Semestre 2 – Sujet n°1
Problème 1. Pédagogie de l'informatique

1.1 Analyse critique d'un document

Le document 1 (pages 2 à 4) sont des extraits d'un manuel scolaire de première NSI concernant un cours sur les booléens.

Question 1. En vous appuyant sur les programmes (fournis séparément), précisez les points du programme abordés par ce cours.

Question 2. Effectuer une analyse critique, paragraphe par paragraphe (on prendra soin de bien désigner chaque paragraphe), de ce cours. En particulier :
- relever les erreurs, imprécisions au niveau du vocabulaire ;
- relever les erreurs conceptuelles ;
- expliquer comment de telles erreurs peuvent impacter l'apprentissage des élèves.

Question 3. Expliciter le plan choisi pour ce cours et le critiquer.

Question 4. Éventuellement, préciser les points positifs de ce cours.

1.2 Rédaction d'énoncés

Les documents 2 (pages 5 et 6) et 3 (page 7) sont extraits du même manuel de NSI. Ce sont des exercices concernant les booléens.

Question 5. En vous appuyant sur l'analyse de programmes de première NSI, discuter de l'adéquation de chaque exercice avec la compétence qui lui est associée.

Question 6. Que pensez-vous de la correction de chaque exercice du document 3 fournie dans le document 4 (page 8) ?

Question 7. Proposer un exercice et sa correction pour la compétence « manipuler des booléens ».

Question 8. Proposer un exercice et sa correction pour la compétence « savoir interpréter une expression booléenne ».

Question 9. Proposer un exercice et sa correction pour la compétence « Utiliser des instructions conditionnelles » (en lien avec les booléens).

Permutation de deux variables

Python permet la permutation du contenu affecté à deux variables de manière très aisée.

```
1.  a = 1 #a vaut 1
2.  print("a vaut:",a)
3.  b = 2 #b vaut 2
4.  print("b vaut:",b)
5.  a,b = b,a #permutation
6.  print("a vaut:",a,", b vaut:",b)
```

Après exécution, l'affichage dans le shell permet de vérifier la permutation effectuée :

```
a vaut: 1
b vaut: 2
a vaut: 2 , b vaut: 1
```

6 Un autre type de base : les booléens

Valeurs booléennes : True et False

Un booléen est une variable qui peut prendre deux états : soit l'état vrai (`True`) qui correspond à 1, soit l'état faux (`False`) qui correspond à 0. L'intérêt des booléens en informatique est qu'ils permettent de tester si une expression logique est vraie ou fausse.

Pour s'exercer sur quelques tests, on affecte tout d'abord les valeurs ci-dessous à trois variables `a`, `b`, `c` dans le shell :

```
>>> a,b,c = 4,2,1
```

On se doute alors que : le test `a<b` sera faux, le test `b>c` sera vrai. Dans un script Python, lorsque l'on essaie de déterminer si une expression logique est vraie ou fausse, le shell renvoie un booléen : le booléen `True` si l'expression est vraie et le booléen `False` si l'expression est fausse. Il suffit de taper les lignes suivantes dans le shell pour le vérifier :

```
>>> a<b

>>> b<c
```

Document 1 – Cours sur les booléens (page 1/3)

On peut continuer à s'entraîner en tapant les instructions de la première colonne du tableau ci-dessous et vérifier le résultat qu'indique le shell :

Instruction Python	Signification	Résultat
`a==b`	a est égal à b	`False`
`a>b`	a est strictement supérieur à b	`True`
`a<b`	a est strictement inférieur à b	`False`
`a>=b`	a est supérieur ou égal à b	`True`
`a<=b`	a est inférieur ou égal à b	`False`
`a!=b`	a est différent de b	`True`
`a<b and b>c`	a est inférieur à b ET b est supérieur à c	`False`
`a<b or b>c`	a est inférieur à b OU b est supérieur à c	`True`
`a<b ^ b>c`	a est inférieur à b « ou exclusif » b est supérieur à c	`False`
`not(True)`	Le contraire de `True`	`False`
`not(False)`	Le contraire de `False`	`True`

Vocabulaire à connaître

Chaque test (en colonne de gauche) correspond à une *expression booléenne*.
Les mots `and`, `or` et `not` sont appelés des *opérateurs booléens*.

