On découvre

- comment manipuler des booléens, les valeurs de vérité *vrai* et *faux*
- comment comparer des nombres ou des chaînes de caractères
- comment construire des expressions booléennes
- comment définir des prédicats, fonctions renvoyant *vrai* ou *faux*

1 Les booléens

Le type de données booléen a seulement deux valeurs - *vrai* et *faux* -.

Python définit deux littéraux `True` et `False` pour noter ces valeurs :

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

La casse est signifiante : `True` et `False` s'écrivent obligatoirement avec une majuscule en début de mot.

```
>>> true
[...]
NameError: name 'true' is not defined. Did you mean: 'True'?
```

Une expression dont la valeur ne peut être que vrai ou faux est appelée **expression booléenne**.

2 Opérateurs de comparaison

Les opérateurs de comparaison, comme leur nom l'indique, permettent de comparer deux valeurs et de produire une valeur booléenne.

La valeur 12 est inférieure à la valeur 42, vrai ou faux ?

```
>>> 12 < 42
True
```

Python définit les opérateurs suivants :

comparaison	opérateur Python	commentaire
égalité	`==`	à ne pas confondre avec l'affectation `=`
différence	`!=`	l'ordre des caractères est important
inférieur	`<`	
inférieur ou égal	`<=`	l'ordre des caractères est important
supérieur	`>`	
supérieur ou égal	`>=`	l'ordre des caractères est important

```
>>> 12 > 42
False
```

Une erreur courante pour les débutants est de penser que l'expression précédente déclenche une erreur car "ça ne se peut pas". Écrire `12 > 42` n'affirme pas que `12` est plus grand que `42`. On demande : "`12` est strictement plus grand que `42`, vrai ou faux ?". C'est totalement différent.

2.1 Comparer des nombres

Supposons que l'état de la mémoire soit le suivant :

a	6
b	4
c	2

Nous pouvons comparer des nombres :

```
>>> a < b
False
>>> a > c
True
>>> a < b+c
False
>>> a == b+c
True
>>> c != a-b
False
```

Nous pouvons comparer des nombres de différents types :

```
>>> (1 + 1) == 2.0
True
```

Un entier et sa "version flottante" sont des valeurs égales.

```
>>> "top" == 4
False
```

On peut se demander quel sens a cette dernière comparaison, car on a souvent appris à l'école à ne pas mélanger les torchons et les serviettes. Pourtant, en programmation, il est tout à fait légitime de se demander si 2 valeurs, quel que soit leur type, sont égales.

Par contre la relation d'ordre n'est pas toujours définie :

```
>>> "top" > 4
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '>' not supported between instances of 'str' and 'int'
```

Attention à ne pas confondre l'opérateur de comparaison `==` avec l'opérateur d'affectation `=`.

```
>>> x = 3 # une affectation qui modifie la mémoire
>>> x == 3 # une comparaison vrai ou faux ?
True
>>> 1 == 3 # une comparaison vrai ou faux ?
False
>>> 1 = 3
[...]
SyntaxError: cannot assign to literal here. Maybe you meant '==' instead of '='?
```

On comprend maintenant ce message d'erreur !

On pourra écrire par exemple:

```
x = (a == b)
```

pour affecter à `x` la valeur `True` ssi `a` et `b` sont associés à la même valeur en mémoire et la valeur `False` sinon.

2.2 Égalité de chaînes de caractères

Deux chaînes de caractères sont égales si et seulement si :

- elles ont le même nombre de caractères
- le premier caractère de chacune des deux chaînes sont identiques
- le deuxième caractère de chacune des deux chaînes sont identiques

- etc.

On utilise les opérateurs `==` et `!=` sur les chaînes :

```
>>> 'oui' == 'o' + 'u' + 'i'
True
>>> 'oui' == 'oui '
False
>>> 'oui' != 'Oui'
True
>>> "C cool !" == 'C cool !'
True
```

Notons que la comparaison de chaînes de caractères est sensible à la casse : les majuscules et minuscules sont des caractères différents. De plus, l'espace est aussi un caractère. Rappel :

```
>>> len(' ')
1
```

Par ailleurs, des valeurs littérales différentes permettent d'écrire une même chaîne de caractères (par exemple en utilisant les apostrophes `'` ou les guillemets `"`). Il s'agit cependant bien de la même valeur, la comparaison par `==` donne vrai.

2.3 Comparer des flottants

Les flottants sont des approximations de valeurs entières. La comparaison de flottants nécessite de prendre des précautions.

```
>>> z = 3.0
>>> 3.0 == z
True
```

```
>>> 3 * 0.1 == 0.3
False
>>> 3 * 0.1
0.30000000000000004
```

Nous avons vu que L1test propose un mécanisme propre au test de valeurs de type flottant.

2.4 L'oracle de L1Test est basé sur l'opérateur de comparaison

On peut maintenant expliquer comment L1test calcule un verdict.

