Programmation 2

Slides modified from **https://github.com/UGE-IGM/courspython**

Chapitre 2 : Conditionnelles et expressions booléennes

Dans ce chapitre vous allez apprendre à :

1. effectuer des traitements différents selon si une condition est réalisée ou non ;
2. manipuler les expressions logiques ;
3. écrire correctement en Python une structure conditionnelle, aussi complexe soit-elle ;
4. expliquer l'ordre dans lequel les lignes d'un programme contenant des blocs conditionnels seront exécutées en fonction des données fournies au programme ;
5. simplifier, quand c'est possible, une imbrication de blocs conditionnels en utilisant des opérateurs booléens ;
6. éliminer les opérateurs booléens d'une condition en introduisant de nouveaux blocs conditionnels.

I Introduction

Au chapitre précédent, nous avons appris à écrire un programme court réalisant des entrées/sorties avec l'utilisateur, pouvant affecter des valeurs à des variables et effectuant des calculs arithmétiques sur ces variables.
Les **instructions** de nos programmes sont exécutées dans l'ordre, de manière linéaire les unes après les autres. On va maintenant ajouter la possibilité de faire des **choix** dans le programme, pour exécuter des instructions différentes selon les cas.
Mais avant tout, commençons par un peu de révisions !

Exercice 1 : Révisions

Que réalise le programme suivant ?

```
note_cc1 = float(input('Note du premier contrôle : '))
note_cc2 = float(input('Note du second contrôle : '))
note_exam = float(input("Note de l'examen : "))

moyenne_cc = (note_cc1 + note_cc2) / 2
note_finale = (moyenne_cc + note_exam) / 2

print('Note finale :', note_finale)
```

In [ ]:

```
note_cc1 = float(input('Note du premier contrôle : '))
note_cc2 = float(input('Note du second contrôle : '))
note_exam = float(input("Note de l'examen : "))

moyenne_cc = (note_cc1 + note_cc2) / 2
note_finale = (moyenne_cc + note_exam) / 2

print('Note finale :', note_finale)
```

Nous allons maintenant voir comment on pourrait :

- ne prendre en compte que la `note_exam` si celle-ci est supérieure à la``note_finale`;
- féliciter l'étudiant qui a une `note_finale` supérieure à `10` et encourager celui qui en a une entre `8` et `10`.

Tout au long du cours, nous allons faire évoluer ce programme.

II Expressions booléennes

Les instructions conditionnelles (vues ci-dessous) sont écrites à l'aide d'expressions booléennes, c'est à dire d'expressions qui s'évaluent en un valeur de type `bool` (`True` ou `False`). Elles peuvent contenir des opérateurs de comparaison, des opérateurs logiques, etc.

1) Opérateurs de comparaison

Plusieurs opérateurs ont des résultats booléens

- Comparaisons : `a < b` `a <= b` `a >= b` `a > b`
- Égalité ou inégalité : `a == b` `a != b`

Ces opérateurs fonctionnent sur de nombreux types de valeurs

- Sur les `int` et `float` : ordre naturel
- Sur les `str` : ordre lexicographique (dictionnaire)
- Sur d'autres types qu'on verra plus tard

In [ ]:

```
1 == 3 - 2
```

In [ ]:

```
1 < 3
```

In [ ]:

```
3 <= 3
```

In [ ]:

```
'aboyer' < 'abime'
```

In [ ]:

```
1 < 4.0
```

Attention : on ne peut pas ordonner des valeurs de types différents (sauf des nombres) !

In [ ]:

```
'1' < 2
```

In [ ]:

```
3.0 < 2 + 2
```

Par contre, les opérateurs `==` et `!=` acceptent des opérandes de types différents

In [3]:

```
2 == '2'
```

Out[3]:

```
False
```

In [ ]:

```
'bonjour' != None
```

In [ ]:

```
2 == 2.0  # Cas particulier : vrai car float(2) == 2.0
```

Ne pas confondre l'opérateur d'égalité (==) avec l'opérateur d'affectation (=) !

Exercice 2 : Opérateurs de comparaison

Que valent les expressions suivantes ?