Table de l'expression booléenne : `a and b`

Dans le shell, si l'on tape l'expression `True and True`, c'est tout à fait équivalent à taper `1 and 1` : le résultat obtenu est `True`. On obtiendra un résultat identique en tapant l'expression booléenne :

```
>>> 2>1 and 3 != 0 #True and True
```

Ainsi le shell affichera `True`.

On peut établir la table de l'expression booléenne `a and b` en effectuant plusieurs tests successifs comme :

```
>>> 0 and 0
0
>>> 0 and 1
0
>>> 1 and 0
0
>>> 1 and 1
1
```

Document 1 – Cours sur les booléens (page 2/3)

Ceci se résume dans la table :

Table d'une expression : a and b (ET)		
a	b	a and b
0	0	0
0	1	0
1	0	0
1	1	1

Table de l'expression booléenne : a or b

Si l'on tape l'expression booléenne :

```
>>> 2>1 or 3 != 0 #True or True
```

Le shell affichera aussi `True`.

Comme on l'a fait pour la table du `and`, si on effectue divers tests avec l'opérateur `or` (tels que `0 or 0, 0 or 1 etc.`), on peut établir la table de l'expression `a or b` :

Table d'une expression : a or b (OU)		
a	b	a or b
0	0	0
0	1	1
1	0	1
1	1	1

Table de l'expression booléenne : a xor b

À noter qu'il existe aussi un opérateur booléen appelé « ou exclusif » appelé `xor` et noté `^` en Python :

```
>>> 1 ^ 0 #True
```

Le résultat d'un test `a ^ b` ne sera vrai que si seulement un des deux (a,b) n'est vrai.

Table d'une expression : a ^ b (xor càd OU exclusif)		
a	b	a xor b
0	0	0
0	1	1
1	0	1
1	1	0

Document 1 – Cours sur les booléens (page 3/3)

Compétence attendue

- Utiliser une bibliothèque.

Exercice 1.3

Concevoir, traduire

Soit un angle dont la mesure en degrés est connue avec une précision de 1°.

- Écrire un script (donc un court programme dans l'éditeur) qui réalise les opérations suivantes :
  - Affecter une valeur de mesure d'angle en degrés à la variable `deg`.
  - Convertir cet angle en radians et stocker cette grandeur dans une variable `rad`.
  - Calculer le sinus et le cosinus de `rad` et afficher le résultat de ces deux calculs de manière conviviale (c'est-à-dire avec un peu de texte puis les résultats).

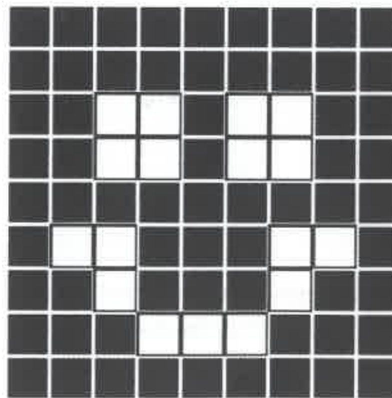
Compétence attendue

- Manipuler des booléens.

Exercice 1.4

Développer

L'image suivante peut être codée en binaire grâce à des 0 et des 1.



Document 2 – Exercices sur les booléens (page 1/2)

- Compléter son codage à l'aide des booléens 0 et 1 :

1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1
1	1	0	0	1	0	0	1	1
1	1	1	0	0	0	1	1	1
1	1	1	1	1	1	1	1	1

Compétence attendue

- Savoir interpréter une expression booléenne.

Exercice 1.5

Développer

- Dessiner la forme définie à l'aide des expressions booléennes ci-dessous (un 1 donne une case noire, un 0 une case blanche) :

1	1	1	1	1	1	1	1	1
1 and 1	1 or 1	0 and 0	1 or 1	1 or 1	1 or 1	0 and 0	1 or 1	1 or 1
1 and 1 or 1	0	0 and 1 or 0	0	1	0 and 1 or 0	0	0	1 and 1 or 0
1 and 1 or 1	0	0 and 1 or 0	0	0	0 and 1 or 0	0	0	1 and 1 or 0
1 and 1 or 1	0	0 and 1 or 0	0	0	0 and 1 or 0	0	0	1 and 1 or 0
1	0 xor 1	1	0	0	0	0 xor 1	1	1
1	1	1	0	0	0	1	1	1
1	1	1	1	1 xor 1	1	1	1	1
1	1	1	1	1	1	1	1	1

Document 2 – Exercices sur les booléens (page 2/2)

Exercices

Compétence attendue

- Utiliser des instructions conditionnelles.

