TD 1.2 : Opérateurs booléens

Bases de la Programmation Impérative – Chapitre 1 : bases
version du 20 octobre 2021

L'objectif de ce TD est de se familiariser avec les opérateurs logiques manipulant des valeurs booléennes, c'est-à-dire `True` et `False` en python, et avec la structure de contrôle conditionnelle `if`.

Exercice 1 : opérateurs logiques

On considère le programme suivant :

```
1  #!/usr/bin/env python3
2  """Premier programme avec variables booléenes et opérateurs logiques."""
3
4  a = 10
5  b = 20
6  c = 30
7  d = a < b
8  print("d =", d)
9  e = b < c and a > b
10 print("e =", e)
11 d = d or e
12 print("d =", d)
13 f = (not d) or (a + b > c and e)
14 print("f =", f)
```

Question 1
Exécuter le programme en notant, comme dans le TD précédent, l'évolution des variables au cours de l'exécution. Noter également ce qu'affiche le programme sur la sortie standard.

Cliquez ici pour révéler la correction.
Après avoir exécuté les lignes 4, 5 et 6, les variables sont les suivantes :

Nom	Type	Valeur	Portée
a	int	10	Globale
b	int	20	Globale
c	int	30	Globale

L'opérateur `<` appliqué à des entiers renvoie `True` si l'opérande de gauche est strictement plus petite que celle de droite et `False` sinon. Après avoir exécuté la ligne 7, les variables sont donc les suivantes :

Nom	Type	Valeur	Portée
a	int	10	Globale
b	int	20	Globale
c	int	30	Globale
d	bool	True	Globale

L'exécution de la ligne 8 va afficher `d = True`. Ici il est important de noter plusieurs choses concernant la fonction `print`, qui :

- peut recevoir un nombre variable d'arguments et que chacun d'entre eux sera affiché, deux arguments ;
- applique la fonction `str` sur chacun des arguments qui n'est pas une chaîne de caractère pour le convertir en chaîne, `d` dans notre exemple ;
- possède un paramètre optionnel `sep` dont la valeur par défaut est `" "` qui indique la chaîne de caractère à afficher entre chacun des arguments, c'est pourquoi il y a un espace dans l'affichage entre `=` et `True` ;

Pour comprendre ce qu'il se passe lorsque la ligne 9 est exécutée, il faut :

- se rappeler que dans une affectation, on commence par évaluer la partie droite (logique : il faut connaître la valeur à affecter à la variable pour faire l'affectation)
- connaître les priorités entre les opérateurs, autrement dit savoir ici dans quel ordre on va appliquer les opérateurs `<`, `and` et `>`. Pour python, la table des priorité est disponible ici. Pour ce qui nous intéresse ici, `<` et `>` sont prioritaires sur le `and`.
- connaître la table de vérité du *et* logique (ci-dessous), autrement dit savoir ce que fait l'opérateur `and`.

a	b	a et b
Vrai	Vrai	Vrai
Vrai	Faux	Faux
Faux	Faux	Faux
Faux	Vrai	Faux

Donc pour l'exécution de la ligne 9, l'interpréteur va d'abord évaluer `b < c` qui donne `True` puis `a > b` qui donne `False` et enfin faire le `and` entre les deux qui donne `False`. L'affectation à `e` peut ensuite avoir lieu, et donc une fois la ligne 9 exécutée, les variables sont les suivantes :

Nom	Type	Valeur	Portée
a	int	10	Globale
b	int	20	Globale
c	int	30	Globale
d	bool	True	Globale
e	bool	False	Globale

L'exécution de la ligne 10 va donc afficher `e = False` sur la sortie standard.

