Numérique et sciences informatiques
Classe de première

A Langages et programmation
2 Tests et structure conditionnelle

I Tests et booléens

I.1 Les tests

a) Opérateurs d'égalité et de différence

• Pour tester si deux variables ou objets ont la même valeur on utilise les opérateurs `==` et `!=`.

* Le test `a == b` signifie « `a` est égal à `b` ». Il renvoie respectivement les objets `True` ou `False` selon que les variables `a` et `b` ont ou non la même valeur.

* Le test `a != b` signifie « `a` est différent de `b` ». Il correspond donc à la négation du test `a == b` : `True` si `a == b` renvoie `False` et `False si a == b` renvoie `True`.

On notera que les objets `True` et `False` s'écrivent avec une majuscule au début.

Exemples :

```
>>> 7*8 == 56
True
>>> 14 != 2*7
False
>>> "bonjour" != "hello"
True
```

Il est essentiel de bien se souvenir que l'opérateur de comparaison qui signifie « égal » s'écrit avec == *et non pas avec un seul signe* =. Nous avons vu qu'en python on utilise un seul signe = pour signifier une instruction d'affectation et non pour effectuer un test.

• Deux nombres peuvent avoir la même valeur même si l'un est de type **`int`** et l'autre de type **`float`**.

Exemples :

```
>>> 5 == 5.0
True
>>> 1200 != 1.2e3
False
```

• En revanche, un objet de type **`int`** ou **`float`** ne peut jamais être égal à un objet de type **`str`**.

Exemple :

```
>>> 789 != "789"
True
```

b) Opérateurs de relation d'ordre

• On peut tester la relation d'ordre entre deux nombres (même s'ils sont de types différents) avec les opérateurs <, >, <= et >=.

• On peut aussi tester une relation d'ordre entre deux objets de type **`str`**. Dans ce cas, c'est l'**ordre lexicographique** qui fournit de critère d'ordre.

Exemples :

```
>>> "Bonjour"<"Hello"
True
>>> "145">"86"
False
```

• Il peut être utile de savoir que, selon cet ordre lexicographique, les chiffres sont placés avant les lettres majuscules qui sont elles-mêmes avant les lettres minuscules.

Exemples :

```
>>> "BONJOUR"<"bonjour"
True
>>> "12"<"DOUZE"
True
```

• La recherche d'une relation d'ordre entre un objet de type **`int`** ou **`float`** avec un objet de type **`str`** génère un message d'erreur « TypeError ».

Exemple :

```
>>> "15"<15
Traceback (most recent call last):
  File "<pyshell#16>", line 1, in <module>
    "15"<15
TypeError: '<' not supported between instances of 'str'
and 'int'
```

I.2 Le type booléen

*a) Les objets de type **`bool`***

• Les objets `True` et `False`, générés par les opérateurs de comparaison, possèdent eux-mêmes un type. On dit que ce sont des **booléens** en hommage au mathématicien et logicien britannique George Boole (1815-1864). En Python ces objets sont dit de type **`bool`**. **True** et **False** sont donc les deux seules valeurs possibles d'un objet de type **`bool`**. On dit qu'un booléen a une **valeur de vérité**.

Exemples :

```
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
>>> type(4<=9)
<class 'bool'>
```

• On peut effectuer des opérations logiques avec des objets de type **`bool`**. Les opérateurs sont **`and`**, **`or`** et **`not`**.

b) Les trois principaux opérateurs logiques en Python

• L'opérateur **`not`** agit sur un unique booléen et il a pour effet d'inverser la valeur de vérité de ce booléen. Il exprime donc la négation en logique .

