# Contrôle du flot d'exécution d'un programme

## L'alternative

## Chap. 4

---

## Table des matières



---

# 1 Booléens et expressions booléennes

**Définition 1.** *Une **grandeur booléenne** est une grandeur qui ne peut prendre que deux valeurs : Vrai ou Faux, ou 0 ou 1, ou ...).*

- Dans le langage Python, les valeurs booléennes s'écrivent `True` et `False`.
- La fonction `bool` transforme n'importe quel argument en valeur booléenne — tout argument à valeur nulle (*entier* `0`, *flottant* `0.0`, *chaîne de caractères* `""`, *liste* `list()`, ...) est converti en valeur `False`, toute autre valeur pour l'argument est devient la valeur `True`.

**Définition 2.** *Une **expression booléenne** est une expression dont la **valeur** est une grandeur booléenne.*

*Une expression booléenne comporte soit un **opérateur de comparaison**, soit une **fonction booléenne**.*

| Opérateur | Expression booléenne | Description |
|---|---|---|
| `<` | `expr1 < expr2` | retourne `True` si `expr1` est strictement inférieure à `expr2` |
| `>` | `expr1 > expr2` | retourne `True` si `expr1` est strictement supérieure à `expr2` |
| `<=` | `expr1 <= expr2` | retourne `True` si `expr1` est inférieure ou égale à `expr2` |
| `>=` | `expr1 >= expr2` | retourne `True` si `expr1` est supérieure ou égale à `expr2` |
| `==` | `expr1 == expr2` | retourne `True` si `expr1` est égale à `expr2` |
| `!=` | `expr1 != expr2` | retourne `True` si `expr1` est différente de `expr2` |
| `not` | `not exprBool` | retourne le complément logique de `exprBool` |
| `and` | `expr1 and expr2` | retourne le résultat d'un `ET` logique |
| `or` | `expr1 or expr2` | retourne le résultat d'un `OU` logique |

**Tableau 1.** Opérateurs de comparaison et fonctions booléennes dans le langage Python

**Définition 3.** *Un **opérateur** est une fonction spéciale dont l'identificateur s'écrit généralement avec des caractères non autorisés pour le nom des fonctions ordinaires (**symboles** ou **ponctuations**). Il s'agit souvent des équivalents aux opérateurs mathématiques pour le langage de programmation.*

*Les opérateurs de comparaison **retournent un booléen** à partir de nombres (ou de chaînes de caractères).*

**Définition 4.** *Une **fonction booléenne** est une fonction qui fait correspondre à un ou plusieurs booléens (selon son **arité**) un booléen.*

*Le comportement d'une fonction booléenne ne peut pas être décrit par une courbe*. Cependant, l'ensemble des booléens n'étant constitué que de deux éléments, on peut établir la **table de vérité** d'une telle fonction, c'est à dire *l'ensemble des valeurs qu'elle retourne en fonction des arguments qui lui sont passés*.

| $A$ | $B$ | Et$(A, B)$ |
|---|---|---|
| Vrai | Vrai | Vrai |
| Vrai | Faux | Faux |
| Faux | Vrai | Faux |
| Faux | Faux | Faux |

| $A$ | $B$ | Ou$(A, B)$ |
|---|---|---|
| Vrai | Vrai | Vrai |
| Vrai | Faux | Vrai |
| Faux | Vrai | Vrai |
| Faux | Faux | Faux |

| $A$ | Non$(A)$ |
|---|---|
| Vrai | Faux |
| Faux | Vrai |

**Tableau 2.** Tables de vérité des opérateurs `Et` (`and`), `Ou` (`or`) et `Non` (`not`). $A$ et $B$ sont deux grandeurs booléennes ou deux expressions booléennes.

