Introduction

La mémoire d'un ordinateur étant finie, il est impossible de représenter en informatique tous les nombres réels. On a déjà étudié les représentations des entiers signés et non signés en ***Python*** *qui constituent le type* `int`*. On a vu que la seule limite était la taille de la mémoire. Pour des nombres comme* π, $\sqrt{2}$ *ou des nombres rationnels dont la partie après la virgule est infinie (il en existe dans toutes les bases), un tel choix n'est pas possible.*
Dans ce cours, on introduit la représentation approchée des réels non entiers sous forme de flottants, type `float` *en* ***Python****, et on présente certaines de leurs caractéristiques qui méritent une attention particulière dans les calculs.*

Sources :

- *« Manuel de première NSI » de Balabonski, Filliâtre, N'Guyen chez Ellipses.*
- *Fiches pédagogiques du MOOC NSI sur la plateforme* `https://www.fun-mooc.fr/fr/`.
- *Cours de mon collègue Pierre Duclosson.*

Tous les codes sont à compléter dans ce notebook Capytale :

`https://capytale2.ac-paris.fr/web/c/78a9-386046`

1 Représentation binaire

1.1 Rappels sur la représentation binaire des entiers

Exercice 1

1. Convertir en base dix l'entier dont la représentation binaire est $\overline{101101}^2$.

   .....

2. Convertir en base deux l'entier dont la représentation en base dix est 29.

   .....

   .....

   .....

   .....

3. Dans le carnet Capytale, compléter les corps des fonctions suivantes en respectant leur spécification et validant leurs tests unitaires :

   a. `entier_vers_binaire(n)` renvoie la représentation en base deux de l'entier positif `n`;

   b. `binaire_vers_entier(tab_bits)` renvoie l'entier positif dont la représentation en base deux est donnée par le tableau de bits `tab_bits`.

c. `bourrage_zero_gauche(tab_bits, nb_bits)` complète le tableau de bits `tab_bits` avec des 0 à gauche pour obtenir un tableau de taille `nb_bits`.

1.2 Représentation binaire d'un nombre décimal

Définition 1

Soit x un réel positif.

- La **partie entière** de x est le plus grand entier noté $\lfloor x \rfloor$ tel que $\lfloor x \rfloor \leqslant x$.

  La propriété suivante est vérifiée par la partie entière de x : $\lfloor x \rfloor \leqslant x < \lfloor x \rfloor + 1$.
- La **partie fractionnaire** de x est la différence entre x et sa partie entière : $x - \lfloor x \rfloor$.
- Dans la représentation d'un nombre, on sépare **partie entière** et **partie fractionnaire** par une virgule ou un point.

En **Python**, la partie entière d'une valeur `x >= 0` est donnée par `int(x)` et sa partie fractionnaire par `x - int(x)`.

Exercice 2

1. Déterminer la partie entière et la partie décimale du nombre $3,14$ en base dix.

.....

2. On considère le nombre de représentation $\overline{110,1011}^2$ en base deux. Déterminer la valeur de ce nombre en base dix.

.....

.....

.....

.....

Méthode *Conversion en binaire d'une partie fractionnaire décimale*

Soit x un nombre décimal tel que $0 < x < 1$. Pour obtenir la liste de bits (ou **développement en base deux**) la partie fractionnaire de x en binaire, on initialise une liste de bits à une liste vide et on applique l'algorithme suivant :

- **Étape 1 :** On calcule $y = 2 \times x$ et on insère sa partie entière $\lfloor y \rfloor$ (0 ou 1 car $0 < x < 1$) à la fin de la liste de bits.
- **Étape 2 :** On remplace x par la partie fractionnaire de $y = 2 \times x$. Si cette nouvelle valeur est nulle, on s'arrête, sinon on reprend à l'étape 1.

Donnons un exemple avec $0,3$.

x	$y = 2x$	Liste de bits
$0,3$	$0,6$	0
$\boxed{0,6}$	$1,2$	1
$0,2$	$0,4$	0
$0,4$	$0,8$	0
$0,8$	$1,6$	1
$\boxed{0,6}$	$1,2$	1
$0,2$	$0,4$	0
$0,4$	$0,8$	0
$0,8$	$1,6$	1

On peut faire deux remarques importantes :

- La partie fractionnaire de 0,3 a un développement en base dix fini mais un développement en base deux infini. Le développement de 0,3 en base deux peut s'écrire :

$$\overline{0,0\underbrace{1001}_{\text{période}}\ldots}^{2}$$

- Le développement infini de 0,3 en base deux est périodique. Plus généralement tout nombre rationnel (fraction d'entiers) a une partie fractionnaire de développement fini ou infini périodique dans une base quelconque. Les nombres irrationnels comme $\sqrt{2}$ ou π ont une partie fractionnaire de développement infini non périodique dans n'importe quelle base.

