"Décimal: {0:d}; Hexadécimal: {0:x}; Binaire: {0:b}".format(42)
```

permet d'obtenir la représentation de 42 en décimal, hexadécimal et binaire. Le nombre avant les deux points dans les accolades correspond au numéro du paramètre de l'appel de la méthode *format* alors que la chaîne de caractères après les deux points spécifie le format voulu : `d` pour décimal, `b` pour binaire et `h` pour hexadécimal.

On peut aussi utiliser les fonctions `bin` et `hex` pour obtenir la représentation d'un entier en binaire ou hexadécimal.

1.1.2 Construire un entier à partir d'une représentation dans une base

Il est aussi possible en Python de créer un entier à partir d'une représentation dans une base quelconque grâce au constructeur *int*. Un appel `int(string_base_representation, base_value)` permet d'obtenir l'entier correspondant à la représentation `string_base_representation` en base `base_value`.

1.1.3 Exercice

- Calculer la représentation en base 2 du nombre x dont la représentation en base 16 est : $FEDCBA9876543210|_{16}$.
- Calculer la représentation en base 16 du nombre y dont la représentation en base 2 est : $1111000010101111011 1|_2$.
- Que remarquez-vous entre les deux représentations des nombres x et y ?
- Est-ce que vous pouvez en déduire une méthode facile pour passer d'une représentation en base 2 à une représentation en base 16 et inversement ?

1.2 Entiers relatifs en binaire

1.2.1 Complément à deux

Il serait naturel de représenter les entiers relatifs en utilisant un bit pour le signe et les autres pour la valeur absolue. Ainsi, avec des mots de 16 bits, on utiliserait 1 bit pour le signe et 15 bits pour la valeur absolue.

Néanmoins, la plupart des langages appliquent une autre méthode appelée complément à deux. Le complément à deux ne s'applique qu'à des nombres ayant tous la même longueur : codés sur N bits, les nombres

binaires peuvent représenter les valeurs entières de -2^{N-1} à $2^{N-1}-1$. Si on utilise des mots de 16 bits, on peut donc représenter les entiers relatifs compris entre -32768 et 32767. Sur 16 bits, on représente un entier relatif r positif ou nul comme l'entier naturel $x = r$ et un entier relatif r strictement négatif comme l'entier naturel $x = r + 2^{16} = r + 65536$.

Ainsi, les entiers naturels de 0 à 32 767 correspondent aux entiers relatifs positifs ou nuls et les entiers naturels de 32 768 à 65 535 représentent les entiers relatifs strictement négatifs.

Il existe une manière facile de calculer la représentation de l'opposé d'un nombre représenté en complément à deux qui consiste à :

— inverser tous les bits ($1 \to 0$ et $0 \to 1$)

— puis ajouter 1 au résultat en ignorant le bit supplémentaire si on dépasse le nombre de bits fixé

En python, on peut calculer le complément à deux d'un entier à l'aide de la fonction suivante :

