Numérique et Sciences Informatiques
Chapitre I - Les booléens

Classe de première

I. Généralités

I.1. Définition

Définition

Un booléen est une variable à deux états : vrai ou faux.

Remarque

Certains langages utilisent le bit pour représenter des booléens : 0 pour faux et 1 pour vrai.
En python, vrai est représenté par `True` et faux par `False`.
Attention, en Python, les valeurs `1` et `1.0` sont équivalentes à `True` et n'importe quel autre nombre est équivalent à `False`.

I.2. Opérations booléennes

L'algèbre de Boole est une partie des mathématiques qui s'intéresse aux opérations et aux fonctions sur les variables logiques.
La logique est le domaine définissant les lois formelles du raisonnement et est utilisée pour le calcul des propositions ($\vee$, $\wedge$, $\neg$, $\implies$, $\iff$, $\forall$, $\exists$, $=$, etc.).
L'algèbre de Boole est utilisée dans la conception des circuits électroniques.

Les opérateurs booléens que nous allons étudier sont :

Description	définition	en français	symbole en algèbre de Boole	opérateurs logiques en Python
La négation		NON	$\neg$	not
La conjonction	renvoie `True` si les deux opérandes valent `True`	ET	$\wedge$ (ou .)	and
La disjonction	renvoie `True` si au moins un des opérandes vaut `True`	OU (inclusif)	$\vee$ (ou $+$)	or

I.3. Table de vérité

Table de vérité de la négation (not, $\neg$)

x	$\neg x$
0	1
1	0

Table de vérité de la conjonction (and, $\wedge$ ou .)

x	y	$x \wedge y$
0	0	0
0	1	0
1	0	0
1	1	1

Table de vérité de la disjonction (or, $\vee$ ou $+$)

x	y	$x \vee y$
0	0	0
0	1	1
1	0	1
1	1	1

I.4. Propriétés des opérateurs booléens

Dans cette partie on considère trois variables booléennes x, y et z, qui valent donc soit 1 (vrai) ou 0 (faux).

Propriété d'associativité

$(x \wedge y) \wedge z = x \wedge (y \wedge z) = x \wedge y \wedge z$
$(x \vee y) \vee z = x \vee (y \vee z) = x \vee y \vee z$

Propriété de commutativité

$x \wedge y = y \wedge x$
$x \vee y = y \vee x$

Propriété de distributivité

$x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$
$x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$

Propriété d'impotence

$x \wedge x = x$
$x \vee x = x$

Eléments neutres

$x \wedge 1 = x$
$x \vee 0 = x$

Eléments absorbants

$x \wedge 0 = 0$
$x \vee 1 = 1$

Propriété de complémentarité

$\neg(\neg x) = x$
$x \vee (\neg x) = 1$
$x \wedge (\neg x) = 0$
$\neg(x \vee y) = (\neg x) \wedge (\neg y)$
$\neg(x \wedge y) = (\neg x) \vee (\neg y)$

Remarque

Python évalue les expression booléennes de façon paresseuse : dès que le résultat final est connu (grâce aux propriétés des opérateurs booléens), l'évaluation est stoppée. L'ordre d'écriture des expressions booléennes est donc important en Python.
Exemple :

```
>>> a = True
>>> b = False
>>> c = True
>>> a or (b and c)
True
```

Dans ce cas, il évalue `a` qui vaut `True`, puis voit l'opérateur `or` et enfin une expression entre parenthèses. Python renvoie donc `True` puisque, peu importe la valeur de l'expression entre parenthèses, le résultat sera `True`.

I.5. Exercices

I.5.a. Exercice 1

L'opérateur booléen Xor, symbolisé par $\oplus$, renvoie True si et seulement si les deux opérandes ont des valeurs différentes.

1. Dresser la table de vérité de l'opérateur booléen Xor.
2. Dresser la table de vérité de l'expression booléenne suivante :

$$A = (x \wedge (\neg y)) \vee ((\neg x) \wedge y)$$

3. Que remarquez-vous ?

I.5.b. Exercice 2

Simplifier les expressions suivantes, c'est-à-dire les écrire avec le moins d'opérateurs et avec les variables qui apparaissent le moins souvent.

