Données de base 2
Les nombres flottants et les textes

I Représentation des nombres flottants

Rappels

Nous avons déjà vu qu'en Python, il se passe des choses étranges dès que l'on manipule les nombres à virgule.

```
0.1 + 0.2 == 0.3
```

```
False
```

Ceci résulte du fait que tous les nombres décimaux ne sont pas représentables par des « nombres flottants » et qu'il y a nécessairement des approximation.

I.1 Approximation des nombres décimaux

Le résultat de la séquence suivante est troublant :

```
x = 0.1
x
```

```
0.1
```

```
y = 0.3
y
```

```
0.3
```

```
x + x + x == y
```

```
False
```

Ceci vient du fait que Python « ment » lors de l'affichage des nombres à virgule.

```
format(0.1, ".60f")
```

```
'0.1000000000000000055511151231257827021181583404541015625000000'
```

```
format(0.1 + 0.1 + 0.1, ".60f")
```

```
'0.3000000000000000444089209850062616169452667236328125000000000'
```

```
format(0.3, ".60f")
```

```
'0.2999999999999999888977697537484345957636833190917968750000000'
```

Astuce

En Python, il existe le module `Decimal` qui permet de représenter exactement, sans approximation les nombres décimaux. On peut alors se rendre compte de l'approximation qui est faite par les nombres flottants.

```
from decimal import Decimal

Decimal.from_float(0.1)
```

```
Decimal('0.1000000000000000055511151231257827021181583404541015625')
```

```
x = Decimal.from_float(0.1)
y = Decimal("0.1")

x - y
```

```
Decimal('5.55111512312578270211815834 0E-18')
```

Info

Certains nombres décimaux peuvent être représentés exactement par des nombre flottants. C'est par exemple le cas de 0,25.

```
format(0.25, ".60f")
```

```
'0.250000000000000000000000000000000000000000000000000000000000'
```

```
0.25 + 0.25 + 0.25 + 0.25 == 1
```

```
True
```

Attention

D'une manière générale, on évite, en Python et dans n'importe quel langage de programmation, de tester l'égalité de deux flottants.

Astuce

Pour vérifier que deux nombres flottants sont proches, on utilisera plutôt

```
tolerance = 1e-15
x = 0.1 + 0.1 + 0.1
y = 0.3

abs(x - y) < tolerance
```

```
True
```

Pour comprendre l'approximation qui est faite par les nombres flottants, il faut comprendre comment ils sont représentés en machine.

I.2 Représentation des nombres décimaux : les nombres à virgule flottante

I.2.1 Rappels sur la notation scientifique

L'encodage des nombres flottants est inspiré de l'écriture scientifique des nombres décimaux qui se compose :

- d'un signe (+ ou −) ;
- d'un nombre décimal m appelé *mantisse* et compris dans l'intervalle $[1\,;10[$;
- et d'un entier relatif n appelé *exposant*.

Par exemple, avec cette notation,

$$\begin{aligned} 2156 &\quad \text{s'écrit} \quad +2{,}156 \times 10^{3} \\ -398\,879{,}62 &\quad \text{s'écrit} \quad -3{,}988\,796\,2 \times 10^{5} \\ 0{,}000\,142 &\quad \text{s'écrit} \quad +1{,}42 \times 10^{-4} \\ 1{,}34 &\quad \text{s'écrit} \quad +1{,}34 \times 10^{0} \end{aligned}$$

Ainsi, de manière générale, l'écriture scientifique d'un nombre décimal est de la forme

$$\pm m \times 10^n.$$

On remarque que 0 n'a pas d'écriture scientifique.

