Chapitre 4

Booléens

4.1 Un nouveau type de variable

Dans les systèmes digitaux (systèmes informatiques et autres automatismes numériques) toutes les données sont traitées et enregistrées à partir d'éléments d'informations binaires.
Ces informations binaires, à la manière des contacts électriques, n'ont que deux états possibles. Un contact électrique peut être ouvert ou fermé, de même le bit est une information élémentaire qui ne peut prendre que deux valeurs : 0 ou 1.

Une **variable booléenne** [1] est une variable ne pouvant prendre que 2 valeurs : **Vrai** (*True*) ou **Faux** (*False*).
La valeur *Vrai* est associée à 1, et la valeur *Faux* à 0.

En Python, les variables booléennes sont de type `bool`. Les valeurs associées sont `True` pour *Vrai* et `False` pour *Faux*.

☞ *Attention aux majuscules !*

```
>>> a = True
>>> a
True
>>> b = False
>>> b
False
>>> type(a)
<class 'bool'>
```

1. Nommée ainsi d'après *George Boole* (1815-1864), logicien, mathématicien et philosophe britannique. Il est le fondateur de la logique moderne.

En pratique, il est rare de définir directement une variable avec la valeur `True` ou `False`. Les variables booléennes sont souvent obtenues lors de **comparaisons**.
On peut par exemple comparer deux variables numériques avec les opérateurs de comparaisons mathématiques suivants :

Opérateur	Signification
`==`	Égalité
`!=`	Différence
`<=`	Inférieur ou égal
`<`	Inférieur strict
`>=`	Supérieur ou égal
`>`	Supérieur strict

▷ ***Il y a une erreur dans la première ligne du tableau, il y a 2 fois le signe égal***

Non, ce n'est pas une erreur. En Python, il faut faire la différence entre :

- `=`, qui est l'opérateur d'**affectation** (pour donner une valeur à une variable)
- `==`, qui est un opérateur de **comparaison**, permettant de savoir si deux variables sont égales

```
>>> 3 == 5
False
>>> 3*4 == 2*6
True
>>> a = 3 <= 5
>>> a
True
>>> b = 4 > 4
>>> b
False
```

Il est également possible de comparer des entiers et des flottants, mais à cause de l'imprécision sur les flottants, les résultats peuvent parfois être incohérents.

```
>>> 5 == 5.0
True
>>> 3.14 >= 3.1415
False
>>> 0.1 + 0.1 == 0.2
True
>>> 0.1 + 0.2 == 0.3
False
```

La dernière comparaison devrait afficher `True`, mais c'est sans compter sur l'approximation de 0,1 en langage machine.
Pour Python, 0,1 + 0,2 est égal à 0,30000000000000004, qui n'est pas égal à 0,3.
Attention donc à la comparaison entre flottants.

Une fonction peut tout à fait retourner un booléen, comme dans l'exemple suivant :

```
1 def egalite(a, b):
2     return a == b
```

```
>>> egalite(3*12, 6*6)
True
```

Question 01 À l'aide de l'opération modulo %, écrire une fonction `divisiblePar7(n)` qui retourne un booléen égal à `True` si `n` est divisible par 7, et `False` sinon.

```
>>> divisiblePar7(37)
False
>>> divisiblePar7(84)
True
```

Question 02 Écrire une fonction `divisiblePar(d,n)` qui retourne `True` si `n` est divisible par `d`, `False` sinon.

```
>>> divisiblePar(3, 10203)
True
>>> divisiblePar(10, 987)
False
```

En plus des entiers et flottants, il est possible de comparer des chaînes de caractères.

```
>>> "Bonjour" == "Au revoir"
False
>>> "abc" == "abc"
True
>>> "un" < "deux"
False
```

▷ ***Euh... Je n'ai pas compris le dernier...***

Deux remarques :

- Python compare ici des chaînes de caractères, et non des nombres. Il ne sait pas que `"un"` correspond au nombre 1.
- L'ordre utilisé pour comparer les chaînes de caractères est l'**ordre lexicographique**. C'est l'ordre du dictionnaire. Le mot *un* est après le mot *deux*, donc `"un" < "deux"` est faux.

Grâce à l'opérateur `in`, il est possible de savoir si un caractère ou une chaîne de caractères est contenue dans une autre :

```
>>> "b" in "bonjour"
True
>>> "na" in "banane"
True
>>> "c" in "voyelle"
False
```

Question 03 Écrire une fonction `jeVoisLaLettreE(chaine)` qui retourne `True` si le caractère *e* est dans `chaine`.

```
>>> jeVoisLaLettreE("bonjour")
False
>>> jeVoisLaLettreE("au revoir")
True
```

Question 04 La séquence de chiffres 123 apparaît-elle dans l'écriture décimale du nombre 2^{2019} ? Et la séquence 1234 ?

☞ *On pourra utiliser la fonction* `str` ...

4.2 Opérateurs et Fonctions logiques

Tout comme il existe des opérations sur les nombres (addition, multiplication...), il existe des opérations sur les booléens. Ces opérations sont qualifiées de **logiques**, car leur utilisation fait intervenir des notions de logique.
Il existe 3 opérations élémentaires sur la variables booléennes : le **ET**, le **OU** et le **NON**.