**Définition 5.** *On appelle **prédicat** une fonction qui retourne un booléen.*

**Remarque.** En pratique, le nom des prédicats devrait toujours commencer par : `est_...`

**Exemple.**

```
def est_negatif(x: float) -> bool:
    """
    Retourne True si x < 0, False sinon.
    """
    return x < 0
```

La fonction précédente respecte la spécification et réalise le traitement attendu. On pourrait cependant regretter qu'elle ne traite pas le cas où l'utilisateur lui transmet un argument qui n'est pas un nombre. Il faut alors modifier la spécification et ajouter une instruction au corps de la fonction[1] :

```
def est_negatif(x: float) -> bool:
    """
    Retourne True si x est négatif, False sinon.
    ERREUR si x n'est pas un nombre.
    """
    if not (isinstance(x, int) or isinstance(x, float)):
        raise TypeError("Impossible, x n'est pas un nombre !")

    return x < 0
```

## 2 Flot d'exécution d'un programme

L'exécution des programmes est toujours effectuée de manière **séquentielle** : *les instructions sont analysées et exécutées les unes à la suite des autres, dans l'ordre choisi par le programmeur*.

La ***séquence*** est la *structure de contrôle implicite*. Elle détermine dans quel ordre sont exécutées les instructions. Le programme s'arrête après l'exécution de la dernière instruction.

On appelle l'enchaînement des instructions « *le flot d'exécution* » du programme.

Le flot d'exécution implicite ne permet cependant d'implémenter qu'un nombre très restreint d'algorithmes ; la plupart d'entre eux nécessitent que *certaines instructions ne soient exécutées que dans des conditions particulières*, ou que *certaines instructions soient répétées*[2] un nombre de fois inconnu du programmeur.

1. Cet exemple ne devra pas forcément être généralisé. Il est souvent préférable d'écrire le code le plus simplement possible et d'être clair au niveau de la spécification.

2. Cette notion sera abordée dans un prochain chapitre.

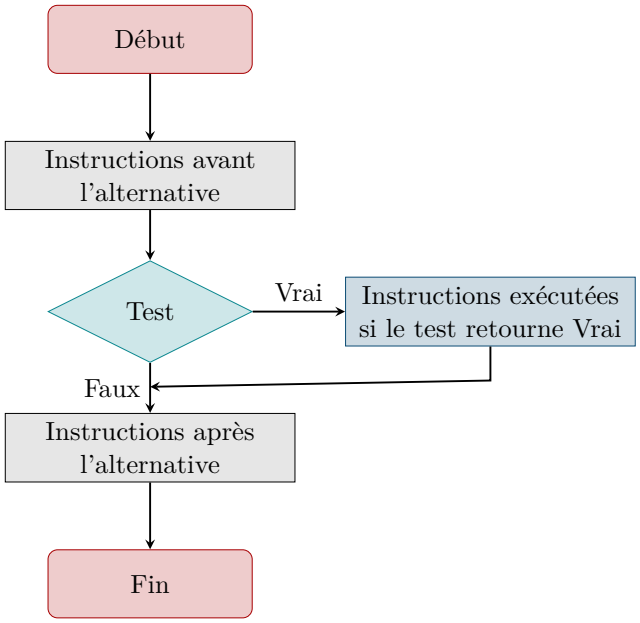
**Définition.** *On appelle* ***structures de contrôle*** *les constructions (fonctions, alternatives, boucles, etc.) qui* ***modifient le flot d'exécution implicite*** *d'un programme.*

# 3 Alternatives

## 3.1 L'alternative simple SI – ALORS

### 3.1.1 Définition

**Définition.** *L'alternative simple est une structure de contrôle qui permet de ne faire exécuter certaines instructions que si* ***une expression booléenne a pour valeur « Vrai »****. Si l'expression booléenne a pour valeur « Faux », le flot d'exécution n'est pas modifié.*




Cette structure s'écrit en pseudo-code :

```
SI <booléen ou expression booléenne>
    ALORS (conséquent)
FIN SI
```

- *(conséquent)* est constitué de l'ensemble des instructions qui sont exécutées si <booléen> est Vrai.

