Booléens - cours

qkzk

pdf

Le terme *booléen* vient du nom du mathématicien britannique George Boole. Il est le créateur de la logique moderne qui s'appuie sur l'algèbre qui porte désormais son nom : l'*algèbre de Boole*.

Booléen

Un booléen est un type de variable à deux états : *vrai* ou *faux*.

On note : `vrai = 1` et `faux = 0`.

En électronique, on emploie les booléens pour modéliser une situation à deux états possibles.

"Le courant passe" : 1, "le courant ne passe pas" : 0.

Cela peut aussi représenter une tension définie : +5V=1, -5V=0 ou +3V=1, -3V=0. etc.

Python et les booléens

En Python, les booléens sont `True` et `False`, ils sont du type `bool`

```
>>> True
True
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
```

Comparaison

Les opérateurs de comparaison courants sont identiques à ceux des mathématiques mais ATTENTION, il ne faut pas confondre l'égalité et l'affectation

```
variable = 5    # une affectation
5 == 8          # une égalité (qui est fausse)
```

Le résultat d'une comparaison est toujours un booléen

Comparaisons des nombres

Comparaison	Symbole	Exemple	Résultat
Égalité	`==`	`1 + 2 == 3`	`True`
Différence	`!=`	`1 + 2 != 3`	`False`
Supérieur	`>`	`4 > 3`	`True`
Inférieur	`<`	`2.2 < 2 * 3`	`True`
Supérieur ou égal	`>=`	`5 >= 6`	`False`
Inférieur ou égal	`<=`	`8 <= 3`	`False`

Appartenance à une structure

On peut tester qu'un élément appartient à une structure avec le mot clé `in`

```
"a"     in "bonjour"        # False
"bon"   in "bonjour"        # True
1       in [2, 3, 4]        # False
```

Casting

En programmation, *caster* signifie *changer le type d'une valeur*.

Par exemple : on peut caster des entiers vers les flottants :

```
>>> type(12)
<class 'int'>
>>> float(12)
12.0
>>> type(12.0)
<class 'float'>
```

Python permet de caster à peu près n'importe quoi en booléen et les booléens en entiers :

```
>>> bool(12)
True
>>> bool(0)
False
>>> int(True)
1
>>> int(False)
0
>>>
```

Plus exotique

```
>>> bool([]) # liste vide: faux
False
>>> bool([1, 2, 3]) # liste non vide: vrai
True
>>> bool("") # chaîne vide: faux
False
>>> bool("Super") # chaîne non vide: vrai
```

De manière générale, cela permet d'écrire des boucles comme ça :

```
tab = [5, 4, 3, 2, 1]
total = 0
while tab:
    x = tab.pop()
    total = total + x
```

C'est un exemple peu utile (on pourrait faire `sum(tab)` ou utiliser une boucle `for`) mais vous rencontrerez peut-être cette construction.

Fonction qui renvoie un booléen

On peut tout à fait renvoyer un booléen.

```python
def est_majeur(age: int) -> bool:
    """Vrai ssi l'age est d'au moins 18 ans"""
    return age >= 18
```

Et quand on l'exécute :

```
>>> est_majeur(22)
True
```

Comment reconnaître un débutant ?

Il écrit des trucs comme ça :

```python
def est_majeur(age):
    if age >= 18:
        return True
    else:
        return False
```

Exercice 1

Simplifier les fonctions suivantes :

def est_assez_grand(taille: int) -> bool: “‘ “La personne peut entrer si elle mesure au moins 130cm” “’ if taille < 130: return False else: return True

print(est_assez_grand(129)) print(est_assez_grand(131))

def contient_un_a(mot: str) -> bool: “‘ “Faux si la lettre ‘a’ ne figure pas dans le mot et vrai sinon” “’ if a not in mot: return False else: return True

print(contient_un_a(“elephant”)) print(contient_un_a(“rhinoceros”))

def est_vide(tableau: list) -> bool: “‘ “Vrai si le tableau ne contient aucun élément” “’ if len(tableau) > 0: return False else: return True

print(est_vide([1, 2, 3])) print(est_vide([]))

Une fonction qui renvoie un booléen est appelée un *prédicat*.

Opérations sur les booléens

Comme pour les entiers, on peut opérer sur les booléens.

Les opérateurs sur les booléens sont de deux types :

- opérateur unaire : prend *un* booléen et en renvoie *un*.
- opérateur binaire : prend *deux* booléens et en renvoie *un*.