In [ ]:

```
1. < 2
```

In [ ]:

```
"""1 >= 0
0 <= 1
"""
```

In [ ]:

```
None != 0
```

In [ ]:

```
"to" * 2 == "toto"
```

2) Opérateurs logiques

On peut combiner plusieurs expressions booléennes à l'aide d'opérateurs logiques :

- `a and b` vaut `False` dès que l'une des variables vaut `False`, et vaut `True` si les deux variables valent `True`.
- `a or b` vaut `True` si l'une des deux variables vaut `True`, et `False` sinon.
- `not a` vaut `True` si `a` vaut `False`, et vaut `False` si `a` vaut `True`.

In [ ]:

```
True and False
```

In [ ]:

```
not (3 + 4 != 7)
```

In [ ]:

```
4 < 1 or 'Bonjour' >= 'Au revoir'
```

On peut résumer le comportement de ces opérateurs à l'aide de tableaux, appelés **tables de vérité** :

a	not a
True	False
False	True

a	b	a and b
True	True	True
True	False	False
False	True	False
False	False	False

a	b	a or b
True	True	True
True	False	True
False	True	True
False	False	False

In [ ]:

```
x = 1  # essayer plusieurs valeurs de x
print(x >= 0 and x <= 10)  # x appartient à l'intervalle fermé [0, 10]
print(x < 0 or x > 10)  # x n'appartient pas à l'intervalle fermé [0, 10]
print(not (x < 0 or x > 10))  # comment interpréter cette ligne ??
```

Exercice 3 :

Pour quelles valeurs des variables `n`, `somme` et `seuil` l'expression suivante vaut-elle `True` ?

```
(n <= 3 and somme + n > seuil) or n > 3
```

In [ ]:

```
n = 2.1
somme = 0
seuil = 2
b = (n <= 3 and somme + n > seuil) or n > 3
print(b)
```

3) Analogies avec les opérations arithmétiques usuelles

Un `True` peut être vu comme l'entier 1. De même, `False` s'interprête comme l'entier 0.
Remarque 1 : Pour garder le fait que `not True == False`, en réalité dans de nombreux langages de programmation, `True` est un entier non nul
Remarque 2 : `True or True = True`, d'où avec cette analogies : $1 + 1 = 1$!!

In [ ]:

```
bool(10)
```

In [ ]:

```
bool(0)
```

In [ ]:

```
print(int(True))
print(int(False))
```

Soit `P` et `Q` deux propositions logiques (`True` ou `False`) s'interprêtant respectivement comme l'entier `p` ou `q`.
Alors : l'opérateur `and` peut être interprété comme une **multiplication**. En effet, `P and Q` s'interprête comme l'entier $p \cdot q = pq$
Alors : l'opérateur `or` peut être interprété comme une **addition**. En effet, `P and Q` s'interprête comme l'entier $p \cdot q = pq$
L'opérateur `not` change l'entier `p` en `1 - p`.

4) Exemples simples de tautologies

Une *tautologies* est une expression logique qui s'évalue toujours à `True`.

In [ ]:

```
P = False
P and False
```

Quelque soit la proposition logique `P`, `P and False == False`.

In [ ]:

```
P = False
P or True
```

Quelque soit la proposition logique `P`, `P or True == True`.

In [ ]:

```
P = False
not not P
```

Quelque soit la proposition logique `P`, `not not P == P`.

In [ ]:

```
P = False
P and (not P)
```

Quelque soit la proposition logique `P`, `P and (not P) = False`
En effet, si `p = 0 ou 1` est l'entier interprétant la proposition `P`, alors `P and (not P)` s'interprête comme : $p(1-p) = 0$.

In [ ]:

```
P = True
P or (not P)
```

Quelque soit la proposition logique `P`, `P or (not P) = True`
En effet, si `p = 0 ou 1` est l'entier interprétant la proposition `P`, alors `P or (not P)` s'interprête comme : $p + (1 - p) = 1$.

5) Exemples plus avancé de tautologies

De manière générale, on peut donc déduire des tautologies à partir des relations vérifier par l'adition et la multiplication des entiers.

