

Chapitre 6 : Les booléens

La traduction des raisonnements logiques par des opérations algébriques proposée par le mathématicien Georges Boole est un des fondements de l'informatique.

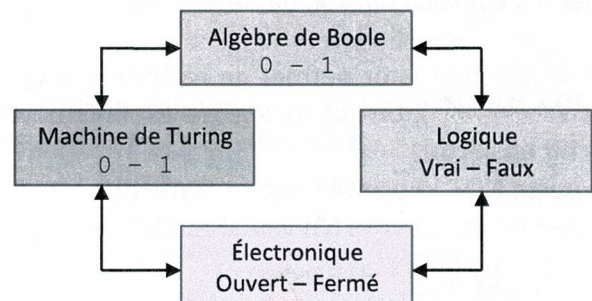
1. L'algèbre de Boole

1.1. Rappels historiques

À partir de 1847, le Britannique Georges Boole propose un mode de calcul permettant de traduire des raisonnements logiques par des opérations algébriques.

Il crée une branche des mathématiques qui définit des opérations dans un ensemble qui ne contient que **deux éléments** noté 0 et 1, ou bien Faux et Vrai (lien avec la logique), ou bien Ouvert et Fermé (lien avec l'électronique) ou encore un Python `False` et `True`.

En 1938, l'américain Claude Shannon prouve que des circuits électriques peuvent résoudre tous les problèmes que l'algèbre de Boole peut résoudre. Avec les travaux d'Alan Turing 1936, cela constitue le fondement de ce qui deviendra l'informatique.



1.2. Les opérations fondamentales

Les opérations fondamentales de l'**algèbre de Boole** ne sont plus l'addition et la multiplication des entiers mais :

conjonction $\rightarrow$ notée "&" ou "et" ou "."

désjonction $\rightarrow$ notée "+" ou "ou" ou "I"

négation $\rightarrow$ notée "~" ou "-" ou "non"

Ces opérations fondamentales sont des exemples de *fonctions logiques*.

Puisque l'algèbre de Boole ne contient que deux éléments, pour chacune de ces fonctions, on peut étudier tous les cas possibles et les regrouper dans un tableau appelé *table de vérité*.

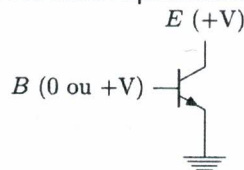
Une table de vérité récapitule dans un tableau les valeurs de vérité de la sortie en fonction des valeurs de vérité des entrées.

On représente souvent les opérateurs booléens à l'aide de **portes logiques** qui représentent à la fois la fonction logique qui lui est associée et le circuit électronique de la machine qui laisse passer le courant (Vrai) ou non (Faux) selon que le courant y entre ou non. Le calcul booléen permet donc d'utiliser la puissance du calcul algébrique pour régler des problèmes de logique et se traduit sur machine par des composants électroniques.

1.3. Les portes logiques

Les circuits d'un ordinateur manipulent uniquement des chiffres binaires 0 et 1 qui, en interne, sont simplement représentés par des tensions électriques. Ainsi, le chiffre 0 est représenté par une tension basse (proche de 0 volt) et le chiffre 1 par une tension haute (que l'on notera $+V$ volts, car cette tension varie selon les circuits).

Les opérateurs (logiques ou arithmétiques) sur ces nombres binaires sont construits à partir de **circuits électroniques** dont les briques élémentaires sont appelées *transistors*. Les transistors que l'on trouve dans les circuits des ordinateurs se comportent comme des interrupteurs qui laissent ou non passer un courant électrique, selon les modes du **tout ou rien**, comme représenté graphiquement de la manière suivante.



Dans ce schéma, la commande de l'interrupteur est jouée par la broche B. Lorsqu'elle est sous tension haute, la broche B laisse passer le courant entre la broche E (sous tension haute) et la masse (le sens du courant est indiqué par la petite flèche), ce qui a pour effet de passer E sous la tension basse. Inversement, quand B est sous tension basse, la broche E reste sous tension haute.