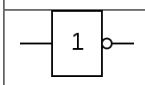
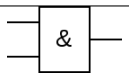
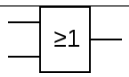
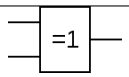
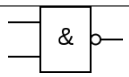
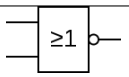
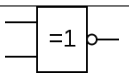
1. $a \wedge (a \vee b)$
2. $(a \vee b) \wedge (\neg a \vee \neg b)$
3. $(a \wedge b) \vee c \vee (c \wedge (\neg a \vee \neg b))$
4. $a \vee (a \wedge b \wedge c) \vee (\neg a \wedge b \wedge c) \vee (\neg a \wedge b) \vee (a \wedge d) \vee (a \wedge \neg d)$
5. $(a \wedge b \wedge \neg c) \vee (b \wedge (a \vee \neg c)) \vee \neg(\neg a \vee b \vee (\neg a \wedge c))$

I.5.c. Exercice 3

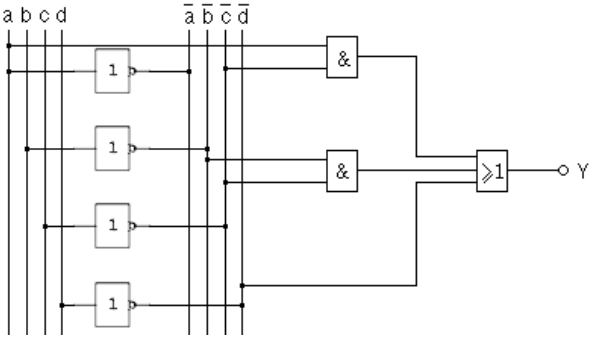
Quatre responsables d'une société (A, B, C et D) peuvent avoir accès à un coffre. Ils possèdent chacun une clé différente (a, b, c et d) et il a été convenu que :

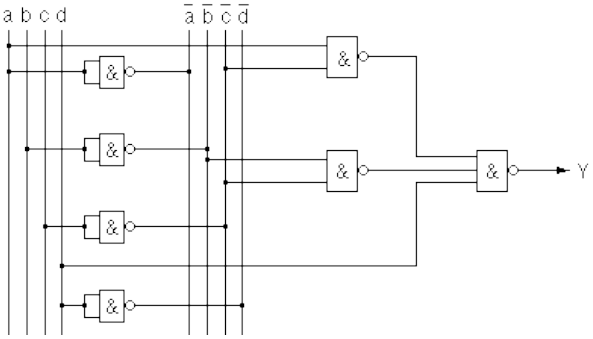
- A ne peut ouvrir le coffre que si au moins un des responsables B ou C est présent.
- B, C et D ne peuvent ouvrir le coffre que si au moins deux des autres responsables sont présents.

1. Donner une expression logique de la serrure S du coffre.
2. Simplifier cette expression.
3. Etablir le logigramme de cette serrure.
   Un logigramme est composé d'entrées, d'une sortie et de portes logiques ayant une sortie et de 2 à 4 entrées (à l'exception de la porte logique NON qui ne possède qu'une entrée)
   Ils sont symbolisés par :

NOT	AND	OR	XOR	NOT AND	NOT OR	NOT XOR
						

   Un logigramme peut ressembler aux schémas ci-dessous :





   a, b, c et d sont les entrées et Y est la sortie.

4. Combien de circuits intégrés de la série 7400 sont nécessaires à cette serrure ? Vous pourrez chercher des informations sur ces circuits intégrés ici.
5. Refaire les questions 3 et 4 avec des portes logiques à 2 entrées uniquement.
6. Les circuits intégrés NON-ET sont moins chers. Refaire les questions 3 et 4 uniquement avec des portes logiques NON-ET à deux entrées.

II. Notions de Python - constructions élémentaires

II.1. Généralités

Définition

Une **instruction** est une étape dans un programme informatique.

Définition

Une **séquence** est une suite d'instructions. Elles sont exécutées les unes à la suite des autres, dans l'ordre d'écriture.

II.2. Les opérations

Python maîtrise les opérations élémentaires.

