TD2 : Logique Booléenne

Préliminaire : 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'apelle 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)` renvoit la $n^{\text{ème}}$ lettre de l'aphabet (écrite en majuscule).

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

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

Affichage d'un arbre

On appelle booléens les objets qui sont soient "vrais" (`True`) soit "faux" (`False`)

On peut manipuler ces objets par le biais de fonctions "Et", "Ou", etc, que l'on peut imbriquer les uns dans les autres pour former des expressions complexes.

Les expressions ainsi obtenues seront représentées par des arbres :

Par exemple, l'arbre

```
Ou
├──Non
│   └────A
└──B
```

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

On va considérer ces arbres come des listes imbriquées les unes dans les autres. Par exemple,

`["Ou", "A", "B"]` désigne l'arbre

```
Ou
├──A
└──B
```

qui représente l'expression "A ou B"

De même, `["Ou", ["Non","A"], "B"]` désigne l'arbre précédant, qui correspond à l'expression "(non A) ou B"

La fonction ci-dessous affiche l'arbre qu'on lui demande :

```
def affiche_arbre(arb,prefix=""):
    print(prefix+str(arb[0]))
    branches=arb[1:]
    nb_branches=len(branches)
    for i in range(nb_branches):
        branche=branches[i]
        prf=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)
```

À noter : cette fonction prend deux arguments : "`arb`" et "`prefix`". Mais si le deuxième argument n'est pas donné, alors python considère par défaut qu'il vaut `""` (la chaîne de caractères vide)

Par exemple, on peut afficher les arbres suivant

```
arbre_1=["Ou", ["Non","A"], "B"]
arbre_2=["Et",arbre_1,["Ou", "A",["Non","B"]]]
affiche_arbre(arbre_1)
affiche_arbre(arbre_2,"   --         ")
```

Question optionnelle : comprendre le fonctionnement de cette fonction et expliquer le rôle de l'argument `prefix`

Affichage d'un tableau

Dans la suite, on calculera les tables de vérités de différentes fonctions booléennes.
Pour cela, il est utilise de disposer d'une fonction qui affiche des tableaux :
Les "tableaux" seront stockés comme des "listes de listes" :
la première liste est la première ligne, la deuxième liste est la deuxième ligne, etc

```
def affiche_tableau(tab):
    nblign=len(tab) #nombre de lignes du tableau
    nbcol=len(tab[0]) #nombre de colonnes du tableau
    largeur=[]  #on va y calculer la largeur maximale de chaque colonne
    for j in range(nbcol):
        largeur=largeur+[max(len(str(tab[i][j])) for i in range(nblign))]
    #afichage ligne par ligne:
    for ligne in tab:
        l="";m=""
        for col in range(nbcol):
            case=str(ligne[col])
            l=l+case+" "*(largeur[col]-len(case))+"│"
            m=m+"─"*largeur[col]+"┼"
        print(l)
        print(m)
```

Par exemple, on peut afficher le tableau suivant

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

affiche_tableau(tbl)
```

Exemple de fonction booléenne : A⇒B

```
def implique(l):
    if l[0]: return l[1]
    else: return True
def et(l):
    if l[0]:
        if len(l)>1: return et(l[1:])
        else: return True
    else: return False
```

1. **Expliquez en quoi "`implique([A,B])` calcule bien `A⇒B`**
2. **Noter que `et(l)` détermine si tous les éléments de `l` sont `True`**
3. **Écrire de même des fonction "`ou`" et "`equivalent`".**

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

```
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 definit l'arbre `equiv` ci-dessous :

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

Determiner (avec un papier et crayon) ce que fait ce code quand on execute `eval_arbre(equiv,[True,False])`, afin de vous convaincre de ce que fait cette fonction

Liste de cas

On souhaite établir des tables de vérités. Par exemple pour "$A \Leftrightarrow B$" on aimerait produire la table ci-dessous :

A	B	A<=>B
False	False	True
False	True	False
True	False	False
True	True	True

Définir tout d'abord une fonction qui, quand on lui donne le nombre de variables, détermine la liste des cas à considérer dans la table de vérité

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

Table de vérité de fonctions et d'arbres

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]]
    affiche_tableau(tbl)
```

exemple :

```
arbre_3=["Ou", "A", "B"]
table_verite([equivalent,implique],[arbre_1,arbre_2,arbre_3],2)
```

Déduire que l'arbre `arbre_1` **défini ci-avant correspond à la fonction** `A⇒B`**, et que l'arbre** `arbre_2` **correspond à la fonction** `A⇔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.

Arbre correspondant à une table de vérité fixée

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

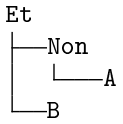
Elle ne prend la valeur `True` que si A=`False` et B=`True`. Elle correspond donc à l'arbre

```
Et
├──Non
│  └──A
└──B
```

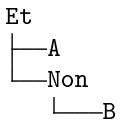
De même, s'il y avait eu deux lignes pour lesquelles f vaut `True`, on aurait trouvé un arbre pour chacune de ces deux lignes, et on les aurait regroupé avec un "Ou", comme pour la table de vérité ci-dessous :

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

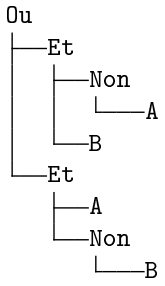
pour le quel la première ligne `True` correspond à l'arbre



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



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



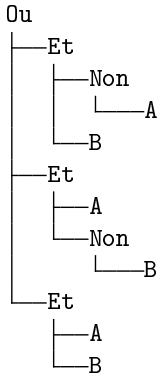
Dans ce qui suit, on cherche à construire un arbre pour n'importe quelle table de vérité.

En répondant aux questions ci-dessous, on prendra garde que les programmes que l'on écrit devront fonctionner 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)

Déterminer à la main (sans l'ordinateur) quel arbre obtiendrait cette méthode pour l'expression "A et (B⇒C)"

Écrire un programme qui construit un arbre reproduisant une fonction arbitraire (dont on lui indique le nombre de variables)

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



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. On appelle "Ni" la fonction qui à A et B associe " `Non(A ou B)`**". Montrer que cette fonction est suffisante pour produire n'importe quelle fonction booléenne.**