Exercice 3

1. Déterminer le développement en base deux du décimal 0,1.

.....

.....

.....

.....

.....

.....

.....

.....

2. Dans le carnet Capytale, compléter les corps des fonctions suivantes en respectant leur spécification et validant leurs tests unitaires :

a. `partie_frac_vers_binaire(partie_frac, nb_bits)` renvoie la représentation en base deux sur `nb_bits` de la partie fractionnaire `partie_frac`;

b. `binaire_vers_partie_frac(tab_bits)` renvoie une partie fractionnaire décimale représentée sur `nb_bits` par `tab_bits`.

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

3. `binaire_vers_partie_frac(partie_frac_vers_binaire(0.1, 10))` renvoie-t-il `0.1`?
`binaire_vers_partie_frac(partie_frac_vers_binaire(0.25, 10))` renvoie-t-il `0.25`?
Comment pouvez-vous expliquer ces évaluations de l'interpréteur **Python**?

```
>>> (0.1 + 0.1 + 0.1) == 0.3
False
>>> (0.1 + 0.1) == 0.2
True
```

2 Les flottants

On étudie différents formats de représentation des réels qui ne sont pas des entiers.

2.1 Représentation en virgule fixe

Exercice 4 *Représentation en virgule fixe*

Supposons que pour représenter en binaire un réel x on utilise le format suivant sur 32 bits :

- un bit de signe 0 pour positif et 1 pour négatif;
- puis quinze bits pour représenter la partie entière de la valeur absolue de x;
- enfin seize bits pour représenter le développement (éventuellement tronqué) de la partie fractionnaire de la valeur absolue de x .

1. Représenter le nombre décimal $-14,625$ dans ce format.

.....

.....

.....

.....

.....

.....

.....

.....

2. Citer un grand et un petit nombre qui ne sont pas représentables dans ce format.

.....

.....

.....

2.2 Représentation en virgule flottante

Méthode *Virgule flottante*

Pour remédier aux limites d'un format à *virgule fixe*, on peut utiliser un format dit à *virgule flottante*, où la position de la virgule n'est pas spécifiée a priori, mais où la représentation binaire du nombre (tant sa partie entière que sa partie fractionnaire) s'accompagne d'une information qui indique où positionner la virgule. Ce format s'inspire de la **notation scientifique** des nombres décimaux, qui consiste à exprimer tous les nombres (en base 10) sous la forme :

$$\boxed{(-1)^s \times m \times 10^e}$$

- s marque le **signe** (1 pour négatif et 0 pour positif) ;
- la **mantisse** m est un nombre dans l'intervalle $[1\,;\,10[$;
- l'**exposant** e un entier relatif

Exemples de **notation scientifique** :

- $-42,625 = -4,625 \times 10^{1}$;
- $0,00042625 = 4,625 \times 10^{-4}$.

Définition 2 *Norme IEEE 754*

La norme IEEE 754 fixe la **représentation binaire approchée** des réels sous un format à *virgule flottante*. Un réel x est représenté de façon approchée par un **flottant** $f(x)$ qui est un nombre dont le développement de la partie fractionnaire en base deux est fini.

Par analogie avec la notation scientifique, un flottant est de la forme :

$$\boxed{(-1)^s \times m \times 2^{n-d}}$$

- ☞ le *signe* s est codé sur 1 bit ;
- ☞ l'*exposant* $n-d$ est un entier relatif mais il est codé comme un entier non signé n avec décalage de d.
- ☞ la *mantisse* $m = 1,f$ est un réel tel que $1 \leqslant m < 2$ dont seuls les bits de la partie fractionnaire f sont codés, la partie entière 1 étant implicite.

La norme définit un format simple précision sur 32 bits et un format double sur 64 bits :

Format double (décalage $d = 2^{8-1} - 1 = 127$)

signe	exposant	fraction
1 bit	8 bits	23 bit

Format double (décalage $d = 2^{11-1} - 1 = 1023$)

signe	exposant	fraction
1 bit	11 bits	52 bit

Hors programme :