Ce simple transistor permet de réaliser une opération élémentaire appelée *porte non ou not* (par *de porte ou*). Une porte logique est une fonction qui prend **un ou plusieurs bits en entrée** et qui produit **un bit en sortie**.

La porte NOT implantée par un transistor est la plus simple de toutes les portes. Elle n'a qu'un seul bit en entrée (P) et sa sortie (Q) vaut 0 quand l'entrée vaut 1, et inversement elle vaut 1 quand son entrée est à 0.

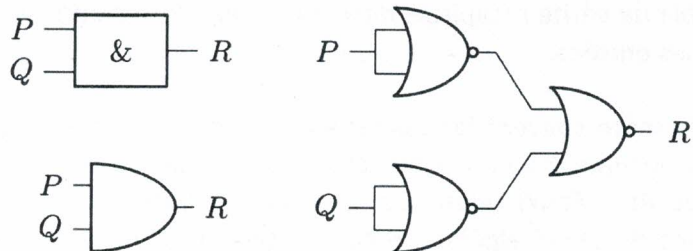


Pour représenter le calcul réalisé par une porte logique, on utilise une **table de vérité** qui relie les valeurs des entrées à la valeur de résultat. La table logique de la porte NOT est donnée ci-dessous.

Porte NOT ($Q = \sim P$)	
P	Q
0	1
1	0

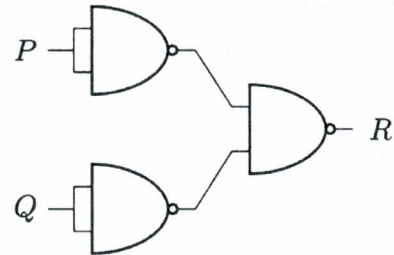
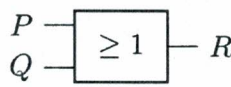
Une porte logique très importante est la porte ET (AND en anglais). Elle peut aussi être construite avec plusieurs portes NOR. Voici ci-dessous la table de vérité de la porte ET, ses représentations symboliques américaine (en bas) et européenne (en haut) ainsi qu'un schéma de construction avec des portes NOR.

Porte AND ($R = P \& Q$)		
P	Q	R
0	0	0
0	1	0
1	0	0
1	1	1

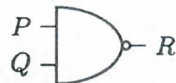
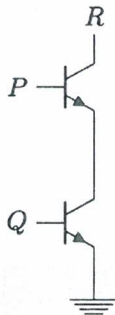


De la même manière, la porte OU (OR en anglais) peut aussi être construite avec plusieurs portes NAND.

Porte OR ($R = P + Q$)		
P	Q	R
0	0	0
0	1	1
1	0	1
1	1	1

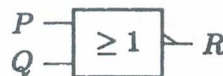
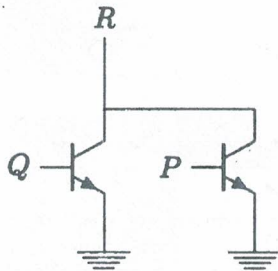


On peut fabriquer d'autres portes logiques en combinant plusieurs transistors. Par exemple, en combinant deux transistors en série comme ci-dessous on peut fabriquer la porte qui, pour deux entrées P et Q, produit un résultat R dont le calcul est donné par la table de vérité.



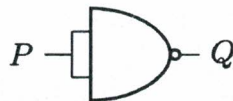
Porte NAND ($R = P \uparrow Q$)		
P	Q	R
0	0	1
0	1	1
1	0	1
1	1	0

De la même manière, en combinant deux transistors en parallèle, on obtient la porte NON OU (NOR en anglais) donnée ci-dessous (avec les schémas pour la représenter et sa table de vérité).