- la somme : `+`

```
>>> 13 + 5
 18
>>> 0.2 + 0.1
 0.30000000000000004
```

- la différence : `-`

```
>>> 13 - 5
 8
```

- le produit : `*`

```
>>> 13 * 5
 65
```

- le quotient : `/`

```
>>> 13 / 5
 2.6
>>> 13 / 3
 4.333333333333333
```

- le quotient de la division euclidienne : `//`

```
>>> 13 // 5
 2
```

- le reste de la division euclidienne : `%`

```
>>> 13 % 5
 3
```

- la puissance : `**`

```
>>> 13 ** 5
 371293
```

II.3. Les variables

Définition

Une **variable** est un emplacement mémoire. Dans cet emplacement est stockée la **valeur** de la variable.

En python, on affecte une valeur à une variable avec le symbole `=` :

```
nom_de_la_variable = valeur_de_la_variable
```

II.4. Les tests conditionnels

Définition

Une **condition** est une instruction qui renvoie un booléen.

Les comparaisons

Les opérateurs de comparaison renvoient un booléen.

Symbole	Signification
==	est égal à
!=	n'est pas égal à (ou est différent de)
<	est inférieur à
>	est supérieur à
<=	est inférieur ou égal à
>=	est supérieur ou égal à

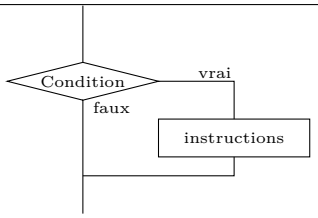
L'appartenance

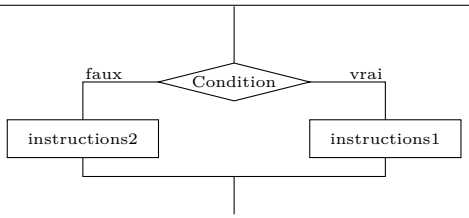
L'opérateur d'appartenance `in` renvoie un booléen.

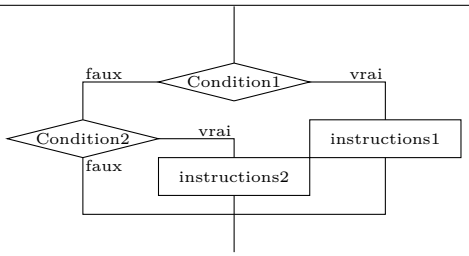
```
>>> chaine = "abcde"
>>> "b" in chaine
True
>>> "f" in chaine
False
>>> "f" not in chaine
True
```

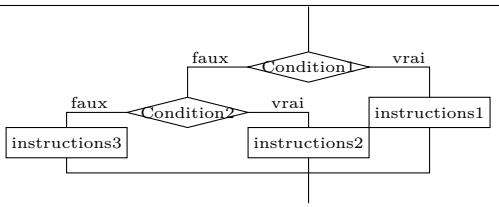
Définition

Une **instruction conditionnelle** est une instruction qui s'effectue en fonction de l'évaluation d'une condition booléenne.

Algorithmique	En python	Diagramme de flux
Si *condition* alors *instructions*	`if condition:` `    instructions`	

Si *condition*
alors *instructions1*
sinon *instructions2*
	`if condition:`
	`    instructions1`
	`else:`
	`    instructions2`
		

Si *condition1*
alors *instructions*
et si *condition2*
alors *instrucitons2*
	`if condition1:`
	`    instructions1`
	`elif condition2:`
	`    instructions2`
		

Si *condition1*
alors *instructions*
et si *condition2*
alors *instrucitons2*
⋮
sinon *instructions3*
	`if condition1:`
	`    instructions1`
	`elif conditions2:`
	`    instructions2`
	`...`
	`else:`
	`    instructions3`
		

Exemple

```
nb = ...
if nb % 2 == 0:
    resultat = nb // 2
else:
    resultat = 3 * nb + 1
```

Par exemple :

- Si la variable `nb` a pour valeur 15, alors la variable `resultat` aura pour valeur 46.
- Si la variable `nb` a pour valeur 16, alors la variable `resultat` aura pour valeur 8.

```
nb = ...
if nb % 2 == 0:
    resultat = nb // 2
elif nb % 3 == 0:
    resultat = nb + 2
else:
    resultat = 3 * nb + 1
```

Par exemple :

- Si la variable `nb` a pour valeur 15, alors la variable `resultat` aura pour valeur 17.
- Si la variable `nb` a pour valeur 16, alors la variable `resultat` aura pour valeur 8.
- Si la variable `nb` a pour valeur 17, alors la variable `resultat` aura pour valeur 52.

