Récapitulatif sur les conditions dans un `if`

Complément - niveau basique

Dans ce complément nous résumons ce qu'il faut savoir pour écrire une condition dans un `if`.

Expression *vs* instruction

Nous avons déjà introduit la différence entre instruction et expression, lorsque nous avons vu l'expression conditionnelle:

- une expression est un fragment de code qui "retourne quelque chose"
- alors qu'une instruction permet bien souvent de faire une action, mais ne retourne rien

Ainsi parmi les notions que nous avons vues jusqu'ici, nous pouvons citer dans un ordre arbitraire:

Instructions	Expressions
instruction `if`	expression conditionnelle
instruction `for`	compréhensions
affection	appel de fonction
`print`	opérateurs `is` et `==`
`import`	opérateur `in`

Toutes les expressions sont éligibles

Comme condition d'une instruction `if`, on peut mettre n'importe quelle expression.

Comme on l'a déjà signalé, il n'est pas nécessaire que cette expression retourne un booléen:

```
for n in [18, 19]:
    if n % 3:
        print n, "n'est pas divisible par trois"
    else:
        print n, "est divisible par trois"
```

Une valeur est-elle "vraie" ?

Se pose dès lors la question de savoir précisément quelles valeurs sont considérées comme *vraies* par l'instruction `if`.

Parmi les types de base, nous avons déjà eu l'occasion de l'évoquer, les valeurs *fausses* sont typiquement

- 0 pour les valeurs numériques
- les objets vides pour les chaînes, listes, ensembles, dictionnaires, etc.

Pour en avoir le cœur net, pensez à utiliser dans le terminal interactif la fonction `bool`. Comme pour toutes les fonctions qui portent le nom d'un type, la fonction `bool` est un constructeur qui fabrique un objet booléen.

Si vous appelez `bool` sur un objet, la valeur de retour – qui est donc par construction une valeur booléenne – vous indique cette fois sans ambiguïté – comment se comportera `if` avec cette entrée.

```
def show_bool(x):
    print 'condition', x, ' considérée comme', bool(x)

for exp in [ None, "", 'a', [], [1], (), (1, 2), {}, {'a': 1}, set(), {1}]:
    show_bool (exp)
```

Quelques exemples d'expressions

Référence à une variable et dérivés

```
a = range(10)
if a:
    print "a n'est pas vide"
if a[1]:
    print "a[1] n'est pas nul"
```

Appels de fonction ou de méthode

```
chaine = "jean"
if chaine.upper():
    print "la chaine mise en majuscule n'est pas vide"


# on rappelle qu'une fonction qui ne fait pas 'return' retourne None
def procedure(a, b, c):
    "cette fonction ne retourne rien"
    pass

if procedure(1, 2, 3):
    print "ne passe pas ici car procedure retourne None"
else:
    print "attendu"
```

Compréhensions

Il découle de ce qui précède qu'on peut tout à fait mettre une compréhension comme condition, ce qui peut être utile pour savoir si au moins un élément remplit une condition, comme par exemple:

```
inputs = [23, 65, 24]
# y a-t-il dans inputs au moins un nombre dont le carré est 10*n+5
def condition(n):
    return n * n % 10 == 5

if [input for input in inputs if condition(input)]:
    print "au moins une entrée convient"
```

Je vous rappelle que nous aborderons la compréhension en semaine 5. Donc, si l'exemple précédent vous paraît obscur, pas d'inquiétude, vous pourrez revenir dessus après avoir vu la compréhension en semaine 5.

Opérateurs

Nous avons déjà eu l'occasion de rencontrer la plupart des opérateurs de comparaison du langage, dont voici à nouveau les principaux

Famille	Exemples
Égalité	`==`, `!=`, `is`, `is not`
Appartenance	`in`
Comparaison	`<=`, `<`, `>`, `>=`
Logiques	`and`, `or`, `not`

Complément - niveau intermédiaire

Remarques sur les opérateurs

Voici enfin quelques remarques sur ces opérateurs

opérateur d'égalité ==

L'opérateur == ne fonctionne en général (sauf pour les nombres) que sur des objets de même type; c'est-à-dire que notamment un tuple ne sera jamais égal à une liste:

```
[] == ()
```

```
[1, 2] == (1, 2)
```

opérateur logiques

Comme c'est le cas avec par exemple les opérateurs arithmétiques, les opérateurs logiques ont une ***priorité***, qui précise le sens des phrases non parenthésées. C'est-à-dire pour être explcite, que de la même manière que

```
12 + 4 * 8
```

est équivalent à

```
12 + ( 4 * 8 )
```

pour les booléens il existe une règle de ce genre et

```
a and not b or c and d
```

est équivalent à

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

Mais en fait, il est assez facile de s'emmêler dans ces priorités et il est très fortement conseillé de parenthéser.

opérateurs logiques (2)

Les opérateurs logiques peuvent être appliqués à des valeurs qui ne sont pas booléennes:

```
2 and [1, 2]
```

Expression conditionnelle dans une instruction `if`

En toute rigueur on peut aussi mettre un `<> if <> else <>` - donc une expression conditionnelle - comme condition dans une instruction `if`. Nous le signalons pour bien illustrer la logique du langage, mais cette pratique n'est pas du tout conseillée.

```
# cet exemple est volontairement tiré par les cheveux
# pour bien montrer qu'on peut mettre n'importe quelle expression comme condition
a = 1
# ceci est franchement illisible
if 0 if not a else 2:
    print "une construction illisible"
# et encore pire
if 0 if a else 3 if a + 1 else 2:
    print "encore pire"
```

Pour en savoir plus

https://docs.python.org/2/tutorial/datastructures.html#more-on-conditions
(https://docs.python.org/2/tutorial/datastructures.html#more-on-conditions)

Types définis par l'utilisateur

Pour anticiper un tout petit peu, nous verrons que les classes en python vous donnent le moyen de définir vos propres types d'objets. Nous verrons à cette occasion qu'il est possible d'indiquer à python quels sont les objets de type `MaClasse` qui doivent être considérés comme `True` ou comme `False`.

De manière plus générale, tous les traits natifs du langage sont redéfinissables sur les classes. Nous verrons par exemple également comment donner du sens à des phrases comme

```
mon_objet = MaClasse()
for partie in mon_objet:
    <faire quelque chose sur partie>
```

Mais n'anticipons pas trop, rendez-vous en semaine 5.