A. `P AND (Q OR R) = ???`

On va utiliser l'interprétation : `P AND (Q or R) --> p (q + r) = pq + pr --> (P AND Q) OR (P AND R)`

In [ ]:

```
P = True
Q = True
R = False
```

In [ ]:

```
P and (Q or R)
```

In [ ]:

```
(P and Q) or (P and R)
```

Une fois une conjecture posée avec l'analogie, on la démontre avec une table de vérité !

P	Q	R	Q or R	P and (Q or R)	P and Q	P and R	(P and Q) or (P and R)
True	True	True	True	True	True	True	True
True	True	False	True	True	True	False	True
True	False	True	True	True	False	True	True
True	False	False	False	False	False	False	False
False	True	True	True	False	False	False	False
False	True	False	True	False	False	False	False
False	False	True	True	False	False	False	False
False	False	False	False	False	False	False	False

Donc : `P AND (Q or R) == (P AND Q) OR (P AND R)`

B. `P AND (Q AND R) == (P AND Q) AND R`

Interprétation :

`P and (Q and R)` --> p(qr) = pqr

`(P and Q) and R` --> (pq)r = pqr

Vérification informatique :

In [ ]:

```
P = True
Q = True
R = True
```

In [ ]:

```
P and (Q and R)
```

In [ ]:

```
(P and Q) and R
```

Donc, `P and (Q and R) == (P and Q) and R`. On ne met donc pas de parenthèse, puisqu'il n'y a aucune ambiguité d'évaluation des opérateurs `and`.

C. `NOT (P AND Q) = ?`

P	Q	P and Q	not (P and Q)	not P	not Q	(not P) or (not Q)
True	True	True	False	False	False	False
True	False	False	True	False	True	True
False	True	False	True	True	False	True
False	False	False	True	True	True	True

Donc : `not(P and Q) == (not P) or (not Q)`.

6) Vérifier vos connaissances

A ce stade, vous devez être capable de :

- citer les six opérateurs de comparaisons en Python ;
- savoir évaluer le résultat d'une expression logique simple ;
- comprendre les mécanisme de conversions implicites lors de l'évaluation d'une expression logique ;
- citer les trois opérateurs logique que nous avons vus ;
- donner la table de vérité des trois opérateurs logique que nous avons vus ;
- savoir simplifier une expression logique simple.

Exercice 4 :

Donner le résultat des expressions logiques suivantes :

In [ ]:

```
a = 10
b = 2
c = 6
```

In [ ]:

```
a < b or a > c
```

In [ ]:

```
a + b < 2 * c
```

In [ ]:

```
a - b == b + c
```

In [ ]:

```
(a > b and a > c) or (b > a and b > c)
```

In [ ]:

```
a < b < c
```

In [ ]:

```
a == b == c
```

In [ ]:

```
(a <= b and a <= c) or not (b < a)
```

In [ ]:

```
not (a > b and a > c) or (b > a and b > c)
```

Exercice 5 :

variables ne servent à rien ou etre simplifier
Les expressions logiques suivantes ne dépendent pas des variables `x`, `y`, `z`.
peuvent être simplifiées. Cela peut avoir une grande importance pour la lisibilité d'un programme.
Simplifiez les autant que possible.

In [1]:

```
x = False
y = False
z = False
```

In [2]:

```
(x + y > 3 * z) or True
```

Out[2]:

```
True
```

In [3]:

```
n = input("abc")
```

```
abc34
```

In [5]:

```
(x == y and y == z) or (x == y and y == z)
```

Out[5]:

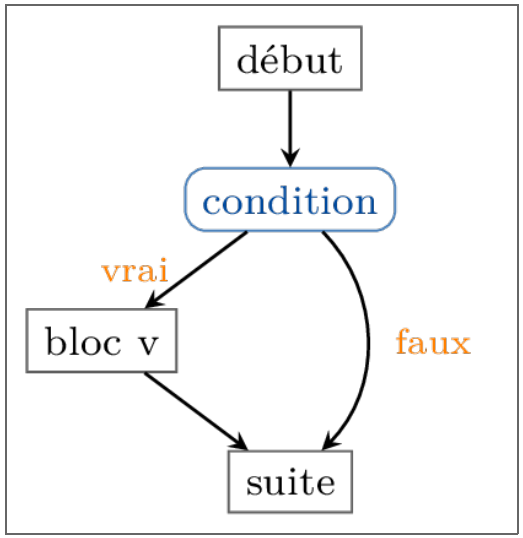
```
True
```

III Instructions conditionnelles

1) Cas simple : Conditionnelle `Si`

On peut maintenant modifier le flot d'instructions selon la valeur d'expressions booléennes, ou conditions :