4.2.1 Opérateur *ET* (*AND*)

Considérons deux variables logiques a et b. Pour que (a ET b) soit vrai, il faut que a soit vrai et b soit vrai.
On peut résumer ces informations dans un tableau, où 0 représente la valeur *Faux* et 1 la valeur *Vrai* :

a	b	a *ET* b
0	0	0
0	1	0
1	0	0
1	1	1

Ce tableau est appelé la **table de vérité** de l'opérateur *ET*.
On peut remarquer que l'opération logique a ET b est équivalente à l'opération mathématique $a \times b$.
Ainsi, il est courant d'écrire $\boxed{a \cdot b}$ en lieu et place de a ET b.

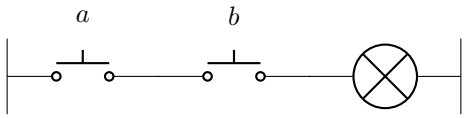
▷ ***OK, mais quel est le rapport avec l'informatique ?***

Il faut garder en tête qu'un ordinateur est composé de circuits électriques, et de fait, raisonne de manière binaire :

- le courant **ne passe pas** : correspond à l'état 0 (*Faux*)
- le courant **passe** : correspond à l'état 1 (*Vrai*)

Les seules variables que peut manipuler une machine sont les 0 et les 1, et les opérations associées sont les opérations logiques comme le *ET*. Et avec ces opérations, il est possible de réaliser des opérations mathématiques, comme des additions.

D'ailleurs, l'opérateur *ET* peut se représenter sous forme de **schéma de contacts** :



Sur le schéma précédent :

- a et b sont nos variables logiques, représentées par des **boutons poussoirs fermants**.

  Pour fermer le circuit - et laisser passer le courant - il faut appuyer sur le bouton.

- Le composant de droite est une lampe, qui s'allume lorsque le circuit est fermé.

On comprend alors que la lampe ne s'allume qu'à une seule condition : si a ET b sont enclenchés.

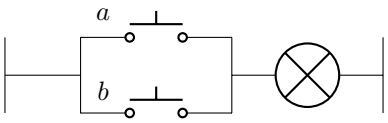
4.2.2 Opérateur *OU* (*OR*)

La table de vérité de l'opérateur *OU* est :

a	b	a *OU* b
0	0	0
0	1	1
1	0	1
1	1	1

On remarque que (a OU b) est vrai à condition que a soit vrai ou b soit vrai.
Comme pour *ET*, on utilise une notation mathématique pour *OU*. On écrit $\boxed{a+b}$ pour a *OU* b.

Le schéma de contacts associé est le suivant :

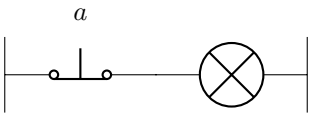


4.2.3 Opérateur *NON* (*NOT*)

Ce dernier opérateur est un peu spécial, car il ne s'applique qu'à une seule variable logique. Sa table de vérité est la suivante :

a	*NON* a
0	1
1	0

On utilise la notation $\boxed{\overline{a}}$ pour *NON* a.
Quant au schéma de contacts, on le qualifie d'**inverseur**.
Le bouton poussoir utilisé est dit **ouvrant**, car il est fermé par défaut et ouvert lorsqu'on appuie dessus :



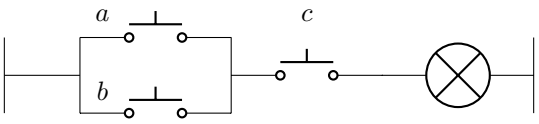
4.2.4 Fonctions logiques

En combinant les opérateurs précédents, on définit des **fonctions logiques**.

Exemple. Considérons l'opération $(a+b)\cdot c$, correspond à (a *OU* b) *ET* c. On peut écrire sa table de vérité :

a	b	c	$(a+b)\cdot c$
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

Il est même possible de représenter le schéma de contacts associé :



Question 05 Donner la table de vérité et le schéma de contacts associé de l'expression : $(a\cdot b)+\overline{c}$

Question 06 Donner les tables de vérité des expressions suivantes. Que remarque-t-on ?

$$\overline{a\cdot b} \qquad \overline{a}+\overline{b}$$

Il existe des fonctions logiques plus importantes que les autres. C'est le cas des fonctions suivantes :

La fonction *NAND* (*NON ET*)

La fonction *NAND* est définie par :

$$\boxed{a \ \textit{NAND} \ b = \overline{a\cdot b}}$$

On a l'égalité $\overline{a\cdot b}=\overline{a}+\overline{b}$.

a	b	a *NAND* b
0	0	1
0	1	1
1	0	1
1	1	0

La fonction *NOR* (*NON OU*)

La fonction *NOR* est définie par :

$$\boxed{a \ \textit{NOR} \ b = \overline{a+b}}$$

On a l'égalité $\overline{a+b}=\overline{a}\cdot\overline{b}$.

a	b	a *NOR* b
0	0	1
0	1	0
1	0	0
1	1	0

La fonction *XOR* (*OU EXCLUSIF*)

La fonction *XOR* est définie par la table de vérité ci-contre.

a	b	a *XOR* b
0	0	0
0	1	1
1	0	1
1	1	0

Question 07 Proposer une expression de a *XOR* b utilisant les opérateurs de base *ET*, *OU* et *NOT*.

