Chapitre 2
Booléens et tests

1 Booléens

Un **booléen**[1], de type `bool`, est une proposition logique pouvant prendre deux valeurs : `True` (vrai) ou `False` (faux).

```
True

type(False)
```

Les connecteurs logiques entre booléens sont les suivants.

Connecteurs logiques	
`and`	Et
`or`	Ou
`not`	Non (négation)

De plus, les opérations suivantes sur les nombres produisent des expressions booléennes.

Opérations dont le résultat est booléen	
==	Égal (test d'égalité)
!=	Différent
<	Inférieur strictement
<=	Inférieur ou égal
>	Supérieur strictement
>=	Supérieur ou égal

1. Le nom vient du logicien anglais George Boole (1815 - 1864).

```
2 + 2 == 4

2 == 3

type(2 * 7 == 13)

condition = (10 <= 5) or (2 > 1)

condition

type(condition)
```

⚠ Il ne faut pas confondre le symbole ==, qui teste l'égalité entre deux expressions, avec le symbole =, qui est l'opérateur d'affectation.

```
x == 2

x = 2

x

x == 2

type(x == 2)

type(x = 2) # L'instruction x = 2 est une affectation.
```

Comme nous l'avons vu au chapitre 1, Python peut faire des arrondis dans le calcul flottant. Il arrive alors que Python déclare vraies ou fausses des propositions qui ne le sont pas.

```
1 == 0.99

1 == 0.99999999999999999

1e-330 == 1e-331

0.1 + 0.2 == 0.3
```

Voici enfin un exemple important de fonction dont la sortie est booléenne : une fonction qui détermine la parité d'un entier.

```
def est_pair(n):
    """
    Entrée : un entier n.
    Sortie : True si n est pair, False sinon.
    """
    return (n % 2 == 0) # Le résultat est un booléen.
```

Testons cette fonction dans l'interprète de commandes.

```
est_pair(4)

est_pair(-1)
```

2 Instructions conditionnelles

2.1 Tests simples : commande `if`

Le test **`if`** (*si* en français...) permet de n'exécuter un bloc d'instructions que dans le cas où une certaine condition est satisfaite. Un test s'écrit de la manière suivante.

```
if condition :
    bloc indenté d'instructions
    à exécuter si, et seulement si,
    la condition condition est vraie
```

Lorsque la condition `condition` est une expression booléenne vraie, Python exécute alors le bloc d'instructions donné.
Dans le cas contraire, Python ne fait rien et passe à la suite du programme !

Voici quelques remarques sur la syntaxe des tests en Python.

- ⋆ Les deux points à la fin de la première ligne sont un symbole discret mais toujours indispensable !
- ⋆ Le bloc d'instructions dépendant du `if` est délimité par l'indentation. Tout ce qui est indenté d'un cran par rapport au `if` est à exécuter dans le cas où la condition est vraie, tout ce qui ne l'est pas est à l'extérieur du test.
- ⋆ La condition booléenne doit être écrite sur une seule ligne (quitte à ce que la ligne soit longue).

Écrivons et testons les fonctions suivantes. Que font-elles ?

```
def mauvaise_fonction(x):
    """ Pourquoi est-elle mauvaise ? """
    if x <= 2:
        y = x + 1
    return y

mauvaise_fonction(-4)

mauvaise_fonction(3)
```

```
def fonction_correcte(x):
    """ Que fait, correctement, cette fonction ? """
    y = x + 3
    if x <= 2:
        y = y - 2
    return y
```

```
fonction_correcte(-4)

fonction_correcte(3)
```

```
def autre_fonction(x):
    if x <= 2:
        return x + 1
        # Au premier return exécuté, la fonction s'arrête,
        # donc si la condition x <= 2 est vérifiée,
        # la fonction se termine.
    return x - 1
    # On passe à la suite du programme si et seulement si
    # la condition x <= 2 n'est pas vérifiée.

autre_fonction(-4)

autre_fonction(3)
```

Une remarque importante d'esthétique/d'efficacité (que le jury du concours Agro-Véto aime à faire dans ses rapports) : si `condition` est un booléen, il est plus élégant d'écrire

```
if condition:
```

que

```
if condition == True:
```

même si ces deux instructions font exactement la même chose, de la même manière que, pour un réel x donné, il est plus naturel de dire «Si x est positif» que «Si la proposition "x est positif" est vraie».
On pourra alors affecter des conditions booléennes à des variables pour plus de lisibilité.

Par exemple, la fonction `autre_fonction` se réécrit également ainsi :

```
def autrement(x):
    condition = (x <= 2)
    if condition:
        return x + 1
    return x - 1
```

Dans les fonctions `autre_fonction` et `autrement`, on observe que l'instruction `return x - 1` n'est exécutée que lorsque la condition `(x <= 2)` n'est pas vérifiée : c'est une alternative au bloc d'instruction du `if`. Une meilleure manière de présenter cette alternative est d'utiliser la commande *else*, détaillée dans la sous-partie suivante.

