TD 1.3 : Conditionnelles

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

L'objectif premier est de continuer à travailler avec la structure de contrôle conditionnelle `if` mais sur des programmes plus complexes que dans le TD précédent. Le second objectif consiste à s'interroger sur la qualité du code que nous allons écrire.

Exercice 1 : heure à la seconde suivante

Question 1
Écrire une fonction `tictac_1(heures, minutes, secondes)` renvoyant l'heure obtenue en incrémentant d'une seconde le temps donné en paramètre.

Cliquez ici pour révéler la correction.
On décide de renvoyer un tuple à trois éléments pour représenter l'heure :

```
def tictac_1(heures, minutes, secondes):
    """Version avec conditionnelles."""
    secondes += 1
    if secondes == 60:
        secondes = 0
        minutes += 1
        if minutes == 60:
            minutes = 0
            heures += 1
            if heures == 24:
                heures = 0

    return (heures, minutes, secondes)
```

Question 2
Écrire une fonction `tictac_2(heures, minutes, secondes)` renvoyant l'heure obtenue en incrémentant d'une seconde le temps donné en paramètre sans utiliser de structure de contrôle conditionnelle.

Cliquez ici pour révéler la correction.
Ici on utilise l'opérateur division entière `//` et l'opérateur modulo `%` calculant le reste de la division entière pour se passer des structures de contrôle conditionnelles.

```
def tictac_2(heures, minutes, secondes):
    """Version sans conditionnelles."""
    secondes += 1
    # L'opérateur // effectue une division entière
    minutes = minutes + secondes // 60
    # L'opérateur % (modulo) donne le reste de la
    # division entière
    secondes = secondes % 60
    heures = heures + minutes // 60
    minutes = minutes % 60
    heures = heures % 24
    return (heures, minutes, secondes)
```

Comme la spécification de ce que doit retourner la fonction n'est pas très précise, on pourrait décider de ne renvoyer que l'heure et avoir d'autres solutions comme celles-ci :

```
def tictac_3(heures, minutes, secondes):
    """ Version simplifié ne renvoyant que l'heure"""
    if secondes >= 59 and minutes >= 59:
        return (heures + 1) % 24
```

```
    # Un else ici serait inutile car si la
    # branche if a été prise alors on est sorti
    # de la fonction.
    return heures

def tictac_4(heure, minutes, secondes):
    """Une autre version ne renvoyant que l'heure.

    Les fonctions peuvent être internes à d'autres fonctions.
    Cela permet "d'encapsuler" le code uniquement où il sera
    utiliser.
    """
    def heure_vers_secondes(heure, minutes, secondes):
        """Convertit un triplet heure, minutes, secondes en secondes"""
        return (heure * 60 + minutes) * 60 + secondes

    def heure_depuis_secondes(secondes):
        """Convertit des secondes en heure"""
        heure = secondes // 3600
        heure = heure % 24
        return heure

    secondes_absolues = heure_vers_secondes(heure, minutes,
                                            secondes)
    secondes_absolues += 1
    return heure_depuis_secondes(secondes_absolues)
```

Exercice 2 : carrés

Un carré peut être caractérisé de nombreuses façons, par exemple :

- avec uniquement des tests d'orthogonalité (il en faut 4) ;
- avec uniquement des tests d'égalité de longueur (4 également) ;
- avec des tests d'égalité de longueur (il en faut 4) et un test d'orthogonalité.

Question 1
Écrire une fonction `est_carre(quadrilatere)` prenant en argument un tuple de `Points` (`namedtuple` vu en TP) formant un quadrilatère (les points sont donc ordonnés) et renvoyant s'ils forment un carré ou non. Penser à découper son code en sous-fonctions quand cela est pertinent.

Cliquez ici pour révéler la correction.
Un carré peut être caractérisé de nombreuses façons, par exemple :

- avec uniquement des tests d'orthogonalité (il en faut 4) : 3 angles du carré plus l'angle entre les diagonales ;
- avec uniquement des tests d'égalité de longueur (4 également) : côté 1 == côté 2, côté 2 == côté 3, côté 3 == côté 4 et diag1 == diag2 ;
- avec des tests d'égalité de longueur (il en faut 4) et un test d'orthogonalité : 4 côtés égaux + un angle droit.

On utilise dans la correction ci-dessous la troisième façon pour caractériser un carré. On utilise un namedtuple `Point` comme vu en TP. On note également que nous avons **testé** notre fonction.

```
#!/usr/bin/env python3
"""Detection de carre"""

from collections import namedtuple

def get_distance2(point1, point2):
    """Renvoie le carre de la distance entre 2 points"""
    diff_x, diff_y = point1.x - point2.x, point1.y - point2.y
    return diff_x * diff_x + diff_y * diff_y

```

```python
def est_losange(quadrilatere):
    """Renvoie si le quadrilatere donne est un losange"""

    # Quand une ligne est trop longue, on la "coupe" en python
    # avec le caractère `\`
    return get_distance2(quadrilatere[0], quadrilatere[1]) == \
        get_distance2(quadrilatere[1], quadrilatere[2]) == \
        get_distance2(quadrilatere[2], quadrilatere[3]) == \
        get_distance2(quadrilatere[3], quadrilatere[0])

def forme_angle_droit(point1, point2, point3):
    """Renvoie si l'angle p1 p2 p3 est droit"""
    # on verifie avec pyth et gore
    return get_distance2(point1, point2) + get_distance2(point2, point3) == \
        get_distance2(point3, point1)

def est_carre(quadrilatere):
    """Renvoie si le quadrilatere donne est un carre"""

    # On peut également "couper" une ligne où l'on veut quand on se situe
    # à l'intérieur d'une parenthèse.
    return est_losange(quadrilatere) and forme_angle_droit(quadrilatere[0],
                                                           quadrilatere[1],
                                                           quadrilatere[2])

if __name__ == "__main__":
    Point = namedtuple("Point", "x y")
    CARRE1 = (
        Point(0.0, 0.0),
        Point(0.0, 2.0),
        Point(2.0, 2.0),
        Point(2.0, 0.0)
    )
    print("est carre : ", est_carre(CARRE1))
    CARRE2 = (
        Point(0.0, 0.0),
        Point(0.0, 2.0),
        Point(2.0, 3.0),
        Point(2.0, 0.0)
    )
    print("est carre : ", est_carre(CARRE2))
```

