Lycée Jean Moulin **Draguignan**
NSI 1re Année .....

Python – Algèbre de Boole

Les booléens et Python

En Python, les booléens sont représentés par les valeurs **True** et **False**. Les opérations booléennes de bases sont **and**, **or** et **not**.

```
>>> True and False
False
>>> True or False
True
>>> not True
False
```

Comme pour les opérations mathématiques, il y a des priorités sur les opérations booléennes.

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

Le **and** est prioritaire sur le **or**. De même, **not** est prioritaire sur les autres opérations.

```
>>> not False and False
False
>>> not (False and False)
True
>>> (not False) and False
False
```

En programmation, les booléens sont en général obtenus à l'aide de tests. Les tests de bases sont des tests arithmétiques mais il y a d'autres opérations donnant des booléens (appartenance à une chaîne de caractères ou une liste, inclusion d'un ensemble dans un autre...)

En math	En python
$a = b$	`a == b`
$a \neq b$	`a != b`
$a > b$	`a > b`
$a \geq b$	`a >= b`
$a < b$	`a < b`
$a \leq b$	`a <= b`

Exercice 1 : Python permet d'écrire directement $a < b < c$. Écrire cette expression à l'aide de deux comparaisons et d'un opérateur logique.

Exercice 2 :

1) Expliquer pourquoi l'expression "`3 == 3 or x == y`" est vraie pour toute valeur de `x` et de `y`.
2) Expliquer pourquoi l'expression "`1 == 2 and x == y`" est fausse pour toute valeur de `x` et de `y`.

Lorsque Python évalue une expression booléenne, il le fait de façon **paresseuse**. C'est à dire que si la partie gauche d'un **or** est vraie, il n'évalue pas la partie droite. De même si la partie gauche d'un **and** est fausse, la partie droite n'est pas évaluée.
Cela permet d'écrire les choses suivantes :

```
>>> x = 0
>>> x == 0 or 0 < 1/x < 1
True
>>> x != 0 and 0 < 1/x < 1
False
```

Si la division `1/x` était évaluée, il y aurait une erreur, puisqu'on le peut pas diviser par 0. Mais dans les deux cas, l'évaluation n'est pas faite puisque le résultat de l'expression a déjà pu être déterminée avec la partie gauche.

Tout est un booléen

Dans Python, n'importe quelle valeur ou objet peut être converti en booléen à l'aide de la commande **bool**.

```
>>> bool(0)
False
>>> bool(1)
True
>>> bool(2)
True
>>> bool(-1)
True
>>> bool(0.0)
False
>>> bool("False")
True
>>> bool("")
False
```

Pour les nombres, seul 0 correspond à **False**. Toute valeur non nulle correspond à **True**. L'expression **bool**`(a)` est donc équivalente à `a!=0`.
Pour les chaînes de caractères, **bool**`(texte)` est donc équivalente à `texte != ""`.
La conversion peut également se faire de façon implicite, mais attention à ne pas rendre son programme illisible dans ce cas là :

```
def bonjour(nom):
    if nom:
        print("Bonjour", nom)
    else:
        print("Le nom est vide")
```

Algèbre de Boole

Nous allons recréer les fonctions booléennes et les booléens dans Python, tout en évitant d'utiliser ceux déjà définies.
Les booléens seront représentés par les entiers `0` et `1`. Pour construire les opérateurs booléens, nous allons juste utiliser l'égalité sur les entiers.

Il y a plusieurs façons de construire les opérateurs booléens. On pourrait directement définir **et**, **ou** et **non**, mais nous allons commencer par définir une seule opération, **nand**, qui correspond à **non et**. Cette opération est **universelle**. C'est à dire qu'elle permet de définir toutes les autres.

a	b	a nand b
0	0	1
0	1	1
1	0	1
1	1	0

Dans un fichier `feuille-algebre-boole-NOM-PRENOM.py`, rajouter la fonction suivante :

```
def nand(a, b):
    if a == 0:
        return 1
    if b == 0:
        return 1
    return 0
```

La fonction retourne 1 dès qu'un des deux paramètre vaut `0`. Sinon, elle retourne `0`.
Afin d'obtenir la table de vérité de cet opérateur, nous allons créer une fonction qui affiche la table de valeur de n'importe quelle fonction booléenne à 2 paramètres :

```
def table2(f):
    print("a", "b", "f(a,b)")
    for a in range(2):
        for b in range(2):
            print(a, b, f(a,b))
```

On obtient :

```
>>> table2(nand)
a b f(a,b)
0 0 1
0 1 1
1 0 1
1 1 0
```

On retrouve bien la même table de vérité.
À partir de **nand**, il est facile de définir **non**. En effet, on remarque que a nand a est équivalent à non a. On rajoute donc :

```
def non(a):
    return nand(a, a)
```

On peut ensuite définir le **et**.

```
def et(a, b):
    return non(nand(a, b))
```

Et on peut vérifier la table de vérité :

```
>>> table2(et)
a b f(a,b)
0 0 0
0 1 0
1 0 0
1 1 1
```

Exercice 3 : En utilisant **nand** et **non**, définir la fonction `ou(a, b)` correspondant à l'opérateur **ou**.