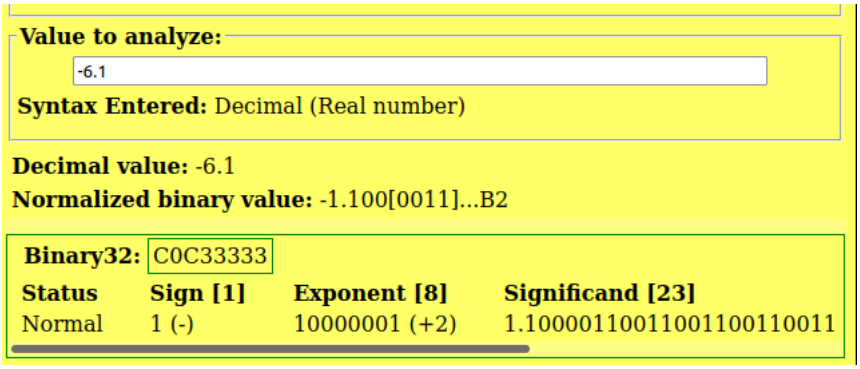
*De plus la norme définit des valeurs spéciales pour les valeurs extrêmes de l'exposant décalé (*0 *et* $2^8 - 1$ *ou* $2^{11} - 1$*) : deux zéros, des infinis positif et négatif et une valeur* `Not A Number` *pour des valeurs indéterminées dans des calculs comme* $\frac{0}{0}$ *ou* $\frac{\infty}{\infty}$.
Pour un flottant dénormalisé le bit implicite de 1 *pour la mantisse n'est pas défini et la partie codée représente exactement la mantisse.*

Signe	*Exposant*	*Mantisse (partie fractionnaire)*	*Valeur spéciale*
0	*0*	0	*+0*
1	*0*	0	*−0*
quelconque	*0*	*non nulle*	*flottant dénormalisé*
0	*valeur maximale*	*nulle*	*infini positif*
1	*valeur maximale*	*nulle*	*infini négatif*
quelconque	*valeur maximale*	*non nulle*	*NAN valeur non définie*

Enfin la norme définit plusieurs méthodes d'arrondi.

Méthode

On donne ci-dessous un exemple de représentation au format IEE 754 simple sur 32 bits du nombre décimal $-6,1$ obtenu avec le convertisseur https://babbage.cs.qc.cuny.edu/IEEE-754/.



Voici les étapes pour obtenir cette représentation :

- **Étape 1 :** On code le signe de la valeur sur le premier bit, ici 1 puisque le nombre est négatif.
- **Étape 2 :** On représente en binaire la partie entière et la partie fractionnaire de la valeur absolue $6,1$ comme dans la partie **1.** du cours :

$$6,1 = \overline{110,000\ \underbrace{1100}_{\text{période}}\ \ldots}^{2}$$

- **Étape 3 :normalisation** On détermine l'exposant réel e qui est le grand entier relatif e tel que $2^e \leqslant 6,1$. Ici $= 2$, on en déduit l'exposant décalé $n = e + d = 2 + 127 = 129$ de représentation non signée sur 8 bits égale à $\overline{10000001}^2$.

$$-6,1 = \overline{(-1) \times 2^2 \times 1,10000\ \underbrace{1100}_{\text{période}}\ \ldots}^{2}$$

- **Étape 4 :** Pour coder la partie fractionnaire de la mantisse, on concatène les représentations binaires de la partie entière (sans le premier bit implicitement égal à 1) et de la partie fractionnaire en ne conservant que 23 bits significatifs.

Exercice 5

1. Commentez l'affichage ci-dessous de `0.1` en dans une console **Python** avec 40 chiffres après la virgule.

```
>>> f"{0.1:.40f}"
'0.1000000000000000055511151231257827021182'
```

.....

.....

2. Déterminer un nombre décimal représenté par chacune des représentations de flottants au format

simple précision sur 32 bits.

a. `'01000000101010000000000000000000'`

b. `'00000000100000000000000000000000'`

c. `'10111111001000000000000000000000'`

.....

.....

.....

.....

.....

.....

.....

.....

.....

3. Représenter au format simple précision sur 32 bits le nombre décimal $-5,1$.

.....

.....

.....

.....

.....

.....

Exercice 6

Ouvrir le carnet Capytale.

1. Compléter le corps de la fonction `calcul_exposant(d)` qui prend en paramètre un décimal positif `d` non nul et renvoie le plus petit entier relatif `e` tel que $2^e \leqslant d$. Vérifier les tests unitaires proposés.

.....

.....

.....

.....

.....

.....

.....

.....

2. Compléter le corps de la fonction `float32_vers_decimal(tab_bits)` qui prend en paramètre un tableau de bits représentant un décimal au format simple précision sur 32 bits et renvoie un décimal correspondant. Vérifier les tests unitaires proposés.

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

3. Commenter la fonction `decimal_vers_float32(d)` qui prend en paramètre un décimal et renvoie sa représentation comme tableau de bits au format simple précision sur 32 bits.