I.2.2 Les nombres flottants simple et double précision

C'est a norme IEEE 754 qui est la plus couramment utilisée pour définir la représentation des nombres flottants [1]

Selon la précision souhaitée, la norme définie un format de données sur 32 bits appelé *simple précision* et un autre sur 64 bits appelé *double précision*. Dans les deux cas, la représentation d'un nombre flottant est similaire à l'écriture scientifique d'un nombre décimal avec un signe, une mantisse et un exposant. De manière générale, un nombre flottant s'écrit

$$\pm m \times 2^{n-d}.$$

Les différences avec l'écriture scientifique sont les suivantes :

- la base choisie n'est plus la base 10 mais la base 2 ;
- la mantisse est donc maintenant dans l'intervalle $[1\,;2[$;
- l'exposant n est décalé d'une valeur d qui dépend de la précision.

Dans le format 32 bits, le bit de poids fort représente le signe : 0 pour + et 1 pour −. Les 8 bits suivants sont utilisés pour stocker l'exposant et les 23 derniers bits sont réservés à la mantisse.

signe	exposant	mantisse
1 bit	8 bits	23 bits

Pour représenter les exposants négatifs et positifs, ce n'est pas la méthode du complément à 2 qui est utilisée. C'est un entier relatif, sur 8 bits, dont les valeurs peuvent aller de 0 à 255 à qui on applique un décalage $d = 127$ pour obtenir des nombres de -127 à 128.

La mantisse étant toujours une nombre de la forme $1,...$, les 23 bits de la mantisse sont utilisés pour représenter les chiffres après la virgule appelée la *fraction*.

Exemple

On cherche le nombre flottant représenté par le mot de 32 bits suivant

1 10000110 10101101100000000000000

1. La norme décrit en fait beaucoup d'autres choses comme par exemple les opérations de calcul sur les nombres flottants.

En double précision, sur 64 bits, les différences sont résumées dans le tableau ci-dessous.

	exposant	mantisse	décalage
32 bit	8 bits	23 bits	127
64 bits	11 bits	52 bits	1023

Exercice 1

Quelle est la plus petite et la plus grande valeur représentable en simple précision ? En double précision ?

Exercices de fin de chapitre

Exercice 2

Donner la représentation binaire de $-10{,}125$ en simple précision.

Exercice 3

Donner la représentation binaire de 0,25 en simple précision.

Exercice 4

Donner la représentation binaire de 0,1 en simple précision.

Exercice 5

Déterminez la représentation au format simple précision d'un tiers en binaire.

II Représentation d'un texte en machine : notion d'encodage

Qui n'a pas déjà été confronté à la lecture d'un texte dont certains caractères (`é`, `è`, `à`, `ç`, ...) sont remplacés par des caractère étranges et rendent le texte peu lisible ?

On peut voir un exemple de cette situation ici : `https://nsi.infobrisson.fr/files/Cours/03_Donnees_base_2/vct.html`.

Ceci vient du fait que plusieurs systèmes de représentation des chaînes de caractères co-existent même si certains tendent à devenir des standards.


Astuce

Puisque l'on sait très bien représenter des nombres en machine, pour représenter des caractères, il suffit de les numéroter ! On appelle ceci un *système d'encodage*.


Attention

Si le système de numérotation choisi n'est pas le même, c'est là que les problèmes apparaissent !

II.1 Le codage ASCII

Dans les années 50, beaucoup d'encodages existaient. Pour tenter de mettre un peu d'ordre et éviter les conversion d'encodage, l'*American National Standard Institute* (ANSI) propose au début des années 60 une norme de codage des caractères appelée ASCII pour *American Standard Code for Information Interchange*. Cette norme définit un jeu de 128 caractères, chaque caractère étant représenté par un octet.

USASCII code chart

b_4	b_3	b_2	b_1	Row ↓ / Column →	0 (0 0 0)	1 (0 0 1)	2 (0 1 0)	3 (0 1 1)	4 (1 0 0)	5 (1 0 1)	6 (1 1 0)	7 (1 1 1)
0	0	0	0	0	NUL	DLE	SP	0	@	P	`	p
0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q
0	0	1	0	2	STX	DC2	"	2	B	R	b	r
0	0	1	1	3	ETX	DC3	#	3	C	S	c	s
0	1	0	0	4	EOT	DC4	$	4	D	T	d	t
0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u
0	1	1	0	6	ACK	SYN	&	6	F	V	f	v
0	1	1	1	7	BEL	ETB	'	7	G	W	g	w
1	0	0	0	8	BS	CAN	(	8	H	X	h	x
1	0	0	1	9	HT	EM	)	9	I	Y	i	y
1	0	1	0	10	LF	SUB	*	:	J	Z	j	z
1	0	1	1	11	VT	ESC	+	;	K	[	k	{
1	1	0	0	12	FF	FS	,	<	L	\	l	|
1	1	0	1	13	CR	GS	-	=	M	]	m	}
1	1	1	0	14	SO	RS	.	>	N	^	n	~
1	1	1	1	15	SI	US	/	?	O	_	o	DEL