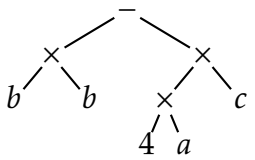
Préfixée, infixée ou postfixée ?

On peut remarquer qu'avec nos opérateurs booléens, nous devons mettre l'opérateur avant les deux valeurs :

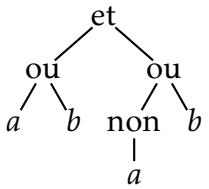
- a **et** b devient `et(a, b)`
- a **ou** b devient `ou(a, b)`
- a **ou** (b **et** c) devient `ou(a, et(b,c))`

On dit que la notation est **préfixée**. C'est à dire qu'on met d'abord l'opérateur et ensuite les paramètres. La notation "classique" est appelle **infixée**. Il existe également une notation **postfixée**.

Par exemple, en Postscript, le langage développé par Adobe à partir de 1982 pour décrire les documents et utilisé par la plupart des imprimantes, les expressions mathématiques sont postfixée.
Ainsi "`b b mul 4 a mul c mul sub`" correspond à $b^2 - 4ac$. Cela peut se traduire par l'arbre ci-contre.



De la même façon, on peut représenter les expressions booléeenes afin ensuite de les écrire sous forme préfixée. Ainsi, l'expression (a **ou** b) **et** ((**non** a) **ou** b) correspond à l'arbre ci-contre. Ce qui se traduit par la notation préfixée `et(ou(a,b), ou(non(a),b))`.



Pour obtenir la table de vérité de cette expression, on peut définir une nouvelle fonction :

```
def expression(a,b):
    return et(ou(a,b),ou(non(a),b))
```

Puisqu'il y a des chances pour qu'on n'utilise cette fonction qu'une seule fois, il est possible d'utiliser une **fonction anonyme** :

```
>>> table2(lambda a,b : et(ou(a,b),ou(non(a),b)))
a b f(a,b)
0 0 0
0 1 1
1 0 0
1 1 1
```

La notation est :

`lambda` PARAM : EXPRESSION

S'il y a plusieurs paramètres, il faut les séparer par des virgules.
C'est l'équivalent de la notation mathématique $f : x \mapsto x^2$ pour $f(x) = x^2$.
En Python, cela se noterait `lambda x : x**2`.

Exercice 4 : À l'aide des fonctions que vous venez de définir, rajouter une fonction `xor(a,b)` qui correspond à a **xor** b.

Exercice 5 : À l'aide de `table2(f)`, afficher les tables de valeurs des expressions de l'exercice 4 de la feuille sur l'algèbre de Boole.

Exercice 6 : En vous inspirant de `table2(f)`, définir une fonction `table3(f)` qui affiche la table de vérité d'une fonction à 3 paramètres.

Exercice 7 : À l'aide des fonctions que vous venez de définir, rajouter une fonction `impl(a,b)` qui correspond à $a \Rightarrow b$.

Exercice 8 : À l'aide des fonctions `table2(f)` et `table3(f)`, afficher les tables de vérités des axiomes de Hilbert et vérifier que toutes les expressions sont toujours vraies.

Pour aller plus loin

Il est possible de définir les fonctions booléennes sans utiliser **nand** et en définissant directement **et**, **ou**, **non**, **xor** et $\Rightarrow$.

Exercice 9 : Proposer une autre définition de **et**, **ou** et **non** en n'utilisant que des tests d'égalité, des `if` et des `return`, en vous inspirant de la définition de la fonction `nand(a,b)`.

Exercice 10 : Proposer une autre définition de **et** et **ou** en n'utilisant que les fonctions `min(a,b)` et `max(a,b)` prédéfinies en Python.

Exercice 11 : Une **Tautologie** est une formule qui est toujours vraie. Par exemple "1 ou a" est vraie quelque soit a.
Créer des fonctions `tauto2(f)` et `tauto3(f)` qui retourne `True` si la fonction `f` est une tautologie et `False` sinon. La fonction `tauto2(f)` prend en paramètre des fonctions à 2 paramètres et `tauto3(f)` des fonctions à 3 paramètres.