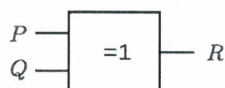


Porte NOR ($R = P \downarrow Q$)		
P	Q	R
0	0	1
0	1	0
1	0	0
1	1	0

Les portes NAND et NOR sont fondamentales dans les circuits électroniques car elles sont **complètes**, c'est-à-dire que n'importe quel circuit peut être conçu en utilisant **uniquement** ces deux portes. Par exemple, la porte NOT peut être fabriquée à partir d'une porte NAND en reliant les deux entrées de cette porte comme ci-dessous.



On peut aussi construire des portes plus complexes telle que la porte OU EXCLUSIF (XOR en anglais). Cet opérateur retourne « Vrai » quand seulement l'un des deux booléens est vrai, et « Faux » sinon.



Porte XOR ($R = P \oplus Q$)		
P	Q	R
0	0	0
0	1	1
1	0	1
1	1	0

Exercice 1 : vérité à partir d'une expression

Calculez la table de vérité de l'expression suivante :

a	b	c	(a ou b) et c
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Exercice 2 : Démontrer les lois de De Morgan

À l'aide des tables de vérité ci-dessous, démontrer les lois de De Morgan :

$$\sim(x \mid y) = \sim x \& \sim y \quad ; \quad \sim(x \& y) = \sim x \mid \sim y ;$$

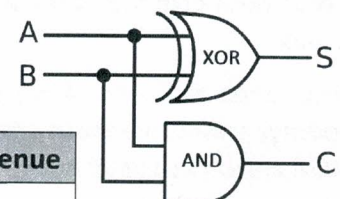
x	y	$x \mid y$	$\sim(x \mid y)$	$\sim x$	$\sim y$	$\sim x \& \sim y$
0	0	0	1	1	1	1
0	1	1	0	1	0	0
1	0	1	0	0	1	0
1	1	1	0	0	0	0

x	y	$x \& y$	$\sim(x \& y)$	$\sim x$	$\sim y$	$\sim x \mid \sim y$
0	0	0	1	1	1	1
0	1	0	1	1	0	1
1	0	0	1	0	1	1
1	1	1	0	0	0	0

Exercice 3 : Table de vérité d'un demi-additionneur

Le schéma ci-contre représente un circuit appelé demi-additionneur.

Construire la table de vérité correspondant à ce circuit.



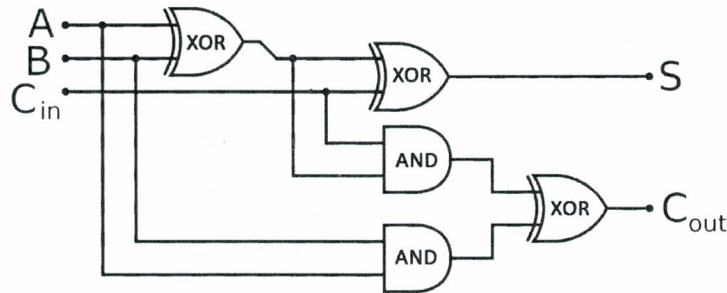
A	B	$S = A \oplus B$	C = Retenue
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Pourquoi, d'après vous, appelle-t-on ce circuit un demi-additionneur ?

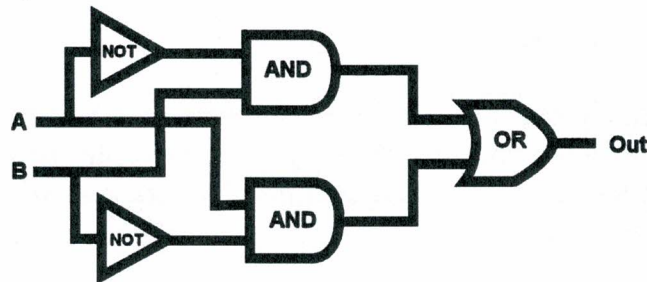
parce que la moitié de l'addition

Remarque :

Le circuit correspondant à l'additionneur complet, est composé de deux demi-additionneurs en série accompagnés d'une logique pour calculer la retenue (un OU EXCLUSIF entre les deux retenues générales par chacun des demi-additionneurs) :



Exercice 4 : Table de vérité à partir d'un schéma



Calculez la table de vérité à partir du schéma ci-dessus :

a	b	$\sim A$	$\sim B$	$\sim A \& B$	$A \& \sim B$	$(\sim A \& B) (A \& \sim B)$
0	0	1	1	0	0	0
0	1	1	0	1	0	1
1	0	0	1	0	1	1
1	1	0	0	0	0	0

Quelle table de vérité retrouve-t-on ?