Opérateur unaire : la négation

C'est le seul opérateur *unaire*, il donne le contraire de ce qu'on lui passe.

En français l'opérateur de négation est noté NON. En Python, l'opérateur de négation est `not`.

```python
not True    # s'évalue à False
not False   # s'évalue à True
```

Table de vérité avec `True` et `False`

a	not a
True	False
False	True

Table de vérité avec des bits

Les *tables de vérité* sont abregées en notant :

- `1` pour `True`
- `0` pour `False`

a	not a
1	0
0	1

Opérateur binaire : le OU, noté `or`

Il est vrai si l'un des deux booléens est vrai.

```
False or False   # False
False or True    # True
True  or False   # True
True  or True    # True
```

a	b	a or b
0	0	0
0	1	1
1	0	1
1	1	1

Opérateur binaire : le ET, noté `and`

Il est vrai si les deux booléens sont vrais.

```
False and False   # False
False and True    # False
True  and False   # False
True  and True    # True
```

a	b	a and b
0	0	0
0	1	0
1	0	0
1	1	1

Opérateur binaire : le XOR noté `^`

Il est vrai si EXACTEMENT un des deux booléens est vrai

```
False ^ False   # False
False ^ True    # True
True  ^ False   # True
True  ^ True    # False
```

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

Expressions complexes avec des booléens

Il est parfaitement possible de combiner plusieurs expressions booléennes pour former une expression complexe avec autant d'entrées que l'on veut.

not not not not not...

Deux négations successives s'annulent.

"Je ne suis pas en désaccord avec ce propos..." = "Je suis d'accord avec ce propos".

En logique : `not (not a) = a`

a	not a	not (not a)
0	1	0
1	0	1

Cela signifie aussi que : $a = \text{not } b \iff \text{not } a = b$

Exemple : simplifier une expression complexe

Essayons de simplifier :

`not ((not a) and (not b))`

Mais comment ?

On construit une table de vérité étape par étape.

1. Combien de colonnes ?

   - a
   - b

   - not a
   - not b
   - (not a) and (not b)
   - une pour la sortie tout à droite

   On va utiliser 6 colonnes. On réduire mais il faut au moins 1 colonne par entrée et 1 pour la sortie.

2. Combien de lignes ?

   Avec 2 entrées, il faut $2^2 = 4$ lignes.

   Avec 7 entrées, il en faudrait $2^7 = 128$, difficile...

3. Par où commencer ? Énumérer les valeurs de a et b en partant de 00 jusque 11 comme si on comptait en binaire. Ainsi :

   - c'est toujours rangé pareil,
   - on n'oublie aucun cas.

a	b	not a	not b	(not a) and (not b)	not ((not a) and (not b))
		x	y	x and y	
0	0	1	1	1	0
0	1	1	0	0	1

a	b	not a	not b	(not a) and (not b)	not ((not a) and (not b))
1	0	0	1	0	1
1	1	0	0	0	1

Reconnait-on quelqu'un ?

Examinez les colonnes d'entrée `a`, `b` et la colonne de sortie `not((not a) and (not b))`

Oui ! C'est la table de `a or b` !

Donc, on a démontré que `not((not a) and (not b)) = a or b`.

On peut aussi dire :

Première loi de Morgan :

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

Exercice 2

1. Construire une expression équivalente à `a and b` qui n'utilise que les opérateurs `not` et `or`.
2. Vérifier sa validité dans une table de vérité.

Distributivité

On connait les règles de distributivité de la multiplication des nombres : `x(y + z) = xy + xz`.

Qu'en est-il des booléens ?

Exercice 3

1. Vérifier à l'aide d'une table de vérité que `a and (b or c) = (a and b) or (a and c)`
2. Donner l'expression développée de `a or (b and c)` et la vérifier dans une table.

Lois de Morgan

On a déjà rencontré la première loi de Morgan : `not (a or b) = (not a) and (not b)`

1. Échanger les opérateurs `or` et `and` dans l'égalité précédente.
2. Déplacer le `not` tout à gauche vers le membre de droite.
3. Vérifier cette égalité dans une table de vérité.
4. Énoncer la seconde loi de Morgan.

Booléens et électronique : les portes logiques

On peut modéliser les opérations booléennes par des circuits électronique. C'est la base de l'informatique moderne.

Non logique

Cliquez sur le rond avec 0 ou 1 à gauche pour en changer l'état et voir le résultat.

