Représentation des données : types et valeurs de base

Écriture d'un entier positif en base $b \geq 2$

Attendu : passer d'une base à une autre

Commentaire : les bases 2, 10 et 16 sont privilégiées

Il faut toujours indiquer la base dans laquelle un nombre est exprimé (sauf, par usage et commodité, en base 10) : $1010_2 = 10 = \mathrm{A}_1 6$

Note : la base par défaut dans du code Python est la base 10. Mais, avec certaines notations, il est possible d'utiliser d'autres bases.

```
>>> 10 == 0b1010 == 0xA == 0o12
True
```

Toutes ces représentations correspondent au nombre dix. Peu importe la représentation donnée en entrée, en interne toutes les données sont en binaire.

Algorithme de conversion d'un entier en base $b \geq 2$:

```
Entrée : b la base de numération, n un entier naturel.
Sortie : x0 , x1 ,. . . xp-1 les chiffres de l'écriture de n en base b.

m := n
i := 0
tant que m $ \geq $ b faire
 r := m (mod b)
 m := m ÷ b
 xi := chiffre correspondant à r
 i := i + 1
fin tant que
xi := chiffre correspondant à m
renvoyer x0 , x1, . . . , xi.
```

Passage de la base 2 à 16, et inversement

La base 16 est fréquemment utilisée. Pourquoi ? 16 a le bon goût d'être une puissance de 2 (24) et deux chiffres hexadécimaux permettent de décrire un octet.

On peut aborder le passage de la base 2 à la base 16 (et inversement). Quatre bits correspondent à un chiffre hexadécimal. On peut donc convertir un nombre de la base 2 à la base 16, sans passer par la 10, par paquets de 4 bits. Si le nombre de bits n'est pas un multiple de 4, on peut de toute façon « ajouter » des 0 non significatifs avant le bit de poids fort.

Taille des données

À noter : 1ko c'est 1 000 octets, pas 1 024. Le préfixe kilo correspond toujours à 1000, l'informatique ne fait pas exception (de même pour méga, giga, téra, etc.). Il existe cependant les préfixes kibi (210=1024), mébi (220), gibi (230), tébi (240), etc. qui sont respectivement abbréviés en ki, Mi, Gi, Ti, etc.

Représentation binaire d'un entier relatif

> Attendu : Évaluer le nombre de bits nécessaires à l'écriture en base 2 d'un entier, de la somme ou du produit de deux nombres entiers. Utiliser le complément à 2.
>
> Commentaires : Il s'agit de décrire les tailles courantes des entiers (8, 16, 32 ou 64 bits). Il est possible d'évoquer la représentation des entiers de taille arbitraire de Python.

Taille d'un entier dans l'écriture en base 2

Le fait que l'entier soit relatif ne change finalement pas grand chose. Il faut uniquement prendre en compte un bit en plus afin de stocker le signe.

En utilisant exactement n bits on peut représenter les entiers naturels de 2^{n-1} à $2^n - 1$. Si on veut connaître le nombre de bits pour représenter un entier donné, il faut utiliser la fonction inverse $\log_2$ (mais qui n'est pas présentée en 1è). De ce fait un entier naturel n s'écrit sur $\lfloor \log_2 n \rfloor + 1$ bits.

D'un point de vue plus pratique les entiers naturels représentables sur 8, 16, 32 ou 64 bits sont donc ceux inférieurs à 28=256, $216 = 65\,536$, $232 = 4\,294\,967\,296$, $264 = 18\,446\,744\,073\,709\,551\,616$.

Complément à 2

Pour représenter un nombre signé on pense intuitivement qu'il suffit d'ajouter un bit de signe. Mais cette représentation (appelée *signe-valeur absolue*) ne permet pas d'additionner directement deux nombres.

Par exemple si le bit de poids fort est le bit de signe et qu'on représente les entiers relatifs sur 3 bits et qu'on fait 2 + (-1). Cela donne, en représentation signe-valeur absolue : 101 + 010 = 111. Or 111 en représentation signe-valeur absolue correspond à -3, ce qui est incorrect.

