# Instructions conditionnelles et expressions booléennes

## Objectifs

### Objectifs

Dans cette séquence, vous allez découvrir

- le type booléen,
- les opérateurs de comparaison,
- les opérateurs logiques,
- l'instruction `assert`,
- l'instruction conditionnelle.

## Les valeurs booléennes

Les valeurs de vérité vrai et faux sont représentées respectivement en Python comme en anglais par `True` et `False` .

Par exemple, voici une expression booléenne qui est vraie:

```
>>> True
True
```

On dit que `True` est une valeur litérale.

**Avertissement:** La majuscule est indispensable. En Python, comme beaucoup d'autres langages de programmation, il faut accorder de l'importance à la casse

```
>>> true
Traceback (most recent call last):
  File "<pyshell#15>", line 1, in <module>
    true
NameError: name 'true' is not defined
```

Voici une autre expression booléenne qui est vraie aussi ::

```
>>> 1 < 3
True
```

L'expression précédente utilise l'opérateur de comparaison strictement inférieur à (noté <).

Bien sûr la valeur faux se note `False` la plus simple des expressions booléennes fausses s'écrit donc

```
>>> False
False
```

En utilisant l'opérateur de comparaison <

```
>>> 3 < 1
False
```

On a déjà vu la fonction `type` qui permet de connaître le type d'une expression:

```
>>> type(True)
<class 'bool'>
```

## Opérateurs de comparaisons

On a déjà rencontré précédemment l'opérateur de comparaison *strictement inférieur à* . Passons en revue les autres opérateurs de comparaisons:

### table des opérateurs de comparaison

| | | |
|---|---|---|
| strictement inferieur à | se note | < |
| inférieur ou égal à | se note | <= |
| strictement supérieur à | se note | > |
| supérieur ou égal à | se note | >= |
| égal à | se note | == |
| différent de | se note | != |

### Exemples

Voici quelques exemples:

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

**Avertissement:** L'ordre est important ! L'opérateur =< n'est pas correct.

```
>>> 1 =< 3
SyntaxError: invalid syntax
```

```
>>> 1 < 1
False
>>> 3 < 1
False
```

### test d'égalité de valeur

**Avertissement:** L'opérateur qui teste l'égalité n'est pas = .Le = sert pour l'affectation.

```
>>> 3 = 1
SyntaxError: can't assign to literal
```

L'opérateur == permet de tester l'égalité de valeur.

```
>>> 3 == 3
True
>>> 3 == 2 + 1
True
```

En Python, le test d'égalité entre des valeurs de types différents est autorisé. Il donne la valeur `False` dans tous les cas.

```
>>> 3 == "3"
False
```

L'opérateur est différent de se note en Python `!=`. Cet opérateur teste la différence.

```
>>> 4 != 3
True
>>> 4 != 3 +1
False
>>> 3 != "3"
True
```

## Opérateurs booléens

On a appris précédement à fabriquer des expressions booléennes simples. On peut composer de nouvelles expressions booléennes grâce aux opérateurs booléens.

- et
- ou
- non

### la table de vérité du ou.

| or | False | True |
|---|---|---|
| False | False | True |
| True | True | True |

**Note:** On peut obtenir de l'aide (en anglais) sur `or` en tapant `help("or")`

### la table de vérité du et.

| and | False | True |
|---|---|---|
| False | False | False |
| True | False | True |

**Note:** On peut obtenir de l'aide (en anglais) sur `and` en tapant `help("and")`

### le ou exclusif

Aucun opérateur n'est défini pour le *ou exclusif* qui peut être réalisé de différentes manières. Ainsi la commande suivante illustre l'inexistance de `xor`.

```
help( "xor")
```

::

no Python documentation found for 'xor'

### Exercice:

réaliser la fonction `ou_excl` qui possède deux paramètres, dont le résultat est un booléen. La table de vérité est:

| ouexcl | False | True |
|---|---|---|
| False | False | True |
| True | True | False |

### la table de vérité du non.

| x | not(x) |
|---|---|
| False | True |
| True | False |

**Note:** On peut obtenir de l'aide (en anglais) sur `not` en tapant `help("not")`

### Exercice

Donnez une expression python signifiant que `x` est compris entre 0 et 10 au sens large.

Donnez une expression python signifiant que `x` est compris entre -3 et -1 ou bien compris entre 2 et 3.

**Note:** Aussi encore à la différence d'autres langages, les expressions comme `a < b < c` ont une interprétation qui est conforme à celle utilisée en mathématique.