```
def two_complement_representation(value: int, nb_bits: int):
    """Return the two's complement of value with nb_bits bits"""
    assert -(2 ** (nb_bits-1)) <= value < 2 ** (nb_bits-1), \
        "Value " + str(value) + " should have been between " + \
        str(-(2 ** (nb_bits-1))) + \
        " and " + str(2 ** (nb_bits-1)-1)
    return ("{0:{fill}" + str(nb_bits) + "b}").format(value % (2 ** nb_bits), fill='0')
```

1.2.2 Exercice

— Calculer (en Python avec la fonction ci-dessus) la représentation en complément à deux sur 16 bits de : -3, -15, -256 et -1024. Vérifier à la main (à l'aide de la méthode facile pour calculer l'opposé) que la fonction donne bien le bon résultat.

— Comment détecter rapidement à partir de la représentation à 2 d'un nombre si celui-ci est positif ou négatif ?

— Additionner en base 2 les nombres correspondant à la représentation en complément à deux de -15 et +16. Quel résultat obtenez-vous ?

— Pourquoi, d'après vous, utilise-t-on le complément à deux dans les ordinateurs ?

— Pourquoi la fonction `two_complement_representation` calcule bien la représentation en complément à deux de `value` sur `nb_bits` bits ?

2 Représentation des réels

2.1 Codage en virgule flottante : IEEE754

Dans la plupart des ordinateurs actuels, les nombres réels sont représentés avec un codage à virgule flottante. L'idée est de retenir un nombre fixe de **chiffres significatifs** quelque soit la valeur du nombre.

2.1.1 Principe du codage

On rappelle qu'un nombre réel peut être représenté en binaire grace à la formule suivante (potentiellement avec un nombre infini de chiffre) :

$$c_n c_{n-1} \dots c_1 c_0, c_{-1} c_{-2} c_{-3} \dots |_2 = \sum_{i=-\infty}^{i=n} (c_i 2^i)$$

Puisque les seuls chiffres en base 2 sont 0 ou 1, cela signifie qu'on représente un nombre comme une somme (finie ou infinie) de puissance de deux.

Pour représenter un nombre en virgule flottante, on commence par écrire le nombre en notation scientifique avec signe, mantisse et exposant, en binaire, $(-1)^s \times m \times 2^e$:

$25,90625 \Rightarrow 11001,11101 \Rightarrow (-1)^0 \times 1,100111101 \times 2^4$
On choisit ensuite une variante : simple précision, double précision (ou plus).

	mantisse	exposant	signe	taille totale
Simple précision	23 bits	8 bits	1 bit	32 bits
Double précision	52 bits	11 bits	1 bit	64 bits

2.1.2 Détails du codage

- **Signe :** 0 pour positif et 1 pour négatif
- **Mantisse :** on n'a besoin que de coder les chiffres après la virgule car le chiffre avant la virgule est forcément 1. Pour `1,101001` on code donc `101001`.
- **Exposant :** On le code avec l'entier $E = e + 2^{n_e-1} - 1$ où n_e est le nombre de bits utilisés pour coder l'exposant. En simple précision, on a $E = e + 127$. Selon les valeurs de l'exposant décalé E, le nombre final peut appartenir à l'une ou l'autre des catégories suivantes. En simple précision, la norme IEEE 754 définit les catégories suivantes :
  - Normalisés : $0 < E < 255$
  - Dénormalisés : $E = 0$ et $m \neq 0 \Rightarrow 0,m \times 2^{-126} \times (-1)^s$
  - Infinis : $E = 255$ et $m = 0 \Rightarrow (-1)^s \infty$
  - Indéfini : $E = 255$ et $m \neq 0 \Rightarrow$ NaN
  - Zéro : $E = 0$ et $m = 0$

Prenons nombre $x = -6,625|_{10}$. En binaire sa valeur absolue est $110,101|_2$ car $-6,625|_{10} = 2^2 + 2^1 + 2^{-1} + 2^{-3}$. En normalisant la mantisse on obtient : $1,10101.2^2$. Cela nous donne donc une mantisse suivante sur 23 bits : `1010 1000 0000 0000 0000 000`. L'exposant décalé est donc $2 + 127|_{10} = 10000001|_2$. Le signe est 1 car le nombre est négatif.

On obtient donc :

```
1 10000001 10101000000000000000000
```

2.1.3 Exercice

- Écrivez $z = -2^{127}$ en IEEE754 simple précision.
- Quel est le nombre qui le suit immédiatement (premier nombre supérieur à z représentable en IEEE754 simple précision) ?
- Quel est le plus petit/le plus grand nombre positif normalisé ?
- Quel est le plus petit/le plus grand nombre positif dénormalisé ?

3 Limites de la représentation des nombres en machine

Le nombre de codes différents est fini, alors que les objets à représenter sont en nombre infini... Quel que soit le codage choisi (virgule fixe ou flottante), la quantité d'information, donc de chiffres significatifs, est finie. Si le nombre à représenter en a trop, il faudra en sacrifier certains. Des nombres comme π ou e ne peuvent pas être représentés exactement en virgule fixe ou flottante.

3.1 Problématiques de l'encodage des nombres flottant en binaire

3.1.1 Accéder à la représentation flottante en Python

Pour accéder à la représentation en mémoire d'un entier, on le met sous la forme d'une séquence d'octets (*bytes*). La fonction `pack` du module struct permet d'effectuer cette transformation. On peut utiliser la fonction

ci-dessous pour obtenir la représentation d'un nombre en norme IEEE 754 en simple précision. on note en particulier l'usage de " !f" pour fixer une représentation gros-boutiste et la simple precision (on peut mettre `!d` pour obtenir la double précision).

