Chapitre 6

Opérations bit à bit

« cinq et trois font un, cinq ou trois font sept »

À retenir

- les opérateurs & et | procèdent bit à bit et diffèrent de `and` et `or` ;
- on utilise très souvent | lors d'appels de fonctions qui demandent des flags.

1 De nouveaux opérateurs

Il n'y a qu'à lire la phrase d'introduction pour constater que « et » et « ou » sont deux nouvelles opérations qui n'ont rien à voir avec l'addition ou la multiplication usuelles.
On dit que ce sont des opérations bit à bit (*bitwise operations* en Anglais) car elles se basent sur les écritures binaires des entiers.

Définition : opérateur &

Soient `a` et `b` deux `int` positifs, `a & b` est un `int` dont la représentation binaire s'obtient en faisant un « et » logique sur les bits correspondants de `a` et `b`.

Exemple

Écriture décimale	Écriture binaire			
13	1	1	0	1
11	1	0	1	1
13 & 11 = 9	1	0	0	1

De la même manière, $3 = (011)_2$, $5 = (101)_2$ de sorte que $3 \,\&\, 5 = 1$.

Définition : opérateur |

Soient a et b deux `int` positifs, `a | b` est un `int` dont la représentation binaire s'obtient en faisant un « ou » logique sur les bits correspondants de a et b.

Exemple

Écriture décimale	Écriture binaire			
13	1	1	0	1
11	1	0	1	1
13 | 11 = 15	1	1	1	1

De la même manière, $3 = (011)_2$, $5 = (101)_2$ de sorte que $3 \,|\, 5 = 7$.

Définition : opérateur <<

Soient a et n deux `int` positifs, `a << n` est un `int` dont la représentation binaire est celle de a, suivie de n zéros.
Il s'agit d'un décalage vers la gauche des bits de a, d'où le symbole <<.
D'après ce que nous avons vu en première, `a << n` vaut `a * 2 ** n`.

Exemples

1. `7 << 3` vaut 56.
2. `1 << 10` vaut 1 024.

Définition : opérateur >>

Soient a et n deux `int` positifs, `a >> n` est un `int` dont la représentation binaire est celle de a décalée de n crans vers la droite avec troncature avant la virgule. D'après ce que nous avons vu en première, `a >> n` vaut `a // 2 ** n`.

Exemple

On veut déterminer la valeur de `147 >> 5` :

1. on commence par déterminer que $147 = (1001\,0011)_2$;
2. on décale de 5 crans vers la droite en oubliant les bits qui passent après la virgule, on obtient $(100)_2$;
3. `147 >> 5` vaut donc 4.

Attention

Les opérateurs `&` et `|` ne doivent pas être confondus avec `and` et `or` :

1. `&` et `|` portent sur des `int` et renvoient des `int` ;
2. `and` et `or` portent traditionnellement sur des `bool` ;

Ceci dit, PYTHON accepte d'évaluer `5 or 3` :

1. il évalue logiquement 5, qui ne vaut pas 0, donc est considéré comme `True` ;
2. puisqu'il procède paresseusement, il n'évalue pas 3 et renvoie… 5.

De la même manière pour `0 or 3` :

1. il évalue logiquement 0, qui vaut `False`
2. ensuite il évalue logiquement 3 qui vaut `True` et renvoie donc 3.

On a le même phénomène avec `and`

Exercice 1

Calculer à la main

1. `(3 << 5) | (5 << 4)`
2. `(120 & 117) >> 2`

Exercice 2 rigoureusement inutile donc indispensable

Peux-tu prédire la valeur des expressions suivantes ?

1. `2 or 7`
2. `2 and 7`
3. `0 and 7`

4. `2 or 0`

Exercice 3

On imagine une fonction qui prend (entre autres) en paramètre un `int` appelé `flags` qui représente 8 flags qu'on peut combiner entre eux comme on le désire : pour n compris entre 0 et 7, le flag f_n n'est autre que 2^n.
Cela veut dire que si par exemple `flags` vaut 28, alors puisque $28 = (0011100)_2$, on a mis f_2, f_3 et f_4 à 1 et les autres flags à 0.