- **Si** une certaine condition est vraie, exécuter un certain groupe (ou bloc) d'instructions
- **Sinon**, passer directement à la suite du programme



La syntaxe d'une instruction conditionnelle est :

```
# début
if condition:
    # bloc d'instructions V
# suite
```

- Les instructions du bloc **v** sont exécutées uniquement si `condition` est évaluée à `True`
- Dans tous les cas, l'exécution reprend à l'instruction suivant le bloc **v**

Exemple 1 :

In [ ]:

```
prenom = input("Quel est votre prénom ? ")
if prenom == 'IRONMAN':
    print('Très joli prénom !')
print('Bonjour', prenom, '!')
```

Exemple 2 :

Reprenons maintenant notre exemple introductif :

In [ ]:

```
# La note finale est le maximum de la note d'examen
# et de la moyenne entre examen et moyenne de contrôle
# continu
note_cc1 = float(input('Note du premier contrôle : '))
note_cc2 = float(input('Note du second contrôle : '))
note_exam = float(input("Note de l'examen : "))

moyenne_cc = (note_cc1 + note_cc2) / 2
note_finale = (moyenne_cc + note_exam) / 2

if note_finale < note_exam:
    # Les deux instructions suivantes ne s'exécutent que si
    # la condition est vraie (remarquer le décalage des lignes)
    note_finale = note_exam
    print('Notes de contrôle non prise en compte.')

print('Note finale :', note_finale)
```

La notion de bloc

Sur cet exemple, on a vu un groupe de lignes commençant par des espaces, appelé **bloc**. Un bloc est utilisé pour regrouper plusieurs instructions dépendant de la même condition.

- Un tel groupe d'instructions est appelé un **bloc**
- Le décalage du début de ligne est appelé **indentation**
- Commencer une ligne avec une indentation supérieure à la précédente commence un nouveau bloc (sur le second exemple : `note_finale = note_exam`)
- Un bloc se termine quand une ligne **moins indentée** apparaît (sur l'exemple : `print('Note finale :', note_finale)`)
- Pour indenter la ligne courante : touche "tabulation" (⇥)
- Pour désindenter une ligne : "Shift + tabulation" (⇧ + ⇥)
- Changer l'indentation change le sens du programme (essayer !)

Erreurs fréquentes liées à l'indentation :

In [ ]:

```
if note_finale < note_exam  # oubli des deux points (:)
    note_finale = note_exam
    print('Note de contrôle non prise en compte.')
```

In [ ]:

```
if note_finale < note_exam:
    note_finale = note_exam
  print('Note de contrôle non prise en compte.')  # ligne pas assez indentée
```

In [ ]:

```
if note_finale < note_exam:
    note_finale = note_exam
        print('Note de contrôle non prise en compte.')  # ligne trop indentée
```

In [ ]:

```
if note_finale < note_exam:
note_finale = note_exam  # oubli d'indentation
```

Vérifier vos connaissances

A ce stade, vous devez être capable :

- d'écrire correctement en Python une structure conditionnelle simple (utilisation des :, indentation, ...) ;
- d'évaluer le résultat d'une structure conditionnelle simple en Python ;
- d'identifier les différents blocs dans un programme

Exercice 6 :

Écrire un programme qui demande un entier saisi au clavier par l'utilisateur et affiche `strictement positif` s'il est strictement positif.