À la place la représentation complément à 2 est généralement préférée. Soit n l'entier relatif à représenter sur p bits (avec $|n| < 2^{n-1}$): - si $n \geq 0$: n est représenté en binaire sur p bits. - sinon : le complément à 2 de $-n$ est représenté sur p bits.

Attention C'est le bit de poids fort qui sert de bit de signe. Il est donc très important de préciser le nombre de bits dans la représentation afin de savoir

quel bit est le bits de poids fort.

Le complément à 2 d'un entier positif N sur p bits est tel que la somme de N et de son complément à 2 soit nulle sur p bits. Il peut se calculer de deux manières : 1. On prend le complément de la représentation binaire de N et on lui ajoute 1 2. $2^p - N$ qu'on représente en binaire sur p bits.

Attention *complément à 2* désigne à la fois l'opération mathématique de conversion et une méthode de représentation des entiers relatifs (qui n'implique pas forcément de calculer un complément à 2 !)

Avec la représentation en complément à 2 sur p bits il est possible de représenter tous les entiers de -2^{p-1} jusqu'à $2^{p-1} - 1$ (représentés respectivement par 10…0 et 01…1).

Exemples

On souhaite représenter 13 sur 5 bits dans la représentation en complément à 2. On a $13 = 1101_2$. Donc dans la représentation en complément à 2 sur 5 bits 13 s'écrit : 01101.

On souhaite représenter -13 sur 5 bits dans la représentation en complément à 2. On a toujours $13 = 1101_2$. Nous devons calculer le complément à 2. Voyons avec les deux méthodes 1. Le complément de 01101_2 est 10010_2, auquel on ajoute 1. On obtient donc 10011_2. La représentation de -13 en complément à 2 est donc 10011. 2. $2^5 - 13 = 32 - 13 = 19$. $19 = 10011_2$. Donc la représentation de -13 en complément à 2 est 10011.

Quel est le nombre entier relatif qui correspond à la représentation en complément à 2 suivante 11001 ?

Il s'agit d'un nombre négatif puisque le bit de poids fort est à 1. 1. On prend le complément et on lui ajoute 1 : 00111_2, ce qui correspond à l'entier 7. L'entier représenté était donc -7 2. En binaire $11001_2 = 25$. $2^5 - 25 = 7$ Donc l'entier représenté était -7.

Si la représentation en complément à 2 est 01001, alors le nombre entier est positif (bit de poids fort à 0). Il suffit donc de convertir le nombre en déciaml pour connaître la valeur de l'entier, ici 9.

Représentation des entiers de taille arbitraire en Python

Dans un langage où les entiers sont de taille fixe (par exemple sur 32 bits), ajouter 1 à $2^{31} - 1$ donnera un nombre négatif (et cela donnera -2^{31} si les nombres sont représentés en complément à 2).

Il n'est pas possible d'illustrer cela sous Python, ou alors de manière très détournée, car les entiers peuvent être arbitrairement grand (l'unique limite étant la mémoire disponible sur la machine).

Pour information les nombres entiers en Python sont représentés comme une suite de chiffres en base 230. Plus d'informations ici.

Représentation approximative des nombres réels : notion de nombre flottant

> Attendu : Calculer sur quelques exemples la représentation de nombres réels : 0.1, 0.25 ou 1/3.
>
> Commentaire : 0.2 + 0.1 n'est pas égal à 0.3. Il faut éviter de tester l'égalité de deux flottants. Aucune connaissance précise de la norme IEEE-754 n'est exigible.

Un nombre flottant n s'écrit sous la forme $n = (-1)^s \times b^e \times m$, où s vaut soit 0 soit 1, b vaut soit 2 soit 10 et m, appelée la mantisse, dont on peut considérer qu'il s'agit d'un nombre entier[$\widehat{}$mantisse]. [$\widehat{}$mantisse]: Ce n'est pas tout à fait vrai dans la norme IEEE-754 mais, dans le cadre de cette norme, on peut néanmoins se ramener à ce cas-là.

L'intérêt d'une telle notation est qu'elle permet de représenter avec une même précision de très petits nombres que de très grand nombre. Cependant on ne peut représenter que des nombres rationnels, mais pas tous.

