Une analyse de la notion de booléen et de son usage dans l'enseignement de la programmation

Sylvie Alayrangues, Olivier Baudon, Emmanuel Beffara, Ronan Charpentier, Sébastien Daniel, Christophe Declercq, Emmanuel Delay, Aslı Grimaud, Sébastien Hoarau, Anne Héam, et al.

Une analyse de la notion de booléen et de son usage dans l'enseignement de la programmation*

Commission Inter-IREM Informatique

octobre 2022

Résumé. La notion de booléen est fondamentale en informatique. Même si elle paraît simple à première vue, son introduction dans les premiers apprentissages de la programmation révèle un certain nombre de difficultés liées au fait qu'elle mobilise plusieurs concepts généraux de l'informatique : types de données, valeurs de vérité, invariants, structures de contrôle… Partant de ce constat, l'objectif de cet article est de proposer quelques réflexions sur le rôle des booléens en programmation. On détaillera certains points relatifs aux « bonnes pratiques » possibles, en particulier dans le cadre de l'enseignement de l'informatique. On s'intéressera notamment au booléen comme type de données et aux principales structures de contrôle faisant appel à des booléens. On s'intéressera également à la notion générale d'expression booléenne utilisée en programmation. Au travers d'une étude de cas, on illustrera quelques usages typiques des variables booléennes.

Mots-clés. booléen, logique, programmation, structure de contrôle, variable

Abstract. The notion of boolean is fundamental in computer science. Even if it seems simple at first sight, its introduction in the first stages of programming education reveals a number of difficulties related to the fact that it involves several general concepts of computer science: data types, truth values, invariants, control structures… Starting from this observation, the point of this article is to propose some reflections on the role of booleans in programming. We will elaborate on some points concerning possible "good practices", in particular in the context of computer science teaching. In particular, we will focus on boolean as a data type and on the main control structures using booleans. We will also focus on the general notion of boolean expression used in programming. Through a case study, we will illustrate some typical use cases of boolean variables.

Keywords. boolean, logic, programming, control structure, variable

*Ont participé à l'élaboration de ce document : Sylvie Alayrangues (IREM de Poitiers), Olivier Baudon (IREM de Bordeaux), Emmanuel Beffara (IREM de Grenoble), Ronan Charpentier (IREM de Caen), Sébastien Daniel (IREM de Lorraine), Christophe Declercq (IREM de Réunion), Emmanuel Delay (IREM de Clermont-Ferrand), Aslı Grimaud (IREM de Lille), Sébastien Hoarau (IREM de Réunion), Anne Héam (IREM de Besançon), Philippe Marquet (IREM de Lille), Jean-Christophe Masseron (IREM de Paris), Antoine Meyer (IREM de Paris), Malika More (IREM de Clermont-Ferrand), Florence Nény (IREM de Marseille), Cécile Prouteau (IREM de Paris), Jean-Marc Vincent (IREM de Grenoble), Emmanuel Volte (IREM de Paris), Nathalie Weibel (IREM de Caen). Contact : emmanuel.beffara@univ-grenoble-alpes.fr

Table des matières

Introduction

Cet article s'adresse aux enseignants de toute discipline concernée par l'algorithmique, la programmation, et plus généralement l'informatique, que ce soit au collège, au lycée ou dans l'enseignement supérieur. Son objectif est de proposer quelques réflexions sur le rôle des booléens en programmation et de détailler certains points relatifs aux difficultés rencontrées par les élèves et aux « bonnes » pratiques possibles. L'intention n'est pas de proposer des exercices ou activités immédiatement utilisables en classe, mais une discussion générale des divers enjeux relatifs à cette notion. On suppose que les lecteurs ne sont pas néophytes en programmation et en particulier qu'ils sont en mesure de comprendre un programme simple en Python ou en Scratch.

L'objet n'est pas non plus de traiter toute la logique telle qu'elle apparaît en informatique. En particulier, les questions telles que la correction d'algorithmes et les invariants ne sont pas abordées ici en tant que telles (mais pourront faire l'objet de travaux ultérieurs). On cherchera néanmoins à souligner les points communs et différences qui peuvent exister entre certaines notions de logique mathématique (tables de vérité, connecteurs logiques) et leurs analogues en programmation.

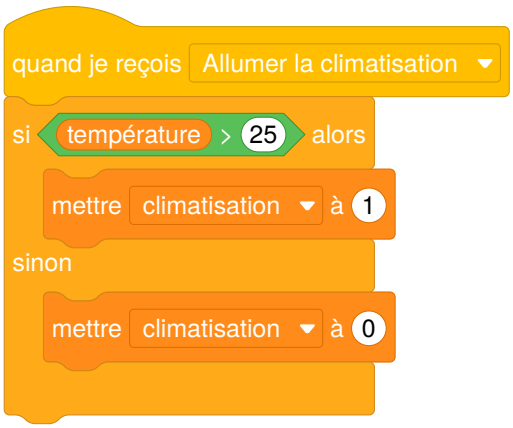
Après avoir décrit la notion de booléen en tant que type de données, notre analyse identifie deux enjeux importants dans la compréhension des booléens :

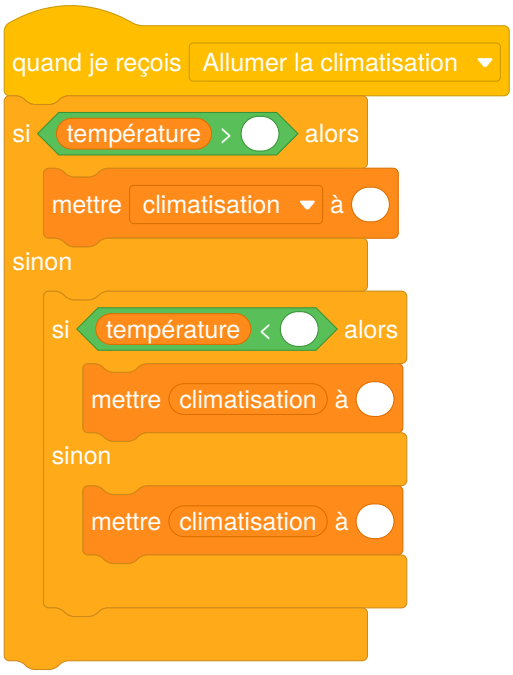
- dans les structures de contrôle et les expressions booléennes, l'articulation entre l'aspect statique (valeurs de vérité, etc) et l'aspect dynamique (mécanisme d'évaluation, flot de contrôle) ;
- la variété des usages des variables booléennes en programmation (indicateur, accumulateur, etc).

Nous amorçons notre discussion par la description de deux exemples d'exercices de programmation tirés des examens nationaux français.

Un exemple tiré d'une épreuve de brevet

L'exercice suivant est tiré du sujet de Diplôme national du brevet (DNB) de Polynésie, session de septembre 2018, en série professionnelle.

> Il est possible de piloter la climatisation de son domicile à distance à l'aide de deux programmes.
>
> **1.** Le programme A est le suivant :
>
> 
>
>
> Indiquer ce qu'il se passe si la commande « Allumer la climatisation » est sélectionnée et que la température de la pièce est de 27°.
>
> **2.** Le programme B permet de régler la puissance de la climatisation en fonction de la température. Ainsi lorsque la température est :
>
> – supérieure à 28°C la climatisation est sur le niveau 2 ;
>
> – entre 28°C et 25°C, la climatisation est sur le niveau 1 ;
>
> – inférieure à 25°C, elle s'arrête.
>
> Compléter les cases contenant des pointillés du programme B :



On constate que cet exercice fait appel à une compréhension relativement fine de plusieurs concepts relatifs à l'usage des booléens :

- perception visuelle de la catégorie syntaxique des expressions booléennes en Scratch (hexagones verts) ;
- instructions conditionnelles imbriquées ;
- interprétation des comparaisons numériques ;
- négation implicite d'une expression booléenne.

Le dernier point fait référence à la perception du statut des différentes variables dans les blocs « sinon » : par exemple, dans le programme A, l'instruction « mettre *climatisation* à 0 » est exécutée dans les cas où *température* ≤ 25. De même dans la question 2, il est nécessaire à la réalisation de l'exercice de pouvoir attribuer mentalement des propriétés au sujet de la valeur des variables (par exemple : « *température* est supérieure à … » ou « *température* est comprise entre … et … ») à chacun des blocs.

Ces observations nous semblent justifier, dès le collège, une réflexion suffisamment approfondie sur ces sujets.

Un exercice tiré de l'épreuve de l'enseignement de spécialité NSI

L'exercice suivant est tiré du sujet n°6 de l'épreuve de l'enseignement de spécialité Numérique et Science Informatique (NSI) session 2022, partie pratique.

> La fonction `recherche` prend en paramètre deux chaînes de caractères `gene` et `seq_adn` et renvoie `True` si on retrouve `gene` dans `seq_adn` et `False` sinon.
>
> Compléter le code Python ci-dessous pour qu'il implémente la fonction `recherche`.

```
def recherche(gene, seq_adn):
    n = len(seq_adn)
    g = len(gene)
    i = ...
    trouve = False
    while i < ... and trouve == ... :
        j = 0
        while j < g and gene[j] == seq_adn[i+j]:
            ...
        if j == g:
            trouve = True
        ...
    return trouve
```

> Exemples :

```
>>> recherche("AATC", "GTACAAATCTTGCC")
True
>>> recherche("AGTC", "GTACAAATCTTGCC")
False
```

Cet exercice nécessite une bonne compréhension de la structure complexe de ce programme, des comparaisons utilisées dans les boucles `while` ainsi que dans la structure conditionnelle `if`. Il requiert également de comprendre l'utilisation faite dans le programme de la variable booléenne `trouve`, qu'on appelle un « drapeau de continuation de boucle ». En outre, on peut considérer que le style de programmation qui transparaît dans cet extrait de programme dénote un certain nombre d'habitudes et de principes qui peuvent être discutés, et méritent *a minima* d'être repérés.

1. Booléen comme type de données

Dans cette partie, on décrit la notion de booléen du point de vue informatique. Ce qui la caractérise et la distingue de la notion mathématique d'algèbre de Boole est son statut de *type de données*, c'est-à-dire d'objet susceptible d'être calculé, stocké et utilisé dans la dynamique d'un programme.

1.1. La notion de booléen et sa transposition en programmation

Le type *booléen* est le type de données comprenant deux valeurs, habituellement appelées *vrai* et *faux*.

Certains langages de programmation disposent d'un type explicite pour représenter cette structure (Python, C++, Java, etc.) et d'autres utilisent d'autres types pour la représenter (versions anciennes du C, shell Unix, etc.). Quoi qu'il en soit, la notion de booléen est toujours présente d'une façon ou d'une autre dans tous les langages de programmation.

En Scratch, les expressions booléennes sont distinguées visuellement : elles sont marquées par des blocs de contour hexagonal et non arrondi comme les expressions arithmétiques. Cette forme d'expressions est imposée dans les structures conditionnelles.

En Python, la documentation précise qu'il existe des valeurs primitives `True` et `False` mais que les valeurs de la plupart des autres types peuvent aussi être interprétées comme valeurs de vérité, selon des conventions assez générales (ce qui ressemble à zéro ou à un ensemble vide est considéré comme faux, le reste est considéré comme vrai). Cela permet d'utiliser n'importe quelle expression comme condition dans une structure conditionnelle, comme dans l'exemple suivant où `liste`, si elle est non vide est interprétée comme `True` :

```
if liste:
    print("la liste commence par", liste[0])
else:
    print("la liste est vide")
```

Les règles précises de cette interprétation des objets en tant que booléens sont un peu complexes. On peut en outre considérer que cette pratique rend le code moins explicite et moins facile à interpréter, elle est donc évitée par certains enseignants. Une table en annexe détaille la situation.