In [ ]:

```
n = int(input("Entrez un nombre : "))
if n > 0:
    print("strictement positif")
```

EXERCICE 7 :

Écrire un programme qui demande à l'utilisateur le poids de son bagage en kilos. Si le bagage pèse plus de 20 kilos, le programme affichera le message :

```
Vous devez payer un supplément.
```

Exercice 8 :

Qu'affiche le programme suivant :

```
a = 10
if a > 1000:
    print("grand")
print("nombre")
```

2) Conditionnelles `Si ... Sinon ...`

On peut ajouter un second bloc d'instructions

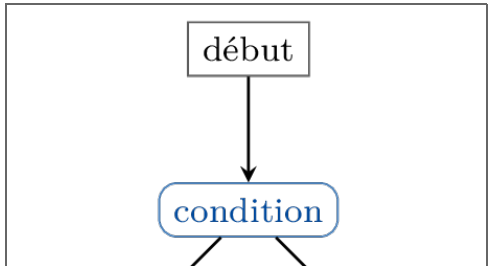
- **Si** une certaine condition est vraie, exécuter le premier bloc
- **Sinon**, exécuter le second
- Enfin, continuer l'exécution normale du programme

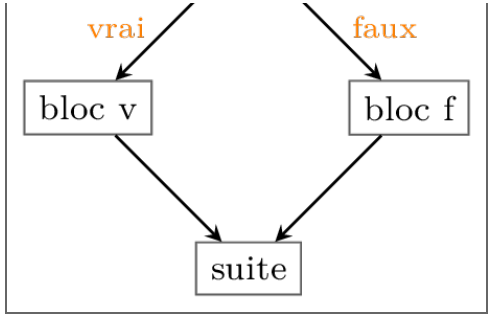
Syntaxe :

```
# début
if <condition>:
    # bloc v
else:
    # bloc f
# suite
```

Seul *l'un des deux* blocs d'instructions, **v** *ou bien* **f**, est exécuté

- Le bloc **v** uniquement si `condition` est évaluée à `True`
- Le bloc **f** uniquement si `condition` est évaluée à `False`
- Dans tous les cas, reprise à l'instruction suivant le bloc **f**




vrai
faux
bloc v
bloc f
suite

Notre exemple favori

En plus de la dernière version, on cherche à savoir si le contrôle continu a été pris en compte.

In [ ]:

```python
# La note finale est le maximum de la note d'examen
# et de la moyenne entre examen et moyenne de contrôle
# continu

note_cc1 = float(input('Note du premier contrôle : '))
note_cc2 = float(input('Note du second contrôle : '))
note_exam = float(input("Note de l'examen : "))

moyenne_cc = (note_cc1 + note_cc2) / 2

if moyenne_cc < note_exam:
    # Bloc à exécuter si la condition est vraie
    note_finale = note_exam
    print('Notes de contrôle non prise en compte.')
else:  # moyenne_cc >= note_exam
    # Bloc à exécuter si la condition est fausse
    note_finale = (moyenne_cc + note_exam) / 2
    print('Notes de contrôle prise en compte.')

# Instruction exécutée dans tous les cas
print('Note finale :', note_finale)
```

Un autre exemple : la division euclidienne

In [ ]:

```
dividende = int(input("Donner moi un dividende : "))
diviseur = int(input("Donner moi un diviseur : "))
if diviseur != 0:
    print(dividende, '=', dividende // diviseur, '*', diviseur, '+', dividende % diviseur )
else:
    print('Erreur de saisie ? Division par zéro...')
```

Vérifier vos connaissances

A ce stade, vous devriez savoir :