python

En Python, il est possible de trouver le caractère ASCII représenté par un entier donné en décimal ou en hexadécimal.

```
chr(\x50)
```

```
P
```

```
chr(\x40)
```

```
@
```

```
chr(65)
```

```
A
```

On peut aussi aller dans l'autre sens et trouver la représentation d'un caractère :

```
ord("a")
```

```
97
```

```
hex(ord("a"))
```

```
'0x61'
```

Info

Pourquoi seulement 128 caractères si on utilise 1 octet, seuls 7 bits suffiraient ? Le bit de poids fort sert de somme de contrôle, on l'appelle le *bit de parité*. Ceci était utilisé pour détecter les erreurs de transmission lorsque les textes étaient transmis sur des réseaux.

Exercice

Exercice 6

Écrire une fonction Python qui prend une chaîne de caractère et retourne le tableau contenant le code ASCII de ses caractères.

II.2 Les normes ISO 8859

Les caractères imprimables de la table ASCII se sont vite avérés insuffisants pour transmettre des textes dans des langues autres que l'anglais. En effet, il manque dans la table ASCII les caractères accentués, les symboles de monnaie, *etc.*

Pour remédier à ce problème l'*Organisation Internationale de Normalisation* (ISO), a proposé la norme ISO 8859. C'est une extension de l'ASCII qui utilise les 8 bits de chaque octet pour représenter des caractères. Malgré un nombre de caractère doublé, cela reste insuffisant pour représenter tous les caractères utilisés rien que dans les langues latines.

Pour représenter le plus de caractères possible, l'ISO 8859 définit plusieurs tables de correspondance appelées *pages* et notée ISO-8859-n où n est le numéro de la page. Bien qu'indépendantes les unes des autres, ces tables ont été conçues pour que les 128 premiers caractères soient compatibles avec ceux de la norme ASCII.

Code ISO	Diminutif	Zone
8859-1	latin-1	Europe occidentale
8859-2	latin-2	Europe centrale ou de l'est
8859-3	latin-3	Europe du sud
8859-4	latin-4	Europe du nord
8859-5		Cyrilique
8859-6		Arabe
8859-7		Grec
8859-8		Hébreu
8859-9	latin-5	Turc, Kurde
8859-10	latin-6	Nordique
8859-11		Thaï
8859-12		(abandonné)
8859-13	latin-7	Balte
8859-14	latin-8	Celtique
8859-15	latin-9	Révision du latin-1 (avec €)
8859-16	latin-10	Europe du sud-est

python

En Python, on peut facilement encoder un texte comme ceci :

```
pangramme = "Dès Noël, où un zéphyr haï me vêt de glaçons würmiens, je dîne
 d'exquis rôtis de boeuf au kir, à l'aÿ d'âge mûr, acætera"
pangramme.encode('iso8859-1')
```

```
b"D\xe8s No\xebl, o\xf9 un z\xe9phyr ha\xef me v\xeat de gla\xe7ons w\xfcrmiens, je d\xee
```

On peut aussi convertir des chaînes entre différents encodages :

```
pangramme.encode('iso8859-1').decode('iso8859-15')
```

II.3 L'Unicode

Les pages de la norme ISO 8859 permettent d'encoder un très grand nombre de caractères, elles ne conviennent pas lorsque l'on cherche à écrire un texte avec un mélange de caractères de différentes tables.

