

# Instructions conditionnelles

## 1 Instructions conditionnelles

### 1.1 Test simple

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 :

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

### 1.2 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 :

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

Un essai naïf pourrait être :

```
def recurrence_Syracuse(n):
    if (n%2==0):
        return n/2
    if not (n%2==0):
        return 3*n+1
```

Si cette fonction a le mérite d'être correcte, 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 n/2` par `n=n/2`), on se retrouverait avec une instruction qui ne réalise pas ce que l'on souhaite (car la valeur de `n` 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...) :

```
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 :

```
def recurrence_Syracuse(n):
    if (n%2==0):
        return n/2
    else:
        return 3*n+1
```

### 1.3 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 code suivant :

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

L'indentation rend difficile la lecture de ce simple programme. Ces instructions peuvent être mises au même niveau en utilisant le mot clé **elif** qui est une contraction de *else* et *if*. Sur l'exemple précédent :

```
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")
```

*Exercice 1* : Écrire une fonction d'en-tête `Est_Bissextile(Annee)` en Python qui prend en entrée une année et renvoie 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 booléens, 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 booléens, 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 électronique 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 :  $A \text{ and } B$  est True si et seulement si  $A$  est True et  $B$  est True.

Cette loi est aussi notée :  $\cdot$  (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 complémentaire booléen en notation préfixe « not »

Le contraire de  $A$  est True si et seulement si  $A$  est False. Le contraire de  $A$  est noté :  $\text{non-}A$ ;  $\bar{A}$ ; « not » en Python; « ! » dans certains langages informatiques, parfois « ~ » 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 :  $(A \text{ and } B) \text{ and } C = A \text{ and } (B \text{ and } C)$ ;  $(A \text{ or } B) \text{ or } C = A \text{ or } (B \text{ or } C)$
- Commutativité de « and » et de « or » :  $A \text{ and } B = B \text{ and } A$ ;  $A \text{ or } B = B \text{ or } A$
- Distributivité  $A \text{ and } (B \text{ or } C) = (A \text{ and } B) \text{ or } (A \text{ and } C)$ ;  $A \text{ or } (B \text{ and } C) = (A \text{ or } B) \text{ and } (A \text{ or } C)$
- Existence d'éléments neutres :  $\text{True and } A = A$ ;  $\text{False or } A = A$

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

- $\text{not}(\text{not } A) = A$
- $A \text{ or not } A = \text{True}$
- $A \text{ and } (\text{not } A) = \text{False}$

## 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 De Morgan

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

$$\text{not } (A \text{ or } B) = (\text{not } A) \text{ and } (\text{not } B)$$

*Exercice 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 De Morgan (négation de la disjonction)** s'exprime :

$$\text{not } (A \text{ and } B) = (\text{not } A) \text{ or } (\text{not } B)$$

*Exercice 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)

*Exercice 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 ;

*Exercice 5* : À l'aide des théorèmes de De Morgan et des distributivités mutuelles de « and » et de « or » montrer que :

$$\text{not}(A \text{ or } \text{not}(B \text{ or } \text{not } C)) = (\text{not } A \text{ and } B) \text{ or } (\text{not } A \text{ and } \text{not } C)$$