En supposant que `x` est une variable numérique précédemment définie, on peut donc écrire :

```
>>> (-3<x<-1) or (2<x<3)
```

## Séquentialité

**Note:** Les opérateurs `and` et `or` sont séquentiels en python. Cela signifie que l'opérande droite n'est évaluée que si l'opérande gauche ne permet pas de trouver le résultat.

```
a=0
a!=0 and (0<1/a)
```

```
False
```

```
(0<1/a) and (a!=0)
```

::

Traceback (most recent call last): File "<pyshell#4>", line 1, in <module> (0<1/a) and (a!=0) ZeroDivisionError: int division or modulo by zero

## lois de Morgan

**Théorème**

Si `a` et `b` sont deux booléens, alors

- `non (a ou b)` est équivalent à `(non a) et (non b)`
- `non (a et b)` est équivalent à `(non a) ou (non b)`

**Preuve:** on réalise les tables de vérité.

| a | b | non a | non b | (non a) et (non b) | a ou b | non(a ou b) |
|---|---|---|---|---|---|---|
| False | False | True | True | True | False | True |
| False | True | True | False | False | True | False |
| True | False | False | True | False | True | False |
| True | True | False | False | False | True | False |

Dans tous les cas, la colonne correspondant à (non a) et (non b) et celle correspondant à non (a ou b), donnent la même valeur. Ces deux expressions sont donc équivalentes.

## Valeurs de verité généralisées

**Note:** Python est assez permissif. Tout élément peut être utilisé comme une valeur de vérité. Toute valeur est alors considérée comme `True` sauf quelques valeurs particulières :

- `None`

- `False`
- les valeurs nulles des types numériques, par exemple, `0`, `0.0`, `0j`.
- les séquences vides, par exemple, `''`, `()`, `[]`.
- les dictionnaires vides, par exemple, `{}`.
- d'autres encore qui dépassent le cadre de ce cours (voir `help( "TRUTHVALUE" )` )

## Assertion

On a vu que certaines fonctions possèdent des contraintes d'utilisation. La fonction `sqrt` définie dans le module `math` attend par exemple un paramétre positif ou nul

```
>>> from math import sqrt
>>> sqrt(-1.0)
Traceback (most recent call last):
  File "<pyshell#39>", line 1, in <module>
    sqrt(-1.0)
ValueError: math domain error
```

On souhaite obtenir le même comportement pour la fonction de conversion de degré Celsius en degré Fahrenheit. En effet, il n'y aurait aucun sens de convertir une température inférieure à la température correspondant au zéro absolu

```
def en_fahrenheit(c) :
    """
      conversion en Fahrenheit d'une température donnée en Celsius
      C.U. c doit être supérieure au zéro absolu.

    """
    assert c>-273.15
    return 9*c/5+32
```

```
>>> help(en_fahrenheit)
Help on function en_fahrenheit in module __main__:

en_fahrenheit(c)
    conversion en Fahrenheit d'une température donnée en Celsius
    C.U. c doit être supérieure au zéro absolu.

>>> en_fahrenheit(-500)
Traceback (most recent call last):
  File "<pyshell#18>", line 1, in <module>
    en_farhenheit(-500)
  File "/home/wl/ex.py", line 21, in en_farhenheit
    assert c>-273.15
AssertionError
>>>
```

L'instruction **`assert`** peut aussi être accompagnée d'une chaîne de caractères qui sera affiché dans le cas où l'instruction déclenche une exception. Cette possibilité peut être utilisée pour préciser les raisons de cette exception.

```
>>> a = 0
>>> assert a != 0, "a doit être différent de zéro."
```

```
Traceback (most recent call last):
  File "<pyshell#56>", line 1, in <module>
    assert a != 0, "a doit être différent de zéro."
AssertionError: a doit être différent de zéro.
```

# Le si

## syntaxe du si

> l'instruction conditionnelle en python dans sa version la plus générale possède la syntaxe suivante :

```
if <condition1> :
   <instructions_a_executer_si_la_condition1_est_vraie>
elif <condition2>:
   <instructions_à_exécuter_si_la_condition1_est_fausse_et_si_la_conditon2_est_vraie>
elif <condition3>:
   <instructions_à_exécuter_si_les_conditions_1_et_2_sont_fausses_et_si_la_conditon3_est_vra
...
else:
   <instuctions_à_exécuter_si_aucune_des_conditions_précédentes_n_est_vraie>
```

Il peut y avoir autant de `elif` qu'on le souhaite.