II.5. Les fonctions, procédures et prédicats

Définition

Une **fonction** est une variable particulière dont la valeur est une séquence d'instructions.

Remarque

En python, pour créer une fonction, on utilise le mot-clé `def`, suivi du nom de la fonction et des éventuels arguments entre parenthèses, séparés par une virgule. Enfin, on place `:` à la fin de la première ligne.

```
1 def nom_fonction(argument1, argument2, ...):
2     instructions
```

Pour appeler une fonction, on écrit le nom de la fonction et les éventuels paramètres entre parenthèses, séparés par des virgules.

```
1 >>> nom_fonction(parametre1, parametre2, ...)
2 le_resultat
```

Les arguments peuvent avoir des valeurs par défaut, dans le cas où on ne renseigne pas les paramètres lors de l'appel de la fonction. Pour donner ces valeurs par défaut, il suffit d'utiliser le symbole `=` puis la valeur par defaut à la suite du nom de l'argument.

```
1 def nom_fonction(argument1, argument2 = valeur_par_defaut, ...):
2     instructions
```

Ainsi dans l'appel de `nom_fonction()`, si on omet le paramètre correspondant à `argument2`, ce dernier prendra automatiquement la valeur `valeur_par_defaut` pour les instructions de la fonction.

Vocabulaire

- Une fonction qui ne renvoie rien (`None`) est appelée **procédure**.
- Une fonction qui renvoie un booléen (`True` ou `False`) est appelée **prédicat**.

Portée d'une variable

Lorsqu'une variable est créée dans une fonction (soit en argument, soit dans les instructions de la fonction), on dit que sa **portée** est **locale**, ou plus simplement qu'il s'agist d'une **variable locale**.
Lorsqu'une variable est créée en dehors des fonctions, on dit que sa **portée** est **globale**, ou plus simplement qu'il s'agist d'une **variable globale**.

Remarque

Pour rendre globale une variable locale, on peut utiliser le mot-clé `global` devant le nom de la variable pour créer cette dernière.

Exemple

Soit les instructions suivantes :

```
1  def carre(x):
2      global resultat
3      resultat = x * x
4      return resultat
5
6  truc = 27
7
8  def affiche_carre(x):
9      print(truc)
10     print(carre(x))
11
12 def est_majeur(age = truc):
13     if age >= 18:
14         return True
15     else:
16         return False
```

Les variables `x` et `age` sont locales.

Par défaut, `age` a la valeur de `truc`.

Les variables `resultat` et `truc` sont globales.

La variable `resultat` ne peut être utilisée que si la fonction `carre(x)` a déjà été exécutée.

La fonction `affiche_carre(x)` possède un argument `x` et ne renvoie rien donc est une procédure.

La fonction `est_majeur(age)` possède un argument `age` et renvoie un booléen donc est un prédicat.

II.6. Spécification et mise au point de programmes

II.7. Spécifications

Définition

Spécifier un programme (ou une fonction), c'est décrire explicitement ce qu'il doit faire et dans quelles conditions.
Les spécifications sont composées de la signature du programme, des éventuelles préconditions et postconditions.

Les commentaires

Commenter son programme est nécessaire pour se rappeler en quoi consiste les instructions écrites.
En Python, en utilise le mot-clé `#`. Tout ce qui suis le `#` sur la ligne n'est pas interprété par Python.

```
1  c = a + b # c est égal à la somme de a et b
2  # Ce qui suit est une fonction à un argument
3  def inverse(a):
4      # instructions de la fonction :
5      resultat = 1 / a
6
7      return resultat
```

Les docstrings

Lorsqu'on écrit une fonction, il faut **documenter** la fonction, c'est-à-dire indiquer :

- la signature de la fonction :
  - le nom de la fonction et ses arguments
  - ce qu'elle fait
  - de quels types sont les arguments
  - de quel type est le résultat renvoyé
- les éventuelles conditions sur le résultat, appelés **postconditions**
- les éventuelles conditions d'utilisation, qu'on appelle des **préconditions**

Une docstrings s'écrit sur les lignes qui suivent la définition de la fonction, entre triple guillemets.

```
def inverse(a: float) -> float:
    """
    Renvoie l'inverse de a
    :param a: (float)  un flottant
    :return: (float) un flottant
    :CU: a != 0
    """
    resultat = 1 / a
    return resultat
```

Un utilisateur qui demande de l'aide sur la fonction verra alors les docstrings et pourra comprendre comment l'utiliser.