On rappelle que dans sa forme la plus simple un L1test s'écrit sur 2 lignes :

```
$$$ <expression_à_évaluer>
<expression_résultat_attendu>
```

L1test compare la valeur obtenue (par évaluation de `<expression_à_evaluer>`) et la valeur attendue (obtenue par évaluation de `<expression_résultat_attendu>`) en utilisant l'opérateur de comparaison `==`.

Le résultat de la comparaison est

- soit vrai (`True`) et alors le test passe
- soit faux (`False`)[1] et alors le test échoue.

3 Prédicats

Un **prédicat** est une fonction qui renvoie une valeur booléenne.

Définissons un prédicat `est_pair()` pour vérifier la parité d'un entier donné. Voici ses signature et docstring :

[1] s'il ne se produit pas une erreur.


```
def est_pair(n : int) -> bool:
    """Renvoie True si et seulement si n est pair, False sinon.

    precondition : aucune
    Exemples:
    $$$ est_pair(42)
    True
    $$$ est_pair(13)
    False
    $$$ est_pair(0)
    True
    """
```

Un entier est pair si le reste de sa division entière par 2 est nul. On pourra donc compléter notre fonction ainsi :

```
def est_pair(n : int) -> bool:
    """
    [...]
    """
    return n%2 == 0
```

3.1 L'opérateur `in`

En plus des opérateurs déjà vus, un prédicat peut utiliser l'opérateur `in`, qui permet de répondre à la question : "ce caractère apparaît dans cette chaîne : vrai ou faux ?". Par exemple :

```
>>> 'a' in 'chat'
True
>>> 'a' in 'ours'
False
>>> 'a' in ''
False
```

Les opérandes de l'opérateur `in` doivent être de type `str` :

```
>>> 1 in '123'
[...]
TypeError: 'in <string>' requires string as left operand, not int
>>> '1' in 12
[...]
TypeError: argument of type 'int' is not iterable
```

Plus généralement l'opérateur répond à la question : "cette chaîne (à gauche) apparaît dans cette autre chaîne (à droite) : vrai ou faux ?" :

```
>>> 'ha' in 'chat'
True
>>> 'ha' in 'hopla'
False
>>> '' in 'chat'
True
```

3.2 Prédicats prédéfinis

Python fournit des prédicats prédéfinis, par exemple :

- cette chaîne de caractères contient uniquement des entiers, vrai ou faux ?
- cette chaîne de caractères contient uniquement des lettres de l'alphabet (parmi `'a'…'z'`, `'A'…'Z'`), vrai ou faux ?
- et bien d'autres !

La syntaxe utilise la notation objet pointée que vous verrez au S2 : `<chaine>.<nom_fonction>()`.

Par exemple :

```
>>> '123'.isdigit()
True
>>> '12h'.isdigit()
False
>>> 'a'.isdigit()
False
>>> ''.isdigit()
False
>>> 'Bonjour'.isalpha()
True
>>> 'Bonjour!'.isalpha()
False
>>> ''.isalpha()
False
```

4 Memento

- le type *booléen* `bool` permet de représenter les valeurs de vérité *vrai* et *faux*
- en Python, on utilise les mots-clés `True` et `False`
- les opérateurs de comparaison testent l'égalité, la différence, l'infériorité, le supériorité de deux valeurs
- un opérateur de comparaison produit une valeur booléenne
- en Python, l'opérateur d'égalité est noté `==`, l'opérateur de non égalité est noté `!=`
- un prédicat est une fonction qui renvoie une valeur booléenne

On découvre

- comment combiner des valeurs booléennes pour en produire d'autres
- les opérateurs logiques *et*, *ou*, *non*
- comment une table de vérité définit un opérateur logique

1 Opérateurs booléens

Les opérateurs booléens, aussi appelés opérateurs logiques, combinent des valeurs booléennes pour produire une nouvelle valeur booléenne.

Il existe principalement trois opérateurs logiques :

- la négation - le *non* -,
- la conjonction - le *et logique* -,
- la disjonction - le *ou logique* -

1.1 La négation – `not`

L'opérateur de *négation* permet d'obtenir la valeur inverse d'une valeur booléenne.

En Python, l'opérateur de négation est noté `not`. On écrit par exemple

```
>>> pair = True
>>> not pair
False
```

ou

```
>>> not (3 < 42)
False
>>> not ('Zim' == 'Zac')
True
```

On décrit couramment les opérateurs booléens par leur table de vérité. Pour l'opérateur de négation :

x	not x
False	True
True	False

1.2 Le et logique — `and`

Le *et logique* permet d'exprimer le fait que deux expressions sont vraies simultanément : x and y est vrai si et seulement si x est vrai, et y est vrai aussi.