Exercice 6.1

Concevoir

Écrire une fonction qui vérifie si un nombre est un entier ou un réel : elle renvoie `True` dans ce cas, `False` sinon.

Exercice 6.2

Concevoir, traduire

Créer une fonction permettant de tester la parité d'un entier. Cette fonction doit renvoyer `True` si l'entier saisi est pair, `False` sinon. Pour cela on peut utiliser le fait que si un entier `a` est divisible par `n`, le résultat du reste de la division euclidienne vaudra 0 : soit en Python `a%n` vaudra 0.

Exercice 6.3

Analyser, traduire

On souhaite tester si une solution chimique est acide, basique ou neutre. Pour cela on rappelle qu'une solution est dite :

- acide si pH < 7
- basique si pH > 7
- neutre si pH = 7

Créer une fonction permettant de tester si une solution est acide, basique ou neutre et d'effectuer un affichage correspondant. Cette fonction admet comme argument d'entrée une valeur `pH` et ne renvoie rien. On vérifiera que `pH` est bien de type entier ou flottant.

Exercice 6.4

Traduire

Écrire une fonction qui renvoie la valeur de la racine carrée d'un nombre `x` s'il est positif ou nul et qui affiche un message d'erreur si ce nombre est strictement négatif et renvoie `False`.

Document 3 – Exercices sur les booléens

Corrigé des exercices

Exercice 6.1

```
1.  def nombre(x):
2.      """Renvoie True si x est un réel ou un entier, False sinon"""
3.      if type(x)==int or type(x)==float:
4.          return True
5.      else:
6.          return False
7.
8.  print(nombre(-2.2))
9.  print(nombre(4))
10. print(nombre("Bonjour"))
```

Exercice 6.2

```
1.  def est_pair(a):
2.      """ Renvoie True si a est pair, False sinon"""
3.      if a%2==0:
4.          print ("a est pair")
5.          return True
6.      else:
7.          print ("a est impair")
8.          return False
```

On teste :

```
9.  x = 4
10. print(est_pair(x)) #On obtient True
```

Exercice 6.3

```
1.  def test_pH(pH):
2.      """ teste la caractère acide, basique ou neutre d'une solution
3.      Effectue un affichage, ne renvoie rien
4.      Teste si pH est un entier ou un flottant"""
5.      assert type(pH)==int or type(pH)==float
6.      if pH<7:
7.          print("solution acide")
8.      elif pH>7:
9.          print("solution basique")
10.     else:
11.         print("solution neutre")
```

Document 4 – Corrigé de trois exercices du document 3

Semestre 2 – Sujet n°1

Problème 2. Le compte est bon

Il s'agit dans ce problème de réaliser un programme capable de résoudre les défis du célèbre jeu télévisé « Le compte est bon ». Pour rappel, il s'agit de trouver une combinaison d'opérations arithmétiques élémentaires (addition, soustraction, multiplication et division) permettant d'atteindre un nombre (choisi au hasard entre 101 et 999) à partir d'un tirage de six nombres (eux aussi choisis au hasard dans un panier comprenant deux fois tous les nombres de 1 à 10, et une fois chacun des nombres 25, 50, 75 et 100), chacun de ces six nombres ne pouvant être utilisés qu'une seule fois au maximum.
Par exemple, avec le tirage des six nombres (4, 25, 50, 1, 4, 8), on peut atteindre le nombre 732 par la suite d'opérations suivante.

$4 + 25 = 29$
$8 \times 29 = 232$
$50 - 1 = 49$
$232 - 49 = 183$
$4 \times 183 = 732$

Une soustraction ne peut être utilisée que si elle donne un nombre positif. Une division ne peut être utilisée que si le reste dans la division euclidienne est nul.

