Écrire un entier dans une base

Doc. 4,1

Objectif

Comprendre comment un ordinateur, qui ne manipule que des 0 et des 1, écrit un nombre entier, et savoir passer de l'écriture en base 10 à l'écriture en base 2 ou en base 16, à la main puis avec Python.

Q1. Un tee-shirt porte l'inscription : « Il y a 10 sortes de personnes : celles qui comprennent le binaire et les autres. » Combien de sortes de personnes la phrase énumère-t-elle vraiment ? Que peut bien valoir « 10 » pour que la phrase soit juste ?

Réponse : La phrase énumère deux sortes de personnes. Elle n'est juste que si « 10 » vaut deux. C'est le cas lorsque 10 est écrit en base 2, comme on va le voir : $(10)_2 = 2$. Le même assemblage de chiffres ne désigne pas le même nombre selon la base dans laquelle on l'écrit.

§1. Ce que l'on fait sans y penser : la base 10

Un ordinateur est fait de circuits électroniques qui ne connaissent que deux états : le courant passe ou ne passe pas. On note ces deux états 0 et 1. Une information qui ne peut prendre que ces deux valeurs s'appelle un **bit** (de l'anglais *binary digit*, chiffre binaire). Pour écrire n'importe quel nombre avec seulement deux chiffres, l'ordinateur utilise la même idée que nous avec nos dix chiffres. Commençons donc par regarder de près ce que nous faisons tous les jours.

Dans le nombre 2026, les deux chiffres 2 n'ont pas la même valeur. Le premier compte pour deux milliers, le second pour deux unités. La valeur d'un chiffre dépend de sa **position** dans l'écriture :

$$2026 = 2 \times 1000 + 0 \times 100 + 2 \times 10 + 6 \times 1 = 2 \times 10^3 + 0 \times 10^2 + 2 \times 10^1 + 6 \times 10^0$$

En base 10, on dispose de dix chiffres, de 0 à 9. Chaque position vaut dix fois plus que celle qui est à sa droite. On dit que le chiffre des milliers a un **poids** de 10^3.

Q2. Écrire 7305 sous la forme d'une somme de chiffres multipliés par des puissances de 10. Pourquoi a-t-on besoin d'un chiffre 0 dans cette écriture ?

Réponse : $7305 = 7 \times 10^3 + 3 \times 10^2 + 0 \times 10^1 + 5 \times 10^0$.

Le 0 indique qu'il n'y a aucune dizaine. Sans lui, on écrirait 735, qui est un autre nombre : le 0 sert à garder chaque chiffre à sa place.

§2. La base 2 : compter avec deux chiffres

§2.1. Du binaire vers le décimal

En base 2, on ne dispose que de deux chiffres, 0 et 1. Chaque position vaut deux fois plus que celle qui est à sa droite. Les poids sont donc, de droite à gauche, $2^0 = 1$, $2^1 = 2$, $2^2 = 4$, $2^3 = 8$, $2^4 = 16$, etc. Pour éviter toute confusion, on note la base en indice : $(1101)_2$ désigne l'écriture en base 2, et 1101 seul désigne le nombre mille cent un.

Exemple résolu : que vaut $(1101)_2$?

On écrit les poids au-dessus de chaque chiffre, en partant de la droite :

poids	$2^3 = 8$	$2^2 = 4$	$2^1 = 2$	$2^0 = 1$
chiffre	1	1	0	1

On additionne les poids des positions où se trouve un 1 :

$$(1101)_2 = 1 \times 8 + 1 \times 4 + 0 \times 2 + 1 \times 1 = 13$$

Les chiffres 0 ne comptent pas : ils servent seulement à garder les autres à leur place, comme en base 10.

Q3. En utilisant la même méthode, déterminer la valeur décimale de :

1. $(10)_2$;
2. $(101010)_2$;
3. $(11111111)_2$.

Réponse :
1. $(10)_2 = 1 \times 2 + 0 \times 1 = 2$: c'est le « 10 » du tee-shirt.
2. $(101010)_2 = 32 + 8 + 2 = 42$.
3. $(11111111)_2 = 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 255$.

Python sait faire ce calcul. La fonction `int`, qui convertit une chaîne en entier, accepte un deuxième argument : la base dans laquelle la chaîne est écrite. On peut aussi écrire directement un nombre en binaire dans un programme en le faisant précéder de `0b`.