Pour remplacer l'utilisation des pages de codes, l'ISO a définit un jeu universel de caractère appelé *UCS* pour *Universal Character Set*.

À chaque caractère de ce jeu, on associe un nom unique ainsi qu'un numéro, un entier positif en base 10 appelé *point de code*. Il y a aujourd'hui plus de 110 000 caractères recensés dans cette norme qui est conçue pour concevoir, à terme, les caractères de n'importe quelle langue. La capacité maximale de la norme a été fixée à 4 294 967 295 caractères. Par soucis de compatibilité, les 256 premiers points de code sont ceux de la norme ISO-8859-1.

On utilise la notation `U+xxxx` pour désigner les points de code où `xxxx` est la représentation hexadécimal du point de code[2]. Ainsi, `U+006F` représente le caractère « `o` ».

Avec une telle représentation on aurait naïvement besoin de 4 octets. Cependant, dans la grande majorité des cas, seuls 1 octet serait utile et cela représenterait un énorme gâchis et une incompatibilité avec la norme ASCII.

C'est là qu'intervient la norme Unicode. Développé par le consortium du même nom, une organisation à but non lucratif, cette norme définit plusieurs techniques d'encodage pour représenter les points de code de manière plus ou moins économique. Ces encodages sont appelés *Universal Transformation Format*, *UTF*, et déclinés en `UTF-`n où n indique le nombre minimal de bits pour représenter un point de code. Dans la suite nous ne parlerons que d'`UTF-8`.

Info

L'UTF-8 est le format le plus utilisé de nos jours. C'est le format par défaut dans les environement GNU/Linux, dans les protocoles réseau, sur le Web, ... Comme son nom l'indique, il faut simplement 8 bits pour coder le premier caractère. L'UTF-8 est entièrement compatible avec les standard ASCII : les 128 premiers caractères sont représentés sur 1 octet. exactement comme en ASCII.

Fonctionnement de l'encodage UTF-8

Le principe de l'encodage UTF-8 est le suivant :

- Si le bit de poids fort d'un octet est à 0, alors il s'agit d'un caractère ASCII codé sur 7 bits.
- Sinon, les premiers bits de poids fort de l'octet indiquent le nombre d'octets utilisés pour encoder le caractère à l'aide d'un séquence de bits à 1 se terminant par un bit à 0. Par exemple, si le premier octet commence par `110xxxxx`, cela signifie que le caractère est codé par 2 octets. De même, si le premier octet commence par `1110xxxx`, cela signifie que le caractère est codé sur 3 octets.
- Enfin, dans le cas d'un encodage sur k octets, les $k-1$ octets qui suivent le premier octet doivent tous être de la forme `10xxxxxx`, c'est-à-dire commencer par deux bits de poids fort valant `10`.

Attention

Le codage UTF-8 utilise 4 octets au maximum !

2. On ajoute des chiffres lorsque cela est nécessaire.

Le tableau ci-dessous résume le principe de l'encodage UTF-8, avec la plage des caractères représentables selon le nombre d'octets utilisés.

Plage	Suite d'octets (en binaire)	bits utiles
U+0000 à U+007F	0xxxxxxx	7 bits
U+0080 à U+07FF	110xxxxx 10xxxxxx	11 bits
U+0800 à U+FFFF	1110xxxx 10xxxxxx 10xxxxxx	16 bits
U+10000 à U+10FFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx	21 bits


Exemple

Quel est le caractère codé en UTF-8 par « 11000011 10100111 » ?


Exercice

Exercice 7

Quel est le caractère codé en UTF-8 par « 11100010 10001000 10011010 » ?


Astuce

Les conversion UTF-8 sont facilement faisables en Python. On peut utiliser les syntaxes suivantes.

```
"\u00a3"
```

```
'£'
```

Ou dans l'autre sens,

```
"ç".encode("utf-8")
```

```
b'\xc3\xa7'
```

Exercices de fin de chapitre

Voir le TP n°3 ici : https://nsi.infobrisson.fr/files/TP/TP_03/.