OU Exclusif $\rightarrow$ XOR

1.4. Opérations bit à bit

On peut généraliser les opérations précédentes à des chaînes de bits.

Exemples :

Addition posée	Avec Python
$\begin{array}{r} 1011011 \\ \& 1010101 \\ \hline 1010001 \end{array}$	<pre>>>> bin(0b1011011 & 0b1010101) '0b1010001'</pre>
$\begin{array}{r} 1011011 \\ 1010101 \\ \hline 1011111 \end{array}$	<pre>>>> bin(0b1011011 0b1010101) '0b1011111'</pre>
$\begin{array}{r} 1011011 \\ ^ 1010101 \\ \hline 0001110 \end{array}$	<pre>>>> bin(0b1011011 ^ 0b1010101) '0b0001110'</pre>

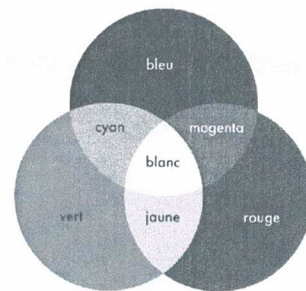
Exercice 5 : Réaliser des opérations bit à bit basiques

Compléter le tableau suivant en évaluant les opérations proposées à partir des octets x et y fournis :

Propriété	Valeur
x	01101001
y	01010101
$x \& y$	01000001
$x y$	01111101
$x \oplus y$	00111100
$x \uparrow y$	10111110

Exercice 6 : Additionner des couleurs

Sur un écran, les couleurs sont créées en mélangeant du rouge, du vert et du bleu, c'est la synthèse additive des couleurs. On imagine un dispositif dans lequel trois lampes de chacune de ces couleurs sont dirigées vers le même endroit et peuvent être allumées ou éteintes.



- a. Justifier que l'on ne peut pas créer plus de huit couleurs différentes dont les noms et les codes binaires sont donnés ci-contre.

car sur 3 bits on ne peut pas faire plus de 8 combinaisons différentes

- b. Le complément d'une couleur est obtenu en allumant les lampes éteintes et en éteignant les lampes allumées. Déterminer les couleurs complémentaires des huit couleurs précédentes.

cool ça me va

Couleur	R	V	B
Noir	0	0	0
Bleu	0	0	1
Vert	0	1	0
Cyan	0	1	1
Rouge	1	0	0
Magenta	1	0	1
Jaune	1	1	0
Blanc	1	1	1

- c. Quelle est la couleur obtenue en effectuant les opérations suivantes :

bleu | rouge

Magenta & Cyan

Vert $\uparrow$ Blanc

101

magenta

001

bleu

101

magenta

2. En Python

2.1. Définition

Un **booléen** est un type de variable à deux états, généralement nommés *Vrai* et *Faux*, et représentés respectivement par 1 et 0. En Python, on peut déclarer un booléen directement par sa valeur : « True » (pour *Vrai*) et « False » (pour *Faux*). Par exemple :

```
Couleur = True
nb = False
```

2.2. Opérateurs booléens

2.2.1. L'opérateur « and »

L'opérateur « and » permet de tester si deux booléens sont vrais.

Exemple :

```
note = 13.5
mention_ab = note >=12 and note < 14
print(mention_ab)
```

Le booléen `mention_ab` est le résultat de : `note >=12` (qui vaut *True*) **ET** `note < 14` (qui vaut *True*). D'après la table de vérité AND, `mention_ab = True`.

2.2.2. L'opérateur « or »

L'opérateur « or » permet de tester si l'un des deux booléens au moins est vrai.