Ainsi, certains langages pratiquent des conversions implicites entre types, par exemple Python interprète un nombre comme un booléen s'il apparaît comme condition. Ce phénomène n'apparaît pas en Scratch du fait de la structure du langage (Scratch n'autorise pas l'introduction de blocs non hexagonaux dans les opérateurs booléens). À l'inverse, Scratch peut interpréter un booléen comme un nombre s'il apparaît dans une expression arithmétique (Scratch autorise l'introduction de blocs hexagonaux dans les opérateurs numériques ou alphanumériques)

En Scratch, il n'y a donc jamais de conversion implicite vers une valeur booléenne. Par contre, les expressions booléennes peuvent être utilisées dans les autres expressions, elles sont alors converties implicitement, en nombres (1 pour *vrai*, 0 pour *faux*) ou en textes (`true` ou `false`) selon l'opérateur utilisé[1].

[1]Pour le détail des conversions implicites utilisées par Scratch 3.0, voir par exemple cet extrait du code source de la machine virtuelle Scratch : https://github.com/LLK/scratch-vm/blob/develop/src/util/cast.js. L'interpréteur Scratch est écrit en Javascript mais n'utilise pas exactement les mêmes conventions que celui-ci.

Python a un comportement similaire dans le cas numérique, où `False` est interprété comme 0 et `True` comme 1:

```
>>> True + True
2
>>> (80 > 50) + (60 > 50)
2
```

Bien entendu, il n'est absolument pas conseillé d'utiliser les conversions implicites des booléens avec des élèves. Nous les mentionnons car elles peuvent aider à comprendre certaines erreurs.

1.2. Les booléens sont des valeurs

Quel que soit le langage de programmation considéré avec les conventions qui s'y appliquent, des booléens sont utilisés dans les structures de contrôle qui impliquent une forme de condition. Comme on le verra plus loin, la détermination d'une condition peut impliquer des calculs de toutes natures. De fait, les booléens sont *calculés*, ils ont donc un rôle de *valeurs*, au même titre que les nombres et les chaînes de caractères par exemple.

En particulier, les valeurs booléennes étant des valeurs comme les autres, il est possible de les stocker en les affectant à des variables. Même dans les langages qui ne disposent pas d'un type spécifique pour les booléens (comme Scratch ou bien C), utiliser une représentation, comme un entier 0 ou 1, permet de stocker une information de nature booléenne.

On désigne parfois sous le nom de *variable booléenne* une variable utilisée pour stocker une information booléenne, qu'elle utilise ou pas un type spécifique aux booléens. On rencontre parfois aussi le terme *drapeau*, calqué de l'anglais *flag*, pour désigner une telle variable, en particulier lorsqu'elle sert à indiquer si une condition a été satisfaite ou pas. Un cas classique consiste par exemple à avoir une variable `est_trouve` dont la valeur de vérité indique si un élément donné a été rencontré lors du parcours d'un tableau.

La section 4 précise plus en détail et illustre différents rôles que peuvent jouer les variables booléennes.

Comme on le verra dans la suite, le fait que les booléens soient des valeurs comme les autres, qui peuvent être stockées, calculées, combinées par des opérateurs spécifiques, mais aussi prises comme valeurs d'entrée et renvoyées comme valeurs de sortie dans des fonctions, explique à la fois la puissance cette notion et les difficultés conceptuelles qui s'y rapportent, notamment à cause du rôle particulier que jouent les booléens dans les structures de contrôle.

2. Structures de contrôle

Le but de cette partie est de mettre en évidence les différentes utilisations fondamentales des booléens dans les structures de contrôle et de décrire les différents cas d'usage pour les variables booléennes.

2.1. Structures conditionnelles

Structure conditionnelle simple

Une des premières rencontres des booléens en programmation est leur usage en tant que condition dans des branchements conditionnels. Les structures « si … alors … » et « si … alors … sinon … » se retrouvent en effet dans la plupart des langages.

Voici un exemple classique d'exécution conditionnelle en Python :

```
s = input("Entrer un nombre entier")
n = int(s)
if n < 0:
    print("Le nombre entré est strictement négatif")
```

Le message à la dernière ligne du programme ne sera affiché que si le nombre `n` est strictement négatif. La version suivante présente une alternative :

```
s = input("Entrer un nombre entier")
n = int(s)
if n < 0:
    print("Le nombre entré est strictement négatif")
else:
    print("Le nombre entré est positif ou nul")
```

Le booléen apparaît donc avec l'expression `n < 0` qui est une simple comparaison numérique. À l'exécution, cette expression sera évaluée à *vrai* ou *faux* selon la valeur reçue par `n` pour déterminer quelle branche emprunter. Ici, le premier message sera affiché si la condition `n < 0` est satisfaite, dans le cas contraire c'est le second message qui sera affiché.

Notons que le bloc d'instructions introduit par le mot-clé `else` correspond à la négation de la condition introduite par le `if` (c'est à dire ici `n >= 0`). On peut rendre cette sémantique explicite sous la forme d'un commentaire portant sur le bloc `else`, par exemple :

```
if n < 0:
    print("Le nombre entré est strictement négatif")
else:  # n >= 0
    print("Le nombre entré est positif ou nul")
```

Attention, la structure `if condition: ... else: ...` n'est cependant pas équivalente en général à la structure `if condition: ...` suivie de `if not condition: ...`. En effet, dans ce dernier cas, si `condition` vaut *vrai*, le bloc d'instructions du premier `if` s'exécute et est susceptible de modifier les valeurs des variables, comme dans l'exemple suivant :

```
if n < 0:
    print("Le nombre entré est strictement négatif")
    n = -n
else:
    print("Le nombre entré est positif ou nul")
```

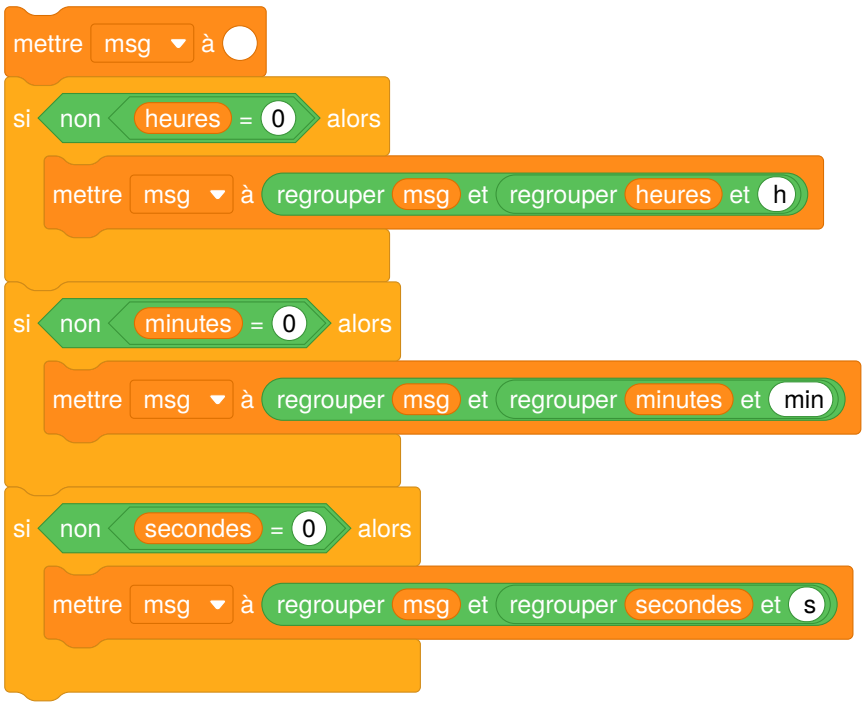
Dans le programme ci-dessus, le texte affiché est correct quel que soit le nombre entré alors que dans l'exemple suivant, si le nombre `n` est strictement négatif, la valeur de `n` est remplacée par son opposé et les deux messages sont affichés.

```
if n < 0:
    print("Le nombre entré est strictement négatif")
    n = - n
if not (n < 0):
    print("Le nombre entré est positif ou nul")
```

Structures conditionnelles successives

Une succession d'instructions conditionnelles « si … alors … » sans « sinon … » est utile pour effectuer une série d'actions « indépendantes ». Supposons qu'on dispose de trois variables entières `heures`, `minutes` et `secondes` représentant une durée. Le programme suivant en Python ou le programme en Scratch construisent dans `msg` un texte décrivant cette durée, en incluant les unités mais en omettant les quantités nulles :

```
msg = ""
if heures != 0:
    msg = msg + str(heures) + " h "
if minutes != 0:
    msg = msg + str(minutes) + " min "
if secondes != 0:
    msg = msg + str(secondes) + " s "
```



Par exemple si `heures`, `minutes` et `secondes` désignent respectivement les valeurs `2`, `0` et `15`, la valeur de `msg` à la fin de l'exécution du programme est `'2 h 15 s '`.

Dans ce second exemple, on s'intéresse à la recherche de la plus grande valeur parmi les trois données. L'ordre d'exécution des deux blocs `if` est indifférent, et il serait incorrect de les lier par `elif` ou par `else` :

```
def maximum(a, b, c):
    """Renvoie la plus grande des trois valeurs a, b, c."""
    mon_max = a
    if b > mon_max:
        mon_max = b
    if c > mon_max:
        mon_max = c
    return mon_max
```

Le troisième et dernier exemple illustre la difficulté à traduire un algorithme en langue naturelle en un programme contenant des branchements conditionnels :

Énoncé

Programmer en Python l'algorithme suivant :

Soit x un nombre,

Si x est inférieur à 1 alors afficher « valeur plus petite que 1 »

Si x est compris entre 1 et 2 alors afficher « valeur comprise entre 1 et 2 »

Sinon afficher « valeur plus grande que 2 »

Une traduction littérale de cet énoncé, dont la formulation est ambiguë (« si… si… sinon »), peut mener à l'écriture de la fonction incorrecte suivante :

```
if x < 1:
    print('valeur plus petite que 1')
if 1 <= x and x <= 2:
    print('valeur comprise entre 1 et 2')
else:
    print('valeur plus grande que 2')
```

Quand `x` a la valeur 0 par exemple, on obtient l'affichage incorrect :

```
valeur plus petite que 1
valeur plus grande que 2
```

Ce programme est incorrect pour toutes les valeurs de `x` strictement plus petites que 1. Ceci est dû au fait que le bloc `else` ne se rapporte qu'au deuxième `if` : il correspond donc à la condition `not (1 <= x and x <= 2)`, qui équivaut à `1 > x or x > 2`. On peut corriger le programme ainsi :

```
if x < 1:
    print('valeur plus petite que 1')
if 1 <= x and x <= 2:
    print('valeur comprise entre 1 et 2')
if x > 2:
    print('valeur plus grande que 2')
```

Ce programme est correct, mais il présente encore plusieurs inconvénients : chaque comparaison effectuée est redondante, et il peut être difficile de déterminer l'expression booléenne à utiliser comme condition du dernier bloc `if`. On peut y remédier à l'aide de structures conditionnelles imbriquées, comme nous le verrons au paragraphe suivant.

Structures conditionnelles imbriquées

Chacune des branches d'une structure conditionnelle peut à son tour contenir des tests et des branchements, on parle alors de structures conditionnelles imbriquées. Cela est souvent nécessaire mais peut rendre plus complexe l'analyse du flot d'exécution du programme.

