Chapitre 2 : Logique Booléenne

I. Introduction à la logique booléenne

1. Fonctions booléennes

On appelle booléens les objets qui sont soient "Vrai" (`True`) soit "Faux" (`False`).

Dans la suite, on notera par $\mathcal{B} = \{Vrai, Faux\}$ l'ensemble des booléens.

Les fonctions booléennes sont des applications dont l'ensemble de départ est de la forme $\mathcal{B}^n$, et l'ensemble d'arrivée est $\mathcal{B}$.

C'est à dire que ce sont des fonctions d'une ou plusieurs valeurs booléennes, dont la valeur est toujours soit "vrai" soit "faux".

C'est par exemple le cas de la fonction Ou définie ci-dessous :

$$Ou : \begin{cases} \mathcal{B}^2 \to \mathcal{B} \\ (A,B) \mapsto \begin{cases} Faux \text{ si } (A,B) = (Faux, Faux) \\ Vrai \text{ sinon} \end{cases} \end{cases}$$

Questions :

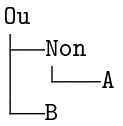
1. Définir de même les fonctions $Et : \mathcal{B}^2 \to \mathcal{B}$ et $Non : \mathcal{B} \to \mathcal{B}$.
2. On pourrait aussi définir la fonction Et sur $\mathcal{B}^3$ ou une puissance arbitraire de $\mathcal{B}$. Quelle en serait la définition ?
3. Même question pour la fonction Ou.

2. Arbre

Une des façons de produire d'autres fonctions booléennes consiste à imbriquer entre elles les fonctions "Et", "Ou", et "Non" définies précédemment.

Les expressions ainsi obtenues seront représentées par des arbres :

Par exemple, l'arbre



représente l'expression "(non A) ou B"

3. Table de vérité

On associe à chaque fonction sa table de vérité, c'est à dire la liste des valeurs qu'elle prend en fonction des arguments (conventionnellement notés "A", "B", etc) qu'on lui donne.

Par exemple, la fonction Ou définie précédemment a la table de vérité ci-dessous :

A	B	Ou(A,B)
False	False	False
False	True	True
True	False	True
True	True	True

Déterminer la table de vérité des expressions $A \Rightarrow B$ et $(NonB) \Rightarrow (NonA)$. En déduire que ces deux expressions correspondent à la même fonction.

II. Fonctions python pour afficher des données

1. Affichage de tableaux

La fonction ci-dessous (que l'on ne vous demande pas de comprendre en détail)

```
def affiche_tableau(tab):
  nblign,nbcol=len(tab),len(tab[0]) #nb de lignes et de colones
  largeur=[max(len(str(tab[i][j])) for i in range(nblign))
    for j in range(nbcol)] #la largeur maximale de chaque colonne
  for l in range(2*len(tab)+1): #afichage ligne par ligne:
    if l%2==0: # affichage d'une ligne horizontale
      if l==0: g,m,d="┌┬┐"
      elif l==2*len(tab): g,m,d="└┴┘"
      else: g,m,d="├┼┤"
      print(g+m.join(["─"*larg for larg in largeur])+d)
    else:
      print("│"+"│".join([case+" "*(largeur[col]-len(case))
        for col in range(nbcol) for case in [str(tab[l//2][col])]])+"│")
```

permet d'afficher des tableaux (on l'utilisera pour afficher des tables de vérités).

Le tableau à afficher est donné comme une liste de listes de chaînes de caractères correspondant au lignes successives, comme dans l'exemple ci-dessous :

```
tbl=[["Nom","Prenom"],["Dalton","Joe"],["Dalton","William"],
     ["Dalton","Jack"],["Dalton","Averell"]]
affiche_tableau(tbl)
```

2. Affichage d'arbres

La fonction ci-dessous (que l'on ne vous demande pas de comprendre en détail)

```
def affiche_arbre(arb,prefix=""):
    print(prefix+str(arb[0]))
    branches,nb_branches=arb[1:],len(arb)-1
    for i in range(nb_branches):
        branche,prf=branches[i],prefix
        for c,d in [("├","│"),("─"," "),("└"," ")]:
            prf=str.replace(prf,c,d)
        if i==nb_branches-1: prf=prf+"└"
        else: prf=prf+"├"
        prf=prf+"─"*len(str(arb[0]))
        affiche_arbre(branche,prf)
```

permet d'afficher un arbre, qui est donné sous la forme d'une liste de listes, comme dans les exemples ci-dessous :

```
arbre1=["Ou", ["Non","A"], "B"]
arbre2=["Et",arbre1,["Ou", "A",["Non","B"]]]
affiche_arbre(arbre1)
affiche_arbre(arbre2)
```

On notera que cette fonction prend deux arguments : "`arb`" (l'arbre que l'on souhaite représenter) et "`prefix`" (qui est uniquement utilisé dans les appels récursifs de la fonction). Mais si le deuxième argument n'est pas donné, alors python considère par défaut qu'il vaut `""` (la chaîne de caractères vide).

