Instructions conditionnelles

1 Instructions conditionnelles

1.1 Test simple

Définition

Une instruction conditionnelle est une instruction pouvant s'exécuter seulement si une condition est vérifiée par l'état courant.

Pour en créer une en Python, on utilise le mot-clé **if**, avec la syntaxe suivante :

```python
if condition :
    #bloc d' instructions
```

À la suite de **if** et avant les **:**, il est possible de mettre toute expression booléenne.

1.2 Opérateurs de comparaison

Algorithme

```
1 2 = 8
2 2 ≠ 8
3 2 ≥ 8
4 2 > 8
5 2 ≤ 8
6 2 < 8
```

```python
>>> 2==8
    False
>>> 2!=8
    True
>>> 2>=8
    False
>>> 2>8
    False
>>> 2<=8
    True
>>> 2<8
    True
```

1.3 Test avec alternative

On souhaite programmer la relation de récurrence liée à la suite de Syracuse. Cette dernière est définie par récurrence avec la formule suivante :

$$u_0 = \mathrm{N} \in \mathbb{N} \text{ et pour tout entier } n \in \mathbb{N}, u_{n+1} = \begin{cases} \frac{u_n}{2} & \text{si n est pair} \\ 3u_n + 1 & \text{si n est impair} \end{cases}$$

Un essai naïf pourrait être :

```python
Un=20
if Un%2==0:
    Un1=Un//2
if not(Un\%2==0):
    Un1=3*Un+1
```

Si cette fonction a le mérite de fonctionner, elle utilise deux tests au lieu de un (ce qui coûte du temps tant au programmeur qu'à la machine). Pire, si on utilisait la même syntaxe en dehors d'une fonction (donc en remplaçant return x par x=x//2), on se retrouverait avec une instruction qui ne réalise pas ce que l'on souhaite (car la valeur de x après division entière par 2 serait potentiellement impaire)

Pour pallier à ces problèmes, on a enrichi la syntaxe des instructions conditionnelles de tests avec alternative (si ... sinon...) :

```python
if condition :
    #bloc d'instructions si la condition est réalisée

else :

    #bloc d'instructions si la condition n'est pas réalisée
```

Par exemple la relation de récurrence de la suite de Syracuse s'écrit :

1.4 Tests imbriqués

Il est possible de réaliser une instruction conditionnelle dans une instruction conditionnelle. On parle alors de tests imbriqués.

Lorsque les tests imbriqués ne servent qu'à séparer une situation en plus de deux cas de figures, l'indentation obligatoire en Python peut rendre difficile la compréhension du programme.

Par exemple considérons le critère de réussite au bac suivant :

```python
N=int(input())
if (N<8):
    print("Echec au bac")
else:
    if (N<10):
        print("Rattrappage")
    else :
        if(N<12):
            print("Bac obtenu au premier tour")
        else:
            if(N<14):
```

```python
            print("Mention AB")
        else:
            if(N<16):
                print("Mention B")
            else: \#On a alors N>=16
                print("Mention TB")
```

L'indentation rend la lecture de ce programme (pourtant très simple) particulièrement peu aisée et peu naturelle. Il est possible de mettre toutes ces instructions au même niveau en utilisant le mot clé **elif** qui est une contraction de *else* et *if*.

Sur l'exemple précédent cela donnerait :

```python
N=int(input())
if (N<8):
    print("Echec au bac")
elif (N<10):
    print("Rattrappage")
elif (N<12):
    print("Bac obtenu au premier tour")
elif (N<14):
    print("Mention AB")
elif (N<16):
    print("Mention B")
else: \# Il n'est pas nécessaire de spécifier la dernière alternative
    print("Mention TB")
```

1.5 *Application 1*

Écrire un programme en Python qui prend en entrée une année et affiche le booléen True si cette année est bissextile et False sinon. On rappelle qu'une année est bissextile si elle divisible par 4 sauf si elle est divisible par 100 (auquel cas elle n'est pas bissextile) sauf si elle est divisible par 400 (auquel cas elle l'est).

2 Introduction à l'algèbre de Boole

2.1 Conditions multiples

Nous n'avons pour le moment pas étudié des tests portant sur plusieurs conditions sur une ou plusieurs variables.

Si en théorie il est possible de décrire à l'aide d'instructions imbriquées et de disjonction de cas n'importe quel test, cela devient vite très compliqué en terme d'analyse a priori, très coûteux en temps de calcul pour évaluer les booléens liés au test, et rend les codes source très difficiles à lire.

La solution consiste à utiliser des opérateurs sur les booléns, qui correspondent aux opérations logiques et qui existent tant dans la vie courante qu'en mathématiques.