Par exemple, en prenant $b = 10$:

- $0,1 = (-1)^0 \times 10^{-1} \times 1$
- $0,25 = (-1)^0 \times 10^{-2} \times 25$
- $1/3 = (-1)^0 \times 10^{-beaucoup} \times 33333\,...$
- $0,0000421 = (-1)^0 \times 10^{-7} \times 421$
- $-421000 = (-1)^1 \times 10^3 \times 421$

Mais lorsque la représentation se fait sur ordinateur, il est plus aisé d'avoir une base $b = 2$. En base 2 le nombre $1{,}110001_2$ est $1 + 1/2 + 1/4 + 1/64$. Voici quelques valeurs pour les puissances de 2 négatives :

x	2^{-x}
1	0.5
2	0.25
3	0.125
4	0.0625
5	0.03125
6	0.015625
7	0.0078125
8	0.00390625
9	0.001953125
10	0.0009765625

Dans ce cas, on a : * $0,1 = (-1)^0 \times 2^{-4} \times 1,6$. Or $1,6 = 1 + 1/2 + 1/16 + 1/32 + 1/256 + 1/512 + \cdots$. De la même manière qu'on ne peut pas représenter $1/3$ de manière exacte avec $b = 10$, on ne pourra pas représenter $0,1$ de manière exacte avec $b = 2$ * $0,25 = (-1)^0 \times 2^{-2} \times 1$

Attention Les calculs sur les nombres flottants ne sont donc pas exacts. Il est parfois nécessaire d'approximer la valeur à représenter. Il ne faut **jamais** tester une égalité entre deux nombre flottants mais utiliser une marge d'erreur relative.

Attention Il ne faut pas se fier à l'affichage de Python (ou d'un autre langage) qui n'affiche pas toutes les décimales stockées du nombre flottant. On peut cependant accéder à plus de décimales en utilisant par exemple les options de formatage de `format`. Attention aux yeux !

```
>>> .1
0.1
>>> .25
0.25
>>> '{:.40f}'.format(.1)
'0.1000000000000000055511151231257827021182'
>>> '{:.40f}'.format(.2)
'0.2000000000000000111022302462515654042363'
>>> '{:.40f}'.format(.25)
'0.2500000000000000000000000000000000000000'
>>> '{:.40f}'.format(.3)
'0.2999999999999999888977697537484345957637'
>>> '{:.40f}'.format(.05)
'0.0500000000000000027755575615628913510591'
>>> .1 + .2 == .3
False
>>> .1 + .1 + .1 == .3
False
>>> .25 + .05 == .3
True
```

Valeurs, opérateurs et expressions booléennes

> Attendu : Dresser la table d'une expression booléenne.
>
> Commentaires : Le ou exclusif (xor) est évoqué. Quelques applications directes comme l'addition binaire sont présentées. L'attention des élèves est attirée sur le caractère séquentiel de certains opérateurs booléens.

Table d'une expression booléenne avec n variables : 2^n cas à évaluer.

Exemple avec $(a \vee b) \wedge c$ ($\vee$: OU ; $\wedge$: ET)

a	b	c	$(a \vee b)$	$(a \vee b) \wedge c$
0	0	0	0	0
0	0	1	0	0
0	1	0	1	0
0	1	1	1	1
1	0	0	1	0
1	0	1	1	1
1	1	0	1	0
1	1	1	1	1

Exemples d'application

L'opérateur ET permet de créer des masques afin de ne conserver que certains bits d'une valeur.

Parité d'un nombre

Un entier naturel pair n a son bit de poids faible à 0. Il suffit donc de consulter ce bit pour connaître la parité du nombre.

$n \wedge 1$ permet de ne conserver que le bit de poids faible (tous les autres bits sont mis à 0). Si le résultat est 1 alors le nombre est impair, sinon le nombre est pair.

Signe d'un nombre représenté en complément à 2

On a vu que dans la représentation en complément à 2, le bit de poids fort désigne le bit de signe. Si on suppose un nombre entier n représenté sur p bits, on peut isoler le bit de poids fort avec un ET également : $n \wedge (2^{p-1})$. Si le résultat est 0, le nombre est positif sinon il est négatif.

Caractère séquentiel

Les expressions booléennes sont évaluées de manière paresseuses : dès que le résultat est connu l'évaluation est stoppée.