### 3.1.2 Syntaxe de l'alternative simple en Python

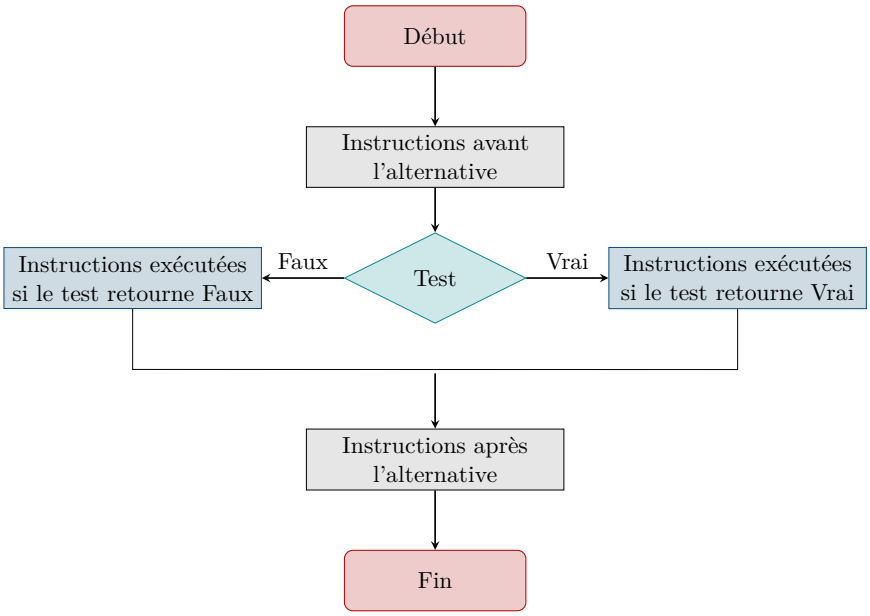
```
if <expression booléenne>:
    instructions si True
```

**Remarque.** C'est cette alternative qui est utilisée dans le code présenté à la section précédente.

## 3.2 Alternative SI – ALORS – SINON

### 3.2.1 Définition

**Définition.** *L'alternative SI – ALORS – SINON est une structure de contrôle qui permet de faire effectuer par le programme* ***deux jeux d'instructions différents*** *selon qu'une expression booléenne a pour valeur « Vrai » ou « Faux ».*




Cette structure s'écrit en pseudo-code :

```
SI <booléen ou expression booléenne>
ALORS (conséquent)
SINON (alternant)
FIN SI
```

- (conséquent) est constitué de l'ensemble des instructions qui sont exécutées si <booléen> est Vrai.

- (alternant) est constitué de l'ensemble des instructions qui sont exécutées si <booléen> est Faux.

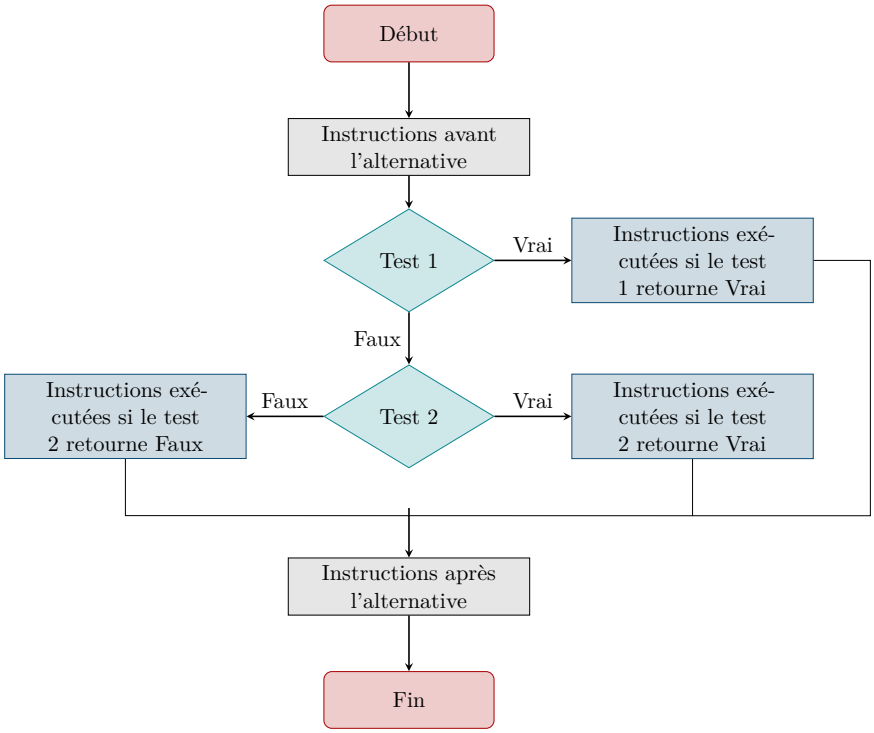
### 3.2.2 Syntaxe de l'alternative SI – ALORS – SINON en Python

```
if <exprBool>:
    instructions du consequent
else:
    instructions de l'alternant
```