Reprenons le dernier exemple de la section précédente. Comme les cas à considérer sont mutuellement exclusifs, il peut être préférable ici d'utiliser des structures conditionnelles imbriquées :

```
if x < 1:
    print('valeur plus petite que 1')
else:  # x >= 1
    if x <= 2:
        print('valeur comprise entre 1 et 2')
    else:  # x > 2
        print('valeur plus grande que 2')
```

En Python, il existe une syntaxe plus concise (utilisant le mot-clé `elif`) permettant d'éviter une trop grande indentation du code :

```
if x < 1:
    print('valeur plus petite que 1')
elif x <= 2:
    print('valeur comprise entre 1 et 2')
else:
    print('valeur plus grande que 2')
```

Instruction `return` dans une structure conditionnelle

La succession et l'imbrication de conditionnelles ont un comportement différent dans une fonction, notamment en présence d'instructions `return` dans les blocs du `if` et du `else`. En particulier, il est possible dans certains cas d'omettre le mot-clé `else` ou de remplacer `elif` par `if`.

Considérons par exemple les fonctions `choix1` et `choix2`, qui reçoivent en argument un nombre `n` et renvoient une chaîne indiquant si la valeur de `n` est négative ou positive et précise dans le cas positif si elle est paire ou impaire:

```
def choix1(n):
    if n < 0:
        return 'négatif'
    if n % 2 == 0:
        return 'positif et pair'
    return 'positif et impair'

def choix2(n):
    if n < 0:
        return 'négatif'
    elif n % 2 == 0:
        return 'positif et pair'
```

```
    else:
        return 'positif et impair'
```

Ces deux fonctions ont le même comportement, car dans les deux cas quand `n` est négatif l'instruction `return 'négatif'` (ligne 3) est exécutée et le reste du corps de la fonction n'est pas pris en compte. Par contre, dans la fonction `choix1` il est bien sûr indispensable de choisir correctement l'ordre des conditions à tester (les intervertir ne serait pas équivalent).

La question de la version à privilégier n'a pas de réponse claire et relève de la préférence et du style. On peut avancer que la fonction `choix2` est plus explicite pour un programmeur débutant car elle souligne le fait que le second bloc n'est exécuté que si la première condition est fausse.

Un autre style de programmation possible est d'utiliser une variable `resultat` dont seule la dernière valeur sera renvoyée par une unique instruction `return` en fin de fonction, comme illustré par la fonction `choix3` ci-dessous :

```
def choix3(n):
    if n < 0:
        resultat = 'négatif'
    elif n % 2 == 0:
        resultat = 'positif et pair'
    else:
        resultat = 'positif et impair'
    return resultat
```

Dans ce cas, il est indispensable de spécifier entièrement la structure conditionnelle, en incluant tous les mots-clés `elif` et `else` permettant de distinguer les différents cas.

2.2. Boucle « tant que » et ses variantes

Pour répéter un ensemble d'instructions dont l'exécution ou l'arrêt dépend d'une certaine condition, la plupart des langages proposent des structures du type « tant que *condition* faire *instructions* ». Elle peuvent prendre plusieurs formes selon les langages.

Itération avec condition au début

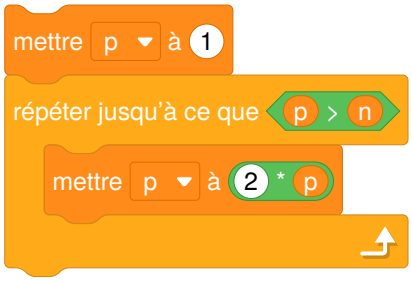
Tous les langages de programmation impératifs fournissent une structure de boucle où la condition est formulée au début. Sous cette forme, la condition est d'abord évaluée, et en fonction du booléen obtenu, soit l'itération est interrompue, soit le corps de la boucle est exécuté une fois puis la boucle recommence.

Ainsi, en Python, on pourra écrire

```
p = 1
while p <= n:
    p = 2 * p
```

et ce morceau de programme se terminera dès que la valeur de `p` devient strictement supérieure à celle de `n`. Si la condition de boucle est fausse dès le début, par exemple ici si `n` vaut 0, le corps de la boucle n'est pas exécuté du tout.

En Scratch, la structure analogue s'écrit « répéter jusqu'à ce que *condition* » ce qui signifie que l'itération est interrompue dès que la condition est vraie. Autrement dit, le corps de la boucle sera exécuté tant que la condition est fausse. Le programme précédent s'écrirait donc



Les deux structures sont équivalentes, la différence ne porte que sur la formulation de la condition, puisque la première forme donne une condition de continuation alors que la seconde donne une condition d'arrêt, or l'une est la négation de l'autre par définition.

Les élèves de collège rencontrent donc d'abord, avec Scratch, des boucles dont la condition est une condition d'arrêt. En seconde, ils découvrent le langage Python où les boucles s'écrivent avec une condition de continuation. Dans les deux cas le corps de la boucle peut ne jamais être exécuté. C'est la négation qui permet de passer de la condition d'arrêt à la condition de continuation. Sur ce point, la transition d'un langage à l'autre doit donc être abordée avec soin, car on sait que la négation logique est une notion difficile.

Scratch dispose également d'une instruction « attendre jusqu'à » où rien ne se passe tant que la condition n'est pas vérifiée. C'est en fait équivalent à une boucle « répéter jusqu'à » avec la même condition et aucune instruction dans le corps de boucle. La condition peut donc être vue comme condition d'arrêt pour la boucle d'attente. Cette instruction est utile surtout en programmation événementielle.

Itération avec condition à la fin

Certains algorithmes s'expriment plus naturellement avec une forme d'itération où le corps de la boucle est toujours exécuté au moins une fois et où la condition d'arrêt ou de continuation est testée après.

Par exemple, pour demander à l'utilisateur un nombre entier positif ou nul, on voudra suivre la procédure suivante:

> **répéter** dire « Entrer un nombre entier positif ou nul »
>
> saisir un entier n
>
> jusqu'à ce que $n \geq 0$.

Comme cette structure est utile, certains langages proposent un type de structure où la condition suit le corps de la boucle. C'est le cas par exemple en C (et dans les langages qui en dérivent comme C++, C#, Java, etc.) avec la structure `do ... while (condition);` ou en Pascal avec la structure `repeat ... until`. Ainsi, l'exemple suivant assure la saisie d'un entier positif ou nul :

```
repeat
  writeln('Entrer un nombre entier positif ou nul');
  readln(s);       (* lit une chaîne de caractères dans s *)
  val(s, n, err);  (* convertit s en entier dans n,
                      rend err non nul en cas d'erreur *)
until err = 0 and n >= 0;
```

Python et Scratch ne disposent pas d'une structure de ce type. Dans ces langages, il est donc nécessaire de se baser sur la boucle avec condition au début pour représenter ce mécanisme. Une façon simple est d'utiliser une boucle `while` en la faisant précéder d'une première exécution du corps de la boucle :

```
print("Entrer un nombre entier positif ou nul")
s = input()
n = int(s)
while n < 0:
    print("Entrer un nombre entier positif ou nul")
    s = input()
    n = int(s)
```

L'inconvénient de cette approche est qu'elle impose de répéter une partie du code, ce qui est généralement considéré comme une mauvaise pratique. Il est parfois possible d'éviter la répétition en choisissant judicieusement les valeurs initiales des variables :

```
n = -1
while n < 0:
    print("Entrer un nombre entier positif ou nul")
    s = input()
    n = int(s)
```

Cette façon de procéder n'est pas toujours applicable.

Itération infinie et interruption d'itération

L'itération avec condition est la forme la plus générale d'itération, elle permet de représenter les autres formes de boucles plus spécialisées comme l'itération numérique « pour i de 1 à n ». La contrepartie de cette expressivité est qu'elle permet d'écrire des programmes dont l'exécution ne termine pas. En effet, il est possible en général que la condition d'arrêt de soit jamais remplie.

Le fait de ne pas terminer peut parfois être souhaité, c'est le cas par exemple dans les programmes interactifs et plus généralement quand on suit un paradigme événementiel. Ainsi, Scratch propose une structure « répéter indéfiniment ». Pour faire la même chose dans un langage qui ne propose pas de structure analogue, on emploie une itération avec une condition de continuation toujours vraie, c'est-à-dire en écrivant `while True`.

Certains langages de programmation proposent par ailleurs un moyen d'interrompre une boucle en cours d'itération, indépendamment de la condition d'arrêt formulée sur la boucle. C'est le cas de Python et de C avec l'instruction `break`, ainsi que l'instruction `return` qui termine l'exécution d'une fonction en renvoyant une valeur. On s'en sert couramment dans des algorithmes de recherche. Dans les exemples suivants, on cherche un diviseur de `n` pour savoir si celui-ci est premier :

```
est_premier = True
for i in range(2, n):
    if n % i == 0:
        est_premier = False
        break
```

Dans le programme précédent, la réponse est contenue dans la variable booléenne `est_premier`. La fonction suivante suit le même principe mais utilise `return` :

```
def est_premier(n):
    for i in range(2, n):
        if n % i == 0:
            return False
    return True
```

L'instruction `break` n'est utile que si elle apparaît sous une condition. Celle-ci jouera alors le rôle d'une condition d'arrêt mais qui peut être placée n'importe où dans le corps de la boucle. Cela donne une autre façon d'écrire l'itération avec condition à la fin, ainsi l'exemple sur la saisie d'entier positif ou nul peut s'écrire de la façon suivante en utilisant une boucle *infinie* en Python :

```
while True:
    print("Entrer un nombre entier positif ou nul")
    s = input()
    n = int(s)
    if n >= 0:
        break
```

L'usage du `break` rend la compréhension et l'analyse des algorithmes plus difficiles puisqu'elle complique la formulation des conditions d'arrêt. On a donc intérêt à l'utiliser avec prudence avec les élèves.

3. Expressions booléennes

Dans cette partie, on étudie les principales façons de calculer des booléens. Même s'il est possible d'utiliser les littéraux[2] représentant des booléens, ces valeurs sont en général calculées à partir d'autres valeurs au moyen d'opérateurs ou de fonctions. On parle alors d'*expressions (à valeurs) booléennes*.

L'objectif n'est pas de chercher l'exhaustivité par rapport à un langage donné mais distinguer les différents ingrédients qui contribuent à l'écriture de telles expressions.

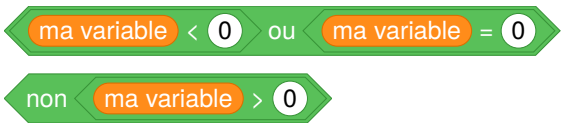
3.1. Produire des booléens : comparaisons et autres relations

Dans tous les langages de programmation, des opérateurs permettent de comparer deux valeurs (égalité, inégalité ou ordre) et de produire une valeur booléenne en conséquence.

Comparaisons en Scratch

Le langage Scratch ne propose que les opérateurs d'égalité et d'ordre strict, ce qui oblige à utiliser des opérateurs booléens pour former d'autres comparaisons. Il peut être intéressant de donner du sens au collège à la relation $\leq$ à l'aide de l'opérateur booléen « ou », ou comme complémentaire de la relation $>$:

[2] En programmation, un *littéral*, ou *valeur littérale*, est une valeur écrite explicitement dans un programme. En Python, on écrit par exemple `3` pour désigner la valeur entière 3, `'a'` pour désigner le caractère `a`, ou `False` pour désigner la valeur booléenne *faux*. *Petit glossaire de termes informatiques*, Commission Inter-IREM Informatique framagit.org/c3i/glossaire/-/wikis/litteral



Comparaisons en Python

En Python, six opérateurs permettent de comparer des valeurs de type quelconque et s'évaluent en une valeur booléenne. Deux concernent l'égalité :

- l'expression `a == b` est vraie lorsque les valeurs de `a` et `b` sont égales ;
- l'expression `a != b` est vraie lorsque les valeurs de `a` et `b` sont différentes.