On peut omettre la partie `else`. On peut omettre les `elif`.

## exemple

L'indice de masse corporel (imc) est un indicateur statistique utilisé par l'organisation mondiale de la santé. Cet indice se calcule en divisant la masse exprimée en kilogramme par le carré de la taille exprimée en mètre. L'unité de l'imc est donc le $kg/m^2$.

Interprétation de l'IMC

| | |
|---|---|
| moins de 16,5 | dénutrition ou famine |
| 16,5 à 18,5 | maigreur |
| 18,5 à 25 | corpulence normale |
| 25 à 30 | surpoids |
| 30 à 35 | obésité modérée |
| 35 à 40 | obésité sévère |
| plus de 40 | obésité morbide ou massive |

Réalisons une fonction nommée `interpretation_imc` qui calcule prend l'imc en paramètre et renvoie la chaîne contenant son interprétation

```
def interpretation_imc(i):
  """ interpretation imc source wikipedia
  """
  if i<16.5:
      return "dénutrition ou famine"
  elif 16.5<=i<18.5:
      return "maigreur"
  elif 18.5<=i<=25:
```

```
        return "corpulence normale"
    elif 25<i<=30:
        return "surpoids"
    elif 30<i<=35:
        return "obésité modérée"
    elif 35<i<=40:
        return "obésité sévère"
    else:
        return "obésité morbide ou massive"
```

**Note:** des tests inutiles sont réalisés dans le code qui précéde... on aurait pu écrire:

```
def interpretation_imc(i):
    """ interpretation imc source wikipedia
    """
    if i<16.5:
        return "dénutrition ou famine"
    elif i<18.5:
        return "maigreur"
    elif i<=25:
        return "corpulence normale"
    elif i<=30:
        return "surpoids"
    elif i<=35:
        return "obésité modérée"
    elif i<=40:
        return "obésité sévère"
    else:
        return "obésité morbide ou massive"
```

Réalisons une fonction qui calcule le nombre de racines réelles d'un trinôme du second degré a*x**2+b*x+c donné par ses coefficients a,b,c où a est non nul.

```
def nbr_rac_trinome(a,b,c):
    """ Calcule le nombre de racines réelles d'un trinôme du second degré a*x**2+b*x+c donné
    C.U.
    a doit être non nul.
    """
    d=delta(a,b,c)
    if d==0:
        return 1
    elif d<0:
        return 0
    else:
        return 2
```

**Avertissement:** Le soin à apporter à l'indentation est **capital**. Car c'est l'indentation qui permet de délimiter le blocs. Par exemple:

```
if test():
    instr_a()
    instr_b()
instr_c()
```

n'a pas du tout la même signification que

```
if test():
   instr_a()
instr_b()
instr_c()
```

dans le premier cas, `instr_a` et `instr_b` n'est exécutées que si `test()` vaut `True` et `instr_c` sera exécutée ensuite inconditionnellement.

dans le second morceau de code, `instr_a` n'est exécutée que si `test()` vaut `True` et `instr_b` et `instr_c` sont exécutées ensuite inconditionnellement.

### exemples

### remarque

**Avertissement:** le code suivant :

```
if test():
   instr_a()
else:
   instr_b()
```

et le code suivant :

```
if test():
   instr_a()
if not(test()):
   instr_b()
```

Ne sont pas équivalents !

Pour s'en convaincre, il suffit de remarquer que dans le second code la fonction `test` est appelée deux fois. Or il se peut que le premier appel à la fonction `test` ou bien que l'instruction `instr_a` ait un effet sur l'environnement.

Exemple:

```
 def code1(x):
     a=x
     if a!= 0:
         a = a - 1
         print(a)
     else:
         print(a)
     print("fin")


def code2(x):
    a=x
    if a!= 0:
        a = a - 1
        print(a)
```

```
    if a==0:
        print(a)
    print("fin")
```

Calculer `code1(2)`, `code2(2)`, `code1(0)`, `code2(0)`, `code1(1)`, `code2(1)`

Exercice: Prévoir puis tester l'effet du code 1 et l'effet du code 2 si on suppose que `a`, `instr_a`, `instr_b` et `test` sont définis comme suit :

```
a=True

def instr_a ():
   print("A")

def instr_b ():
   print("B")

def test () :
   global a
   a:=not(a)
   return(a)
```

## Conclusion

## Pour aller plus loin