## 3.3 Alternative SI – SINON SI – ... (– SINON)

### 3.3.1 Définition

**Définition.** *L'alternative SI – SINON SI – ... (– SINON) est une structure de contrôle qui permet de faire exécuter des jeux d'instructions à un programme selon les valeurs de plusieurs expressions booléennes (examinés les unes à la suite des autres).*




Cette structure s'écrit en pseudo-code :

```
SI <booléen 1> ALORS (conséquent 1)
SINON SI <booléen 2> ALORS (conséquent 2)
...
SINON (alternant)
FIN SI
```

Les différentes clauses agissent comme des court-circuits, *leur évaluation cesse dès l'instant où l'interpréteur trouve le* ***premier booléen avec la valeur Vrai***.

**Remarque.** La présence de la branche SINON n'est pas obligatoire.

### 3.3.2 Syntaxe de l'alternative SI – SINON SI – ... (– SINON) en Python

```
if <exprBool 1>:
    instructions conséquent 1
elif <exprBool 2>:
    instructions conséquent 2
else:
    instructions de l'alternant
```

### 3.3.3 À noter

La structure SI – SINON SI – ... (– SINON) *n'est pas aussi fondamentale que l'alternative* SI – SINON. On peut la remplacer par des alternatives imbriquées, comme le montre l'exemple qui suit. *Son utilisation est cependant conseillée car sa syntaxe est beaucoup plus claire* que celle de plusieurs alternatives imbriquées.

**Exemple.** Écriture en pseudo-code d'une fonction qui compare deux nombres.

```
ENTRÉE : i, j : Nombres
SORTIE : message : Chaîne de caractères

SI i<j ALORS
    message ← "j"
SINON SI i>j ALORS
    message ← "i"
SINON
    message ← "égaux"
FIN SI

RETOURNE message
```

```
ENTRÉE : i, j : Nombres
SORTIE : message : Chaîne de caractères
SI i<j ALORS
    message ← "j"
SINON
    SI i>j ALORS
        message ← "i"
    SINON
        message ← "égaux"
    FIN SI
FIN SI
RETOURNE message
```

# 4 Exercices

### Exercice 1. Découverte des fonctions booléennes

*Afin de répondre aux questions, effectuer tous les tests qui suivent dans la console (shell) du logiciel Thonny.*

1. Quel est le résultat de l'application de l'opérateur logique `not` à une variable booléenne ?

2. Vérifier que les applications des opérateurs logiques `and` et `or` donnent bien les tables de vérités annoncées dans le chapitre.

3. Que peut-on conclure après examen des résultats des expressions suivantes ?

```
(1/0) and True
True and (1/0)
True or (1/0)
```

```
(1/0) or True
True or (1/0)
False and (1/0)
```

4. Supposons que `a` vaille `True` et que `e` et `f` vaillent `False`. On ne connaît pas les valeurs de `b`, `d` et `g`. Que valent les expressions ci-après ? *Déterminer sur le papier avant de lancer les vérification à l'ordinateur.*

```
e or ((not e) and f and d)
(a or g) and (not (b or e))
```

5. On pose que deux expressions sont équivalentes si elles font les mêmes calculs. En utilisant l'alternative, mais pas la fonction booléenne OU, proposer une expression équivalente à : `(not p) and q`.