Il est important de ne pas confondre l'opérateur d'égalité `==` avec l'affectation notée `=`. Il s'agit d'une erreur très fréquente chez les programmeurs débutants[3]. Il existe aussi un opérateur permettant de tester l'identité en mémoire de deux valeurs, que nous mentionnons brièvement plus loin.

```
>>> 2 + 2 == 4
True
>>> 'bonjour'[3:] == 'nuit'
False
>>> [1, 3, 5] != [8, 7]
True
>>> {'a': 8, 'b': [37]} == {'b': [36 + 1], 'a': 8}
True
```

Les quatre autres opérateurs de comparaison concernent les types pour lesquels il y a une notion d'ordre :

- `a < b` est vraie si la valeur de `a` est strictement inférieure à celle de `b`, fausse sinon ;
- `a > b` fait la comparaison réciproque ;
- `a <= b` et `a >= b` font les comparaisons larges.

```
>>> 3 < 8
True
```

Remarques sur les comparaisons

Chaînes de caractères Pour les chaînes de caractères, c'est l'ordre dit *lexicographique* qui s'applique, tant en Scratch qu'en Python : on compare les premiers caractères des deux chaînes, puis les seconds si les premiers sont égaux, et ainsi de suite.

```
>>> 'alphabet' < 'alphonse'
True
>>> 'un' < 'deux'
False
>>> '150' < '90'  # ce sont les caractères qui sont comparés
True
```

Lorsqu'on compare des chaînes de caractères, ce sont les indices de caractères (selon le standard Unicode, spécifiquement) qui servent à comparer les caractères, ce qui ne correspond pas à l'ordre alphabétique :

```
>>> 'Z' < 'a'  # les majuscules sont avant les minuscules
True
>>> 'z' < 'é'  # les lettres accentuées viennent après celles sans accents
True
```

Listes et ensembles C'est également l'ordre lexicographique qui permet de comparer des listes et des n-uplets en Python :

```
>>> (1, 0) > (0, 1)
True
>>> (1, 0, 1) < (1, 1)
True
```

[3]En Python, l'interpréteur signale dans ce cas une erreur car une affectation n'est pas considérée comme une expression. En C, cette utilisation est parfois signalée par le compilateur sous forme d'avertisssement car cette écriture est légale et peut être voulue.

```
>>> [1, 2, 3] > [1, 2, 3, 4]
False
```

Pour le type `set`, les opérateurs de comparaison désignent la relation mathématique d'inclusion (stricte ou large) :

```
>>> {'a', 'b'} < {'b', 'c', 'a'}
True
>>> {'b', 'a'} <= {'a', 'b'}
True
```

Spécificités de Python Python autorise à écrire, par commodité, des comparaisons multiples comme `0 <= i < len(lst)`. Cette expression est équivalente à `0 <= i and i < len(lst)`, ce qui est une originalité de Python.

L'intérêt de cette facilité de syntaxe ne fait pas l'unanimité parmi les enseignants, d'autant plus qu'elle fait figure d'exception et peut être source d'erreur.

```
>>> 0 < 3 > 2  # comment comprendre cette expression ?
True
```

Par ailleurs, les opérateurs de comparaison peuvent être utilisés sur des opérandes de types différents seulement dans les cas où une conversion implicite est possible entre les deux types, par exemple entre types numériques `int` et `float`. Dans les autres cas, les tests d'égalité considèrent en général comme différentes des valeurs de types différents, alors que les opérateurs d'ordre provoquent une erreur[4].

```
>>> 2 == 2.0
True
>>> 8.5 < 3
False
>>> 5 != 'patate'
True
>>> 5 <= 'patate'
TypeError: '<=' not supported between instances of 'int' and 'str'
```

Il est important d'être vigilant sur les conversions automatiques possibles lors de comparaisons entre nombres, qui peuvent avoir des résultats inattendus, en particulier lorsqu'elles impliquent des nombres de type `float` (mais ceci n'est pas notre propos principal, voir par exemple la brochure IREM *Algorithmique et programmation au collège* (Beffara et al, 2017)).

Autres relations

Nous mentionnons ici deux autres relations qui nous semblent utiles dans certains contextes. Selon le langage, il peut exister d'autres opérateurs à valeur booléenne, nous ne cherchons pas ici à en dresser une liste exhaustive.

Appartenance à une collection En Python, `A in B` détermine si la valeur `A` apparaît dans la structure de donnée `B` qui peut être par exemple une liste ou un dictionnaire (dans ce cas le test concerne les clés du dictionnaire). Sa négation peut se noter `A not in B`.

Attention: `not A in B` signifie la même chose que `A not in B` et non `(not A) in B`. Le `not`, qui est prioritaire sur `and` et sur `or`, ne l'est pas sur l'opérateur `in`. Il sera donc préférable d'utiliser `A not in B`.

Sur les chaînes de caractères, cet opérateur teste l'apparition d'une chaîne comme fragment d'une autre :

```
>>> 'ane' in 'banane'
True
```

Ces opérateurs existent également en Scratch pour les chaînes de caractères (en vert) et pour les listes (en orange) :

[4] Les règles précises qui s'appliquent en Python pour la comparaison de valeurs de types différents sont assez subtiles, mais le détail n'est pas notre propos ici. La documentation du langage explique précisément le mécanisme. Voir par exemple https://docs.python.org/fr/3/reference/datamodel.html#object.__lt__

Identité en mémoire Certains langages de programmation proposent un opérateur permettant de tester l'identité en mémoire de deux objets. C'est le cas de la comparaison de pointeurs en C, de `is` en Python, de `==` en OCaml, etc.

Voici un exemple en Python illustrant la différence entre les opérateurs `==` et `is` :

```
>>> lst_a = [1, 2, 3]
>>> lst_b = [1, 2, 3]
>>> lst_a is lst_b
False
>>> lst_a == lst_b
True
```

Mais :

```
>>> lst_c = lst_b
>>> lst_b is lst_c
True
```

Dans ce dernier cas, `lst_b` et `lst_c` désignent en réalité le *même* objet dans la mémoire.

Ces opérateurs et la nuance entre égalité de valeur et identité en mémoire mettent en jeu des concepts difficiles qui dépassent le périmètre de ce texte, nous nous contentons donc de signaler leur existence.

3.2. Opérateurs entre booléens

Comme les valeurs booléennes représentent naturellement des valeurs de vérité, les opérations élémentaires pour combiner des booléens sont de nature logique. Les opérateurs logiques usuels sont la conjonction « et », la disjonction « ou » et la négation « non ». Si A et B sont deux valeurs de vérité :

- conjonction : « A **et** B » est vrai lorsque les deux opérandes sont vrais,
- disjonction : « A **ou** B » est un *ou inclusif* : il est vrai lorsqu'au moins un des deux opérandes est vrai,
- négation : « **non** A » est vrai lorsque A est faux et inversement.

Ces opérateurs existent dans tous les langages de programmation, même ceux qui n'ont pas de type booléen explicite. Dans ce dernier cas, ils renvoient une valeur dont l'interprétation comme valeur de vérité correspond à ces définitions.

On retrouve fréquemment deux syntaxes :

- Python et Scratch utilisent les mots : `A and B`, `A or B`, `not A` (Scratch en français utilise naturellement `et`, `ou` et `non`) ;
- Le langage C et ses dérivés utilisent `A && B` et `A || B` pour conjonction et disjonction et `!A` pour la négation.

Voici un exemple d'évaluation d'expressions booléennes dans l'interpréteur Python :

```
>>> a = True
>>> b = False
>>> a or b
True
>>> a and b
False
>>> not a
False
```

Évaluation séquentielle

Dans la plupart des langages de programmation, comme Python ou C, les opérateurs booléens ont un comportement particulier. Lors de l'évaluation des opérateurs `and` et `or`, le deuxième opérande n'est évalué que si c'est nécessaire, c'est-à-dire lorsque la valeur du premier opérande ne permet pas de déterminer à elle seule la valeur de l'expression.

En Python[5], dans l'expression `a and b` l'expression `b` n'est évaluée que si `a` est interprétée comme vraie. En effet, si `a` vaut `False`, `a and b` vaudra toujours `False` quelle que soit la valeur de `b`. Le fait que l'expression à

[5] https://docs.python.org/3/reference/expressions.html#boolean-operations

droite de `and` ne soit pas évaluée du tout peut être utilisé dans une situation où son évaluation serait susceptible d'échouer. Par exemple, dans l'expression

```
x >= 0 and sqrt(x) < 5
```

la fonction `sqrt` ne sera pas appelée si `x` est négatif (ce qui provoquerait une erreur). De même, si l'on suppose que `ma_liste` désigne une valeur de type `list`, l'évaluation de

```
len(ma_liste) > 0 and ma_liste[0] == 'a'
```

ne provoque pas d'erreur, même si `ma_liste` est vide, car dans ce cas le membre gauche est faux et `ma_liste[0] == 'a'` n'est pas évaluée. Un autre avantage de ce type d'écriture est d'éviter l'imbrication de structures conditionnelles :

```
if len(ma_liste) > 0 and ma_liste[0] == 'a':
    print('la liste commence par un a')
else:
    print('la liste ne commence pas par un a')
```

est équivalent au code suivant, qui est plus explicite sur l'ordre d'évaluation mais nécessite de répéter le deuxième cas :

```
if len(ma_liste) > 0:
    if ma_liste[0] == 'a':
        print('la liste commence par un a')
    else:
        print('la liste ne commence pas par un a')
else:
    print('la liste ne commence pas par un a')
```

Conceptuellement, on peut considérer que la sémantique de l'expression `c = a and b` est proche de celle de la suite d'instructions

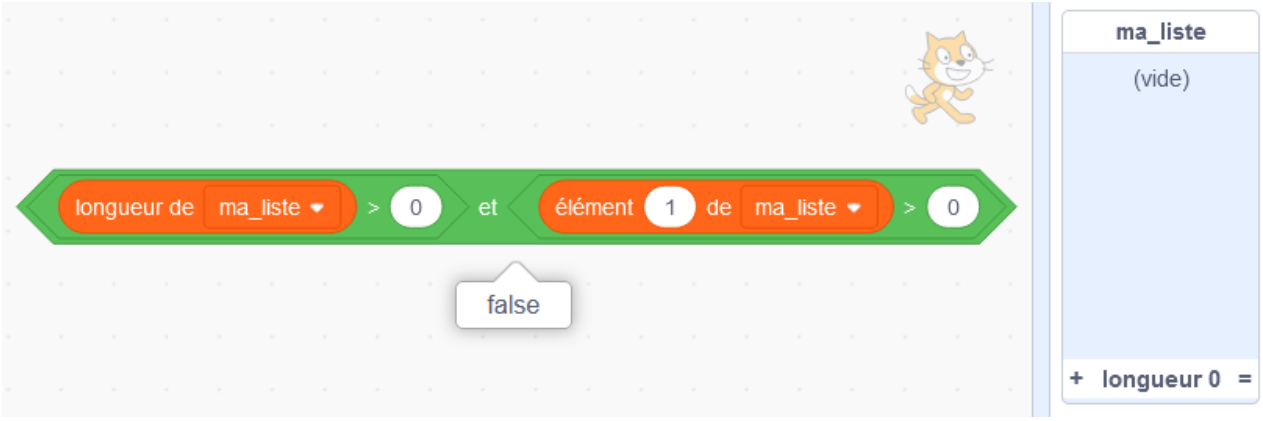
```
if a:
    c = b
else:
    c = a
```

On voit ici que l'expression `b` n'est pas évaluée si `a` est interprétée comme fausse. Ceci montre également que la valeur de l'expression `a and b` est, selon les cas, du type de `a` ou du type de `b`. Elle n'est donc pas systématiquement de type booléen.