2.1 Existence d'une solution

Il n'existe pas toujours de solution à un problème donné. Par exemple, avec le même tirage que celui de l'exemple précédent, il n'est pas possible d'atteindre le nombre 737. Le but de cette partie est d'étudier un prédicat qui teste l'existence d'une solution pour un problème donné comme le montre les deux exemples suivants :

```
>>> existe_solution(732, (4, 25, 50, 1, 4, 8))
True
>>> existe_solution(737, (4, 25, 50, 1, 4, 8))
False
```

Pour cela on se donne les quatre fonctions simples suivantes, ainsi qu'un tuple constitué de ces quatre fonctions.

```
def add(n1, n2):
    return n1 + n2

def sub(n1, n2):
    return n1 - n2 if n1 > n2 else n2 - n1

def mul(n1, n2):
    return n1 * n2

def div(n1, n2):
    if n2 != 0 and n1 % n2 == 0:
        res = n1 // n2
    elif n1 != 0 and n2 % n1 == 0:
        res = n2 // n1
    else:
        res = None
    return res

OPERATORS = (add, sub, mul, div)
```

Voici une réalisation récursive du prédicat `existe_solution`.

```
def existe_solution(cible, tirage):
    n = len(tirage)
    trouve = cible in tirage
    i, j = 0, 1
    while i < n - 1 and j < n and not trouve:
        n1, n2 = tirage[i], tirage[j]
        tirage_restant = tirage[:i] + tirage[i+1:j] + tirage[j+1:]
        ind_op = 0
        while ind_op < len(OPERATORS) and not trouve:
            res = OPERATORS[ind_op](n1, n2)
            if res != None:
                trouve = existe_solution(cible, tirage_restant + (res,))
            ind_op += 1
        if j == n - 1:
            i = i + 1
            j = i + 1
        else:
            j = j + 1
    return trouve
```

Question 1. Tracez le calcul effectué par ce prédicat lorsque le nombre `cible` est 2, et les nombres de `tirage` sont (4, 8, 25).

Question 2. Prouvez que pour toutes valeurs des paramètres `cible` et `tirage`, le calcul effectué par le prédicat `existe_solution` s'arrête. (Attention, il y a deux boucles `while` imbriquées et des appels récursifs.)

Question 3. Quel est l'intérêt du tuple `OPERATORS` dans la programmation du prédicat `existe_solution` ?

2.2 Meilleure approximation

Lorsqu'au jeu du « compte est bon » aucun des compétiteurs n'a trouvé la solution, c'est celui qui s'en approche le plus qui l'emporte. Cette partie consiste donc établir une fonction qui pour un problème donné trouve le nombre le plus proche de la cible à atteindre.

```
>>> meilleure_solution(732, (4, 25, 50, 1, 4, 8))
732
>>> meilleure_solution(737, (4, 25, 50, 1, 4, 8))
738
```

Question 4. Écrivez une fonction nommée `plus_proche` paramétrée par une séquence d'entiers et un nombre qui renvoie un entier de la séquence le plus proche du nombre.

```
>>> plus_proche((5, 25, 50, 1, 4, 8), 50)
50
>>> plus_proche((5, 25, 50, 1, 4, 8), 10)
8
>>> plus_proche((6, 25, 50, 1, 4, 8), 7)
6
```

Question 5. En vous inspirant du prédicat `existe_solution`, écrivez une fonction `meilleure_solution` qui renvoie l'entier le plus proche de la cible passée en premier paramètre que l'on puisse obtenir par les quatre opérations élémentaires et les nombres de la séquence passée en second paramètre (voir exemples ci-dessus).

2.3 Savoir justifier sa réponse

Au jeu du « compte est bon » il ne suffit pas de déclarer avoir atteint la cible ou s'en être approché, il faut aussi justifier sa réponse. C'est l'objectif de cette partie de permettre d'expliquer une solution. Au lieu de calculer le « meilleur nombre », il est préférable de chercher une « expression arithmétique » donnant le meilleur nombre. Par exemple, avec la cible 732 et le tirage (4, 25, 50, 1, 4, 8), une expression arithmétique menant à la solution est $(4 \times ((8 \times (4 + 25)) - (50 - 1)))$.

2.3.1 Expressions arithmétiques

Toute expression arithmétique n'utilisant que les quatre opérations peut être représentée par un arbre binaire. L'expression donnée en exemple ci-dessus peut être représentée par l'arbre de la figure 1.

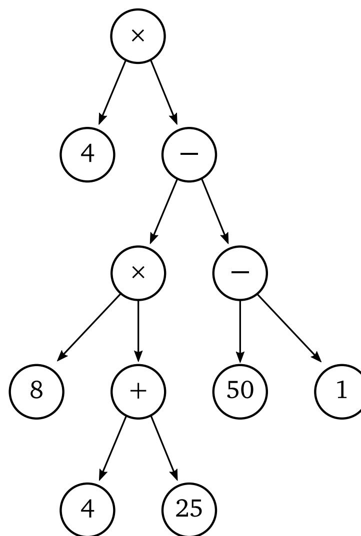


FIGURE 1 – Arbre représentant une expression arithmétique

On peut noter que dans l'arbre binaire représentant une expression arithmétique

(a) tous les nœuds ont 0 ou deux fils ;

(b) les nœuds internes sont tous étiquetés par des opérateurs arithmétiques ;

(c) les feuilles (ou nœuds externes) sont étiquetées par des nombres.