Q4. Dans la console de Thonny, vérifier les réponses de la question précédente :

```Python
>>> int("101010", 2)
>>> int("11111111", 2)
>>> 0b101010
```

Réponse :

```Python
>>> int("101010", 2)
42
>>> int("11111111", 2)
255
>>> 0b101010
42
```

§2.2. Du décimal vers le binaire

Pour passer dans l'autre sens, on cherche à répondre à la question : quels poids faut-il additionner pour obtenir le nombre ? Une méthode systématique consiste à diviser le nombre par 2, encore et encore. On a déjà rencontré cette idée dans le Doc. 2,4 : `n % 10` donnait le chiffre des unités de `n` en base 10, et `n // 10` retirait ce chiffre. En base 2, on fait la même chose avec 2 à la place de 10.

Exemple résolu : écrire 13 en base 2

On effectue des divisions euclidiennes successives par 2, jusqu'à obtenir un quotient nul :

division	**quotient** (`n // 2`)	**reste** (`n % 2`)	**ce que donne le reste**
$13 = 2 \times 6 + 1$	6	**1**	chiffre des unités (2^0)
$6 = 2 \times 3 + 0$	3	**0**	chiffre de poids 2^1
$3 = 2 \times 1 + 1$	1	**1**	chiffre de poids 2^2
$1 = 2 \times 0 + 1$	0	**1**	chiffre de poids 2^3

Le premier reste obtenu est le chiffre de droite. On lit donc les restes **de bas en haut** : $13 = (1101)_2$. On vérifie : $8 + 4 + 1 = 13$.

Q5. Écrire 42 puis 100 en base 2 par la méthode des divisions successives. Vérifier ensuite avec la fonction `bin` de Python : `bin(42)` et `bin(100)`.

Réponse : Pour 42, les restes successifs sont 0, 1, 0, 1, 0, 1 (quotients 21, 10, 5, 2, 1, 0). Lus de bas en haut : $42 = (101010)_2$.

Pour 100, les restes successifs sont 0, 0, 1, 0, 0, 1, 1 (quotients 50, 25, 12, 6, 3, 1, 0). Lus de bas en haut : $100 = (1100100)_2$.

```python
>>> bin(42)
'0b101010'
>>> bin(100)
'0b1100100'
```

La fonction `bin` retourne une chaîne de caractères qui commence par `'0b'`, pour rappeler que c'est une écriture en base 2.

Q6. Un élève écrit 13 en base 2 et obtient $(1011)_2$. Il a pourtant fait toutes les divisions sans erreur. Quelle erreur a-t-il commise ? Comment aurait-il pu la repérer lui-même ?

Réponse : Il a lu les restes de haut en bas, dans l'ordre où il les a obtenus. Or le premier reste est le chiffre des unités, qui s'écrit à droite. Il pouvait repérer l'erreur en vérifiant son résultat : $(1011)_2 = 8 + 2 + 1 = 11$, et non 13. Vérifier une conversion en faisant la conversion inverse est un bon réflexe.

§3. La base 16 : une écriture plus courte du binaire

Les écritures binaires sont vite très longues : il faut 8 chiffres pour écrire 255, et 11 pour écrire 2026. Pour les écrire de façon plus compacte, les informaticiens utilisent la base 16, appelée **hexadécimale**.

En base 16, il faut seize chiffres. On utilise les chiffres de 0 à 9, puis les lettres A, B, C, D, E et F pour les valeurs de 10 à 15 :

base 16	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
base 10	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Les poids sont maintenant des puissances de 16 : $16^0 = 1$, $16^1 = 16$, $16^2 = 256$, etc. Par exemple :

$$(2A)_{16} = 2 \times 16 + 10 \times 1 = 42$$

En Python, un nombre écrit en base 16 commence par `0x`, et la fonction `hex` donne l'écriture hexadécimale d'un entier.

Q7.
1. Déterminer la valeur décimale de $(FF)_{16}$ et de $(7E3)_{16}$.
2. Vérifier avec Python : `0xFF`, `int("7E3", 16)` et `hex(42)`.

Réponse :
1. $(FF)_{16} = 15 \times 16 + 15 = 255$ et $(7E3)_{16} = 7 \times 256 + 14 \times 16 + 3 = 1792 + 224 + 3 = 2019$.
2.
```
>>> 0xFF
255
>>> int("7E3", 16)
2019
>>> hex(42)
'0x2a'
```
Python écrit les lettres en minuscules : `0x2a` et `0x2A` désignent le même nombre.

§3.1. Passer du binaire à l'hexadécimal

Le grand intérêt de la base 16 vient de ce que $16 = 2^4$. Un chiffre hexadécimal, qui va de 0 à 15, correspond exactement à un paquet de 4 chiffres binaires, qui vont de $(0000)_2 = 0$ à $(1111)_2 = 15$.

Exemple résolu : écrire $(10101101)_2$ en base 16

On découpe l'écriture binaire en paquets de 4 bits, **en partant de la droite** : 1010 1101.