Pour comprendre ce que fait l'exécution de la ligne 11, voici la table de vérité de l'opérateur logique *ou*, c'est à dire `or` en python :

a	b	a ou b
Vrai	Vrai	Vrai
Vrai	Faux	Vrai
Faux	Faux	Faux
Faux	Vrai	Vrai

Après l'exécution de la ligne 11, les variables sont donc les suivantes :

Nom	Type	Valeur	Portée
a	int	10	Globale
b	int	20	Globale
c	int	30	Globale
d	bool	True	Globale
e	bool	False	Globale

L'exécution de la ligne 12 va afficher `d = True`.

Pour savoir ce que l'exécution de la ligne 13 va donner, il faut retourner voir la table des priorité entre opérateurs et prendre en compte les parenthèses. L'interpréteur va ici commencer par évaluer la partie gauche du `or`, à savoir `not d`, qui va donner `False` car l'opérateur `not` inverse la valeur booléenne qui lui est donnée. Ensuite, la partie droite est évaluée, c'est à dire `a + b > c and e`. La table des priorités nous dit que l'on commence par appliquer `+` puis `>` et enfin `and`. La partie droite du `or` est donc évaluée à `False` et dont le `or` lui même est évalué à `False` car ses deux opérandes valent `False`. Après l'exécution de la ligne 13, les variables sont donc les suivantes :

Nom	Type	Valeur	Portée
a	int	10	Globale
b	int	20	Globale
c	int	30	Globale
d	bool	True	Globale
e	bool	False	Globale
f	bool	False	Globale

L'exécution de la ligne 14 va afficher `f = False`.

Exercice 2 : opérateurs logiques toujours

On considère le programme suivant :

```
1  #!/usr/bin/env python3
2  """Qu'affiche ce programme ?"""
3
4  def est_pair(entier):
5      """Affiche et renvoie la parité ou non de entier."""
6      parite = entier % 2 == 0
7      print(entier, "est pair =", parite)
8      return parite
9
10 a = 2 < 3 < 4 or est_pair(4)
11 b = est_pair(6) and est_pair(3)
12 print("a =", a, ", b =", b)
```

Question 1
Qu'affiche ce programme sur la sortie standard quand on l'exécute ?

Cliquez ici pour révéler la correction.
Voici le résultat :

```
1  > ./mystere.py
2  6 est pair = True
3  3 est pair = False
4  a = True , b = False
```

Pour comprendre ce qu'il se passe ici, il faut savoir que :

- l'opérateur `a % n` (modulo) renvoie le reste de la division euclidienne de `a` par `n` ;
- `2 < 3 < 4` est un raccourci d'écriture de `2 < 3 and 3 < 4` ;
- les opérateurs `or` et `and` sont des opérateurs dit "court-circuit". C'est à dire que pour l'opérateur `or` l'opérande de droite n'est pas évaluée si l'opérande de gauche est vraie. Pour l'opérateur `and` l'opérande de droite n'est pas évaluée si l'opérande de gauche est fausse.

Comme un appel à la fonction `est_pair` produit des **effets de bords**, c'est à dire des effets visibles à l'extérieur de la fonction, en l'occurrence un affichage sur la sortie standard, le court-circuit est visible sur cet exemple.

Exercice 3 : réécritures

On considère le programme suivant :

```
1  #!/usr/bin/env python3
2
3  ...
4  b = 7
5  if a == 3:
6      b = 4
7      e = a + 1
```

```
8  elif a == 5:
9      b = 0
10     e = a - 1
11 else:
12     e = 4
```

Question 1
Comment simplifier ce code ?

Cliquez ici pour révéler la correction.

```
1  ...
2  b = 7
3  if a == 3:
4      b = 4
5  elif a == 5:
6      b = 0
7  e = 4
```

Question 2
En sachant que `int(False)` vaut `0` et que `int(True)` vaut `1`, trouver une expression arithmétique calculant la valeur de `b` en fonction de `a`.

Cliquez ici pour révéler la correction.

```
1  b = int(a == 3) * 4 + int(a != 3 and a != 5) * 7
```

Question 3
On suppose maintenant que `a` est entier compris entre `0` et `6`. À l'aide d'un tuple de 7 éléments, supprimer toutes les comparaisons pour affecter la bonne valeur à `b`.