De même, dans la condition `a or b`, l'expression `b` n'est évaluée que si `a` est interprétée comme fausse. En effet, si `a` est vraie, `a or b` le sera aussi quelle que soit la valeur de `b`. Lorsqu'on exécute l'affectation `c = a or b`, tout se passe comme si l'on exécutait la suite d'instructions

```
if a:
    c = a
else:
    c = b
```

On a l'impression de retrouver la même chose en Scratch. Avec les instructions suivantes, il semble qu'il n'y a pas de tentative d'évaluation du premier élément de la liste si celle-ci est vide.



En fait c'est un peu plus compliqué que cela, étant donné que Scratch est conçu pour ne pas planter ! Si la liste est vide, une lecture du premier élément donne `rien` mais ne plante pas et sa comparaison avec une autre valeur donne `False`.

Cette petite expérience ne nous permet donc pas de trancher. Cependant, puisque Scratch est programmé en Javascript et qu'en Javascript, les opérateurs `and` et `or` sont, comme en Python, séquentiels, on peut penser qu'il en est de même avec Scratch. Malheureusement, il n'est pas possible d'illustrer cet état de fait avec des exemples. Il est également trop tôt pour abroder ces points avec des élèves de collège.

Une curiosité de Python L'interprétation des valeurs non-booléennes comme valeurs de vérité en Python peut provoquer des phénomènes inattendus. Voici un exemple amusant le démontrant :

```
>>> 'patate' and 'courgette'
'courgette'
>>> type('patate' and 'courgette')
str
```

Quand on évalue l'expression `'patate' and 'courgette'`, `'patate'` en tant que chaîne non vide est considérée comme vraie, l'expression vaut donc `'courgette'`. L'opérateur `or` présente le même genre de phénomène :

```
>>> 'patate' or 'courgette'
'patate'
>>> type('patate' or 'courgette')
str
```

Quand on évalue l'expression `'patate' or 'courgette'`, `'patate'`en tant que chaîne non vide est considérée comme vraie, l'expression vaut donc `'patate'`.

Table de vérité

Dans certains domaines comme la logique mathématique ou l'architecture des circuits, on rencontre des *tables de vérité*. Il s'agit de tableaux qui énumèrent tous les cas possibles pour une opération booléenne. Par exemple :

a	b	non (a et b)
faux	faux	vrai
vrai	faux	vrai
faux	vrai	vrai
vrai	vrai	faux

Dresser ainsi une table de tous les cas est possible parce qu'il n'y a qu'un nombre fini de cas à considérer (2^n possibilités dans le cas d'une expression à n opérandes). Dans l'exemple ci-dessus, avec deux opérandes a et b, il n'y a que les quatre cas du tableau, celui-ci définit donc entièrement la valeur de l'expression considérée.

Voici un exemple de programme permettant d'afficher la table de vérité précédente :

```
print ('a', 'b', 'not(a and b)', sep='\t')
for a in (False, True):
    for b in (False, True):
        print(a, b, not(a and b), sep='\t')
```

L'exécution de ce programme provoque l'affichage suivant :

```
a       b       not(a and b)
False   False   True
False   True    True
True    False   True
True    True    False
```

Utiliser des tables de vérité permet d'établir que certaines expressions sont équivalentes, en montrant qu'elles prennent la même valeur dans *tous* les cas.

a	b	non (a et b)	(non a) ou (non b)
faux	faux	vrai	vrai
vrai	faux	vrai	vrai
faux	vrai	vrai	vrai
vrai	vrai	faux	faux

Le fait que les deux dernières colonnes de ce tableau coïncident montre que les expressions « non (a et b) » et « (non a) ou (non b) » sont équivalentes (propriété dite « de De Morgan »), au sens où leur valeur de vérité est identique pour toutes les valeurs de vérité possibles de a et b.

Pour démontrer cette proposition à l'aide de Python, il est également possible d'énumérer et vérifier tous les cas, comme le montre l'exemple suivant :

```
print('Vérification de : not(a and b) == not a or not b')
for a in (False, True):
    for b in (False, True):
        if (not (a and b)) != (not a or not b):
            print('pas égal pour les valeurs', a, 'et', b)
```

Une autre manière de formuler cette vérification est l'écriture de doctests Python[6] :

```
def distrib_not_and(a, b):
    """
    >>> distrib_not_and(True, True)
    True
    >>> distrib_not_and(True, False)
    True
    >>> distrib_not_and(False, True)
    True
    >>> distrib_not_and(False, False)
    True
    """
    return (not (a and b)) == (not a or not b)
```

Disjonction exclusive

Il existe aussi en logique et dans le langage courant un opérateur à deux opérandes de *disjonction exclusive*, également appelé *ou exclusif*, parfois noté *xor* ou *eor*. La disjonction exclusive de deux valeurs est vraie si et seulement si un seul des opérandes est vrai.

En pratique, l'usage de cet opérateur dans l'écriture de conditions est rare, dans la mesure où il est souvent plus naturel de raisonner en termes de disjonction inclusive (il est par contre très utilisé comme opérateur sur les bits, notamment en cryptographie et en électronique). Pour cette raison, la plupart des langages n'ont pas d'opérateur correspondant. Néanmoins, pour a et b booléens, la disjonction exclusive de a et b est vraie si et seulement si a et b sont distincts, ce qui permet d'employer l'opérateur de comparaison pour représenter cette opération.

Python ne propose pas d'opérateur `xor`. On peut employer à sa place l'opérateur d'inégalité `!=` entre deux valeurs de type `bool` : `a != b`. On pourra se convaincre de l'équivalence avec le ou exclusif en dressant les tables de vérité des deux opérations. Cette écriture ne fonctionne cependant que si `a` et `b` sont bien de type `bool`. Si ce n'est pas le cas, il est nécessaire de les convertir au préalable avec quelque chose comme `bool(a) != bool(b)`. Par exemple :

[6] Cf. notes de cours de l'université de Lille, liens en fin de document.

```
>>> 1 != 2
True
>>> bool(1) != bool(2)
False
```

Confusion possible avec les opérateurs bit à bit

Dans les langages de programmation de la famille du C il existe des opérateurs `&` et `|` mais ce ne sont pas des opérateurs booléens : ils agissent (bit par bit) sur l'écriture binaire des entiers. Ces opérateurs ont un sens très différent des opérateurs `&&` et `||`, et il est facile de confondre leurs syntaxes.

En Python il existe de même `&`, `|`, `^` et un certain nombre d'autres opérateurs qui acceptent exclusivement des opérandes de type numérique (y compris de type `bool`) et calculent des opérations sur l'écriture binaire des nombres[7].

Même si leur comportement est proche de celui des opérateurs booléens quand on les applique à des opérandes de valeur `True` ou `False`, il est considéré comme très risqué voire incorrect de les utiliser de cette manière.

Voici un exemple en Python qui illustre la différence entre les opérateurs `and` et `&` :

```
>>> 4 and 2  # ici 4 est interprété comme 'vrai'
2
>>> 4 & 2    # 4 s'écrit 100 et 2 s'écrit 010 en binaire
0
```

L'exemple suivant montre que l'opérateur `&` n'est pas défini sur les types non numériques :

```
>>> "a" & 1
...
TypeError: unsupported operand type(s) for &: 'str' and 'int'
```

Expressions conditionnelles

En Python, comme dans d'autres langages comme le C, il existe un opérateur ternaire permettant de construire des *expressions conditionnelles*, dont voici un exemple :

```
-1 if n < 0 else +1
```

Il s'agit bien d'une *expression*, qui s'évalue en une *valeur*, et qu'on peut par exemple utiliser comme partie droite d'une affectation :

```
signe = -1 if n < 0 else +1
```

Voici un autre exemple utilisant des chaînes :

```
print("il y a", n,
      ("journal" if n <= 1 else "journaux"))
```

Des constructions équivalentes sont possibles dans la plupart des autres langages de programmation.

Il n'est pas indispensable de présenter cette structure à des élèves débutants. Elle est en particulier susceptible d'entraîner une confusion avec la structure de contrôle `if ... else`.

3.3. Fonctions à valeur booléenne

Pour tester des propriétés plus complexes que de simples comparaisons, on a naturellement recours à des fonctions qui renvoient des valeurs booléennes. On appelle parfois ce type de fonctions des *prédicats* par analogie avec la logique mathématique.

Conceptuellement, les fonctions à valeur booléenne sont similaires aux opérateurs à valeur booléenne comme les comparaisons. Les opérateurs servent simplement à donner une syntaxe plus concise pour des opérations fréquentes.[8]

Dans cette section, on donne quelques exemples de fonctions booléennes prédéfinies du langage Python, puis de définitions de telles fonctions.

[7] https://docs.python.org/fr/3/library/stdtypes.html#bitwise-operations-on-integer-types

[8] Les cas des opérateurs `and` et `or` et de l'opérateur ternaire présenté plus haut sont particuliers parce qu'ils n'évaluent pas forcément toutes leurs opérandes, ce qui ne peut pas se définir avec une fonction.

Fonctions booléennes prédéfinies en Python

Méthodes sur les chaînes de caractères Le type prédéfini `str` fournit un grand nombre de méthodes à valeur booléenne : `isalpha()` – est composé uniquement de lettres –, `isdigit()` – uniquement de chiffres –, `islower()` ou `isupper()` - est entièrement en minuscules ou entièrement en majuscules, `istitle()` - commence par une majuscule puis ne contient que des minuscules - etc.

La liste de ces méthodes et leur description détaillée est disponible dans la documentation en ligne.

L'annexe sur l'exemple d'utilisation de fonctions booléennes prédéfinies sur les chaînes de caractères illustre l'utilisation de la fonction `isdigit()`.

Combinateurs `all()` et `any()` La fonction `all()` reçoit comme paramètre un objet itérable (par exemple une liste), et renvoie `True` si tous les éléments de la liste sont interprétables comme la valeur de vérité « vrai », et `False` sinon. Cette fonction imite le comportement du quantificateur universel ∀ en mathématiques.

```
>>> prenoms = ['Sylvie', 'Cécile', 'Anne', 'Malika']
>>> avec_majuscule = [prenom.istitle() for prenom in prenoms]
>>> all(avec_majuscule)  # tous ont une majuscule en début de prénom
True
```

Cette fonction est souvent combinée avec la fonction `map` (appliquer pour tous) :

```
>>> all(map(str.istitle, prenoms))
True
```

```
>>> all(s.istitle() for s in prenoms)
True
```

La fonction prédéfinie `any()` est similaire. Elle correspond au quantificateur existentiel ∃, elle renvoie `True` si un itérable contient au moins une valeur interprétable comme `True`, et `False` sinon.

```
>>> nombres = [0, 1, 2, 3, 4, 5, 6]
>>> impair = [n % 2 == 1 for n in nombres] # indique les éléments impairs
>>> any(impair)  # il y a au moins un nombre impair
True
```

Remarquons que dans cet exemple, lors du calcul de la liste `impair`, le test `n % 2 == 1` est évalué pour tous les éléments de la liste `nombres`. Si on cherche à savoir si cette liste contient au moins un nombre impair, il n'est en fait pas nécessaire de faire le test sur tous, il suffit de s'arrêter au premier cas trouvé. Python permet de réaliser ce comportement au moyen des *expressions génératrices* qui sont en quelques sorte des listes « paresseuses » dont les éléments ne sont évalués que si nécessaire. On écrirait alors:

```
>>> nombres = [0, 1, 2, 3, 4, 5, 6]
>>> any(n % 2 == 1 for n in nombres)
True
```

On ne détaillera pas plus cette fonctionnalité qui dépasse le cadre du présent article.