On convertit chaque paquet séparément : $(1010)_2 = 8 + 2 = 10$, soit A, et $(1101)_2 = 8 + 4 + 1 = 13$, soit D.

On obtient $(10101101)_2 = (AD)_{16}$. Aucune division n'a été nécessaire.

Q8.
1. Écrire $(11110000)_2$ en base 16.
2. Écrire $(3C)_{16}$ en base 2.
3. Pour écrire $(110101)_2$ en base 16, un élève découpe 1101 01 et obtient $(D1)_{16}$. Que lui répondre ?

Réponse :
1. 1111 0000 donne F et 0 : $(11110000)_2 = (F0)_{16}$.
2. 3 s'écrit 0011 et C (12) s'écrit 1100 : $(3C)_{16} = (00111100)_2 = (111100)_2$. Chaque chiffre hexadécimal doit donner exactement 4 bits, en complétant à gauche par des 0.
3. Il faut découper en partant de la droite : 11 0101, ce qui donne 3 et 5, donc $(35)_{16}$. Vérification : $(110101)_2 = 32 + 16 + 4 + 1 = 53$ et $(35)_{16} = 3 \times 16 + 5 = 53$. Son résultat $(D1)_{16} = 209$ est faux, car il a donné aux bits de gauche un poids trop faible.

Où rencontre-t-on la base 16 ?

En HTML, une couleur s'écrit souvent sous la forme `#FF8000` : trois nombres en base 16, de 00 à FF, donnent les quantités de rouge, de vert et de bleu. Ici, rouge 255, vert 128 et bleu 0 donnent de l'orange.

§4. Programmer les conversions

Les fonctions `int`, `bin` et `hex` font déjà les conversions. Les programmer soi-même oblige à comprendre exactement ce qu'elles font.

§4.1. Du binaire vers le décimal

Une chaîne de caractères se parcourt comme un tuple : `for chiffre in "1101":` donne successivement les caractères `"1"`, `"1"`, `"0"` et `"1"`, de gauche à droite.

Plutôt que de calculer chaque puissance de 2, on peut lire les chiffres de gauche à droite en appliquant une règle simple : à chaque nouveau chiffre, on multiplie le résultat par 2 (tout ce qui a déjà été lu monte d'une position), puis on ajoute le chiffre.

Q9. Compléter le tableau d'évolution de la variable `n` lorsqu'on lit la chaîne `"1101"` en appliquant la règle `n = 2 * n + int(chiffre)`, avec `n` égal à 0 au départ.

étape	chiffre lu	n
départ		0
1		
2		
3		
4		

Réponse :

étape	chiffre lu	n
départ		0
1	`"1"`	$2 \times 0 + 1 = 1$
2	`"1"`	$2 \times 1 + 1 = 3$
3	`"0"`	$2 \times 3 + 0 = 6$
4	`"1"`	$2 \times 6 + 1 = 13$

On retrouve $(1101)_2 = 13$. Le premier 1 lu a été multiplié trois fois par 2 : il compte bien pour $2^3 = 8$.

Q10. Écrire la fonction `recompose` dont la spécification est :

```python
def recompose(binaire: str) -> int:
    """
    Retourne l'entier dont binaire est l'écriture en base 2.

    >>> recompose("1101")
    13
    >>> recompose("11111111")
    255
    >>> recompose("0")
    0
```

```
    """
```

Réponse :

```python
def recompose(binaire: str) -> int:
    n = 0
    for chiffre in binaire:
        n = 2 * n + int(chiffre)
    return n
```

`chiffre` est une chaîne d'un caractère : il faut la convertir avec `int` avant de l'additionner. Sans cette conversion, Python signale une erreur `TypeError`, car on ne peut pas additionner un entier et une chaîne.

§4.2. Du décimal vers le binaire

Q11. On veut écrire la fonction `decompose`, qui traduit en Python la méthode des divisions successives :

```python
def decompose(n: int) -> str:
    """
    Retourne l'écriture en base 2 de l'entier positif n.

    >>> decompose(13)
    '1101'
    >>> decompose(42)
    '101010'
    >>> decompose(0)
    '0'
    """
```

1. Le nombre de divisions n'est pas connu à l'avance. Quelle boucle faut-il utiliser ? Quelle est sa condition d'arrêt ?
2. Chaque reste doit être placé à gauche des chiffres déjà obtenus. Si `bits` est la chaîne déjà construite, quelle instruction ajoute le reste `n % 2` à sa gauche ?
3. Écrire la fonction. Attention au cas `n = 0`.