Par exemple avec `a ET b ET c`. Si `a` est faux, `b` et `c` ne sont même pas évaluées puisque le résultat sera nécessairement faux.

L'ordre dans lequel les expressions sont écrites est donc important. Par exemple il faut d'abord vérifier qu'une clé existe dans un dictionnaire pour ensuite vérifier sa valeur.

Si `d` est un dictionnaire, on peut faire :

```
if 'cle' in d and d['cle'] == 2:
```

Mais on **ne doit pas** faire (une exception sera levée dès que la clé n'existe pas) :

```
if d['cle'] == 2 and 'cle' in d:
```

XOR

Le ou-exclusif ne pose généralement pas de problème car son interprétation correspond à l'interprétation intuitive du OU en français « *resto chinois ou italien* ? ». Le résultat d'un ou-exclusif entre deux valeurs n'est vrai que si exactement une des deux valeurs est vraie.

Et les opérateurs bit-à-bit ?

Les opérateurs bit-à-bit incluent les opérateurs booléens déjà mentionnés mais également les décalages de bits.

- *Décalage à droite*: le décalage à droite de k positions d'un entier n, noté $n \gg k$, est l'entier dont l'écriture binaire est obtenue en supprimant les k bits de poids faibles de l'écriture binaire de n.
- *Décalage à gauche*: le décalage à gauche de k positions d'un entier n, noté $n \ll k$, est l'entier dont l'écriture binaire est obtenue en ajoutant k bits nuls à droite de l'écriture binaire de n.

En Python, décaler un nombre entier positif d'un bit vers la gauche revient à le multiplier par 2 (et le décaler de k bits, revient à le multiplier par 2^k). Si le nombre entier est représenté sur un nombre fixe de bits (ce qui n'est pas le cas de Python), décaler de k bits vers la gauche va également faire perdre les k bits qui étaient originellement de poids fort.

En Python les décalages à gauche et à droite se font respectivement avec les opérateurs `<<` et `>>`

```
>>> 1 << 2
4
>>> 5 >> 1
2
```

Le décalage de bit aurait été utile dans l'exemple d'application précédent de détermination du bit de signe. Nous avions fait : $n \wedge (2^{p-1})$ pour cela et dans ce cas soit le résultat était 0 soit 2^{p-1}. Avec le décalage à droite on peut également faire $n \gg (p-1)$ et dans ce cas le résultat est soit 0 soit 1.

Représentation d'un texte en machine

> Attendu : Identifier l'intérêt des différents systèmes d'encodage. Convertir un fichier texte dans différents formats d'encodage.

Commentaires : Aucune connaissance précise des normes d'encodage n'est exigible.

Pourquoi différents encodages de caractères ?

ASCII

ASCII (*American Standard Code for Information Interchange*) est la première norme largement utilisée pour encoder des caractères. Comme son nom l'indique cette norme est américaine et elle n'inclut donc que les lettres latines non accentuées (en plus des chiffres, opérateurs mathématiques, caractères de ponctuation ou de délimitation et certains caractères spéciaux).

Voici les caractères de la table ASCII (les 33 premiers, et le dernier, ne sont pas imprimables) :