Conversion avec `bool()` La fonction `bool` construit une valeur booléenne à l'aide de son argument, comme toute fonction dont le nom est celui d'un type prédéfini. Cela revient à appliquer les règles qui déterminent la valeur de vérité d'une valeur quelconque en fonction de son type :

```
>>> bool(0)
False
>>> bool('zéro')
True
```

D'autres exemples sont donnés en annexe.

Définition de fonctions à valeur booléenne

La définition d'une telle fonction se base sur d'autres fonctions ou opérateurs booléens pour calculer son résultat. Voici un exemple simple:

```
def est_impair(n):
    return (n % 2) == 1
```

Une fois définie, cette fonction peut être utilisée en tant que condition :

```
if est_impair(n):
    ...
else:
    ...
```

En termes logiques, une telle fonction peut être appelée *prédicat* puisqu'elle détermine une valeur de vérité en fonction de l'objet qu'elle reçoit en argument. Les raisons qui peuvent justifier la définition de fonctions à valeurs booléenne sont les mêmes que pour n'importe quelle définition de fonction : cela permet de nommer une certaine expression pour gagner en lisibilité, d'éviter la répétition si le prédicat sert plusieurs fois, ou encore de mieux structurer le programme si le calcul du prédicat est complexe.

Un exemple typique : recherche exhaustive Un exemple très classique de fonction à valeur booléenne est la recherche exhaustive d'un élément dans une liste :

```
def recherche(liste, valeur):
    for element in liste:
        if element == valeur:
            return True
    return False
```

Dans cet exemple, il serait évidemment incorrect de renvoyer directement une valeur booléenne dans le corps de la boucle, ou de compléter le corps d'un `else: return False`, ce qui est une erreur fréquente :

```
def recherche_fausse(liste, valeur):
    for element in liste:
        if element == valeur:
            return True
        else:
            return False
```

Selon le contexte, le public et les objectifs poursuivis, certains auteurs et enseignants évitent d'utiliser l'instruction `return` au sein d'une boucle. Il est dans ce cas envisageable de restructurer la fonction.

3.4. Écritures redondantes

Comme on l'a vu, les valeurs calculées par des expressions booléennes peuvent aussi bien servir comme conditions dans les structures de contrôle qu'être stockées dans des variables, comme on le fait pour les valeurs numériques ou textuelles. Il y a donc un rôle de contrôle et un rôle de donnée, qui correspondent à des intentions différentes. Le fait qu'une même valeur puisse servir aux deux rôles dans un même programme est conceptuellement difficile pour les débutants en programmation et explique certaines tournures redondantes que nous détaillons dans la suite.

Comparaisons inutiles dans des expressions

Supposons qu'on souhaite déterminer si un nombre `n` est divisible par 3 ou 5. Une fois définies les deux variables

```
est_multiple_de_3 = (nombre % 3 == 0)
est_multiple_de_5 = (nombre % 5 == 0)
```

on peut initialiser la variable booléenne

```
est_multiple_de_3_ou_5 = est_multiple_de_3 or est_multiple_de_5
```

dont la valeur est `True` si et seulement si `n` est divisible par 3 ou par 5 (ou les deux). On rencontre parfois l'écriture équivalente :

```
est_multiple_de_3_ou_5 =
    (est_multiple_de_3 == True) or (est_multiple_de_5 == True)
```

Quoique correcte, cette expression effectue des comparaisons inutiles. En effet, une variable `b` qui contient un booléen a la même valeur de vérité que l'expression `b == True`.

b	b == True
True	True
False	False

On préférera aussi en général utiliser l'expression booléenne `not b` plutôt que `b == False`.

b	b == False	not b
True	False	False
False	True	True

Le guide de bonnes pratiques « PEP 8[9] » signale qu'il n'est pas recommandé de comparer les résultats d'expressions booléennes aux littéraux `True` ou `False`. En particulier, de telles comparaisons peuvent avoir un comportement surprenant lié aux conversions de types. D'expérience, il est cependant fréquent que des programmeurs peu expérimentés recourent à ce genre d'écriture.

Comparaisons inutiles dans des structures de contrôle

On rencontre fréquemment des structures conditionnelles de la forme suivante :

```
if est_bissextile == True:
    nombre_jours = nombre_jours + 1
```

Si `est_bissextile` est une variable booléenne, cela est équivalent à la formulation plus simple suivante :

```
if est_bissextile:
    nombre_jours = nombre_jours + 1
```

Le fait de comparer une variable booléenne à `True` dans une structure conditionnelle est le signe que quelque chose n'a pas été compris dans la notion de valeur booléenne : l'élève pense par exemple qu'une condition doit forcément contenir un opérateur de comparaison. Or l'expression `est_bissextile`, réduite à une variable, n'en contient pas, de ce fait il est possible qu'elle ne soit pas perçue comme une condition valide. Ainsi, une comparaison entre nombres comme `n >= 0` sera plus spontanément perçue comme une condition valable, c'est pourquoi les élèves n'écriront pas

```
if (n >= 0) == True:
    resultat = "positif"
```

mais bien

```
if n >= 0:
    resultat = "positif"
```

Ce phénomène se produit également lors de l'appel à des fonctions qui renvoient une valeur booléenne. Ainsi, au lieu d'une condition comme

```
if est_premier(n):
    ...  # traitement
```

on rencontrera parfois le code redondant suivant :

```
if est_premier(n) == True:
    ...  # traitement
```

On remarque la même chose lors de l'usage de boucle `while`. Si `continuer` est un booléen, on préférera

```
while continuer:
    ...  # traitement
```

à

```
while continuer == True:
    ...  # traitement
```

Structures conditionnelles superflues

Dans les fonctions renvoyant une valeur booléenne, on trouve régulièrement dans les productions d'élèves un code de la forme :

[9] https://peps.python.org/pep-0008/#programming-recommendations

```
def est_pair(n):
    if n % 2 == 0:
        return True
    else:
        return False
```

Cette fonction peut être remplacée par :

```
def est_pair(n):
    return n % 2 == 0
```

En effet, si la condition `n % 2 == 0` vaut `True`, la fonction renvoie `True` et si la condition vaut `False`, la fonction renvoie `False`. Autrement dit dans tous les cas la fonction renvoie la veleur de vérité de la condition. Renvoyer simplement la valeur de `n % 2 == 0` permet donc d'éviter un branchement conditionnel superflu.

Il y a des considérations analogues pour les deux séquences suivantes:

```
def est_impair(n):
    if est_pair(n):
        return False
    else:
        return True
```

Dans tous les cas la valeur renvoyée est celle de `not est_pair(n)`. On préfèrera donc :

```
def est_impair(n):
    return not est_pair(n)
```

Bien entendu, le même principe s'applique quelle que soit la complexité de l'expression booléenne utilisée dans la condition.

```
def est_multiple_de_trois_ou_cinq(nombre):
    if nombre % 3 == 0 or nombre % 5 == 0:
        return True
    else:
        return False
```

Il est donc possible de réécrire cette fonction de manière plus concise :

```
def multiple_de_trois_ou_cinq(nombre):
    return nombre % 3 == 0 or nombre % 5 == 0
```

Si certains élèves ont du mal à s'en convraincre, il est toujours possible d'utiliser une table de vérité pour les y aider :

`n % 3 == 0`	`n % 5 == 0`	`n % 3 == 0 or n % 5 == 0`	Valeur de retour
`True`	`True`	`True`	`True`
`True`	`False`	`True`	`True`
`False`	`True`	`True`	`True`
`False`	`False`	`False`	`False`

Ceci met en évidence le fait qu'une valeur booléenne est une valeur comme une autre (au même titre qu'un entier ou une chaîne de caractères par exemple). Cette version peut être considérée comme plus lisible, à condition d'avoir compris le statut des expressions et valeurs booléennes.

Par ailleurs, la seconde version peut être marginalement plus rapide que la première à l'exécution. Cependant la différence est minime et même difficilement observable, ce qui fait que l'efficacité pure n'est pas un argument fort en faveur d'une forme ou de l'autre. D'autre part, on se gardera bien de confondre la concision et l'efficacité.

On pourra alors réserver la forme longue avec instruction conditionnelle à des blocs nécessitant plusieurs instructions ou des expressions plus complexes, comme dans ce deuxième exemple :

```
def cas_de_base(L):
    if len(L) == 0:
        return True
    else:
        return L[0] == 0
```

On peut néanmoins également écrire :

```
def cas_de_base(L):
    return len(L) == 0 or L[0] == 0
```

Attention, dans ce dernier cas la fonction s'appuie sur le mécanisme d'évaluation séquentiel particulier des opérateurs booléens que nous avons décrit au paragraphe Évaluation séquentielle des opérateurs. En effet, le test sur la nullité du premier élément de la liste ne sera effectué que si la liste n'est pas vide. Si la liste est vide, la deuxième opérande n'est pas évaluée. Écrire l'expression dans l'autre sens ne fonctionnerait pas si la liste `L` était vide.

Comme le montrent ces exemples, il est nécessaire de rechercher un bon compromis entre difficulté conceptuelle pour les élèves, longueur et efficacité du code, respect des bonnes pratiques communément admises, et d'annoncer clairement les éventuelles exigences aux élèves.

Généralement il est préférable de remplacer

```
def fonction():
    if expression_a_valeur_booleenne:   # l'expression est une condition
        return True
    else:
        return False
```

par

```
def fonction():
    return expression_a_valeur_booleenne
```

et

```
def fonction():
    if expression_a_valeur_booleenne:
        return False
    else:
        return True
```

par

```
def fonction():
    return not expression_a_valeur_booleenne
```

Il nous semble préférable de ne pas sanctionner l'emploi de la forme longue dans l'évaluation des productions d'élèves débutants. Cependant, comprendre la notion d'expression à valeur booléenne est un objectif d'apprentissage important, c'est pourquoi il peut être judicieux d'insister sur l'existence de la forme concise quand l'occasion se présente.

Nous formulons l'hypothèse que ces différents cas de comparaisons redondantes sont liées au double rôle du booléen, à la fois donnée et condition influant sur le flot d'exécution. Ainsi, en présence d'une variable `b` contenant un booléen, le fait d'écrire `if b == True` montre que, dans l'esprit de l'apprenant, l'expression `b` a purement le statut de donnée et que le fait d'écrire explicitement la comparaison `b == True` produit une expression qui a le statut de condition. Réciproquement, l'emploi de `return True` et `return False` dans les deux branches d'un `if` montre que l'expression qui suit `if` n'a que le statut de condition et pas celui de donnée qui rendrait légitime son emploi comme valeur de retour de fonction.

4. Étude de cas : archétypes de variables booléennes

On s'intéresse dans cette section à quelques cas d'usage typiques de variables booléennes, en s'inspirant de la catégorisation des rôles de variables par Samurçay (1985), enrichie par Sajaniemi (2002) et reprise par Sorva et al. (2007).