Ou logique

Et logique

XOR logique

Exercice 4

1. Constuire un circuit qui prend `a` en entrée, un bit valant 0 ou 1 et qui renvoie *toujours* 1.
2. Modifier votre circuit pour qu'il renvoie *toujours 0.*
3. Traduire vos deux circuits en expressions logiques.

Exercice 5

On a illustré certaines des expressions de l'exercice 3.

1. Reconnaître les règles et expressions déjà rencontrées
2. Tester tous les cas en changeant les valeurs des entrées et vérifier la validité

Exercice 6

1. Construire les schéma electroniques illustrant les deux autres règles
2. Vérifier que vos circuits sont valides

Exercice 7 : Demi additionneur

1. Rappeler les tables de `XOR` et `AND`

Lorsqu'on additionne deux bits, on obtient un nombre pouvant occuper 2 bits. Par exemple : `1 + 0 = 1` (un bit) et `1 + 1 = 2 = 0b10` (deux bits).

2. En notant `s` la somme (le dernier bit) et `r` la retenue (le bit de gauche), construire dans une seule table la table de `s` et de `r`
3. Comparer avec les tables de `a xor b` et `a and b`
4. Dans le circuit ci-dessous, connecter `a` et `b` à `a and b` et à `a xor b`
5. Vérifier la table de vérité écrite plus haut.

Additionneur complet

Un additionneur complet contient trois entrées : `a`, `b` et `c0` et a 2 sorties `s` et `c1`.

- `a` et `b` sont les bits des nombres qu'on ajoute et `c0` est la retenue du calcul précédent.
- `s` est le bit de la somme à cette position et `c1` est la retenue qui part dans la somme suivante.
- Cette retenue `c1` est calculée en faisant un `or` entre `a and b` et `c0 and (a xor b)` : `c1 = (a and b) or (c0 and (a xor b))`.

Additionneur 4 bits

Comment réaliser l'addition de nombres à plusieurs bits ? La méthode ci-dessous, appelée "*propagation des retenues*" est simple, fastidieuse et peu efficace. Il existe une méthode beaucoup plus rapide que nous n'aborderons pas.

Un additionneur complet sur 4 bits est obtenu en raccordant successivement :

un demi additionneur pour les bits de poids faibles à trois additionneurs complets pour les bits suivant. Le bit de poids fort de la somme est la dernière retenue.

En notant `a = a3 a2 a1 a0` les bits de `a` et de la même manière les bits de `b` et `s = a + b`, on peut dessiner le circuit plus bas.

Utilisation :

- Les bits d'entrée sont à gauche et se lisent de haut en bas, le bit de poids faible est en haut.
- Les bits de sortie sont à droite et se lisent aussi de haut en bas.
- colonne tout à gauche : les bits de `a`. On peut lire `a = 0b1101 = 13`
- colonne juste à droite : les bits de `b`. On lit `b = 0b1001 = 9`

a	a3	a2	a1	a0
valeur	1	1	0	1

b	b3	b2	b1	b0
valeur	1	0	0	1

Donc la somme :

```
  a     1 1 0 1
+ b     1 0 0 1
---------------
= s   1 0 1 1 0
```

$13 + 9 = 22 = 16 + 4 + 2 = 0b10110$

- colonne de droite, les bits de `s=a+b`. On lit de haut en bas : `s = 0b11000`. C'est correct.

s = a+b	s4	s3	s2	s1	s0
valeur	1	1	0	0	0

Changez les bits d'entrée et vérifiez quelques valeurs.

Vous pouvez vérifier ici que les branchements sont corrects.

Propriétés mathématiques de l'algèbre de Boole

On peut résumer ainsi ce qu'on a vu sur les booléens :

Définition

On considère l'ensemble `{0, 1}` ou `{False, True}` muni de trois opérations :

la **négation** `not`, le **et logique** `and`, le **ou logique** `or`.

Elles sont définies par les tables de vérité présentées plus haut.

Complémentarité

- `not(not(a)) = a`
- `a or (not a) = 1`
- `a and (not a) = 0`

Associativité

- `a or (b or c) = (a or b) or c`
- `a and (b and c) = (a and b) and c`

Distributivité

- `a or (b and c) = (a and b) or (a and c)`
- `a and (b or c) = (a or b) and (a or c)`

Ainsi, on retrouve les propriétés des opérations sur les nombres. Pour simplifier, `or` *se comporte comme une somme et* `and` *comme un produit.*

Lois de Morgan

- `not (a and b) = (not a) or  (not b)`
- `not (a or  c) = (not a) and (not b)`