```
from struct import pack

def float_to_bin(f):
    """Returns a string corresponding to the binary representation
     of the given float in simple precision with spaces between the
     sign, the exponent and the significand"""
    bytes = pack('!f', f)
    binary_representation = ""
    for byte in bytes:
        binary_representation += "{:{fill}8b}".format(byte, fill='0')
    return binary_representation
```

On peut réaliser l'opération inverse avec `unpack` et la fonction ci-dessous.

```
from struct import unpack

def bin_to_float(float_representation):
    """Returns a float equal to the value corresponding to
     the given binary representation in float simple precision"""
    bytes = int(float_representation, 2).to_bytes(4, "big")
    return unpack("!f", bytes)[0]
```

3.1.2 Exercice

- Vérifier que la représentation de $z = -2^{127}$ en IEEE754 simple précision que vous avez calculée à l'exercice précédent est correcte.
- Calculer la représentation de $0,1$. Qu'observez-vous ?
- Effectuez les manipulations suivantes dans la console Python :

```
>>> 0.1 + 0.1 + 0.1 == 0.3
>>> a = 1 - 0.2 - 0.2 - 0.2 - 0.2 - 0.2
>>> b = 1 - (0.2 + 0.2 + 0.2 + 0.2 + 0.2)
>>> a == b
```

- Comment expliquez-vous les résultats des tests ?
- Qu'est-ce que ces problèmes impliquent pour la comparaisons de flottants ?

3.2 Calcul des nombre harmonique

3.2.1 Définition

En mathématiques, le n-ième nombre harmonique est la somme des inverses des n premiers entiers naturels non nuls :

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} = \sum_{k=1}^{n} \frac{1}{k}$$

La valeur du millionième nombre harmonique est approximativement 14.392726722865723.

3.2.2 Exercice

— Calculez la valeur du millionième nombre harmonique en sommant les inverses `1/i` pour `i` variant de 1 à `n`.
— Calculez la valeur du millionième nombre harmonique puis pour `i` variant de `n` à 1.
— Que constatez-vous ? Est-ce qu'une des deux approches semble meilleure empiriquement ? Avez-vous une explication ?

3.3 Phénomènes de compensation

Les phénomènes de compensation se produisent lorsque l'on tente d'effectuer des soustractions de valeurs très voisines. Ils peuvent aboutir à des pertes importantes de précision.

3.3.1 Calcul de e^{-10}

Vous allez calculer la valeur de $e^{-10} \approx 4,539992976.10^{-5}$ en utilisant la suite suivante :

$$u_n = \sum_{k=0}^{n} (-1)^k \frac{10^k}{k!}$$

Cette suite converge vers e^{-10} quand n tend vers l'infini.
Vous allez aussi utiliser la suite suivante calculer la valeur de e^{-10} (de manière indirecte) :

$$v_n = \sum_{k=0}^{n} \frac{10^k}{k!}$$

Cette suite converge vers e^{10} (l'inverse de e^{-10}) quand n tend vers l'infini.

3.3.2 Exercice

— Écrivez une fonction qui calcule et renvoie le terme de rang u_n.
— En déduire une valeur approchée de e^{-10} à partir de u_{100}.
— Écrivez une fonction qui calcule et renvoie le terme de rang v_n.
— En déduire une valeur approchée de e^{-10} à partir de v_{100}.
— Comment expliquez-vous les différences constatées ?

3.4 Programme mystère

3.4.1 Petit programme inconnu

On considère le programme suivant (écrit en pseudo-code) :

```
Sortie: un entier i et un flottant b

a := 1.0;
i = 0;
tant que (((a + 1.0) - a) == 1.0 )
    faire a := 2.0 * a
b := 1.0;
tant que (((a + b) - a) != b )
    faire b := b + 1
```

3.4.2 Exercice

— Écrivez une fonction Python réalisant le calcul ci-dessus et renvoyant le couple (i, b).
— Pourquoi cette fonction s'arrête-t-elle ?
— À quoi correspondent les valeurs renvoyées ?
— Que pouvez-vous en déduire sur la représentation des flottant en python ?
— Réessayer la même fonction en utilisant cette fois des `Decimal` (package decimal), des `float16`, des `float32` et des `float64` (package numpy). Quels résultats obtenez-vous ?