On suppose donnée une classe `BinaryTree` permettant de construire un arbre binaire et accéder aux composantes (cf Annexe page 5). Les expressions arithmétiques seront désormais représentées par des arbres binaires dont les feuilles sont étiquetées par les nombres (entiers) et les nœuds internes par l'un des caractères représentant une opération arithmétique (c'est le cas de l'arbre `a3` dans la question suivante qui représente `1 + 2`). La *valeur* d'une expression arithmétique est le nombre (entier) obtenu lorsque toutes les opérations arithmétiques ont été effectuées (la valeur de `a3` est 3).

```
from binary_tree import BinaryTree as BT
vide = BT()
a1 = BT(1, vide, vide)
a2 = BT(2, vide, vide)
a3 = BT('+', a1, a2)
```

Question 6. Réalisez un prédicat `is_leaf` paramétré par un arbre binaire qui renvoie `True` si l'arbre passé en paramètre est une feuille (i.e. un arbre non vide dont les deux sous-arbres le sont) et `False` dans le cas contraire.

```
>>> [is_leaf(a) for a in (vide, a1, a2, a3)]
[False, True, True, False]
```

Question 7. Réalisez une fonction nommée `valeur` qui calcule la valeur de l'expression passée en paramètre.

```
>>> [valeur(a) for a in (a1, a2, a3)]
[1, 2, 3]
```

Question 8. Adaptez les fonctions `add`, `sub`, `mul` et `div` pour qu'elles acceptent des expressions arithmétiques (et non des entiers) en paramètres, et renvoient l'expression arithmétique correspondante.

```
>>> valeur(add2(a2, a3))
5
>>> valeur(mul2(a2, a3))
6
>>> valeur(div2(a1, a3))
3
>>> div2(a2, a3) is None
True
```

2.3.2 Construction d'une expression solution

Question 9. Réalisez une fonction nommée `meilleure_solution2` qui effectue le même travail que son homonyme `meilleure_solution` mais qui renvoie une expression arithmétique au lieu d'un nombre.

```
>>> sol1 = meilleure_solution2(732, (4, 25, 50, 1, 4, 8))
>>> valeur(sol1)
732
>>> sol2 = meilleure_solution2(737, (4, 25, 50, 1, 4, 8))
>>> valeur(sol2)
738
```

2.3.3 Explication de la solution

Expliquer une solution c'est lister la suite d'opérations élémentaires qui mènent à la solution annoncée, chaque opération faisant intervenir des nombres du tirage et/ou des nombres établis par des opérations antérieures.

Question 10. Réalisez une fonction nommée `explique` qui à partir d'une expression arithmétique construit la liste des opérations (sous forme de chaînes de caractères) permettant de parvenir à la solution.

```
>>> explique(sol1)
['4 + 25 = 29', '8 * 29 = 232', '50 - 1 = 49', '232 - 49 = 183', '4 * 183 = 732']
>>> explique(sol2)
['4 * 25 = 100', '100 - 1 = 99', '8 * 99 = 792', '50 + 4 = 54', '792 - 54 = 738']
```

Annexe : arbres binaires

La classe `BinaryTree`, définie dans un module nommé `binary_tree`, permet de représenter des arbres binaires.

```
>>> from binary_tree import BinaryTree as BT
>>> vide = BT()
>>> vide.is_empty()
True
>>> arbre = BT(1, BT(2, vide, vide),
...              BT(3, vide, vide))
>>> type(arbre)
<type 'instance'>
>>> arbre.is_empty()
False
>>> arbre.get_root()
1
>>> arbre.get_left_subtree().get_root()
2
>>> arbre.get_right_subtree().get_root()
3
```

Le constructeur d'arbres (`BinaryTree`), lorsqu'il est utilisé avec trois arguments, déclenche une exception `BinaryTreeError` si les deuxième et troisième paramètres ne sont pas des arbres binaires. Il en va de même des trois méthodes de sélection `get_root`, `get_left_subtree` et `get_right_subtree` si elles sont invoquées pour un arbre vide.