3. Code ASCII

La norme ASCII est une façon de stocker du texte sur un ordinateur. Les ordinateurs manipulent des suites de "0" et de "1", qui correspondent habituellement à des nombres (écrits en binaires).

Les textes sont alors manipulés comme des suite de nombres, en ayant choisi une convention qui indique quelle lettre correspond à quel nombre (cette convention s'appelle ASCII).

En python ; la fonction `chr` indique quel caractère est associé à un nombre, alors que la fonction `ord` indique quel nombre est associé à un caractère (voir exemple ci-dessous) :

```
for i in "ABCDEFGH":
    print(ord(i))
```

Complétez le code ci-dessous afin de définir une fonction `lettre` telle que `lettre(n)` renvoie la $n^{\text{ème}}$ lettre de l'alphabet (écrite en majuscule).

Par exemple `lettre(2)` doit renvoyer "B"

```
def lettre(n):
    return chr(...)
```

III. Exercices

1. Exemple de fonctions Booléennes

1. Fonction "$\Leftrightarrow$"
   (a) Écrire la table de vérité de l'expression $A \Leftrightarrow B$.
   (b) Écrire la table de vérité de $(A\,et\,B)\,ou\,(\,(Non\,A)\,et\,(Non\,B))$
   (c) En déduire un arbre associé à la fonction "$\Leftrightarrow$"
   (d) Justifiez que les expressions $A \Leftrightarrow B$ et $(NonA) \Leftrightarrow (NonB)$ sont associées à la même fonction.
2. Réécritures
   (a) En utilisant uniquement les fonction "Ou" et "Non", écrire un arbre associé à l'expression $A\,Et\,B$.
   (b) En utilisant uniquement les fonction "Et" et "Non", écrire un arbre associé à l'expression $A\,Ou\,B$.
   (c) On définit la fonction Ni par $Ni(A,B) = Et(Non\,A\,,Non\,B)$. En utilisant uniquement cette fonction Ni, écrire des arbres pour les expression $Non\,A$, $A\,Et\,B$ et $A\,Ou\,B$.

2. Négation d'une fonction booléenne

Écrire une fonction qui à partir d'un arbre (auquel on pourrait associer une fonction f), renvoie un arbre associe un arbre associé à la fonction $Non\, f$.

Une option est évidemment d'ajouter un "Non" au début de l'arbre sans rien changer au reste de l'arbre, mais on attendra de vous une fonction qui modifie l'arbre plus significativement.

3. Évaluation paresseuse

Il arrive que certains calculs inutiles puissent être évités. Par exemple quand on détermine $A\, Ou\, B$: si A vaut `True` alors il est inutile de calculer `B`, on peut directement affirmer que $A\, Ou\, B$ vaut `True`.

En informatique, l'expression *évaluation paresseuse* désigne le fait d'éviter ainsi des calculs qui ne sont pas nécessaires pour évaluer une expression.

La fonction ci-dessous permettra de savoir ce que Python a calculé (ou n'a pas calculé) :

```
def evalue(val,position):
    print("python a évalué le terme "+position+" qui était ",val)
    return val
```

Par exemple, en exécutant le code ci-dessous :

```
evalue(True,"de gauche") or evalue(False,"de droite")
```

on se rend compte que pour calculer `True or False` python renvoie `True` après avoir vu le premier terme, sans regarder le terme suivant.

En revanche, en exécutant le code ci-dessous

```
evalue(True,"de gauche") and evalue(False,"de droite")
```

on se rend compte qu'il évalue cette fois ci les deux termes.

1. Écrire une fonction `implique` telle que `implique(A,B)` revoit `True` si $A \Rightarrow B$ est vrai et `False` sinon. On demande d'écrire cette fonction de façon *paresseuse*.
2. De même, définir une fonction `equivalent`. Peut-elle être *paresseuse* ?
3. (a) Définir de même des fonctions `et` et `ou` qui prennent comme argument une liste de valeur Booléennes. Par exemple `et([True,False,True])` devra renvoyer `False`.
   (b) En utilisant la fonction `evalue` constatez que cela ne permet pas d'obtenir une évaluation *paresseuse*.
   (c) À l'aide de la fonction `evalue` constatez que les fonction `any` et `all` calculent elles aussi les fonctions Et et Ou, mais que si l'argument est un générateur au lieu d'une liste, alors l'évaluation est paresseuse.

4. Calcul d'une table de vérité

4.1) Valeur booléenne d'un arbre

Étant donné un arbre "`arb`" et une liste "`l`" de booléens, la fonction ci-dessous évalue l'arbre lorsque A=`l[0]`, B=`l[1]`, C=`l[2]`, etc. On notera qu'il utilise les fonction `ou` et `et` définies dans l'exercice précédent.

```
def eval_arbre(arb,l):
    if arb[0]=="Ou":return ou(
        [eval_arbre(branche,l) for branche in arb[1:]])
    elif arb[0]=="Et":return et(
        [eval_arbre(branche,l) for branche in arb[1:]])
    elif arb[0]=="Non":return False==eval_arbre(arb[1],l)
    for i in range(len(l)):
        if arb==lettre(i+1): return l[i]
    if arb in [True,False]: return arb
```

par exemple, on définit l'arbre `equiv` ci-dessous :

```
equiv=["Et",["Ou", ["Non","A"], "B"],["Ou", "A",["Non","B"]]]
```

Question : Déterminer (avec un papier et crayon) ce que fait ce code quand on exécute `eval_arbre(equiv,[True,False])`, afin de vous convaincre que cette fonction évalue bien les arbres correctement.