Cliquez ici pour révéler la correction.

```
1  valeurs_b = (7, 7, 7, 4, 7, 0, 7)
2  b = valeurs_b[a]
```

On peut s'interroger ici sur la lisibilité du code : cette version est-elle plus lisible que la version avec `if`, `elif` et `else` ? Il n'y a pas de "bonne" réponse à cette question, c'est le contexte ainsi que les habitudes de chacun qui permettront de choisir entre les deux versions.

Question 4
Pour ceux qui connaissent déjà python, comment raccourcir le test suivant sachant que `s` est une chaîne de caractères, `i` un entier et `t` un `tuple` ?

```
1  if len(s) != 0 and i != 0 and len(t) == 0 :
2      // do something
```

Cliquez ici pour révéler la correction.

```
1  if s and i and not t:
2      // do something
```

En python, on peut utiliser une variable de n'importe quel type comme condition d'un `if`. Autrement dit, quand un booléen est attendu et qu'un autre type est fourni, il y a une conversion automatique vers un booléen. Comme on le voit dans la correction ci-dessus, en supposant qu'elle soit juste (à vérifier absolument dans un interpréteur interactif pendant la prochaine séance de TP), :

- une chaîne de caractères est convertie en `True` si sa longueur est strictement plus grande que zéro et en `False` sinon ;
- un entier est converti en `True` si il est différent de `0` et en `False` si il est égal à zéro ;
- un tuple est converti en `True` si sa longueur est strictement plus grande que zéro et en `False` sinon.

Exercice 4 : date correcte

Question 1

Écrire la fonction `date_correcte(jour, mois, annee)` qui renvoie `True` si les trois entiers donnés en argument forment une date correcte et `False` sinon. On considère pour cette première question que le mois de février a toujours 28 jours.

Question 2

Prendre en compte les années bissextiles. Une année est bissextile si elle rentre dans l'un des deux cas suivants :

- l'année est divisible par 4 et non divisible par 100 ;
- l'année est divisible par 400.

Par exemple, l'an 2000 était bissextile mais 2100 ne le sera pas.

Cliquez ici pour révéler la correction.

```python
#!/usr/bin/env python3
"""Exo de dates correctes"""

def date_correcte(jour, mois, annee):
    """Version ne prenant pas en compte les années bissextiles."""

    # Avant ca la terre n'existait pas !
    if annee < -4.54 * 1E9:
        return False

    if not (mois >= 1 and mois <= 12):
        return False

    if mois == 1 or mois == 3 or mois == 5 or mois == 7 or \
            mois == 8 or mois == 10 or mois == 12:
        return jour <= 31
    else:
        if mois == 2:
            return 1 <= jour <= 28
        else:
            return 1 <= jour <= 30


def date_correcte_bissextile(jour, mois, annee):
    """Version prenant en compte les années bissextiles.

    Les fonctions peuvent être internes à d'autres fonctions.
    Cela permet "d'encapsuler" le code uniquement où il sera
    utilisé.
    """

    # Avant ca la terre n'existait pas !
    if annee < -4.54 * 1E9:
        return False

    def est_bissextile(annee):
        """L'année donnée est elle bissextile ?

        Ici on a une fonction dans une fonction,
        c'est à dire une fonction interne.
        """
        return annee % 400 == 0 or (annee % 4 == 0 and
                                    not annee % 100 == 0)

    if not (mois >= 1 and mois <= 12):
        return False

    if est_bissextile(annee):
```

```
49          return 1 <= jour <= 29
50
51      # On simplifie le test du mois à l'aide d'un tuple
52      jours_par_mois = (None, 31, 28, 31, 30, 31, 30, 31,
53                        31, 30, 31, 30, 31)
54      return 1 <= jour <= jours_par_mois[mois]
55
56  # On teste nos fonctions pour s'assurer de leur bon
57  # fonctionnement !
58  if __name__ == "__main__":
59      print("04/05/2014 :", date_correcte(4, 5, 2017))
60
61      print("29/02/2015 :", date_correcte(29, 2, 2015))
62      print("29/02/2015 avec prise en compte bissextiles :",
63            date_correcte_bissextile(29, 2, 2015))
64
65      print("29/02/2100 :", date_correcte(29, 2, 2100))
66      print("29/02/2100 avec prise en compte bissextiles :",
67            date_correcte_bissextile(29, 2, 2100))
68
69      print("29/02/2048 :", date_correcte(29, 2, 2048))
70      print("29/02/2048 avec prise en compte bissextiles :",
71            date_correcte_bissextile(29, 2, 2048))
72
73      print("29/02/-5 milliards :", date_correcte(14, 1, -5 * 1E9))
```