Par exemple :
Venir à pied au lycée = Métro en grève ET Journée travaillée

Venir en métro au lycée = NON(Métro en grève) ET Journée travaillée
Décrocher = (Sonnerie ET Décision de répondre) OU Décision d'appeler

2.2 Algèbre de Boole

L'algèbre de Boole formalise les opérations sur les booléns, et donne un cadre rigoureux (en mathématiques, les algèbres sont des structures algébriques ayant certaines propriétés communes) à celles-ci permettant d'introduire des techniques algébriques pour traiter et simplifier les expressions ne contenant que des booléens. Elle a des applications tant en mathématiques, en logique, en éléctronique qu'en mathématiques. Elle fut initiée en 1854 par le mathématicien britannique George Boole.

On note B l'ensemble constitué de deux éléments appelés valeurs de vérité : $B = \{\text{True}; \text{False}\}$

Sur cet ensemble on définit deux opérateurs, ou lois : les lois "and" et "or" et une transformation, appelé complémentaire, inversion ou contraire : "not"

2.3 Opérateur booléen « and »

Elle est définie de la manière suivante : A and B est True si et seulement si A est True et B est True.

Cette loi est aussi notée : .(comme la multiplication à laquelle elle correspond si on note True=1 et False=0) ou « and» dans Python ou « & » ou « && » dans certains langages informatiques.

Voici un tableau de vérité définissant cette loi :

Table de vérité de « and »		
A	B	A and B
False	False	False
True	False	False
False	True	False
True	True	True

2.4 Opérateur booléen « or »

Il est défini de la manière suivante : A or B est True si et seulement si A est True ou B est True. (En particulier, si A est True et que B est True aussi, alors A or B est True.) Cette loi est aussi notée : +, V, « or » dans Python ou « — » ou « —— » dans certains langages informatiques.

Voici une table de vérité définissant cette loi :

Table de vérité de « or »		
A	B	A or B
False	False	False
True	False	True
False	True	True
True	True	True

2.5 Opérateur « not »

Le contraire de A est True si et seulement si a est False. Le contraire de A est noté : non-A ; $\overline{A}$; « not» en Python ; «!» dans certains langages informatiques , parfois « $\sim$ » comme en SQL.

Voici un tableau définissant cette loi :

Table de vérité de « not »	
A	not(A)
False	True
True	False

2.6 Propriétés d'Algèbre

Certaines propriétés des opérateurs booléens lui confèrent en mathématiques une structure d'algèbre commutative :

— Associativité : (A and B) and C = A and (B and C) ; (A or B) or C= A or (B or C)
— Commutativité de « and » et de « or » : A and B = B and A ; A or B = B or A
— Distributivité A and (B or C) = (A and B) or (A and C) ; A or (B and C) = (A or B) and (A or C)
— Existence d'éléments neutres : True and A = A ; False or A = A

De plus, l'opération « not » a certaines propriétés particulières supplémentaires :

— not(not A)= A
— A or not A = True
— A and (not A) = False

2.7 Priorités

On pose que « and » est prioritaire au « or » (de la même manière que le * est prioritaire au +). Ainsi :

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

2.8 Théorème de DeMorgan

La **première loi de DeMorgan (négation de la conjonction)** s'exprime :

not (A or B) = (not A) and (not B)

Application 2 : prouvez-le avec une table de vérité :

Table de vérité de la négation de la conjonction						
A	B	A or B	**not(A or B)**	not A	not B	**(not A) and (not B)**
False	False					
True	False					
False	True					
True	True					

La **deuxième loi de DeMorgan (négation de la disjonction)** s'exprime :

not (A and B) = (not A) or (not B)

Application 3 : prouvez-le avec une table de vérité :

Table de vérité de la négation de la conjonction						
A	B	A and B	**not(A and B)**	not A	not B	**(not A) or (not B)**
False	False					
True	False					
False	True					
True	True					

2.9 Simplification d'expressions booléennes

Pour simplifier les expressions dont les valeurs sont de type booléen, on utilisera soit directement le théorème de DeMorgan, soit on démontrera l'égalité de deux lois à l'aide de tables de vérités (en testant tous les cas)

Application 4 : Écrire à l'aide de « and », « or » et « not »l'opérateur « xor » : "OU EXCLUSIF" (A xor B est vrai si et seulement si A et B le sont mais pas en même temps)

Application 5 : À l'aide des théorèmes de DeMorgan et des distributivités mutuelles de « and » et de « or » montrer que :

— not(A or not(not B and C) = (not A and B) or (not A and not C)

— not(A or not(not B and C) = not A and (B or not C)