2.2 Tests avec alternative : commande `else`

La commande **`else`** (*sinon* en français...) permet de réaliser un test plus évolué offrant une alternative, c'est-à-dire d'exécuter un bloc d'instructions si une certaine condition est vraie et un autre bloc d'instructions si la condition est fausse. Cela s'écrit de la manière suivante.

```
if condition :
    premier bloc d'instructions, à exécuter si, et seulement si,
    la condition condition est vraie
else :
    second bloc d'instructions, à exécuter si, et seulement si,
    la condition condition est fausse
```

Si le booléen `condition` est vrai, Python exécute alors le premier bloc d'instructions et pas le second. Sinon, Python exécute le second bloc d'instructions mais pas le premier.

Comme précédemment, on prendra garde

- ⋆ à l'indentation, qui délimite les différents blocs d'instructions ;
- ⋆ aux discrets deux points à la fin de la première ligne ainsi qu'après le `else`.

Voici deux fonctions importantes qui s'écrivent avec une instruction conditionnelle : le maximum et la valeur absolue.

```
def maximum(a, b):
    """
    Arguments : deux nombres a et b, entiers ou flottants.
    Sortie : le plus grand des deux nombres a et b.
    """
    if a > b:
        return a
    else:
        return b
```

```
def valeur_absolue(x):
    """
    Arguments : un nombre x, entier ou flottant.
    Sortie : la valeur absolue de x.
    """
    if x >= 0:
        return x
    else:
        return -x
```

Remarquons que l'on peut aussi écrire la fonction précédente avec un seul `return` au lieu de deux :

```
def valeur_absolue_bis(x):
    if x >= 0:
        resultat = x
    else:
        resultat = -x
    return resultat
```

Moins recommandé : comme les fonctions s'arrêtent au premier `return` exécuté, la fonction précédente peut également se réécrire ainsi :

```
def valeur_absolue_ter(x):
    if x >= 0:
        return x
    return -x # On n'arrive ici que dans le cas où
              # la condition (x > 0) n'est pas vérifiée.
```

Cette dernière écriture est certes plus économe en lignes de code, mais a le défaut de briser la symétrie entre le premier bloc d'instructions et son alternative.

Dans la continuité de la remarque de la section précédente : dans le corps d'une fonction, si `condition` est un booléen local à la fonction, il est plus élégant d'écrire

```
return condition
```

que d'écrire

```
if condition:
    return True
else:
    return False
```

même si ces instructions ont exactement le même effet.

2.3 Tests avec plusieurs alternatives : commande `elif`

Si l'on souhaite choisir entre plus de deux alternatives, on peut utiliser la commande **`elif`** (contraction de *else if*) après un test `if`, qui permet alors d'exécuter un bloc d'instructions dans le cas où la condition du test `if` est fausse mais une deuxième condition est satisfaite. Une telle disjonction de cas s'écrit de la manière suivante.

```
if condition_1 :
    premier bloc d'instructions, à exécuter si, et seulement si,
    la condition condition_1 est vraie
elif condition_2 :
    deuxième bloc d'instructions, à exécuter si, et seulement si,
    la condition_1 est fausse et condition_2 est vraie
else :
    troisième bloc d'instructions, à exécuter si, et seulement si,
    les deux conditions condition_1 et condition_2 sont fausses
```

Si la condition `condition_1` est vraie, alors Python exécute le premier bloc d'instructions mais aucun des deux autres.

Si la condition `condition_1` est fausse et si la condition `condition_2` est vraie, alors Python exécute le deuxième bloc d'instructions (et pas les autres).

Si aucune des deux conditions `condition_1` ou `condition_2` n'est vraie, alors Python exécute le troisième bloc d'instructions.

Voici quelques remarques sur l'écriture des instructions conditionnelles multiples.

- ⋆ Comme précédemment, on prendra garde aux deux points et à l'indentation, qui délimitent les différents blocs d'instructions.
- ⋆ La présence du `else` n'est pas obligatoire : si l'on ne précise pas de dernier bloc d'instructions et si les deux premières conditions sont fausses, alors Python ne fait rien (et passe à la suite du programme) !
- ⋆ Il peut y avoir plusieurs cas intermédiaires introduits par des `elif`.
- ⋆ Idéalement, on s'arrange pour que les différentes conditions soient mutuellement exclusives, c'est-à-dire que les différents cas de la disjonction de cas ne se recoupent pas.

  Toutefois, lorsque ce n'est pas le cas, l'ordre des tests a une grande importance.

  En effet, dans l'exemple précédent, si la condition `condition_1` est vraie, Python exécutera le premier bloc d'instructions et non le deuxième, même si la condition `condition_2` est vraie aussi.

Écrivons et testons la fonction suivante.

```
def alternatives(x):
    if x > 0:
        y = x + 2
    elif x > -2:
        y = x - 5
    elif x >= -10:
        y = 3 * x
    else:
        y = x
    return y

alternatives(4)

alternatives(-1)

alternatives(-3)

alternatives(-14)
```