4.3 Opérateurs logiques et Python

Python connaît les trois opérateurs de base `and`, `or` et `not`. On peut les utiliser avec les valeurs `True` et `False` ou avec des 0 et des 1.

```
>>> True and False
False
>>> True or False
True
>>> not True
False
>>> 0 and 0
0
>>> 1 or 0
1
>>> not 1
False
```

Cependant, l'opérateur `not` renverra `True` ou `False`, mais jamais 1 ou 0.
Toutes ces opérations sont combinables :

```
>>> (True or False) and True
True
>>> 0 and 1 and (1 or 0)
0
```

On peut créer des fonctions logiques de la façon habituelle :

```
1 def operation(a,b):
2   return not a and b
```

```
>>> operation(False, False)
False
>>> operation(True, False)
False
>>> operation(False, True)
True
>>> operation(True, True)
False
```

Question 08 Quelle est l'expression de la fonction logique `operation` ? Donner sa table de vérité.

Question 09 Définir les fonctions logiques `nand`, `nor` et `xor`.

▷ ***J'ai testé les opérations suivantes. À quoi correspondent les résultats ?***

```
>>> 5 or 9
5
>>> 4 and 2
2
>>> 0 and 3
0
```

Si vous avez pensé à "bidouiller" comme ci-dessus, c'est une bonne chose. C'est important de comprendre comment fonctionne Python, et le mieux à faire, c'est de faire des tests comme celui-ci. Expliquons tout cela :

```
>>> 5 or 9
5
```

Lorsqu'on effectue l'opération `a or b`, Python lit de gauche à droite les différentes valeurs utilisées.
Pour lui, une valeur nulle (0 ou 0.0) équivaut à `False`, le reste à `True`.
Si `a` est non nul, alors Python renvoie directement `a`, et `b` n'est « pas lu ».
Comme 5 est non nul, alors `5 or 9` renvoie simplement `5`.

```
>>> 4 and 2
2
>>> 0 and 3
0
```

Lorsqu'on effectue l'opération `a and b`, toutes les valeurs sont vérifiées, sauf si la valeur de `a` est nulle, auquel cas Python renverra sa valeur. Sinon, Python renverra la valeur de `b`.

Nous reviendrons plus en détails sur ces subtilités dans le chapitre consacré aux conditions.

Opérateurs bit à bit

Python dispose nativement d'**opérateurs bit à bit** qui agissent sur des valeurs binaires. Il s'agit des opérateurs :

- `&` : *ET*
- `|` : *OU*
- `^` : *XOR*
- `~` : *NOT*

Attention : ces opérateurs agissent sur des **valeurs binaires**. Par exemple :

```
>>> 6 & 3
2
```

Si on passe en binaire, c'est facile :

Décimal	Binaire		
6	1	1	0
3	0	1	1
6 & 3	0	1	0

Avec `&`, on réalise l'opération *ET* sur les premiers, deuxièmes et troisièmes bits des écritures binaires de 6 et 3.
En binaire, on a donc 110 & 011 = 010 soit 2 en décimal.

Les autres opérateurs s'utilisent de la même manière.

☞ *Les opérateurs bit à bit sont plus utiles qu'il n'y paraît. Nous les utiliserons notamment lorsque nous parlerons d'adresse IP et de masque réseau.*

Question 10 **Sans utiliser l'interpréteur Python**, donner le résultat des opérations suivantes :

1. `4 & 2`
2. `12 & 10`
3. `6 | 9`
4. `3 ^ 5`

4.4 QCM

Dans le QCM suivant, chaque question peut admettre **une ou plusieurs bonnes réponses**.

Questions	Réponses
1. Parmi les expressions suivantes, lesquelles sont égales à `True` ?	☐ `23 > 5` ☐ `131 <= 313` ☐ `2*6 > 3*4`
2. Même question :	☐ `12 > 10 and 1 > 3` ☐ `12 < 10 or 1 < 3` ☐ `1 > 2 or 5 > 2 or 31 < 0`
3. Même question :	☐ `"alice" <= "bob"` ☐ `"six" <= "sept"` ☐ `"passé" < "futur"`
4. Le résultat de `(1 == 2) == (3 == 4)` est :	☐ `True` ☐ `False` ☐ 5 ☐ Une erreur
5. Si a = 3 et b = 10, quelles comparaisons sont égales à `True` parmi les suivantes ?	☐ `a + b > 10` ☐ `b**a < 100` ☐ `a != b` ☐ `a + b >= a * b`
6. On considère l'opération logique `not(a or not(b or c))`. Cette opération est équivalente à :	☐ `a and b or not c` ☐ `a and not(b and c)` ☐ `not a and (b or c)`