6. Comment peut-on simplifier les deux expressions suivantes :

```
if p:
    True
else:
    False
```

```
if p:
    False
else:
    True
```

### Exercice 2. Découverte des expressions booléennes

Dans la **console**, écrire et exécuter les expressions booléennes suivantes, *après avoir prédit chacune des valeurs retournées* :

```
a, b = 3, 5
lettre, car = 'i', 'j';
a != b
a + 2 == b
a + 8 < 2 * b
lettre <= car
lettre == 'w'
```

### Exercice 3. Vérifier qu'un nombre est positif

Écrire un prédicat nommé `est_positif` dont la spécification est :

```
def est_positif(x: float) -> bool:
    """ Retourne True si x est positif ou nul, False sinon. """
```

### Exercice 4. Déterminer si deux nombres sont égaux

En Python, on peut tester l'égalité de deux nombres **entiers** (`int`) à l'aide de l'opérateur `==` :

```
>>> 1 == 1
True
```

cependant, cet opérateur est inutilisable avec les nombres à **virgule flottante** (`float`) :

```
>>> 1 / 49 * 49 == 1
False
```

La méthode, avec de tels nombres, consiste à *vérifier que la différence entre ces deux nombres (en valeur absolue) est contenue dans l'intervalle* $[0, \varepsilon]$ *où* $\varepsilon$ *est une constante choisie arbitrairement.*

Écrire un prédicat nommé `est_egal` dont la spécification est :

```
def est_egal(x: float, y: float) -> bool:
    """ Retourne True si abs(x - y) < epsilon où epsilon = 1e-5.

    >>> est_egal(1 / 49 * 49, 1)
    True
    """
```

### Exercice 5. Vérifier qu'un nombre entier est un nombre pair

Écrire un prédicat nommé `est_pair` dont la spécification est :

```
def est_pair(n: int) -> bool:
    """ Retourne True si n est un nombre pair, False sinon. """
```

### Exercice 6. Vérifier qu'un nombre est un nombre entier

Écrire un prédicat nommé `est_entier` dont la spécification est :

```
def est_entier(n: int) -> bool:
    """ Détermine si n est un nombre entier. """
```

**Remarque.** Il existe plusieurs façons de résoudre ce problème : utiliser la fonction intégrée `int` ou la fonction du module `math` nommée `floor` (rechercher son action à l'aide de la fonction `help`).

Il est aussi possible d'utiliser la fonction `isinstance` et le type intégré `int`.

### Exercice 7. Vérifier qu'un nombre est un nombre entier naturel

Écrire un prédicat nommé `est_naturel` dont la spécification est :

```
def est_naturel(n: int) -> bool:
    """ Détermine si un nombre entier est un entier naturel.

    >>> est_naturel(5)
    True
    >>> est_naturel(-5)
    False
    >>> est_naturel_2(2.3)
```

```
    False
    """
```

La fonction `est_naturel` doit réaliser son traitement en utilisant les fonctions `est_positif` et `est_entier` des exercices précédents.

### Exercice 8. Vérifier qu'un nombre est un nombre entier naturel (v2)

Écrire un prédicat nommé `est_naturel_2` dont la spécification est :

```
def est_naturel_2(n: int) -> bool:
    """ Détermine si un nombre entier est un entier naturel.
    ERREUR si n n'est pas un nombre.

    >>> est_naturel(5)
    True
    >>> est_naturel(-5)
    False
    >>> est_naturel_2(2.3)
    False
    >>> est_naturel_2("coucou")
    TypeError: n n'est pas un nombre !!! """
```

Cette fonction doit utiliser la fonction `est_naturel`, la fonction intégrée `isinstance`, les types intégrés `int` et `float` et l'instruction `assert`.