Exemple :

```
note = 16.5
extreme = note < 5 or note > 15
print(extreme)
```

Ici, on teste si l'un ou l'autre des booléens `note < 5` et `note > 15` est vrai. C'est effectivement le cas donc le booléen `extreme` vaut *True*.

2.2.3. L'opérateur « not »

L'opérateur « not » retourne le booléen opposé.

Exemple :

```
couleur = True
nb = not couleur
```

Ici, on définit le booléen `nb` comme étant le contraire du booléen `couleur`. Ainsi, `nb = False`.

2.2.4. En Python : le mot clé « in »

Ce mot-clé permet de vérifier si une chaîne de caractères est contenue dans une autre.

Exemple :

```
phrase = "Le dormeur du val est un poème d'Arthur Rimbaud"
resultat = "val"
print(resultat in phrase)
```

Le programme affiche « *True* » car la chaîne de caractères « val » fait bien partie de la phrase.

Remarque : ce mot-clé est aussi utile pour des listes.

2.2.5. Assertion

Les assertions permettent de vérifier qu'un code est correct, par exemple pour tester une fonction.

Exemple :

```
1. def RacineCarree(n) :
2.     assert n >= 0, "L'argument de la fonction RacineCarree
   doit être positif"
3.     x = 0
4.     while x**2 <= n :
5.         x = x + 0.01
6.     return x
```

Si la condition $n \geq 0$ est vraie, il ne se passe rien et le reste du code s'exécute.

Si la condition $n \geq 0$ est fausse, le code cesse de s'exécuter et le message d'erreur « L'argument de la fonction RacineCarree doit être positif » apparaît.

Remarque : cette instruction peut être utilisée à l'intérieur de la fonction pour tester les pré-conditions, et à l'extérieur pour tester les post-conditions.

Exercice 7 : La valeur des booléens (sans test machine)

Dans chacun des cas suivants, donner la valeur du booléen rep.

a. $x = -3$
 $rep = x**2 == -9$

faux

b. $x, y, z = 3, 4, 5$
 $rep = x**2 + y**2 = z**2$

pas de réponse

c. $a, b = 3, -7$
 $rep = a**3 > 50 \text{ and } b**2 < 50$

faux

d. $a, b = 3, -7$
 $rep = (a**3 > 50 \text{ and } b**2 < 50) \text{ or } (a**2 < 10 \text{ and } b**2 > 10)$

vrai

Exercice 8 : Booléens et chaînes de caractères (sans test machine)**a. Qu'affiche le programme suivant ?**

```
phrase1 = "il est actuellement midi."
phrase2 = "il est temps d'aller manger."
"il" in (phrase1 and phrase2)
```

*mai***b. Qu'affiche le programme suivant ?**

```
phrase1 = "il est actuellement midi."
phrase2 = "il est temps d'aller manger."
"est" in (phrase1 and phrase2) and "na" in (phrase1 or phrase2)
```

*faux***Exercice 9 : La loi « Delta » (sans test machine)**

Soient a et b deux booléens. On définit la loi Delta telle que :

a Delta b = (a or b) and not(a and b)

a. Compléter la table de la loi Delta suivante :

a	b				Delta
0	0				0
0	1				1
1	0				0
1	1				1

b. Que se cache-t-il derrière la loi Delta ?

Exercice 10 : La fonction « Delta »

On considère le programme suivant :

```

1 def Delta(A,B,x) :
.
2     return (x in (A or B)) and not(x in (A and B))
.
3
.
4 A = "ABCDEFGHIJKLM"
.
5 B = "OKLM"
.
6
.
7 print(Delta(A,B,"K"))
.

```

a. Qu'affiche le programme ?

faux

b. Trouver une valeur de x pour qu'il affiche « True ».

A B C D E F G H I J K L M

c. Que se passe-t-il quand x = « A » si on définit la fonction « Delta » de la manière suivante :

```

def delta(A,B,x) :
    return (x in A or B) and not(x in A and B)

```

faux