4. Donner un exemple de nombre décimal trop grand pour être représenté au format simple précision sur 32 bits.

.....

.....

2.3 Le type `float` en Python, précautions à prendre

Méthode

En **Python** les nombres réels sont représentés de façon approchée au format double précisions 64 bits de la norme IEE 754 dans le type `float`.
On désigne ces représentations approchées sous le terme de **flottants**.
Voici une liste non exhaustive de principes à retenir lorsqu'on manipule des flottants en **Python** :

1. Les **flottants** sont des *représentations approchées.*

   Il ne faut pas attendre une réponse exacte d'un test d'égalité entre deux flottants.

   Par exemple des réels, comme 0,3 ont un développement fini en base dix mais infini en base deux

et ne sont donc pas représentés de façon exacte.

```
>>> (0.1 + 0.1 + 0.1) == 0.3
False
>>> (0.1 * 3) == 0.3
False
>>> (0.1 * 4) == 0.4
True
```

2.  En particulier, **il ne faut jamais baser un test sur une comparaison de flottants**.

   Par exemple la boucle ci-dessous ne se termine pas :

```
k = 0
while k != 1:
    k = k + 0.1
```

   Il faudrait plutôt l'écrire ainsi :

```
k = 0
while k < 1:
    k = k + 0.1
```

3.  Si on a besoin de tester l'égalité de deux flottants, il faut plutôt utiliser une inégalité et comparer la valeur absolue de leur différence à un seuil d'erreur.

   Par exemple la fonction `isclose(a, b, rel_tol=1e-09)` du module `math` renvoie `True` si l'erreur relative `abs(a - b)/max(abs(a), abs(b)` est inférieure à `rel_tol`.

```
>>> a, b, c = sqrt(2), sqrt(3), sqrt(5)
>>> (a ** 2 + b ** 2) == c ** 2
False
>>> import math
>>> math.isclose(a ** 2 + b ** 2, c ** 2)
True
```

4.  Les flottants ne sont pas répartis de façon uniforme. En particulier l'écart avec le successeur du flottant x est supérieur à 1 dès que $x \geqslant 2^{53}$.

```
>>> float(2 ** 53 + 1) == float(2 ** 53) # pas d'égalité en float
True
>>> (2 ** 53 + 1) == (2 ** 53) # égalité en int
False
```

5.  Les opérations entre flottants conduisent à de nombreuses erreurs d'approximation :

   - *cancellation* lorsqu'on soustrait deux flottants de même ordre de grandeur : effacement du premier bit implicite de mantisse, décalage de la virgule vers la droite et des bits de précisions

de la mantisse vers la gauche sauf que les places libérées à droite dans les bits de mantisse ne peuvent être remplies autrement que par des 0 d'où la perte de précision ;

- *absorption* lorsqu'on additionne deux flottants d'ordres de grandeur très différents. En effet, il faut dénormaliser le plus petit en déplaçant la virgule vers la gauche au même niveau que le plus grand ce qui entraîne une perte des bits de précision à droite de la mantisse.

```
>>> 1 + 2 ** (-52) == 1 # pas d'absorption
False
>>> 2 + 2 ** (-52) == 2 # absorption
True
```

6.  Les calculs sur les flottants étant approchés, les règles habituelles des opérations algébriques comme l'addition et la multiplication ne sont pas garanties.

```
>>> 10.3 - 10 - 0.3 == 0
False
>>> (10 + 0.3 - 10 - 0.3) == 0
False
>>> (10 - 10 + 0.3 - 0.3) == 0
True
>>> (10.3 - (10 + 0.3)) == 0
True
```

7.  Les flottants sont codés sur un nombre fini de bits et sont donc en nombre fini. Il existe un plus grand et un plus petit flottant positif. Attention donc aux `Overflow` ou `Underflow`.

```
>>> float(2 ** 1024)
.............
OverflowError: int too large to convert to float
>>> 2 ** (-1023-51)
5e-324
>>> 2 ** (-1023-52)
0.0
```

Les détails de l'implémentation du type `float` peuvent être découverts avec `sys.float_info`.

```
>>> import sys
>>> sys.float_info
sys.float_info(max=1.7976931348623157e+308, max_exp=1024, max_10_exp
    =308, min=2.2250738585072014e-308, min_exp=-1021, min_10_exp
    =-307, dig=15, mant_dig=53, epsilon=2.220446049250313e-16, radix
    =2, rounds=1)
```

8. Quelques exemples de problèmes causés par les erreurs d'approximation dans les calculs sur les flottants :

   http://math.univ-lyon1.fr/irem/Formation_ISN/formation_representation_information/nombre/codage_numeriques_des_nombres.html.