- écrire une structure conditionnelle `Si ... Sinon ....`

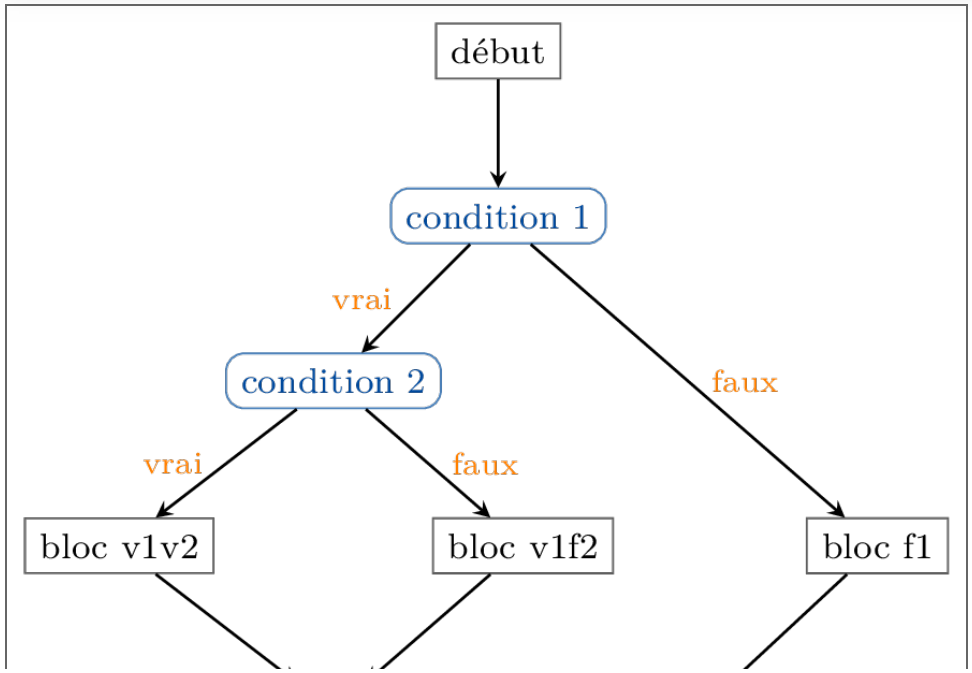
EXERCICE 9 :

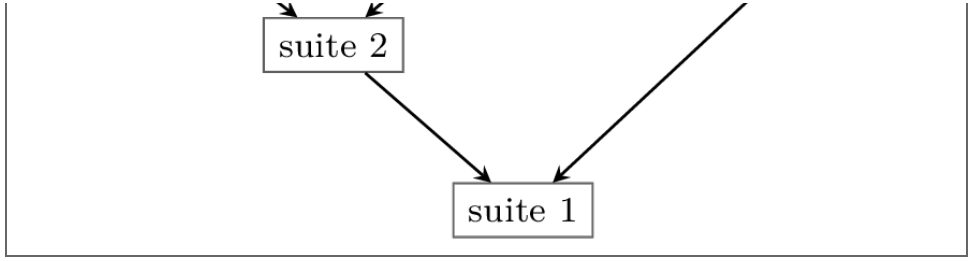
Écrire un programme qui demande un entier saisi au clavier par l'utilisateur et affiche `positif` si l'entier est positif ou nul et `negatif` sinon.

3) Conditionnelles composées

Cette construction peut être imbriquée :

```
# début
if <condition 1>:
    if <condition 2>:
        # bloc v1v2
    else:
        # bloc v1f2
    # suite 2
else:
    # bloc f1
# suite 1
```





Toutes les variantes sont possibles — si chaque `else` correspond à un `if` de même indentation !

Notre exemple favori

On veut désormais ajouter la possibilité de tester si les notes données sont des notes positives.

In [ ]:

```
# La note finale est le maximum de la note d'examen
# et de la moyenne entre examen et moyenne de contrôle
# continu

note_cc1 = float(input('Note du premier contrôle : '))
note_cc2 = float(input('Note du second contrôle : '))
note_exam = float(input("Note de l'examen : "))

# On vérifie qu'aucune note n'est négative
if note_cc1 >= 0 and note_cc2 >= 0 and note_exam >= 0:
    moyenne_cc = (note_cc1 + note_cc2) / 2
    # Condition dans la condition
    if moyenne_cc < note_exam:
        # Indentation supplémentaire
        note_finale = note_exam
        print('Note de contrôle non prise en compte.')
    else:
        note_finale = (moyenne_cc + note_exam) / 2
        print('Note de contrôle prise en compte.')
    print('Note finale :', note_finale)
else:
    print('Erreur de saisie (note négative).')
```

Vérifier vos connaissances

A ce stade, vous devriez être capable :

- d'expliquer quelles lignes seront exécutées dans un programme contenant des blocs conditionnels en fonction des données fournies au programme ;
- simplifier, quand c'est possible, une imbrication de blocs conditionnels en utilisant des opérateurs booléens ;
- éliminer les opérateurs booléens d'une condition en introduisant de nouveaux blocs conditionnels.