Exercice 5 : surprenant non ? (pour aller plus loin)

Question 1
Qu'affiche le programme suivant ?

```
1  s = "toto"
2  i = 41
3  print(s or i)
```

Question 2
Qu'affiche le programme suivant ?

```
1  s = "toto"
2  i = 41
3  print(s and i)
```

Cliquez ici pour révéler la correction.
Faisons quelques tests dans un interpréteur interactif (ipython ci-dessous) :

```
1  In [7]: "toto" or 41
2  Out[7]: 'toto'
3
4  In [8]: "toto" and 41
5  Out[8]: 41
```

Le premier programme affiche donc `toto` et le second `41`. Pour comprendre pourquoi, faut aller là.

Exercice 6 : fizzbuzz ? (pour aller plus loin)

Question 1
Écrire le plus "joliment" possible une fonction `fizzbuzz(nombre)` retournant :

- `"fizz"` si `nombre` est un multiple de 3

- `"buzz"` si `nombre` est un multiple de 5
- `"fizzbuzz"` si `nombre` est un multiple de 3 et de 5
- `str(nombre)` sinon

Cliquez ici pour révéler la correction.

Ce programme est célèbre, paraît-il, parcequ'il est utilisé dans des entretiens d'embauche. Voici différente implémentation dont on peut discuter.

```
#!/usr/bin/env python3
"""Exo fizzbuzz"""

# Pour ne pas rendre la correction dispo sur le site en ligne
def fizzbuzz1(nombre):
    """Version avec elif et else."""
    if nombre % 3 == 0 and nombre % 5 == 0:
        return "FizzBuzz"
    elif nombre % 3 == 0:
        return "Fizz"
    elif nombre % 5 == 0:
        return "Buzz"
    else:
        return str(nombre)

def fizzbuzz2(nombre):
    """Version sans elif et else."""
    if nombre % 3 == 0 and nombre % 5 == 0:
        return "FizzBuzz"
    if nombre % 3 == 0:
        return "Fizz"
    if nombre % 5 == 0:
        return "Buzz"
    return str(nombre)

def fizzbuzz3(nombre):
    """Version avec deux modulos."""
    resultat = ""
    if nombre % 3 == 0:
        resultat += "Fizz"
    if nombre % 5 == 0:
        resultat += "Buzz"
    if not resultat: # La chaîne vide est fausse
        resultat += str(nombre)
    return resultat

def fizzbuzz4(nombre):
    """Version avec deux modulos et plus rigolotte non ?"""
    resultat = ""
    if nombre % 3 == 0:
        resultat += "Fizz"
    if nombre % 5 == 0:
        resultat += "Buzz"
    return resultat or str(nombre)

def fizzbuzz5(nombre):
    """Pour ceux qui aiment bien les expressions arithmétiques, même si c'est
        illisible :-)"""
    return "Fizz" * (nombre % 3 == 0) + "Buzz" * (nombre % 5 == 0) or str(nombre)
```