Coup de pouce

1. Reprendre le tableau de l'exemple résolu pour 13 : on s'arrête quand le quotient devient nul.
2. Une chaîne se construit par concaténation avec `+` ; l'ordre des deux morceaux compte.
3. Que fait la boucle quand `n` vaut 0 dès le départ ?

Réponse :

1. Une boucle `while`, puisqu'on ne sait pas d'avance combien de divisions seront nécessaires. On continue tant que `n > 0`.
2. `bits = str(n % 2) + bits`. Écrire `bits + str(n % 2)` placerait les restes dans l'ordre où on les obtient, c'est-à-dire à l'envers.

3.
```python
def decompose(n: int) -> str:
    if n == 0:
        return "0"
    bits = ""
    while n > 0:
        bits = str(n % 2) + bits
        n = n // 2
    return bits
```

Lorsque `n` vaut 0, la condition `n > 0` est fausse dès le premier test : sans le cas particulier, la fonction retournerait la chaîne vide.

Dans un ordinateur, les nombres sont rangés dans des cases de taille fixe. Une case de 8 bits s'appelle un **octet**. Pour écrire 13 sur un octet, on complète à gauche par des 0 : 00001101. Ces 0 ne changent pas la valeur, comme 0042 vaut toujours 42.

Q12. Écrire la fonction `octet`, qui doit utiliser `decompose` :

```python
def octet(n: int) -> str:
    """
    Retourne l'écriture en base 2 de l'entier n sur 8 bits.
    Précondition : 0 <= n <= 255.

    >>> octet(13)
    '00001101'
    >>> octet(255)
    '11111111'
    """
```

Réponse :

```python
def octet(n: int) -> str:
    assert 0 <= n <= 255, "n doit être compris entre 0 et 255"
    bits = decompose(n)
    while len(bits) < 8:
        bits = "0" + bits
    return bits
```

On peut aussi écrire `return "0" * (8 - len(bits)) + bits` : l'opérateur `*` répète une chaîne. L'assertion vérifie la précondition ; les entiers plus grands que 255 ne tiennent pas sur un octet, comme on le verra dans le Doc. 4,2.

§4.3. Vers la base 16

La méthode des divisions successives marche dans n'importe quelle base : pour la base 16, on divise par 16. Il reste une difficulté : un reste peut valoir de 0 à 15, et il faut l'écrire avec un seul chiffre. On range les seize chiffres hexadécimaux dans une chaîne, à la bonne place :

```python
CHIFFRES = "0123456789ABCDEF"
```

Comme pour un tuple, `CHIFFRES[13]` est le caractère d'indice 13, c'est-à-dire `"D"`.

Q13. Écrire la fonction `decompose_16` en s'inspirant de `decompose` :

```python
def decompose_16(n: int) -> str:
    """
    Retourne l'écriture en base 16 de l'entier positif n.

    >>> decompose_16(42)
    '2A'
    >>> decompose_16(255)
    'FF'
    >>> decompose_16(2019)
    '7E3'
    """
```

Réponse :

```python
CHIFFRES = "0123456789ABCDEF"

def decompose_16(n: int) -> str:
    if n == 0:
        return "0"
    resultat = ""
    while n > 0:
        resultat = CHIFFRES[n % 16] + resultat
        n = n // 16
    return resultat
```

Seules trois choses changent par rapport à `decompose` : on divise par 16, et le reste devient un caractère grâce à `CHIFFRES[n % 16]` au lieu de `str(n % 2)`.

Q14. Sans écrire de nouvelle fonction, expliquer comment modifier `decompose_16` pour obtenir l'écriture d'un entier en base 8. Quels caractères de `CHIFFRES` seraient alors utilisés ?

Réponse : Il suffit de remplacer 16 par 8 dans les deux divisions. Les restes vont alors de 0 à 7 : seuls les caractères `"0"` à `"7"` de `CHIFFRES` sont utilisés. On pourrait même écrire une seule fonction qui prend la base en paramètre.

§5. Bilan

À l'issue de cette activité, je sais…

- expliquer que la valeur d'un chiffre dépend de sa position, en base 10 comme en base 2 ou 16 ;
- calculer la valeur décimale d'une écriture en base 2 ou en base 16 à l'aide des poids ;

- écrire un entier en base 2 par la méthode des divisions successives, en lisant les restes dans le bon ordre ;
- passer de la base 2 à la base 16, et inversement, par paquets de 4 bits ;
- utiliser `int(..., 2)`, `int(..., 16)`, `bin`, `hex` et les préfixes `0b` et `0x` ;
- programmer les conversions de la base 2 vers la base 10 et de la base 10 vers les bases 2 et 16.

Dans le Doc. 4,2, on se demande combien de bits il faut pour écrire un entier donné, et ce qui se passe quand un résultat ne tient plus dans la place prévue.