Exercice 10 : Evaluations d'un programme

Qu'affiche le programme ci-dessus lorsque :

- le premier nombre donné est 1 et le second est 0.
- le premier nombre donné est 2 et le second est 2.
- le premier nombre donné est 3 et le second est 0.
- le premier nombre donné est 4 et le second est 4.

```
a = int(input('Donnez un nombre : '))
b = int(input('Donnez un nombre : '))
print('toto')
if a > 2:
    print('tata')
    if b >= a:
        print('truc')
    else:
        print('bla')
    print('poire')
print('42')
```

Exercice 11 : Transformation de conditions imbriquées

Modifier le programme suivant pour qu'il ne contienne qu'une seule condition.

```
a = int(input("Donnez moi un entier : "))
b = int(input("Donnez moi un entier : "))
if a > 0:
    if b == 2 * a
        print("Le nombre positif b est le double de a")
    else:
        print("a est négatif ou b n'est pas le double de a")
else:
    print("a est négatif ou b n'est pas le double de a")
```

4) Conditionnelles enchaînées :

Cas particulier où le bloc `else` contient seulement un autre `if` : le mot-clé `elif`

Le code...

```
# début
if <condition 1>:
    # bloc **v1**
else:
    if <condition 2>:
        # bloc **f1v2**
    else:
        # bloc **f1f2**
# suite
```

... s'écrit aussi :

```
# début
if <condition 1>:
    # bloc **v1**
elif <condition 2>:
    # bloc **f1v2**
else:
    # bloc **f1f2**
# suite
```

On peut ainsi enchaîner autant de conditions qu'on le souhaite, lorsque les cas ne se recouvrent pas :

```
# début
if <condition 1>:
```

```
    # bloc **v1**
elif <condition 2>:
    # bloc **f1v2**
elif <condition 3>:
    # bloc **f1f2v3**
else:
    # bloc **f1f2f3**
# suite
```

Exemple final :

In [ ]:

```python
# La note finale est le maximum de la note d'examen
# et de la moyenne entre examen et moyenne de contrôle
# continu

note_cc1 = float(input('Note du premier contrôle : '))
note_cc2 = float(input('Note du second contrôle : '))
note_exam = float(input("Note de l'examen : "))

if note_cc1 < 0 or note_cc2 < 0 or note_exam < 0:
    # Négation de note_cc1 >= 0 and note_cc2 >= 0 and note_exam >= 0
    print('Erreur de saisie (note négative).')
else:
    moyenne_cc = (note_cc1 + note_cc2) / 2

    if moyenne_cc < note_exam:
        note_finale = note_exam
        print('Note de contrôle non prise en compte.')
    else:
        note_finale = (moyenne_cc + note_exam) / 2
        print('Note de contrôle prise en compte.')
    print('Note finale :', note_finale)

    if note_finale < 8:
        print("T'as pas assez bossé... Tant pis !")
    elif note_finale < 10:
        print('Encore un petit effort !')
    elif note_finale < 12:
        print('Ça passe !')
    elif note_finale < 14:
        print('Pas mal !')
    else:  # Quelles valeurs possibles ?
        print('Bravo !')
```

Vérifier vos connaissances

A ce stade, vous devriez être capable :

- d'écrire correctement en Python une structure conditionnelle simple ou complexe ;

Exercice 12 : Où est cette valeur ?

Pour quelles plages de valeur de la variable `a` le programme suivant affiche-t-il `X` ?

Que se passe-t-il si `a` n'est pas de type `int` ?

```
if a > 10:
    print('a grand')
elif a < 5:
    print('a petit')
else:
    print('X')
```

In [1]:

```
a = "10.3"
if a > 10:
    print('a grand')
elif a < 5:
    print('a petit')
else:
    print('X')
```

```
---------------------------------------------------------------------------
-
TypeError                                 Traceback (most recent call las
t)
Input In [1], in <cell line: 2>()
      1 a = "10.3"
----> 2 if a > 10:
      3     print('a grand')
      4 elif a < 5:

TypeError: '>' not supported between instances of 'str' and 'int'
```

In [9]:

```
reload_ext nbtutor
```

In [10]:

```
%%nbtutor -r -f
empty = []
foo = 10
pass
```

In [11]:

```
print("hello")
```

```
hello
```

In [ ]:

```

```

In [ ]:

```

```