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

inverse(a: float) -> float
    Renvoie l'inverse de a
    :param a: (float or int)  un flottant ou un entier
    :return: (float) un flottant
    :CU: a != 0
```

II.7.a. Les assertions

Dans le cadre de la programmation défensive, on peut écrire les préconditions et les postconditions sous forme d'**assertions** en utilisant le mot-clé `assert`.

```
def inverse(a):
    """
    Renvoie l'inverse de a
    :param a: (float or int)  un flottant ou un entier
    :return: (float) un flottant
    :CU: a doit être différent de 0
    """
    # une précondition :
    assert a != 0, "a doit être différent de 0"
    # instructions de la fonction :
    resultat = 1 / a
    # une postcondition
    assert resultat != 0, "Il y a un problème."
    return resultat
```

II.7.b. La mise au point

Définition

La mise au point d'un programme (ou d'une fonction) est la création de jeux de tests qui permettent de vérifier si, avec des paramètres donnés, la fonction renvoie bien ce qui est attendu.

Remarque

Pour faire cela, on peut :

- Soit placer ces tests dans les docstrings et on faire appel à la bibliothèque `doctests`.
- Soit utiliser des assertions à l'extérieur de la fonction

Remarque

Afin d'espérer le bon fonctionnement d'une fonction :

- il faut un nombre suffisant de tests
- il faut des tests de bonne qualité (un cas général ou deux et des (tous les) cas particuliers sont nécessaires)

Ce n'est pas parce qu'un jeu de test réussit que la fonction est correcte.

```python
def inverse(a: float) -> float:
    """
    Renvoie l'inverse de a
    :param a: (float)  un flottant
    :return: (float) un flottant
    :CU: a doit être différent de 0
    :Exemples:
    >>> inverse(2)
    0.5
    >>> inverse(0)
    Traceback (most recent call last):
    ...
    AssertionError: a doit être différent de 0
    >>> inverse(3)
    0.3333333333333333
    """
    # une précondition :
    assert a != 0, "a doit être différent de 0"
    # instructions de la fonction :
    resultat = 1 / a
    # une postcondition
    assert resultat != 0, "Il y a un problème."
    return resultat

assert inverse(2) == 0.5, "Il y a une erreur dans la fonction"

if __name__ == "__main__":
    import doctest
    doctest.testmod(verbose=True) # exécute les jeux de test des docstrings
```

II.8. Exercices

Dans tous ces exercices, vous ferez apparaître des spécifications et des mises au point afin de montrer des préconditions, des postconditions et des jeux de test.

II.8.a. Exercice 1

La suite de Syracuse d'un nombre entier strictement positif N est définie ainsi :

- si le nombre N est pair alors le suivant est l'entier qui vaut la moitié de N
- si le nombre N est impair alors le suivant est l'entier qui vaut le triple de N augmenté de 1

Écrire une fonction `suivant_syracuse(nb)` qui renvoie le terme suivant dans la suite de Syracuse de `nb`.

II.8.b. Exercice 2

1. Écrire un prédicat `est_majeur(age)` qui renvoie `True` si `age` correspond à l'age d'une personne majeure, `False` sinon.
2. Écrire un prédicat `peut_passer_permis(age)` qui renvoie `True` si `age` correspond à l'age d'une personne qui peut passer le permis, `False` sinon.

II.8.c. Exercice 3

Dans cet exercice, on n'utilisera pas la fonction `max(*args)` native de Python.

1. Écrire une fonction `maxi2(a, b)` qui renvoie le maximum des deux nombres entrés en paramètres.
2. Écrire une fonction `maxi3(a, b, c)` qui renvoie le maximum des trois nombres entrés en paramètres.

II.8.d. Exercice 4

1. Écrire un prédicat `multiple3(a)` qui renvoie `True` si `a` est un multiple de 3.
2. Écrire un prédicat `multiple(a, b)` qui renvoie `True` si `a` est un multiple de `b`.