En Python, le *et logique* est noté `and`. On écrit par exemple

```
>>> pair = True
>>> positif = False
>>> pair and positif
False
```

ou

```
>>> a = 12
>>> a < 42 and a > 0
True
>>> 'Zim' != 'Zac' and 'Zim' != 'Zou'
True
```

La table de vérité de l'opérateur *et logique* est la suivante

x	y	x and y
False	False	False
False	True	False
True	False	False
True	True	True

1.3 Le ou logique — `or`

Le *ou logique* permet d'exprimer le fait qu'une parmi deux expressions est vraie : x or y est vrai si soit x est vrai, soit y est vrai, soit les deux sont vrais.

Remarquons que x or y est aussi vrai si x est vrai et y est vrai.

En Python, le *ou logique* est noté `or`. On écrit par exemple

```
>>> pair = True
>>> positif = False
>>> pair or positif
True
```

ou

```
>>> a = 12
>>> a < 0 or a > 42
False
>>> 'Zim' == 'Zac' or 'Zim' == 'Zou'
False
```

La table de vérité de l'opérateur *ou logique* est la suivante

x	y	x or y
False	False	False
False	True	True
True	False	True
True	True	True

1.4 Expression booléenne

Un expression booléenne est une expression à valeur booléenne.

Une expression combine librement les différents opérateurs booléens et les différents opérateurs de comparaison. Par ex :

```
>>> not (1 > 3) and "e" in "heu"
True
```

L'opérande gauche `not (1 > 3)` du `and` est évalué. Comme `1 > 3` est évalué à `False`, `not (1 > 3)` est évalué à `True`. Puis l'opérande droit `"e" in "heu"` du `and` est évalué à `True`. L'ensemble de l'expression est donc évaluée à `True and True`, soit `True`.

Il peut être plus simple de faire une table de vérité :

x	y	not x	not x and y
False	True	True	True
False	False	True	False
True	True	False	False

x	y	not x	not x and y
True	False	False	False

On peut aussi écrire des expressions qui semblent absurdes mais valent juste `False`, comme :

```
>>> x = 14
>>> x == 1 and x == 3
False
```

Enfin, quand une erreur se produit lors de l'évaluation d'une sous-expression, le reste de l'expression n'est pas évalué.

```
>>> sqrt(-5) > 0 and 1/0 > 0
[...]
ValueError: math domain error
```

On voit que l'expression `1/0` n'a pas été évaluée, sinon on aurait aussi eu une erreur de division par 0.

2 Priorité des opérateurs

En Python les opérateurs de comparaison `< > == >= <= !=` ont tous la même priorité.

Par ailleurs l'opérateur `not` est prioritaire sur l'opérateur `and` qui est lui-même prioritaire sur l'opérateur `or`.

Les expressions `x and y or z and t` et `(x and y) or (z and t)` sont donc équivalentes.

Finalement les opérateurs arithmétiques sont plus prioritaires que les opérateurs de comparaison, qui sont plus prioritaires que les opérateurs booléens.

Les expressions suivantes sont donc équivalentes :

- `not 3 < 42` et `not (3 < 42)`
- `not 'Zim' == 'Zac'` et `not ('Zim' == 'Zac')`
- `a < 42 and a > 0` et `(a < 42) and (a > 0)`
- `a < b+1` et `a < (b+1)`

Comme pour les priorités des opérateurs arithmétiques, il ne faut pas hésiter à ajouter des parenthèses dès qu'on a un doute. Si on l'a, le lecteur du code pourra l'avoir aussi.

3 Opérateurs booléens séquentiels

Observons la table de vérité de l'opérateur `and` :

x	y	x and y
False	False	False
False	True	False
True	False	False
True	True	True

Pour évaluer `x and y` :

- si `x` est évalué à `False`
- quelque soit la valeur de `y`, `x and y` vaudra `False`

Python va utiliser cette remarque pour définir l'opérateur `and`. Ainsi, pour évaluer l'expression `x and y` :

- Python évalue la partie gauche, le `x`
- si elle vaut `False`, on a terminé
- sinon, Python évalue la partie droite, le `y`

On parle d'opérateur **séquentiel** : la partie gauche est évaluée. Puis, si nécessaire, la partie droite est évaluée.

L'opérateur `or` est aussi séquentiel. Cette fois, c'est `True or x` qui vaut toujours `True`, quelle que soit la valeur de `x`. Ainsi, pour évaluer l'expression `x or y` :

- Python évalue la partie gauche, le `x`
- si elle vaut `True`, on a terminé
- sinon, Python évalue la partie droite, le `y`

La séquentialité des opérateurs `and` et `or` sera fondamentale quand nous verrons les boucles `while`.

4 Memento

- un opérateur logique combine des valeurs booléennes et produit une valeur booléenne
- l'opérateur logique de négation est noté `not` en Python
- l'opérateur *et logique* est noté `and` en Python
- l'opérateur *ou logique* est noté `or` en Python
- on combine ces opérateurs logiques pour former des expressions booléennes
- les opérateurs logiques de Python sont *séquentiels*