Exemple :

```
>>> not True
False
>>> not False
True
>>> not 5 > 8
True
```

• L'opérateur **`and`** fait intervenir deux booléens et exprime le « et » de la logique. Rappelons la « table de vérité » du `and` :

A	B	A and B
False	False	False
False	True	False
True	False	False
True	True	True

• L'opérateur **`or`** fait intervenir deux booléens et exprime le « ou » logique. Sa table de vérité est donc la suivante :

A	B	A or B
False	False	False
False	True	True
True	False	True
True	True	True

• Comme les opérateurs arithmétiques, les opérateurs logiques possède des règles de priorité des opérations :

* Les parenthèses indiquent l'ordre des opérations.
* En l'absence de parenthèses, le **`not`** est prioritaire sur le **`or`** et le **`and`**.

Exemple :

```
>>> not True and False
False
>>> not (True and False)
True
```

* En l'absence de parenthèses, le **`and`** est prioritaire sur le **`or`**.

Exemple :

```
>>> False and False or True
True
>>> False and (False or True)
False
```

II La structure conditionnelle

II.1 L'importance de l'instruction conditionnelle

Un algorithme doit adapter les opérations à effectuer aux différentes situations qui peuvent survenir. Ainsi tout langage de programmation doit permettre d'exprimer les instructions qui seront exécutées dans les différents cas envisagés. On doit donc pouvoir traduire dans le langage de programmation la démarche que l'on écrit en « pseudo-code » de la façon suivante :

si *test 1* **alors**
    bloc d'instructions 1
sinon si *test 2* **alors**
    bloc d'instructions 2
...
sinon
    bloc d'instructions n
fin du si

bloc d'instructions n+1

Remarques :

* Le signe « ... » indique qu'il est possible d'avoir un nombre quelconque de structure « **sinon si** ... **alors** ... » entre la fin du *bloc d'instruction 1* et le **sinon**.

* Il est aussi possible d'avoir une structure qui ne comporte aucun « **sinon s**i ... » et/ou pas de « **sinon** ... ».

II.2 La syntaxe et exécution du « `if ...` » en Python

• En python la structure que nous avons vu plus haut sera traduite de la façon suivante :

```
if test1 :
  bloc1
[elif test2  :
  bloc2]
...
[else :
  blocN]
blocP
```

Convention de présentation de la syntaxe : la partie du code 3placée entre crochets n'est pas obligatoire.

Une remarque fondamentale sur les blocs d'instructions en Python

En python, les *blocs d'instructions* sont marqués par l'indentation (4 espaces à gauche) et *aucun autre signe n'est utilisé pour marquer la fin d'un bloc d'instructions*.

Les indentations sont donc obligatoires pour marquer qu'une succession d'instructions forme un bloc et elles sont donc une *partie intégrantes de la syntaxe du langage*.

Ceci est une particularité de Python qui le distingue de la plupart des autres langages de programmation.

Remarques sur la syntaxe de la structure conditionnelle

• les mots **`if`**, **`elif`** et **`else`** sont des mots réservés du langage. Ils ne peuvent donc pas être utilisés comme noms de variable.

• `bloc1`, `bloc2`, ..., `blocN`, `blocP` sont des successions d'instructions appelées « bloc d'instructions ».

• `test1`, `test2`, etc. sont des expressions qui renvoient une valeur booléenne, donc soit **`True`**, soit **`False`**.

• Le signe **`:`** *est impératif* derrière chacun des tests *mais aussi* derrière le mot-clé **`else`**.

• Le mot-clé **`elif`** (propre au langage Python) est une contraction de « else if ». Cette dernière expression n'est pas acceptée en Python.

Exécution d'une structure conditionnelle « `if ...` »

• Dans une structure conditionnelle de la forme **« `if ...` »**, *au plus un seul des blocs d'instructions sera exécuté*. Dès que l'interpréteur évalue un des tests de la structure comme **`True`**, il exécute le bloc d'instructions correspondant puis il sort de la structure conditionnelle *sans évaluer* les autres conditions. Il exécute ensuite les instructions qui se trouvent sous la structure « `if` ... » (c'est-à-dire, avec nos notations, le bloc d'instruction `blocP`).