1. Comment à l'aide des seuls symboles 1, 2, 7 et des opérateurs bit à bit, faire en sorte que `flags` présente les deux seuls flags f_2 et f_7 ?
2. On imagine qu'on a récupéré une valeur de la variable `flags`. Comment tester si f_4 est bien à 1 ? Comment tester si f_5 est à 0 ?

2 Applications (hors programme)

Sous Windows, le module `pywin` offre un accès à l'API (interface de programmation) `Win32`. Celle-ci, destinée à utiliser des fonctionnalités du système d'exploitation Windows, a vu le jour au début des années 90 et a été écrite en C / C++. On peut donc qualifier cette API d'*archaïque*.
Dans ce module, on trouve la fonction `mouse_event` qui permet de simuler des clics de divers boutons de souris, des déplacements, des glisser-déposer (*drag-n-drop* en Anglais).
Voici un morceau de documentation (elle aussi archaïque) de cette fonction :

win32api.mouse_event

```
mouse_event(dwFlags, dx, dy, dwData, dwExtraInfo)
```

Simulate a mouse event

name	type	comments
`dwFlags=0`	`int`	**Flags** specifying various function options
`dx`	`int`	Horizontal position of mouse
`dy`	`int`	Vertical position of mouse
`dwData`	`int`	Flag specific parameter
`dwExtraInfo=0`	`int`	Additional data associated with mouse event

`dwFlags` controls various aspects of mouse motion and button clicking. This parameter

can be certain combinations of the following values.

name	value	
MOUSEEVENTF_ABSOLUTE	0x8000	The `dx` and `dy` parameters contain normalized absolute coordinates. If not set, those parameters contain relative data : the change in position since the last reported position. This flag can be set, or not set, regardless of what kind of mouse or mouse-like device, if any, is connected to the system. For further information about relative mouse motion, see the following Remarks section.
MOUSEEVENTF_LEFTDOWN	0x0002	The left button is down.
MOUSEEVENTF_LEFTUP	0x0004	The left button is up.
MOUSEEVENTF_MIDDLEDOWN	0x0020	The middle button is down.
MOUSEEVENTF_MIDDLEUP	0x0040	The middle button is up.
MOUSEEVENTF_MOVE	0x0001	Movement occurred.
MOUSEEVENTF_RIGHTDOWN	0x0008	The right button is down.
MOUSEEVENTF_RIGHTUP	0x0010	The right button is up.
MOUSEEVENTF_WHEEL	0x0800	The wheel has been moved, if the mouse has a wheel. The amount of movement is specified in `dwData`
MOUSEEVENTF_XDOWN	0x0080	An X button was pressed.
MOUSEEVENTF_XUP	0x0100	An X button was released.
MOUSEEVENTF_WHEEL	0x0800	The wheel button is rotated.
MOUSEEVENTF_HWHEEL	0x01000	The wheel button is tilted.

Les « flags » dont les valeurs sont données en hexadécimal sont en fait des puissances de 2, donc leur écriture binaire comporte un seul bit à 1.
Les combinaisons dont parle la documentation s'obtiennent avec l'opérateur bit à bit « ou » : `|`.
Voici par exemple comment on simule un clic de souris (bouton gauche), puis un déplacement de 100 pixels vers la droite et 15 vers le bas, puis un relâchement du bouton :

Python

```
from win32api import *
from win32con import *

mouse_event(MOUSEEVENTF_LEFTDOWN | MOUSEEVENTF_MOVE, 100, 15, 0,
 ↪  0)
mouse_event(MOUSEEVENTF_LEFTUP, 100, 15, 0, 0)
```

Remarque

On a utilisé 2 flags lors du premier appel : un pour dire qu'il y a mouvement, l'autre pour dire que le bouton gauche est pressé.
Bien qu'on veuille qu'il y ait ces deux flags en même temps, on utilise un « ou » bit à bit et pas un « et » :

`MOUSEEVENTF_LEFTDOWN | MOUSEEVENTF_MOVE` vaut `2 & 1`
vaut `0b10 & 0b01`

vaut `0b11`
vaut 3

Si on avait utilisé un « et » on aurait obtenu zéro !