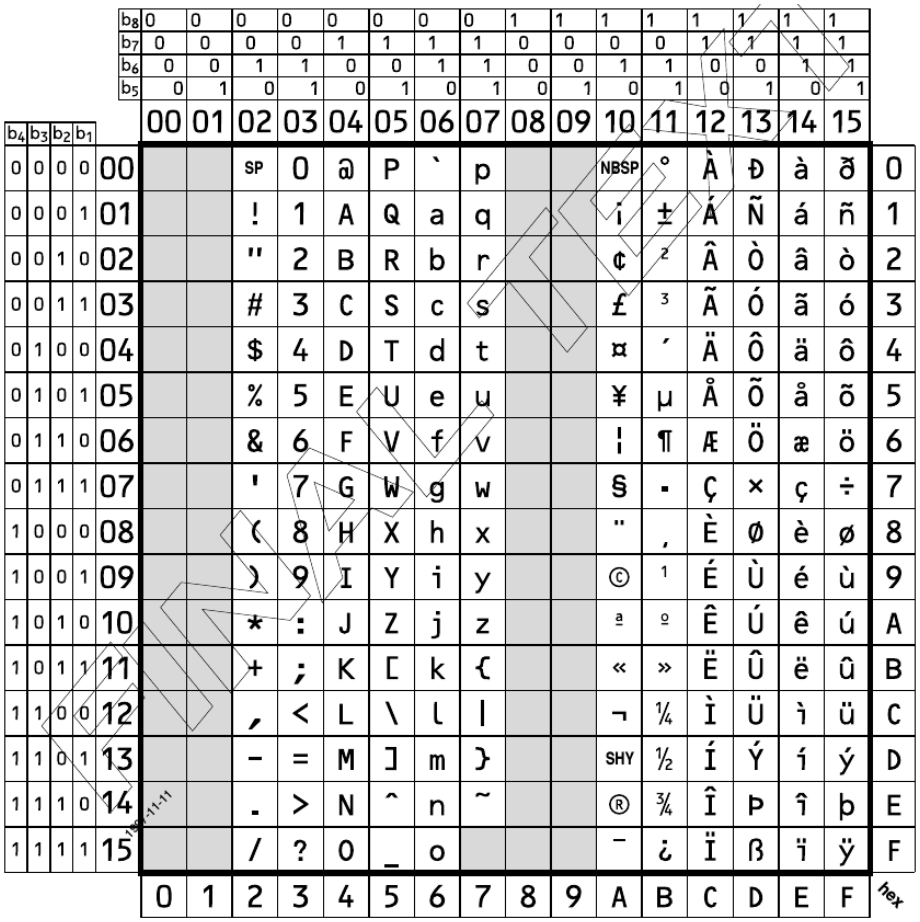
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	ESP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d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

128 caractères composent la table ASCII, ce qui permet de les représenter sur 7 bits (en pratique plutôt 8 bits afin d'occuper un octet complet).

ISO-8859-1

Par la suite d'autres encodages ont vu le jour afin de pallier les limites de l'ASCII. L'ISO-8859-1 (aussi appelé *Latin-1*), pour l'Europe occidentale, a vu le jour en 1986. Celui-ci comble les manques pour la plupart des langues d'Europe occidentale. Pour le français il manque cependant le œ, le Œ et le Ÿ et, bien entendu, le symbole €. L'encodage en ISO-8859-1 utilise 8 bits, les 128 premières valeurs de l'ISO-8859-1 sont identiques à l'ASCII, ce qui assure une compatibilité avec cet encodage.

Voici la table des caractères ISO-8859-1 :



UTF-8

À nouveau le codage ISO-8859-1 (et les autres codages de la famille ISO-8859) présentent des limites. Dans les années 1990, le projet Unicode de codage unifié de tous les alphabets est né. Différents codages sont utilisés pour représenter des caractères Unicode (UTF-8, UTF-16, UTF-32…). Ici nous nous concentrons sur l'UTF-8

Le codage UTF-8 est un codage de longueur variable. Certains caractères sont codés sur un seul octet, ce sont les 128 caractères du codage ASCII. Les autres caractères peuvent être codés sur 2, 3 ou 4 octets. Ainsi l'UTF-8 permet en théorie de représenter $2^{21} = 2\,097\,152$ caractères différents, en réalité un peu moins. Il y a actuellement environ une centaine de milliers de caractères Unicode (incluant les caractères des langues vivantes ou mortes et également de nombreux emojis indispensables

Les caractères en UTF-8 doivent avoir une forme particulière décrite dans la

table ci-dessous :

Nbre octets codant	Format de la représentation binaire
1	`0xxxxxxx`
2	`110xxxxx 10xxxxxx`
3	`1110xxxx 10xxxxxx 10xxxxxx`
4	`11110xxx 10xxxxxx 10xxxxxx 10xxxxxx`

L'encodage UTF-8 est lui aussi compatible avec l'ASCII. En revanche ISO-8859-1 et UTF-8 sont incompatibles entre eux pouvant conduire à ce genre de problèmes :

Aller plus loin

Plus d'informations sur ces différents aspects sont disponibles dans le chapitre 1 du polycopié du cours de Codage de l'information donné en L2 informatique. Le polycopié contient également des exercices.