Exercice 3 : échecs

Dans cet exercice, nous nous intéressons aux mouvements de certaines pièces sur un échiquier entre une case source et une case destination. Une case est representée par deux entiers (ses coordonnées x et y). Nous considérons uniquement des cases valides, c'est-à-dire faisant partie de l'échiquier dont la taille est `8 x 8`. Nous considérons également que toutes les cases entre le départ et la destination sont inoccupées.

Question 1
Donner l'implémentation d'une fonction vérifiant si le déplacement d'une tour est valide.

Cliquez ici pour révéler la correction.
Une tour se déplace horizontalement ou verticalement, il faut donc vérifier :

- que la position de la tour a bien changé ;
- que la position de la tour n'a changé que dans une seule dimension (x ou y).

```python
def teste_deplacement_tour(x_source, y_source, x_destination, y_destination):
    """Test le deplacement pour une tour.

```

```
    pre-condition : pas de cases occupees entre la source et la destination
    pre-condition : cases source et destination valides
    """
    return (x_source, y_source) != (x_destination, y_destination) and\
           (x_source == x_destination or y_source == y_destination)
```

Question 2
Donner l'implémentation d'une fonction vérifiant si le déplacement d'un fou est valide.

Cliquez ici pour révéler la correction.
Un fou se déplace diagonalement, il faut donc vérifier :

- que la position du fou a bien changé ;
- que le déplacement absolu est le "même" dans les deux dimensions. En `python`, la fonction `abs` renvoie la valeur absolu d'un nombre.

```
def teste_deplacement_fou(x_source, y_source, x_destination, y_destination):
    """Test le deplacement pour un fou.

    pre-condition : pas de cases occupees entre la source et la destination
    pre-condition : cases source et destination valides
    """
    return (x_source, y_source) != (x_destination, y_destination) and\
        abs(y_destination - y_source) == abs(x_destination - x_source)
```

Question 3
Donner l'implémentation d'une fonction vérifiant si le déplacement d'un cavalier est valide.

Cliquez ici pour révéler la correction.
Un cavalier se déplace "en L", il faut donc vérifier :

- que la position du cavalier a bien changé ;
- que le produit des déplacements absolus dans les deux dimensions est égale à 2.

```
def teste_deplacement_cavalier(x_source, y_source, x_destination, y_destination):
    """Test le deplacement pour un cavalier.

    pre-condition : pas de cases occupees entre la source et la destination
    pre-condition : cases source et destination valides
    """
    # il faut bouger de 1 sur une dimension et 2 sur l'autre
    return abs((x_destination - x_source)*(y_destination - y_source)) == 2
```

Question 4
Sans se fatiguer, c'est-à-dire en réutilisant les fonctions déjà écrites, donner l'implémentation d'une fonction vérifiant si le déplacement d'une reine est valide.

Cliquez ici pour révéler la correction.
Une reine a les pouvoirs du fou et de la tour, il suffit donc de réutiliser ce que nous avons déjà implémenté.

```
def teste_deplacement_reine(x_source, y_source, x_destination, y_destination):
    """Test le deplacement pour une reine

    pre-condition : pas de cases occupees entre la source et la destination
    pre-condition : cases source et destination valides
    """
    return teste_deplacement_fou(x_source, y_source, x_destination, y_destination)
        or\
        teste_deplacement_tour(x_source, y_source, x_destination, y_destination)
```

Exercice 4 : expressions conditionnelles (pour aller plus loin)

Question 1
Que penser du programme ci-dessous ?

```python
#!/usr/bin/env python3

"""Construire une chaîne de caractère en fonction d'un booléen."""

def construit_chaine_1(est_une_femme):
    """Version avec structure de contrôle conditionnelle."""
    if est_une_femme:
        return "Bonjour madame,"
    return "Bonjour monsieur,"

def construit_chaine_2(est_une_femme):
    """Version avec expression conditionnelle."""
    return "Bonjour " + ("madame," if est_une_femme else "monsieur,")

if __name__ == "__main__":
    print(construit_chaine_1(True) + " (première fonction)")
    print(construit_chaine_1(False) + " (première fonction)")
    print(construit_chaine_2(True) + " (deuxième fonction)")
    print(construit_chaine_2(False) + " (deuxième fonction)")
```

Cliquez ici pour révéler la correction.
Les deux fonctions sont équivalentes. La deuxième introduit l'opérateur ternaire conditionnel de python `a if condition else b`. Cet opérateur peut simplifier le code dans des situations de type "one value or another" sans rien faire d'autre tel que dans l'exemple ci-dessus.

Question 2
Dans la deuxième fonction, quelle est la différence fondamentale avec la structure de contrôle conditionnelle `if` que nous avons vue jusqu'à présent ?

Cliquez ici pour révéler la correction.
Nous avons ici une expression et non pas un "statement". Une expression a une valeur contrairement à un "statement". La **valeur** de `a if condition else b` est soit `a` soit `b` en fonction de `condition`.

Question 3
Est-il possible de faire la même chose que dans la deuxième fonction dans d'autres langages impératifs ?

Cliquez ici pour révéler la correction.
Oui. Par exemple en C et en Java on écrit `condition ? a : b`