4.2) Liste des cas

Afin de calculer une table de vérité, il faut tout d'abord lister les différentes lignes de la table, c'est à dire déterminer toutes les valeurs possibles pour les variables A, B, C, ...

Définissez à cette fin une fonction qui, quand on lui donne le nombre de variables, détermine cette liste des cas à considérer.

Par exemple `liste_cas(2)` doit produire la liste `[[False, False], [False, True], [True, False], [True, True]]` (ou ces même cas dans un autre ordre)

4.3) Détermination d'une table de vérité

La fonctions "`table_verite`" définie ci dessous affiche la table de vérité d'une liste de fonctions et d'une liste d'arbres, à condition qu'on lui indique le nombre de variables.

```
def table_verite(fonctions,arbres,nombre_de_variables):
    for i in range(len(arbres)):
        print("a"+str(i+1)+" designe l'arbre ci-dessous :")
        affiche_arbre(arbres[i])
    tbl=[[chr(ord("A")+i) for i in range(nombre_de_variables)]+[""]+
         [f.__name__ for f in fonctions]+
         ["a"+str(i+1) for i in range(len(arbres))]]
    for valeurs in liste_cas(nombre_de_variables):
        tbl=tbl+[valeurs+[""]+[f(valeurs) for f in fonctions]+
                 [eval_arbre(a,valeurs) for a in arbres]]
    print(tbl)
    affiche_tableau(tbl)
```

Par exemple, on peut déterminer ci-dessous les tables de vérité des fonctions `equivalent` et `implique` (définies précédemment) et des trois arbres :

```
arbre3=["Ou", "A", "B"]
table_verite([equivalent,implique],[arbre1,arbre2,arbre3],2)
```

Question : Déduire que l'arbre `arbre1` défini ci-avant correspond à l'expression $A \Rightarrow B$, et que l'arbre `arbre2` correspond à l'expression $A \Leftrightarrow B$.

De même, à quelle fonction correspond l'arbre ci-dessous ?

```
Non
└──Et
   ├──A
   └──Non
      └──B
```

En déduire que deux arbres distincts peuvent correspondre à la même fonction.

5. Détermination d'un arbre pour une fonction booléenne arbitraire

Considérons la fonction f dont la table de vérité est donnée ci dessous :

A	B	f
False	False	False
False	True	True
True	False	False
True	True	False

Elle ne prend la valeur `True` que si A=`False` et B=`True`. Elle correspond donc à l'arbre

```
Et
├──Non
│  └──A
└──B
```

De même, s'il y avait eu deux lignes pour lesquelles f vaut `True`, on aurait trouvé un arbre pour chacune de ces deux lignes, et on les aurait regroupé avec un "Ou", comme pour la table de vérité ci-dessous :

A	B	f
False	False	False
False	True	True
True	False	True
True	True	False

pour laquelle la première ligne `True` correspond à l'arbre

```
Et
├──Non
│   └────A
└──B
```

tandis que la seconde ligne `True` correspond à l'arbre

```
Et
├──A
└──Non
    └────B
```

En conséquence la table de vérité indiquée correspond à l'arbre

```
Ou
├──Et
│   ├──Non
│   │   └────A
│   └──B
└──Et
    ├──A
    └──Non
        └────B
```

Dans ce qui suit, on cherche à construire un arbre pour n'importe quelle table de vérité.

1. Déterminer à la main (sans l'ordinateur) quel arbre obtiendrait cette méthode pour l'expression "A et (B⇒C)"
2. Écrire un programme qui construit un arbre reproduisant une fonction arbitraire (dont on lui indique le nombre de variables). Vous prendrez prendra garde à écrire un programmes qui fonctionne même quand il y a plus de deux variables (on pourrait avoir trois variables A, B, C comme on pourrait en avoir 5, ou bien une seule).

Par exemple, `construire_arbre(ou,2)` doit construire l'arbre

```
Ou
├──Et
│   ├──Non
│   │   └────A
│   └──B
├──Et
│   ├──A
│   └──Non
│       └────B
└──Et
    ├──A
    └──B
```

Conclusions

L'existence de cet algorithme indique que toute fonction booléenne peut s'exprimer à partir des fonctions "et", "ou" et "non"

1. Montrer que "ou" peut se construire à partir de "et" et "non". En déduire que les fonctions "et" et "non" sont suffisantes pour produire toutes les fonctions booléennes.
2. De même montrer que les fonctions "ou" et "non" sont suffisantes pour produire toutes les fonctions booléennes.
3. Enfin montrer que la fonction "Ni" (définie dans le premier exercice) est suffisante pour produire n'importe quelle fonction booléenne.