### Exercice 9. Déterminer qu'un nombre est plus grand qu'un autre

Écrire un prédicat nommé `est_plus_grand` dont la spécification est :

```
def est_plus_grand(x: float, y: float) -> bool:
    """ Retourne True si x > y, False sinon. """
```

### Exercice 10. Déterminer le plus grand entre deux nombres

Écrire une fonction nommée `le_plus_grand` dont la spécification est :

```
def le_plus_grand(x: float, y: float) -> float:
    """ Retourne x si x > y, y sinon. """
```

**Remarque.** Cette fonction doit utiliser la fonction `est_plus_grand` de l'exercice précédent.

### Exercice 11. Détermination de la valeur absolue d'un nombre

Écrire une fonction nommée `valeur_absolue` dont la spécification est :

```
def valeur_absolue(x: float) -> float:
    """ Retourne la valeur absolue de x. """
```

Cette fonction devra utiliser le prédicat `est_positif` de l'exercice 3.

**Exercice 12. Structures imbriquées**

Écrire deux fonctions qui implémentent les deux algorithmes donnés à la section 3.3.3.

Les spécifications de ces fonctions ne devront pas être oubliées.

**Exercice 13. Message en fonction d'une note**

Écrire une fonction (*et sa spécification !*) nommée `synthese` qui attend une moyenne générale au BAC et qui retourne :

- « Recalé » si elle est inférieure à 8 ;
- « Second tour » si elle est comprise entre 8 (inclus) et 10 (exclus) ;
- « Admis » si elle est comprise entre 10 (inclus) et 12 (exclus) ;
- « Admis, Mention Assez Bien » si elle est comprise entre 12 (inclus) et 14 (exclus) ;
- « Admis, Mention Bien » si elle est comprise entre 14 (inclus) et 16 (exclus) ;
- « Admis, Mention Très Bien » si elle est supérieure à 16 (inclus).

**Exercice 14. Détermination des racines d'un polynôme du second degré dans $\mathbb{R}$**

Écrire une fonction (*et sa spécification !*) nommée `racines` qui résout les équations du second degré *dans l'ensemble des réels*, à l'aide d'instructions if-else imbriquées.

Pour l'équation $a\,x^2 + b\,x + c = 0$, où $a$, $b$ et $c$ sont des nombres réels, l'arbre des choix associés aux solutions réelles peut s'écrire :

1. $a = 0$
   a. $b = 0$
      i. $c = 0 \Longrightarrow$ tout réel est solution
      ii. $c \neq 0 \Longrightarrow$ pas de solution
   b. $b \neq 0 \Longrightarrow$ une seule solution $x = -c/b$
2. $a \neq 0$
   a. $b^2 - 4\,a\,c > 0 \Longrightarrow$ deux solutions : $\begin{cases} x_1 = \left(-b + \sqrt{b^2 - 4\,a\,c}\right)/2\,a \\ x_2 = \left(-b - \sqrt{b^2 - 4\,a\,c}\right)/2\,a \end{cases}$

b. $b^2 - 4\,a\,c = 0 \Longrightarrow$ une solution : $x = -b/2\,a$

c. $b^2 - 4\,a\,c < 0 \Longrightarrow$ pas de solution dans l'ensemble des réels

**Exercice 15. Détermination d'un prix TTC avec remise**

Écrire une fonction (*et sa spécification !*) nommée `calcul_prix` qui attend un prix hors taxe et qui calcule le prix TTC correspondant (avec un taux de TVA de 19,6 %), auquel doit être appliquée une remise qui dépend de sa valeur :

- 0 % pour un montant TTC inférieur à 1000 euros ;
- 1 % pour un montant TTC supérieur ou égal à 1000 euros et inférieur à 2000 euros.
- 2 % pour un montant TTC supérieur ou égal à 2000 euros et inférieur à 5000 euros.
- 5 % pour un montant TTC supérieur ou égal à 5000 euros.

La remise doit être calculée par l'appel d'une fonction nommée `calcul_remise` (penser à écrire sa spécification).

**Exercice 16. Année bissextile**

Écrire un prédicat nommé `est_bissextile` dont la spécification est :

```
def est_bissextile(n: int) -> bool:
    """ Retourne True si l'année n est bissextile, False sinon. """
```

**Remarque.** Une année est bissextile si son millésime est multiple de 4. Cependant, les années dont le millésime est multiple de 100 ne sont bissextiles que si c'est aussi un multiple de 400 (1900 n'était pas bissextile, 2000 l'a été).