À titre d'exemple pour illustrer les différents usages, on étudie l'implémentation d'un prédicat `est_triee` qui vérifie qu'une liste est triée par ordre croissant. On considère qu'une liste à 0 ou 1 élément est triée. Mathématiquement, la liste L est triée si pour tous indices i et j valides avec $i \leq j$ on a $L[i] \leq L[j]$. Ceci peut s'exprimer de manière équivalente et disant que pour tout indice i on a $L[i] \leq L[i+1]$.

Les exemples qui suivent sont formulés en Python et son type booléen, mais les mêmes observations s'appliquent dans n'importe quel langage même s'il n'a pas de type booléen. Par exemple, en Scratch, on pourra utiliser une variable numérique et la convention que la valeur 0 représente le faux et la valeur 1 représente le vrai.

Quelle que soit l'implémentation choisie, on souhaite que la fonction satisfasse les tests suivants :

```
>>> est_triee([])
True
>>> est_triee([6])
True
>>> est_triee([2, 4, 7, 94])
True
>>> est_triee([6, 3, 9])
False
>>> est_triee([3, 9, 6])
False
```

Voici une première écriture possible de cette fonction[10] :

```
def est_triee_v1(lst):
    """
    Teste si une liste est triée en ordre croissant.

    >>> est_triee_v1([])
    True
    >>> est_triee_v1([2, 4, 7, 94])
    True
    >>> est_triee_v1([6, 3, 9])
    False
    >>> est_triee_v1([3, 9, 6])
    False
    """
    for i in range(1, len(lst)):
        if lst[i - 1] > lst[i]:
            return False
    return True
```

L'utilisation de `return` à l'intérieur d'une boucle `for` peut amener des confusions dans la compréhension même des notions d'itérable ou de boucle d'itération, de la même façon que `break` évoqué plus haut. On peut chercher des implémentations ne faisant pas appel à une sortie de boucle prématurée. Celles-ci font apparaître certains usages intéressants des variables booléennes.

4.1. Accumulateur booléen

Dans la version suivante, on utilise une variable booléenne comme *accumulateur* dont la valeur est mise à jour à chaque nouvelle itération de la boucle.

```
def est_triee_v2(lst):
    resultat = True
    for i in range(1, len(lst)):
        resultat = resultat and lst[i - 1] <= lst[i]
    return resultat
```

Cette version revient à écrire

```
resultat = L[0] <= L[1] and L[1] <= L[2] and ... and L[n-2] <= L[n-1]
```

où `n` serait la longueur de la liste, et en remplaçant le `...` par les comparaisons intermédiaires.[11] En effet, la liste L est triée si et seulement si $L[0] \leq L[1]$ et $L[1] \leq L[2]$ et … et $L[n\text{-}2] \leq L[n\text{-}1]$ puisque cette expression est fausse si et seulement si il existe un indice i tel que $L[i\text{-}1] \leq L[i]$ est faux.

La propriété invariante dans cet exemple est que `resultat` vaut `True` si et seulement si la portion de la liste d'indices strictement inférieurs à `i` est triée. En sortie de boucle, on a donc bien la propriété souhaitée que `resultat` vaut `True` si et seulement si la liste entière est triée.

[10] Dans cet exemple nous avons choisi d'inclure une chaîne de documentation (*docstring*) contenant des tests unitaires (*doctests*) dans le corps de la fonction. Par souci de concision, nous n'en inclurons pas dans les exemples qui suivent. Ce type d'annotation des fonctions est néanmoins généralement considéré comme une *bonne pratique*.

[11] Le `...` cache un « et ainsi de suite » qui est le même que celui du $1+2+\ldots+n$ qui représente la somme des n premiers entiers et qui se calcule avec un accumulateur numérique et des itérations.

4.2. Drapeau à sens unique

Dans la variante qui suit, on utilise une variable booléenne `lst_est_triee`, utilisée comme drapeau (comme défini dans la section sur les variables booléennes) mais ne contrôlant pas directement la continuation de la boucle. Cette variable vaut initialement `True`, et on lui donne la valeur `False` dès que deux éléments consécutifs mal ordonnés sont rencontrés dans la liste.

```
def est_triee_v3(lst):
    lst_est_triee = True
    for i in range(1, len(lst)):
        if lst[i - 1] > lst[i]:
            lst_est_triee = False
    return lst_est_triee
```

On peut remarquer que la variable `lst_est_triee` dans cette fonction respecte le même invariant que `resultat` dans `est_triee_v2()`.

Il est important de remarquer que les fonctions `est_triee_v2()` et `est_triee_v3()` ne sont pas algorithmiquement équivalentes à `est_triee_v1()`. En effet, celles-ci parcourent toujours entièrement l'ensemble des indices quelles que soient les valeurs rencontrées,[12] alors que `est_triee_v1()` s'interrompt dès qu'un cas d'inversion de l'ordre est rencontré.

4.3. Drapeau de continuation de boucle

Dans la version suivante de la fonction `est_triee()`, la variable booléenne `continuer` est utilisée comme un drapeau qui indique s'il faut continuer à répéter le corps de la boucle ou non. Dès que deux éléments consécutifs mal ordonnés sont rencontrés, la valeur de cette variable passe à `False` et la boucle se termine. Un soin particulier doit être apporté à la valeur de retour de la fonction.

```
def est_triee_v4_erreur(lst):
    continuer = True
    i = 1
    while i < len(lst) and continuer:
        continuer = lst[i - 1] <= lst[i]
        i += 1
    return i == len(lst) # échec sur [] et [3, 9, 6]
```

La version ci-dessus échoue quand la première inversion se situe à la toute fin de la liste, car on a bien `i == len(lst)` alors que la liste n'est pas triée. D'autre part, quand la liste est vide, le résultat est incorrect. On peut remédier à ce problème en remplaçant `return i == len(lst)` par `return continuer` par exemple.

On peut aussi omettre entièrement la variable `continuer`, comme dans la version suivante. Pour éviter l'erreur de la fonction précédente quand la liste est vide, il est nécessaire d'ajouter un premier test permettant de traiter ce cas particulier.

```
def est_triee_v5(lst):
    if len(lst) == 0:
        return True
    # dorénavant, on sait que len(lst) >= 1
    i = 1
    while i < len(lst) and lst[i - 1] <= lst[i]:
        i += 1
    # ici, on a i >= len(lst) ou lst[i - 1] > lst[i]
    return i == len(lst)
```

Dans l'exemple précédant on utilise aussi l'aspect séquentiel de `and` dans la condition de boucle (voir le paragraphe Évaluation séquentielle des opérateurs). Si `i < len(lst)` vaut `False`, l'expression `lst[i - 1] <= lst[i]` ne sera pas évaluée et il n'y aura pas de tentative d'accès à `lst[i]` pour `i` supérieur ou égal à la longueur de la liste, ce qui ferait planter le programme.

5. Conclusions et perspectives

Nous avons donc pu voir, les différents statuts et usages des booléens en programmation :

[12] Dans le cas de `est_triee_v2()`, même si l'ensemble des indices est parcouru entièrement, toutes les valeurs ne sont pas forcément examinées, du fait du caractère paresseux de l'opérateur `and` expliqué plus loin.

- Le statut de condition
  - dans un branchement conditionnel ;
  - dans une boucle conditionnelle.
- Le statut de type de donnée
  - variable booléenne, expression booléenne ;
  - variable booléenne comme accumulateur ;
  - variable booléenne comme drapeau à sens unique ;
  - variable booléenne comme drapeau de continuation.

Différents opérateurs (`and`, `or`, `not`, `in`, `is`) ainsi que des fonctions prédéfinies (`isdigit()`, etc.) permettent de construire des expressions booléennes et d'autres fonctions booléennes. L'aspect séquentiel de certains opérateurs en Python (et en Scratch) n'est pas anodin et pourra être travaillé à partir du lycée.

Il nous paraît également nécessaire, dès le début du collège, de travailler l'usage des opérateurs `and`, `or` et `not` non seulement dans l'enseignement d'informatique mais également dans le cours de mathématiques. Par exemple lors du cours sur la médiatrice d'un segment. Celle-ci est définie comme la « droite qui passe par le milieu d'un segment *et* qui lui est perpendiculaire ». Vérifier qu'une droite est médiatrice nécessite de vérifier les deux propriétés de la conjonction. Pour affirmer qu'une droite n'est pas médiatrice d'un segment, il suffit de vérifier qu'une des deux propriétés est fausse : une droite qui « ne passe pas par le milieu d'un segment *ou* qui ne lui est pas perpendiculaire » n'est pas sa médiatrice. Nous utilisons donc une des lois de De Morgan. Les exemples ne manquent pas dans le cours de mathématiques et si à chaque occasion (définition, propriétés ou exercice), nous prenons la peine d'en expliciter la structure logique et d'en exprimer la négation, nous faciliterons l'apprentissage des booléens par les élèves.

En ce qui concerne la forme que l'on peut donner aux structures conditionnelles, certaines relèvent du style de programmation. Cependant, si on veut travailler la preuve de programme avec des élèves, il est déconseillé d'utiliser des `return` dans les boucles.

L'apparition d'écritures redondantes dans les programmes des élèves, même s'il n'est pas souhaitable de le sanctionner, fait ressortir le manque de compréhension qu'ils ont du statut des booléens qu'ils manipulent. Au début de leur apprentissage en programmation, les élèves rencontrent les booléens comme des conditions dans des structures conditionnelles et ces conditions ont le plus souvent la forme d'expression avec opérateur de comparaison. Il leur est alors difficile de reconnaître les booléens comme type de données, les expressions conditionnelles comme expression à valeur booléenne et n'importe quelle expression à valeur booléenne comme éventuelle condition. Ceci est d'autant plus vrai qu'en Scratch, les variables de type booléen n'existent pas. Il sera donc nécessaire dans nos classes de faire attention à la terminologie utilisée. Dans une structure conditionnelle, ce que nous appelons habituellement *condition* est en fait une *expression booléenne* ayant le statut de *condition*, mais cette même expression, dans le corps du bloc soumis à cette condition, peut retrouver un statut de *donnée* dont la valeur peut être à renvoyer dans une fonction par exemple.

Des compléments sur ces notions seront proposés à la suite de cet article. Ceux-ci seront accessibles depuis la page de la C3I, sur le site de l'ADIREM. Notamment, des contenus et des réflexions pour la classe seront abordés :

- des exercices (pour pratiquer … pour remédier)
- À quelle occasion introduire les booléens ? Doit-on avoir un chapitre dédié aux booléens ?

Références

- Beffara, E., More, M., & Prouteau, C. (2017). *Algorithmique et programmation au cycle 4 : Commentaires et recommandations du groupe Informatique de la CII Lycée*. Institut de recherche pour l'enseignement des mathématiques. https://www.univ-irem.fr/IMG/pdf/algoetprogaucycle4ciilycee-2.pdf#page=81
- Lagrange, J.-B., & Rogalski, J. (2017). Savoirs, concepts et situations dans les premiers apprentissages en programmation et en algorithmique. *Annales de Didactiques et de Sciences Cognitives*, 41.
- Sajaniemi, J. (2002). An empirical analysis of roles of variables in novice-level procedural programs. *Proceedings IEEE 2002 Symposia on Human Centric Computing Languages and Environments*, 37-39. https://doi.org/10.1109/HCC.2002.1046340
- Samurçay, R. (1985). Signification et fonctionnement du concept de variable informatique chez des élèves débutants. *Educational Studies in Mathematics*, *16*(2), 143-161. https://doi.org/10.1007/PL00020737
- Sorva, J., Karavirta, V., & Korhonen, A. (2007). Roles of Variables in Teaching. *Journal of Information Technology Education: Research*, *6*, 407-423. https://doi.org/10.28945/224
- Notes de cours de l'université de Lille:
  - Tester avec doctest. https://www.fil.univ-lille1.fr/~L1S2API/CoursTP/tp_doctest.html

  - Expressions booléennes et doctests. https://www.fil.univ-lille1.fr/~L1S1Info/last/cours/033-bool.html
  - Test de fonction. https://www.fil.univ-lille1.fr/~L1S1Info/last/cours/024-fonction.html
- Sur Python:
  - Documentation du langage. https://docs.python.org/fr/3/
  - PEP 8 – Style Guide for Python Code. https://peps.python.org/pep-0008/
- Le glossaire https://framagit.org/c3i/glossaire/-/wikis/home

Annexes

Les booléens dans différents langages

Langage	Remarque	Interprété comme *vrai*	Interprété comme *faux*
Scratch	Possède des valeurs booléennes mais pas de blocs pour exprimer directement les valeurs *vrai* ou *faux*	les expressions booléennes vraies	les expressions booléennes fausses
Python	Possède un type booléen mais d'autres valeurs peuvent être interprétées comme des booléens	`True`, nombre non nul, liste ou chaîne non vide, etc.	`False`, 0, 0.0, liste ou chaîne vide, None, etc.
C (avant C99)	ne possède pas de type booléen et utilise des valeurs entières.	1 ou toute valeur non nulle	0, ou toute valeur nulle
C (à partir de C99)[13]	Un type `bool` explicite existe depuis la version C99 du langage C	comme en C, ou `true`	comme en C, ou `false`
Ocaml	Possède un type `bool` primitif	`true`	`false`
Haskell	Possède un type `Bool`	`True`	`False`
Bash	Interprète le code de retour des commandes comme des valeurs de vérité	code de terminaison 0 (renvoyé par la commande `true`)	code de terminaison non nul (la commande `false` renvoie 1)
Java	Possède un type primitif `boolean`[14]	`true`	`false`
Javascript	Possède des valeurs booléennes primitives[15] et pratique des conversions implicites	`true`	`false`

Les conversions de type en Python

Le tableau ci-dessous synthétise une partie des conversions de type impliquant les booléens, soit comme type de départ soit comme type final.

Conversions depuis le type bool

valeur de départ	commande	type d'arrivée	valeur d'arrivée
`True`	`str(True)`	`str`	`'True'`
`False`	`str(False)`	`str`	`'False'`
`True`	`int(True)`	`int`	`1`
`False`	`int(False)`	`int`	`0`
`True`	`float(True)`	`float`	`1.0`
`False`	`float(False)`	`float`	`0.0`

Conversions vers le type bool

[13] Grâce à l'en-tête `stdbool.h`, voir par exemple ce paragraphe sur Wikipedia : fr.wikipedia.org/wiki/Booléen#C,_C++,_Objective-C,_AWK.

[14] Voir la documentation. Il existe également un type objet correspondant Boolean, comme pour la plupart des autres types primitifs.

[15] Le standard ECMAScript stipule l'existence d'un type primitif Boolean, voir aussi sur MDN. Comme en Java, il existe aussi un type *wrapper* (objet) Boolean.

type de départ	valeur de départ	commande	valeur d'arrivée
`str`	la chaine vide `''`	`bool('')`	`False`
`str`	toute chaine non vide	`bool('courgette')`	`True`
`int`	tout entier non nul	`bool(-7)`	`True`
`int`	`0`	`bool(0)`	`False`
`float`	tout flottant non nul	`bool(0.5)`	`True`
`float`	`0.0`	`bool(0.0)`	`False`
`List`	liste non vide	`bool([0,1,3])`	`True`
`List`	liste vide `[]`	`bool([])`	`False`

Dans un exercice on pourrait demander aux élèves les valeurs de `bool('True')` et de `bool('False')` .

Implémentation de structures `case`

Dans certains langages classiques, comme le C ou le Pascal, il existe une structure de contrôle qui permet d'exécuter un bloc d'instructions différent en fonction de la valeur d'une certaine expression. Ces constructions assurent que l'ensemble des cas sont disjoints et que tous les cas sont traitées (dans une alternative `default` finale par exemple).

En C :

```c
switch (reponse) {
    case 401:
        printf("Utilisateur non authentifié.\n");
        break;
    case 403:
        printf("Accès refusé.\n");
        break;
    case 404:
        printf("Page non trouvée.\n");
        break;
    case 500:
    case 503:
        printf("Erreur serveur.\n");
        break;
    default:
        printf("Tout va bien !\n");
}
```

En Pascal :

```pascal
case reponse of
    401: writeln("Utilisateur non authentifié.");
    403: writeln("Accès refusé.");
    404: writeln("Page non trouvée.");
    500, 503: writeln("Erreur serveur.");
    else writeln("Tout va bien !");
end;
```

Dans ces structures de contrôle, chaque alternative correspond à une constante (ou *valeur littérale*), et ces valeurs doivent être toutes différentes. On peut spécifier un comportement par défaut si aucune des valeurs envisagées n'est rencontrée, introduite par les mots-clés `default` en C et `else` en Pascal.

Cette structure n'existe pas en tant que telle en Python, mais peut être implémentée via une traduction systématique utilisant une structure à base de `elif` :

```
if reponse == 401:
    print("Utilisateur non authentifié.")
elif reponse == 403:
    print("Accès refusé.")
elif reponse == 404:
    print("Page non trouvée !")
elif reponse == 500 or reponse == 503:
    print("Erreur serveur.")
else:
    print("Tout va bien !")
```

Contrairement aux structures `switch` en C et `case` en Pascal, la syntaxe de Python permet d'utiliser des conditions booléennes quelconques et non des seules valeurs littérales toutes distinctes. On peut y voir une plus grande souplesse, mais aussi dans une certaine mesure un plus grand risque d'erreur. Les conditions ne sont plus nécessairement *mutuellement exclusives*, dans ce cas le bloc correspondant à la première condition évaluée à `True` est exécuté :

```
if x < 1:
   print('x est plus petit que 1')
elif x <= 2:
    print('x est compris entre 1 et 2')
else:
    print('x est plus grand que 2')
```

Attention, une erreur fréquente consiste à utiliser le mot-clé `if` au lieu de `elif`, ce qui peut changer le comportement du programme. Par exemple, si `x` vaut 0, le programme suivant affiche à la fois `x est plus petit que 1` et `x est compris entre 1 et 2`.

```
if x < 1:
   print('x est plus petit que 1')
if x <= 2:
    print('x est compris entre 1 et 2')
else:
    print('x est plus grand que 2')
```

Dans certains cas, il est bien sûr nécessaire d'utiliser une succession de `if`. Dans l'exemple suivant, étant données trois variables heures, minutes, secondes ayant une valeur entière, on souhaite afficher la durée totale dans un format concis :

```
heures, minutes, secondes = ...
if heures != 0:
    print(heures, end='h ')
if minutes != 0:
    print(minutes, end='min ')
if secondes != 0:
    print(secondes, end='s')
print()
```

On peut remarquer que la prochaine version de Python3 (3.10) propose une nouvelle construction qui permet d'utiliser la structure `case`, appelée *filtrage par motifs*.[16]

En voici un exemple :

```
match reponse:
    case 401:
        print("Utilisateur non authentifié.")
    case 403:
        print("Accès refusé.")
    case 404:
        print("Page non trouvée !")
    case 500 | 503:
        print("Erreur serveur.")
    case _:
        print("Tout va bien !")
```

[16] https://docs.python.org/3/tutorial/controlflow.html#match-statements

Cette fonctionnalité offre en fait des possibilités plus larges, que nous n'explorerons pas en détail dans ce document.

Exemple d'utilisation de fonctions booléennes prédéfinies sur les chaînes de caractères

L'utilisation de fonctions booléennes sur les chaînes de caractère permet par exemple de contrôler la saisie au clavier. La fonction isdigit() appliquée à une chaine de caractère renvoie `True` si celle-ci n'est constituée que de chiffres et `False` sinon. L'exemple fourni ici assure que la chaîne saisie correspond à un entier avant de tenter une conversion qui pourrait planter le programme :

```
def saisie_entier(n):
    echec = True
    while echec:
        reponse = input("Entrez un entier : ")
        if not reponse.isdigit():
            print("Ce n'est pas un entier !")
        else:
            n = int(reponse)
            echec = False
    return n
```

Une amélioration progressive d'une première version sans contrôle de la saisie peut être la suivante :

```
def saisie_entier_v1(n):
    reponse = input("Entrez un entier : ")
    return int(reponse)

def saisie_entier_v2(n):
    reponse = input("Entrez un entier : ")
    if not reponse.isdigit():
        print("Ce n'est pas un entier !")
        ## ... ???
    return int(reponse)

def saisie_entier_v3(n):
    reponse = input("Entrez un entier : ")
    if not reponse.isdigit():
        print("Ce n'est pas un entier !")
        reponse = input("Entrez un entier : ")
    return int(reponse)

def saisie_entier_v4(n):
    reponse = input("Entrez un entier : ")
    while not reponse.isdigit():
        print("Ce n'est pas un entier !")
        reponse = input("Entrez un entier : ")
    return int(reponse)
```

Calculs de point fixe

Voici un exemple d'utilisation de drapeaux à sens unique de continuation. Il s'agit de calculer le point fixe d'une fonction `transformer` agissant sur un objet mutable pris dans un domaine donné. On suppose que `transformer` renvoie un booléen `changement` indiquant si `transformer` a modifié l'objet. On écrit une fonction `point_fixe(objet, transformer)` qui utilise une variable booléenne `changement` dont la valeur est mise à jour à chaque exécution de la fonction `transformer`. L'intérêt d'utiliser un drapeau booléen est que le nombre d'itérations n'est pas connu à l'avance.

```
def point_fixe(objet, transformer):
    changement = True
    while changement:
        changement = transformer(objet)
```

Voici un exemple d'utilisation de la fonction `point_fixe` pour implémenter un algorithme simple de type « tri à bulle ».

```
def bulle(lst):
    echanges = False
    for i in range(len(lst)-1):
        if lst[i] > lst[i+1]:
            lst[i], lst[i+1] = lst[i+1], lst[i]
            echanges = True
    return echanges

def tri_a_bulle(lst):
    point_fixe(lst, bulle)
```

Un algorithme du calcul du PGCD, plus grand commun diviseur peut également s'écrire à l'aide de `point_fixe()` :

```
def etape_pgcd(lst):
    if lst[1] == 0:
        return False
    else:
        lst[0], lst[1] = lst[1], lst[0] % lst[1]
        return True

def pgcd(a, b):
    lst = [a, b]
    point_fixe(lst, etape_pgcd)
    return lst[0]
```

On peut citer d'autres exemples de calcul de point fixe susceptibles de se prêter à une implémentation du même type : heuristique de construction d'un stable de graphe, heuristique de croisement pour le voyageur de commerce, construction des paires dans un graphe d'amis, automate cellulaire avec convergence, *bin packing*.

Correction de l'exercice de bac NSI

```
def recherche(gene, seq_adn):
    n = len(seq_adn)
    g = len(gene)
    i = 0
    trouve = False
    while i < n - g + 1 and trouve == False :
       j = 0
       while j < g and gene[j] == seq_adn[i+j]:
          j = j + 1
       if j == g:
          trouve = True
       i = i + 1
    return trouve
```

```
def recherche(gene, seq_adn):
    n = len(seq_adn)
    g = len(gene)
    for i in range(n-g+1):
        for j in range(g):
            if gene[j] != seq_adn[i+j]:
                break
        else:
            return True
    return False
```
